Node counting in a wireless sensor network

Negi, A.

Award date:
2013

Node Counting in a Wireless Sensor Network

Amit Negi, B.E.
0828045

Master Thesis

Eindhoven University of Technology
Department of Electrical Engineering
Chair of Electro-Optical Communications
And
Manipal Institute of Technology
Department of Information and Communication Technology

In cooperation with:
DevLab
Eindhoven, The Netherlands

Supervisors at TU/e:
Prof. dr. A. Liotta
Asst. Prof. dr. G. Exarchakos

Supervisor at DevLab:
Ing. Frits van der Wateren

Supervisor at Manipal:
Prof. dr. Radhika Pai M.

Abstract

Wireless Sensor Networks (WSN) are used in many applications like tracking, monitoring etc. Many of these applications require a decentralized approach where all the nodes are equally involved in a network and there is no central controlling unit or a sink node. Moreover, a decentralized approach makes a WSN robust to network failures using the property of self organizing. These requirements stem the need for the distributed applications for the WSN. One such application is counting the number of nodes in a network. This has numerous uses like monitoring the movement of a herd or for a decision making process which requires the size of the network.

This need of distributed application and the properties of the underlying WSN platform imposes certain challenges for developing applications. In this work we have used the *MinTopK* and the *Static Bucket* algorithm for counting the number of nodes in MyriaNed. MyriaNed is a WSN which uses a gossiping approach for data dissemination. We have evaluated the performance of these two algorithms on certain performance metrics under different scenarios. Based on these evaluations, we have suggested the application where each of these algorithms can be used.

Acknowledgements

This thesis is written as part of the master project that I have performed at the chair of Electro-Optical Communications at Eindhoven University of Technology, the Netherlands, in cooperation with DevLab, Eindhoven, the Netherlands. First, I would like to express my gratitude to dr. Antonio Liotta, for believing in me and giving the freedom to explore the problem domain, while simultaneously guiding, encouraging and supporting me during the course of the project.

I am thankful to Lex van Gijsel for providing me with this opportunity and the resources at DevLab. I wish to thank Frits van der Wateren for giving me his invaluable suggestions during our weekly discussions and his flexibility for our meetings.

I would like to express my heartfelt gratitude to dr. George Exarchakos for his invaluable guidance, constant encouragement, support, timely feedback and most importantly his pep talks. I am thankful to Matthew Dobson for sharing his practical knowledge, help and most importantly hosting me at Vrije University, Amsterdam, whenever it was possible and needed within a short notice.

I am grateful to dr. Radhika M. Pai, dr. Manohara M. Pai and dr. Mark van den Brand for giving me the opportunity to pursue the dual degree program at Manipal University and TU/e. I would like to thank my seniors Abhinav, Ajith and Alok for proofreading my thesis and giving their valuable feedback. I would like to thank Madhura for being my study companion during the course of my study at TU/e with whom, working was always fun.

Last, but absolutely not the least, I would like to thank my Maa, Papa and my sisters Vandana and Ruchi for their constant support and encouragement. They are the pillars of my life.

Contents

Chapter 1

Introduction

Advances in the peer-to-peer overlay networks [1] and Wireless Sensor Networks (WSN) [2, 3] has resulted in the use of distributed networks for achieving scalability and robustness. Many a times it is required that each node should have a count on the number of participating nodes in the network. This can be used for the applications like load balancing [4] and habitat monitoring [5]. The absence of a central controlling system or a storage point in the distributed networks makes it a further challenging task [6]. This project focuses on the node counting techniques in such distributed and resource constrained networks.

1.1 Context

The work presented in the thesis is carried out at DevLab[1]. It is an alliance of 13 small and Medium Enterprises (SMEs) that was started in the year 2004 in close association with Technical Universities and institutions. The goal of this co-operation is to facilitate the exchange of knowledge between universities and the industry.

1.1.1 MyriaNed

MyriaNed[2] is a WSN platform developed at DevLab. It uses the concept of *gossiping* [7], a type of an epidemic protocol achieved through the gMAC [7](gossiping Medium Access Controller (MAC)[8]). The gMAC makes use of Time Division Multiple Access (TDMA) [9] scheme for medium access and gossiping as a communication protocol [10].

[1]http://www.devlab.nl/
[2]http://wsn.chess.nl

The gMAC makes it robust to failures and efficient [11, 12]. Some salient features of the MyriaNed are[3] :

- The absence of an addressing scheme, makes MyriaNed very low cost protocol (in terms of energy spent, message overhead and memory consumed) which is robust to network failures.
- Self-configuration, the absence of a top-down structure[13] and transparency in data dissemination aids the nodes to easily join or leave the network any time. Thus, achieving scalability.
- A heterogeneous network can be formed, allowing nodes to easily exchange data from different applications without introducing any further complexity.
- It uses a TDMA technique to resolve medium access thus helping reduce collisions.
- MyriaNed uses synchronization between nodes to increase the duty cycling [14] in which node remains active for some period and can go to sleep for most of the time for limiting the energy consumption.
- A very tight synchronization mechanism that is also capable of coping with the clock drift [15], enabling all nodes in the network to communicate in the same time frame[16].

The above mentioned properties make MyriaNed a really distributed platform (no central unit or sink node) making it apt for not just typical WSN applications [17], but also for social ad-hoc networks [16] and Body Area Networks (BAN)[11].

1.1.2 TDMA Frame Structure

The TDMA protocol divides time in a series of *frames*. Figure 1.1 illustrates a frame structure in MyriaNed. Each frame is further divided into 2 parts the *active* part and an *inactive* part.

Active (Awake/ ON) part: In this mode, all the components of the node are active, including the transceiver.

Inactive (Sleep/ Off) part: In this mode, the transceiver is turned Off as it consumes most of the power, thus increasing lifetime of a node.

The *active* part is further divided into *slots*. From the number of available slots in the active part, every node can transmit only in its *transmission(Tx)* slot. The remaining

[3]MyriaNed is described in more detail in chapter 2

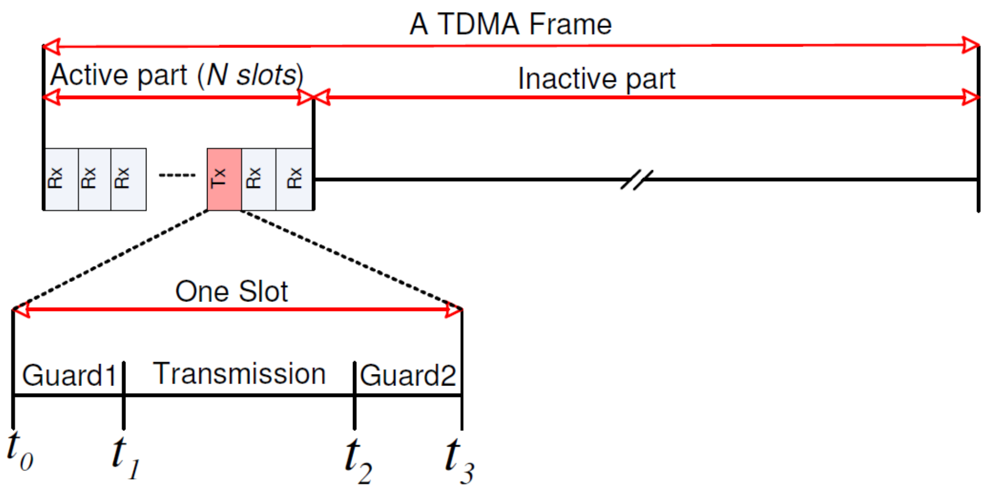


Figure 1.1: TDMA Frame Structure

slots in the active period are *receiving (Rx)* slot, during which, the node just listens to the channel and receives the messages that arrive. The received messages are then further processed.

1.1.3 Synchronization in MyriaNed

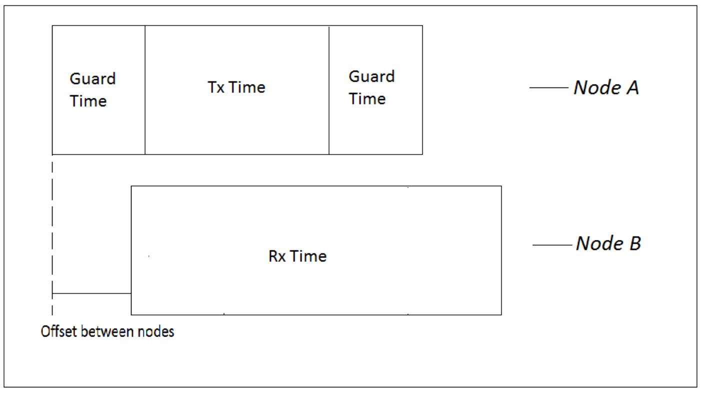


Figure 1.2: Syncronization offset

In context of MyriaNed, when the active part of two nodes overlap, then the nodes are said to be synchronized. Consider figure 1.2, in which two nodes A and B are in the transmission range of each other and the transmission (Tx) slot of a node A overlaps with the receiving (Rx) slot of another node B, implying that one node can hear other node, then we say that two nodes are synchronized.

The synchronization *offset* shown in figure 1.2, is the relative time difference by which the transmission (Tx) of node A and receiving slot of node B are separated. It shares an inverse relation with tightness of synchronization [16]. This means that, the lesser the offset, tighter the synchronization between two nodes. It also depicts that for the nodes to be synchronized, the active period of the nodes should overlap at least a bit, in fact lesser the offset, the better synchronized the network is. As the amount of the nodes increase, it becomes really challenging to maintain synchronization between the nodes, so that they can be aware of the existence of all the other nodes in the range.

This was achieved by Dobson et al [16] and they were able to maintain synchronization in the network size of up to 4096 nodes, even accounting for the clock frequency offsets and clock drift, that are inherent in a physical clock.

1.1.4 Clusters

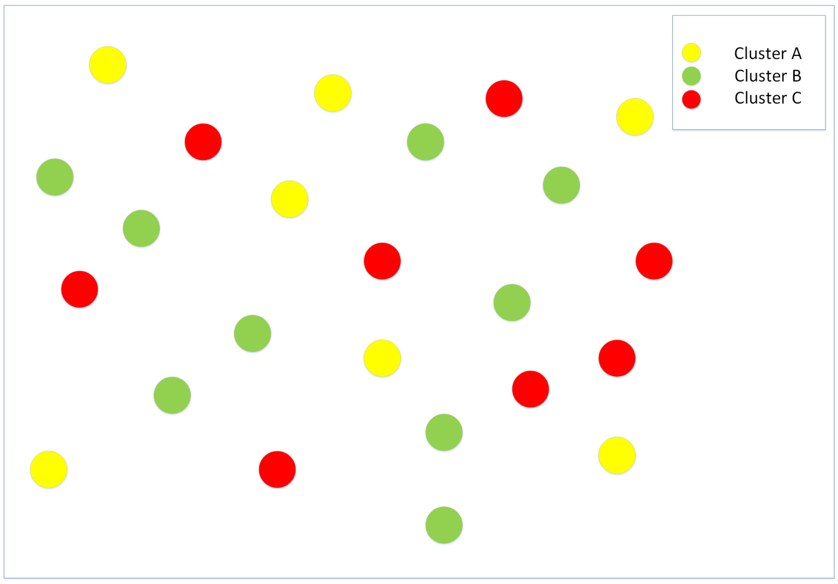


Figure 1.3: Different clusters

When the nodes start up, there might be different groups of a few nodes, which are synchronized together depicted by different color in figure 1.3. Thus, resulting in a separate cluster of nodes within a network.

As the active part of a node is very short, the node can receive messages from other nodes only for a limited duration. In a network, if this duration of all the nodes does not overlap then it results in the formation of numerous clusters. As shown in figure 1.3. Even though all the nodes are in the transmission range of each other, lack of complete synchronization results in formation of the subnetwork of nodes.

This issue was solved by merging all the clusters together to form a network. It has been achieved in a network of up to 4096 nodes [14] by sharing the active period even in very low duty cycle networks. Thus, achieving complete synchronization between all the nodes in the whole network. The solution to this problem lies in factoring it into 3 sub-problems and then addressing them separately [14]. The 3 phases are as follows:

Detection: In this phase the nodes detect the existence of different clusters.

Decision: When a node from cluster A detects existence of another cluster B, then it has to decide, whether to merge with cluster B or remain in cluster A.

Notification: In this phase, after a node has decided to merge into a new cluster, then its decision is propagated to all the nodes in its cluster.

1.2 Motivation

The process of merging of smaller clusters eventually resulting in a completely synchronized single network leads to implicit questions.

1. How should merging of the clusters take place?
2. Which cluster should merge with which one?

Once a node from cluster A is aware of the existence of another cluster B, the decision of whether to join the new cluster or remain in the same cluster has to be taken and this decision should be based on a condition. If this is left to chance and done randomly, then there is a certain possibility that the nodes might start jumping from one cluster to another and sometimes back and forth, resulting in an unstable network, thus creating further issues. Furthermore this rudimentary approach results in a significant waste of time and energy, both of which are critical in WSN.

The authors in [14] used the concept of cluster Ids. In this approach, each node would have its own unique cluster Id. Whenever, a node hears from a node with a different cluster Id, it can make the decision whether to join that cluster or not based on which cluster Id is higher. If its own cluster Id is higher, it will not change its cluster but if its cluster Id is lower then it will join the other cluster. Using this condition, to choose the cluster Id solves the problem, albeit not efficiently. The merging of the clusters is rather slow [14] increasing the convergence time of the network. Also, in certain cases, a large cluster of nodes, which has a lower cluster Id, has to merge with a much smaller cluster, whose cluster Id is incidentally higher. Thus, reducing the optimality. In the trade-off between convergence and optimality, convergence is of prime importance which was achieved by this method. What if we do not have to do a trade off between convergence and optimality? This can be achieved as described below.

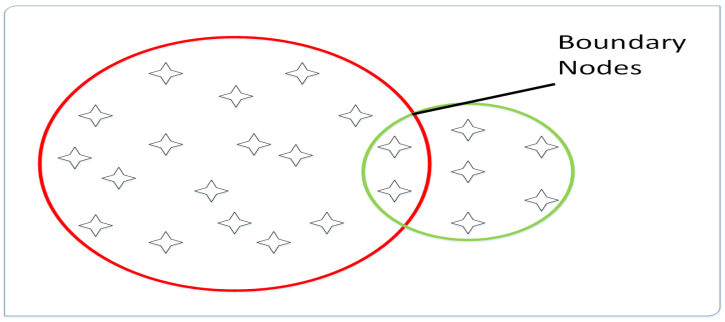


Figure 1.4: Boundary condition

Consider the figure 1.4. It shows a scenario in which there are two clusters/networks of nodes which are not synchronized. The nodes at the boundary of two networks can identify the existence of both clusters thus facing the dilemma of which network to join. The simple solution is, for a smaller network to join a bigger network, thus reducing convergence time.

For the above solution to work, it is really necessary that all the nodes should have the knowledge of the network/cluster size, prompting us to find a solution to count the size of the network. Besides the knowledge of network size can be useful for numerous other reasons like

1. Optimal decision making in the process of cluster merging or voting process.
2. Counting the number of participants in large scale social ad-hoc networks [16].
3. The vendors can build various services on it, suiting their requirements.

The inherent properties (described in section 1.1.1) of the MyriaNed like gossiping and the need for complete decentralized structure makes node counting in MyriaNed a very challenging task.

1.3 Project Goals

1.3.1 Assignment

The benefit of knowing the size of each cluster aids the process of decision making when choosing which cluster to merge to. The work described and carried out in this thesis is based on two network size estimation algorithms, the *MinTopK* [18] and the *Static*

Bucket [19]. Both these algorithms are distributed algorithms which give an estimate of the size of the whole network. The use of these algorithms is not limited by the need of any extra hardware in the form of a peripheral device or need of any sink or data collection node. Also, it perfectly suits the gossip based broadcasting scheme of MyriaNed, thus making it an apt protocol for estimating the network size. The thesis mainly focuses on the Performance analysis and comparison between MinTopK and Static Bucket algorithm with different parameters under a range of diverse practical scenarios in a simulator[20].

1.3.2 Requirements

The node counting algorithm should satisfy certain requirements. They are as follows:

1. It should be able to do so without making use of any specialized hardware device or a sink node.
2. It should result in the message size of less than 26 bytes. As, this is the available packet size for the applications in MyriaNed.
3. It should be distributed. Every node should be able to calculate the size of the network.
4. It should be trigger independent. This means, an algorithm in which an external event (like query, interrupt) results in the initiation of the algorithm, is not suitable.
5. It should be independent of the routing protocol. This means that the algorithm should not alter the way in which routing happens.
6. All the nodes in the network should be able to calculate the size simultaneously.
7. It should not require the WSN platform to have a addressing scheme.
8. The algorithm, once started on a node, should continuously provide the network size. Unless and until there is a change in the network due to nodes leaving or joining the network.

1.3.3 Objectives

The starting point of this assignment was the work of Evers et al[18], which presented some solutions for performing node counting in a WSN. The main objectives of this assignment are as follows:

1. A detailed study of all the algorithms proposed in [18] and choose a suitable candidate that satisfies the requirements mentioned in section 1.3.2.

2. Study the state of the art distributed node counting algorithms and select a candidate that meet the requirements of MyriaNed (as mentioned in section 1.3.2).

3. Implement the chosen algorithms (from step 1 and 2) on MyriaSim (a simulator based on the framework of MiXim and OMNeT++ and simulates the behaviour as MyriaNed), which lays the foundation for the evaluation of algorithms on a simulator.

4. Investigate the performance of the algorithms of step 1 and 2 under different scenarios by analysis and comparison.

1.4 Approach

The problem of counting the number of nodes in the network combined with the requirements mentioned in the section 1.3.2 results in the choice of estimation algorithms. Here, the term estimation is used because, the network size cannot be calculated exactly. Thus, using estimation algorithms provides an estimate of the network size[4]. Therefore, a study is done to find suitable algorithms.

This family of estimation algorithms uses different methods and parameters to estimate the size of the network. It would be very naive to not look into, how these parameters affect each algorithm. Hence, we experiment with different sets of parameters for each algorithm thus presenting their results and discussing and the reasons behind them.

The comparison between the algorithms is done based on their functioning and by experimenting them under various scenarios. So, we can evaluate their performance, observe the result and discuss the reason behind it, to decide which is better under what set of conditions.

1.5 Contributions

In this project, after studying the various estimation algorithms, we chose two algorithms : *MinTopK* and *Static Bucket* that are suitable to MyriaNed and present the results of the comparison between the two. The contributions are listed as follows:

[4]The accuracy of the estimate varies with the algorithm

1. Selected the best suited algorithm, from the present algorithms that address the issue of node counting in WSN.

2. Implemented the MinTopK algorithm in MyriaSim to obtain the results in a MyriaNed set-up.

3. An empirical study on the change of parameter for both the algorithms is addressed.

4. An empirical study between two algorithms is presented based on the important performance metrics under various scenarios.

1.6 Structure of the thesis

The rest of the report is organized as follows: chapter 2 describes the basic functioning of the MyriaNed. It also describes the functioning of the MyriaSim concerned with using the algorithm for estimating the network size. Chapter 3 provides an insight into the approaches that can be used for node counting. Chapter 4 describes the selected algorithms and the performance metrics used. The various scenarios, experiments and the results are discussed in chapter 5. Chapter 6 concludes with an insight into future work.

Chapter 2

MyriaNed

2.1 Introduction

A wireless sensor network (WSN), is distinguished from other WSN primarily by the device, that is being used as the node in that network and secondly, the way it functions. Different types of WSN platforms are listed in [21] and it is extensively surveyed in [2, 3]. The WSN that we are investigating for our research is *MyriaNed*. Therefore, it is necessary to have a clear understanding of the functioning of *MyriaNed*.

In this chapter, we describe the MyriaNed architecture, its scheduling, the basics of gMAC, its benefits and MyriaSim, the simulator used for performing the simulations.

The Technical Reports [13, 22] explains MyriaCore and gMAC in detail and this chapter is largely based on these reports.

2.2 MyriaNed

One of the challenges in a WSN, is to maintain the network configuration for efficient routing of messages. This necessity leads to need of extra resources in the form of memory, network resources and energy. Some WSN applications that require frequent changes in the network topology, adds to this plight and most of the energy and time is spent on updating the routing tables or maintaining the network structure. DevLab created MyriaNed with a goal of providing a cheap solution and addresses these issues, in the following way:

1. For propagating the data in the network, a *gossiping* scheme is used. The use of

gossiping provides the following benefits

(a) No need of maintaining a topology, thus reducing overhead involved in maintaining network topology and routing the data through the network.

(b) Network becomes tolerant to node failures, as there is no single path of communication

(c) The data diffusion is done without any added complexity in terms of message

(d) If a transmitted packet is lost, that doesn't mean that the data is lost. Since, the data remains in the network through other nodes.

(e) A node joining or leaving the network does not affect the configuration and data dissemination through the network.

2. The gMAC (Gossiping Medium access control) protocol used to broadcast the messages in the network employs a Time Division Multiple Access (TDMA) technique for dividing a channel/medium among the nodes. Thus compared to unicast, where each node communicates with other nodes one at a time, gossiping utilizes broadcast for communicating with multiple nodes and this results in energy saving.

3. The operating system of the sensor nodes in the *MyriaNed* is*MyriaCore*. The gMAC is tightly integrated into it. MyriaCore helps in reducing energy consumption by using very low duty cycles.

Gossiping

Instead of using a conventional approach of point-to-point communication, in which a node communicates with just a specified node, *MyriaNed* employs a gossiping scheme, in which each node periodically broadcasts its messages over the WSN to all the other nodes. This is a very quick and efficient way of flooding the data in a network and has the advantages as mentioned in section 2.2.

MyriaCore

MyriaCore is the software that is used in the *MyriaNode* for scheduling the communication applications tasks. The ability of the *MyriaNed* to attain very low duty cycles for reducing the power consumption, is attained by keeping the network very tightly synchronized. This tight synchronization is achieved by giving the *MyriaCore* complete control over the interrupt system of the processor. It makes use of the local timer to achieve synchronization with the other nodes in the network. Figure 2.1 depicts the functioning of the *MyriaCore*. It shows that the gMAC makes use of the events from the timer to synchronize the network. The *Scheduler* and *dispatcher* of the *MyriaCore* is controlled by these synchronized events and they in turn call the application. It also shows that the control of sensors and actuators is completely independent of *MyriaCore* and lies with the application itself, which uses the API's[1] from the *MyriaIo* library to control them (sensors and actuators).

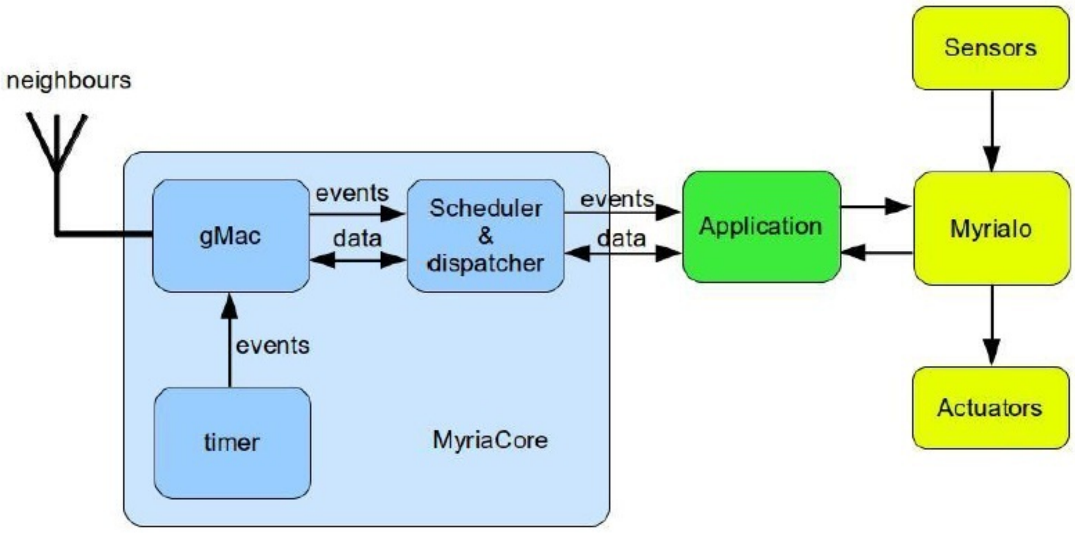


Figure 2.1: MyriaCore functioning

Gossip MAC

The gMAC makes the use of *gossiping* as a routing mechanism and a TDMA protocol for medium access. This approach helps to achieve low duty cycle by turning on the radio for a small duration. Figure 2.2 shows the timing diagram of gMAC.

The TDMA protocol employed in gMAC works as follows: Time is divided into a number of *frames* and a collection of one or more frames form a frame schedule. Each frame is also referred to as one gossiping round. The frame is further divided into *active* and *inactive* periods. The difference between these two being that a transceiver in on during active period, whereas it is off during inactive period. The active period of a frame consists of one or multiple slot schedule. Each slot schedule comprises of multiple *slot* of equal length. The slots are assigned to perform the role of either receiving a message (Rx), transmitting a message (Tx) or remaining idle (idle). Each frame has just one active slot schedule, whereas, the number of slot schedule in a frame is variable and is

[1]Application programming interface

directly dependent on the number of neighbours. Figure 2.3 shows the timing diagram of receive and transmit slot.

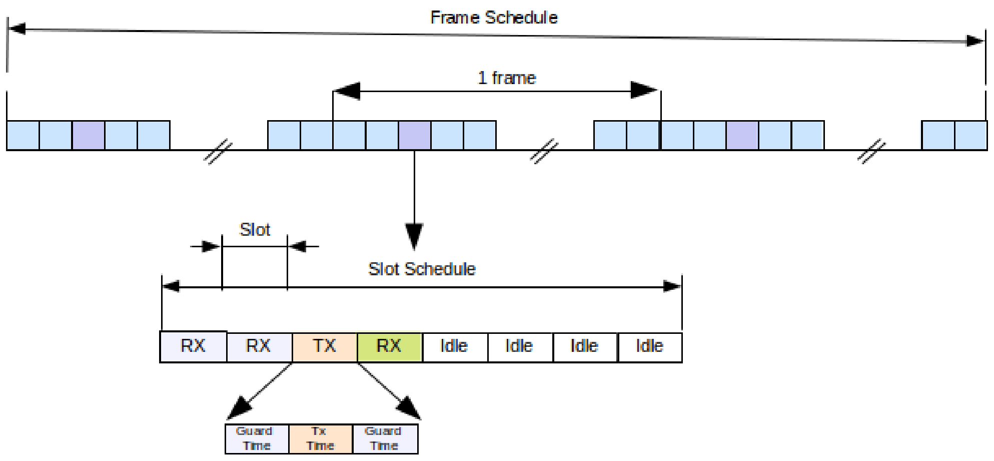


Figure 2.2: Timing Diagram of a frame

2.3 Distributed Architecture of MyriaNed

MyriaNed has a distributed architecture, which enables it to be robust to network failures, adds scalability to it and helps it to become self organized. To achieve these things along with low duty cycles, add to the challenge of maintaining synchronization between the nodes. This is achieved as follows: In *MyriaNed*, for forming a distributed network, the procedure starts with at least 2 nodes with other nodes joining it later. The steps involved are as follows:

1. *Network Initialization:* Figure 2.4 depicts the network initialization procedure in *MyriaNed*. When the node is powered on, or reset it enters the *INITIAL_LISTEN* state, if it detects an already existing network, it enters the *SYNCHRONIZED* state. Otherwise, it enters *SAY_HELLO* state, where it broadcasts that it is still not in the *SYNCHRONIZED* state. If it still does not receive any message from the other nodes, then it enters *KEEP_LISTENING* state, where it continuously monitors for messages from other nodes. It enters the *SYNCHRONIZED* state, as soon as it receives a message.

2. *Network Discovery:* At times, if, due to some reasons the node looses its connectivity with the other nodes and it cannot find any neighbours within a specified number of frames, then it becomes *disconnected* from the network. In such a case,

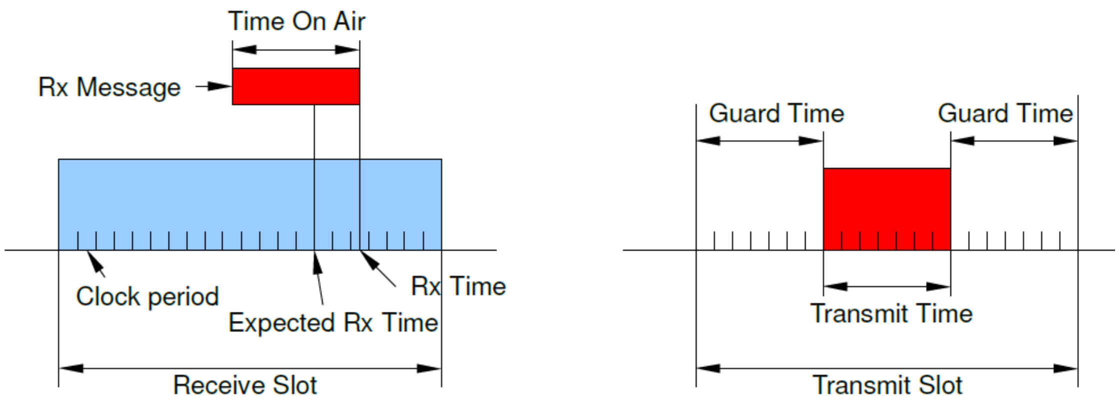


Figure 2.3: Timing Diagram of receive and transmit slot

instead of starting the node again, it enters *SEARCH* state, where it constantly monitors all the frames for any existing network, to join.

3. *Network Join:* When two small networks cannot join, then the mechanism is as follows: Every node transmits a join message in the idle period of each frame by randomly selecting a slot. This information about the slot number is carried in the gMAC header of its message. When this message is received by other nodes, then based on this information the receiving nodes will synchronize to it in the next frame. Thus, joining a network.

4. *Network Execution Model:* MyriaNed follows a *"Locally Asynchronous Globally Synchronous"* (LAGS) architecture. This means, that even though the processes that run on each node are not in sync with others, globally the whole network is synchronized. Figure 2.5 depicts the execution model of the network. At each round all the nodes *potentially* receive a message from the neighbouring nodes (denoted by blue blocks). Upon receiving, these messages are then evaluated (denoted by green blocks). The word potentially is used because due to the broadcasting nature of the nodes, there is no guarantee that every transmitted message will be received. The height of each block represents the power consumed by that activity. As, it can be seen from the figure that, *communication* consumes more power than *computation*.

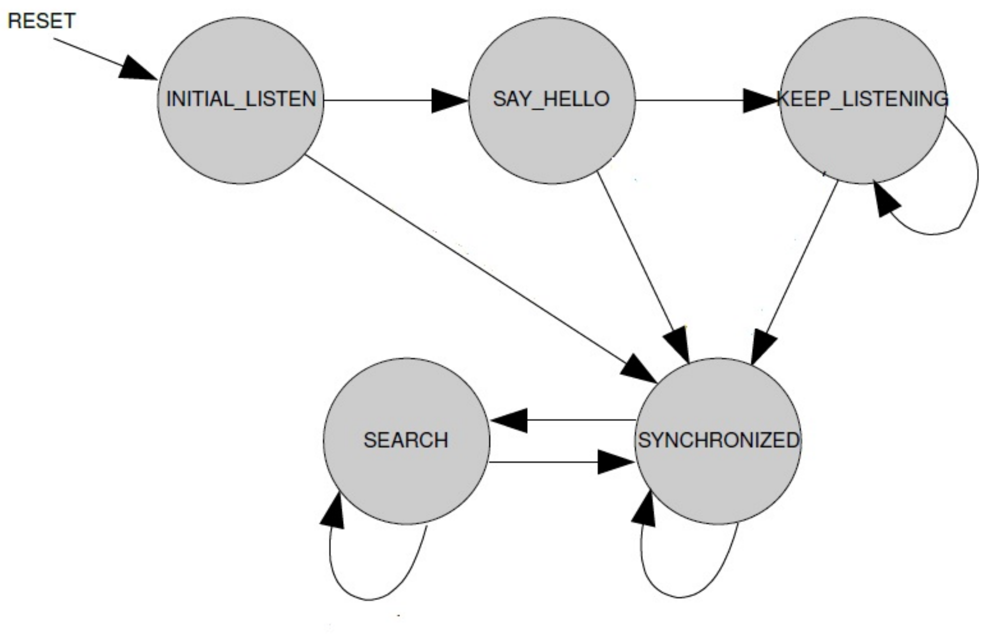


Figure 2.4: Network Initialization

2.4 Scheduling

As we explained in section 2.2 that *MyriaCore* is the operating system of nodes in *MyriaNed* and gMAC is the timing master of *MyriaCore* which determines the scheduling of frames. *MyriaCore* deals with the scheduling of the necessary functions that facilitate the execution of the applications. There are 3 such compulsory functions, which every node is bound to execute and they should be part of the application code. They are as follows:

1. *appInit():* This function is called just once during the entire lifetime of a node, when the node is first started. It is mainly used to initialize the application and set the data structures, pointers, variables to their default values.

2. *appEvalRxMsg():* This function is embodied in the application and *MyriaCore* uses this function, every time it receives a *valid* message. A message is called valid, if it has been received from a node in the same frequency band with a specified header and has a correct CRC[2]. The data in the this message is lost after the completion of the function, therefore the application should either consume this data or store it in the memory.

[2]Cyclic redundancy check

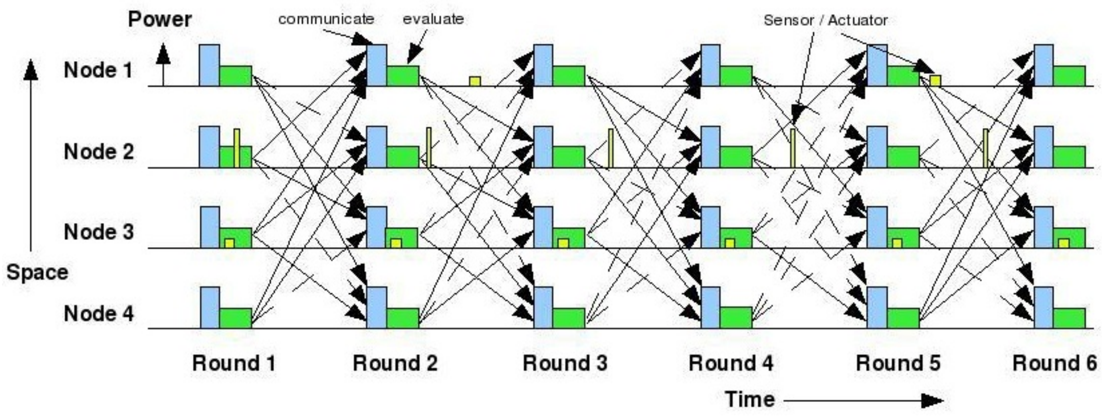


Figure 2.5: Network Execution Model

3. *appPrepareTxMsg():* During the active period *MyriaCore* calls *appEvalRxMsg()* function and it calls *appPrepareTxMsg()* after the end of active period. It is during this duration, that the application should process the received data and prepare a message to be broadcast in the next round.

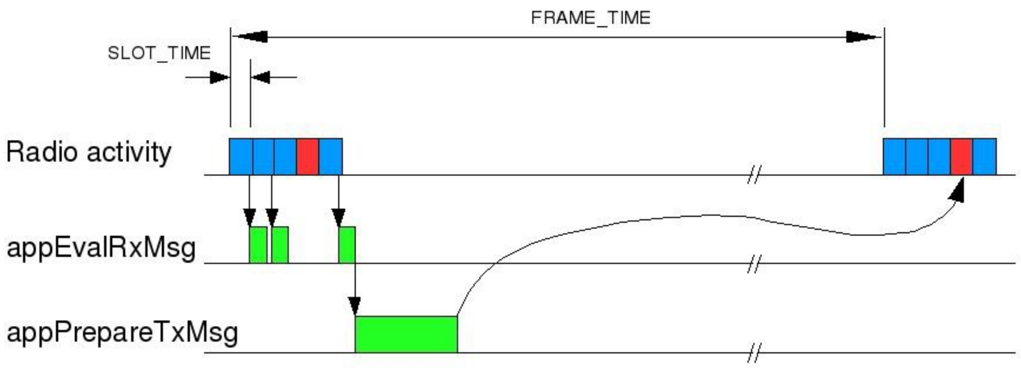


Figure 2.6: MyriaCore: Scheduling

Figure 2.6 depicts the timing diagram for the application scheduling by *MyriaCore*. We can see that the active period of each frame is divided into 5 slots each (numbered from left to right). The transmission (4^{th}) slot is shown in red whereas the receiving slot is in blue. After a valid message is received in receiving slot $(1, 2, 5)$, the *appEvalRxMsg()* function is called by the *MyriaCore*. At the end of the active period of the frame, when all the messages have been received, *MyriaCore* calls the *appPrepareTxMsg()* function. The processing on received data is performed by this function and the resulting message is then transmitted in the next active period. After finishing *appPrepareTxMsg()* function, *MyriaNode* will go to sleep mode and will wake up at the start of the next frame.

2.5 MyriaNed Components

The main components of *MyriaNed* are as follows:

1. *MyriaNode:* The wireless devices which are used as nodes in the *MyriaNed* consists of the nRF24L01 as the radio device and the processor is Atmel ATXMega128a1.

2. *Transceiver:* The transceiver used in *MyriaNed* is Nordic nRF24L01, which is a packet based radio capable of message encoding, decoding and assembling/disassembling. It operates in the 2.4 GHz ISM band and has a bit rate of 2 Mbps.

3. *Message Structure:* The actual packet size which the Nordic radio can carry is 32 bytes. Out of this, 6 bytes are needed by gMAC, for its own functioning. Therefore, the effective size that is available for an application in each packet is 26 bytes.

2.6 MyriaSim

The simulator used is *MyriaSim* [23], which is based on OMNeT++ [24], an open-source simulator and MiXiM [20], which provides extensions for mobility and wireless network protocols.

In *MyriaSim*, OMNeT++ is used to extend the MiXiM framework to provide support for *MyriaNed* by adding the gMAC protocol. Also, the simulator makes use of the modules, which uses the parameter of actual hardware. Example: In *MyriaSim*, the module for the radio of the nodes is modelled according to the specifications of actual radio nRF24L01, which is used in *MyriaNode.*

Since, we are interested in the implementation of the node counting algorithm and its functioning in *MyriaSim*, we explain in brief the structure of the modules of *MyriaSim* for achieving it.

Figure 2.7 depicts the functioning of the modules in *MyriaSim* and how it can be used for the implementation of the algorithms that estimates the size of the network.

The base one is the physical layer module that deals with the actual transmission/ reception of the data just like in the physical nodes. The gMAC module encapsulates the functioning of the gMAC and it interacts with the physical module. The application module, consists of the application that are to be used in *MyriaNed.* The application

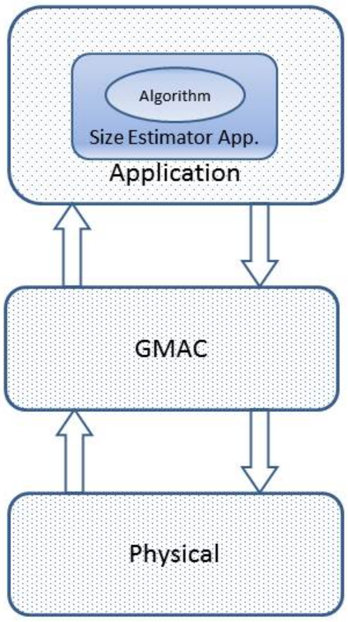


Figure 2.7: MyriaSim: Structure

module is independent of the physical module and interacts with the it through the gMAC module.

One of the application modules is the *Size Estimator Application.* The size estimator application module does the work of calling the *algorithm* module. It is this algorithm module, where algorithms for estimating the size of networks are placed. The size estimator application module takes the simulation parameters from the *.ini* file and calls the specified algorithm (Example: MinTopK, Static Bucket) from the *algorithm* module with the set parameters. The parameters which are specific for running the simulation are set in the *ini* [24] file of the simulator.

Chapter 3

State of the art

3.1 Introduction

Estimating the network size (or node counting in a network) is a topic of great interest, evident from the active research in this field [25, 26]. Some of the applications of it are listed in section 1.2. The choice of the algorithm for counting the nodes is influenced by many factors, few of them are enumerated as follows as follows

1. The application for which it is being used.
2. The underlying hardware approach. Example: The type of nodes being used, use of a sink node etc.
3. The protocol being used for routing. Example: Gossiping [7], ZigBee [27]
4. The desired accuracy. Example: accurate or just a rough estimate of network size is acceptable which has certain error.

For our work, we are investigating the performance of the node counting algorithm in *MyriaNed*. Therefore, the requirements for the appropriate approach are enumerated as follows:

1. The working of the approach should be independent of the knowledge of the structure of the network.
2. The approach should be capable of working in a distributed environment/network.
3. The approach should be able to work in a gossiping network.

4. Each and every node in the network should be able to estimate the network size at any given instant.

5. The message size is limited to 32 bytes by *MyriaNed*[1] bytes.

6. It should not make use of the extra hardware.

7. It should work independently of the knowledge of the degree[2] of the nodes.

8. The approach should work even in the networks, where collision detection of the transmitted messages is not possible

9. the approach should be memory efficient and should work on the minimal amount of memory.

10. A single node should not act as a trigger or query generator, for estimating the network size.

11. It should be able to use the algorithm as a standalone application; use of approaches, which does not require altering the routing scheme are required.

12. The approach should provide a continuous estimate of the network size.

13. The approach should not work in phases. This means that for giving an estimate of the network size, the approach should not execute multiple phases. Example: capture-recapture.

14. Algorithms, which approximately estimate the network size and lead to the saving of resources, in terms of memory and power, with a certain accuracy are preferred over the ones which give accurate estimates, but with huge penalty in terms of memory requirement and message overhead.

In this chapter we present few well known approaches which are used for estimating the network size. We start with presenting some simple approaches, then we discuss some node aggregation approaches, which are used for aggregating different function values (like count, sum and average [25]) in a node and classify them according to certain criteria like computation technique or network structure [25]. We then, present the algorithms that we chose to evaluate and reason the motivation behind it.

[1]The effective space available is 26 bytes, since *gMAC* uses 6 bytes for its own purpose.
[2]A degree of a node specifies the number of direct physical connections to other nodes

3.2 Naive approaches

In this section, we describe some of the very basic or simple approaches which can be used for estimating the network size, but do not seem appropriate for our use due to certain reasons.

3.2.1 Epidemic Protocol

With the issue of counting the number of nodes in a network, the simplest idea that occurs, is to store the unique identifiers (Node Id) of all the nodes in the network at each and every node, where the number of Node Id's gives a count of network size, N. Every time the node propagates, the style of propagation for every node would be to select a random Node Id from its memory and broadcast it.

Though this idea is very simple, the main problem with this approach is that it requires a huge memory size allocated for storing Node Ids. In a network of N nodes, each node would need a space of $(N * k)$ bytes, where k is the number of bytes required to denote node-id of each node. This grows with the size of the network, thus demanding huge memory requirements, which is a scarce resource in WSN. Also, the time required to propagate all the node-id's till each node has all the Node Id's in the network, would be very large. Therefore, this approach is not suitable for our use.

3.2.2 German Tank Problem

Another very simple approach used to find the estimates, was actually used by the allied forces during world war II for counting the number of German tanks. This is famously known as the German tank problem [28].

It works on the principle, that if the population is *serially* numbered, then based on k samples, which are sampled from the population, it is possible to determine the total size of the population using the formula mentioned in the equation 3.1. m is the maximum value observed for k samples.

$$N \approx (m-1) + \left(\frac{m}{k}\right) \tag{3.1}$$

The detailed study of this approach is done in [29], wherein it is mentioned that this procedure was used to estimate with great success many things like rockets, tires, tanks to name a few.

The problem with this approach is that, due to the applications of a WSN, it is going to be a tough task for all the nodes in a network, to be numbered serially at any given instant.

3.3 Distributed data aggregation

Distributed data aggregation is a referred to as the process to gather some aggregates like *count, sum, average* of various functions or values in a distributed network. It can also be used to calculate the size of the network in WSN [25]. The techniques used for data aggregation are classified and surveyed in [30, 25]. We discuss some of the existing techniques that are pertaining to *MyriaNed*.

The suitability of the data aggregation technique being used, depends mainly on how the *network structure* is and what *computation principle* is used to compute aggregates. [25].

3.3.1 Network structure

Structured

Some aggregation algorithms, as classified in [30], require the network to be *structured* in some sort of structure like *tree*, *ring* or *clusters*. Although, this approach helps to compute aggregate in as faster way, this is not a suitable approach for us. Because, this requires an additional overhead (in terms of time) to create a network in the form of a structure first. Also, in mobility scenarios, where the topology changes fast, creating and maintaining these sort of structure requires unnecessary demand in terms of power and memory. In certain cases, this approach requires the routing algorithm to work in a desired way. Hence, we look for an *unstructured* approach, where the routing scheme used is *flooding* or *gossiping*, since, this is what the *MyriaNed* uses.

Unstructured network

This type of approach does not require the network to be in a particular structure, thus perfectly suiting to the principle of *gossiping*, that is embodied in *MyriaNed*. Here, we study some popular approaches, which have been employed in unstructured networks.

Randomized Reports This is a simple approach mentioned in [31] wherein, a node generates a query (known as source node) for estimating the network size, with a sampling probability p and propagates it. The receiving node replies with its response (whether present or not), based on this p. After generating a query within a pre-defined time interval, the source node can calculate the network size N, based on the number of received replies r by using the formula 3.2.

$$N = \frac{r}{p} \tag{3.2}$$

This approach requires, the query to initiate at one node, thus making it unsuitable for our requirement.

Random Tour With this approach, each node starts a query, and every other node receiving that query propagates it further, after processing it in a certain way, thus, creating a *random walk*. When the query is ultimately reached at the initiating node, it is termed as a *random tour*.

A scheme proposed by Massoulie et al. [32] works in the following way for estimating the size of the network. Let every node store a value 1. The source node x (node starting the query) initiates a random variable X with value $\frac{1}{d_x}$, where d_x, denotes the number of nodes, node x is connected to (i.e. degree of node x). After receiving X from node x, node y processes it based on equation 3.3

$$X \leftarrow X + \left(\frac{1}{d_y}\right) \tag{3.3}$$

When the tour is completed, the network size N can be found out by the source node x using equation 3.4

$$N = d_x * X \tag{3.4}$$

Although, the algorithm is very simple, it is not resilient to a scenario, where the neighbouring nodes change frequently, because the network size estimation for node x is directly related to the degree of the node x. Such situation, where the degree changes, can occur due to node failures or in a mobile scenario, where the neighbouring nodes keep changing frequently.

Push-Pull Protocol This approach was presented by Evers et al [18], which is the adaptation of the work done by Jelasity et al [33, 34] and Kempe et al [35] wherein, they have proved that, using an epidemic style of anti-entropy protocol [36], in the network which is fully connected, with all the nodes having bidirectional links, it is possible to estimate the size of the network.

In this approach, for estimating the size of the network, a node x at time instant $t = 0$, starts with a weight of 1, denoted by $W_x^0 = 1$. Whereas, all other nodes $y \neq x$ start with value $W_y^0 = 0$. In each round, half of the actual weight, is transferred to the node which is selected *uniformly* at random. Thereafter, at every round t, upon receiving a value from node x, each node y updates its value according to the equation 3.5. At any time instant t, node x estimates, the network size, N, according to the equation 3.6. The accuracy increases as time increases, thus slowly converging to the accurate value [35]. The authors proved this based on the *conservation* property , which states that at time instant t, the weight carried by all the N nodes in the network is 1. This property is formalized in equation 3.7.

$$W_y^t \; = \; W_y^t \; + \; W_x^t \tag{3.5}$$

$$N = \frac{1}{W_x^t} \tag{3.6}$$

$$\sum_{i=1}^{N} W_i^t \; = \; 1 \tag{3.7}$$

As we can observe, the problem with this approach is that it is too idealistic, in the sense, that it does not take into account the message loss encountered while transmitting or receiving (very common factor in WSN). The authors in [35] assume a *detection mechanism*, which alerts the node about the failure of the transmitted message, in that case, the node restores its weight by again sending the message to itself. This mechanism does not exist in *MyriaNed*. Also, contrary to our need that all the nodes should be able to start the process at any given instant t, this approach requires a single node x to initiate the process.

3.3.2 Computation Principle

In this section, we discuss some approaches, which use a particular type of computation principle for the aggregation of data. Thus, the network size.

Hierarchical

This sort of approach requires the network to be arranged in a structured way as mentioned in the section 3.3.1, so that the computation is hierarchical in nature. The simple example of its working is as follows: Upon receiving the query for the network size, the head of each cluster known as *cluster head*[3] provides the information on the number of nodes in that cluster to the source node, the source node (from which the query is generated) then combines the information it receives from all the cluster heads and uses it to count the network size. Algorithms, which employ this mechanism for aggregation produces fairly accurate results, but they are not fault tolerant [25], and they seldom have a single source of failure. An example would be, failure of the *cluster head* in a cluster of nodes, or a *root node*[4] in a tree structure. We avoid use of this approach for reason mentioned in section 3.3.1.

Sketches

As the name suggests, the use of this approach involves creating a data structure of fixed size, which is termed as the *sketch.* This sketch carries the information of all the nodes in the network about the desired function. There are various ways of making, updating and merging these sketches and different sketching techniques have different computational complexities and accuracy [37]. The special properties of using these sketches are that they are *duplicate insensitive*[5], thus enabling it to be implemented independent of routing topology and it is very fast [19].

A function f is said to be duplicate insensitive, if its functioning is unaffected by duplicate values. Example: Let f_{count} be a function which can count the number of unique elements in a set

$$f_{count}(3, 2, 1, 1, 4, 4, 5, 4, 5) = f_{count}(1, 4, 2, 5, 3) = 5$$

We can say that function f_{count} is *duplicate insensitive.*

Sketches are used widely in the field of database [38] and peer-to-peer computing for counting the number of distinct elements in a a stream of data. The various methods used for counting the distinct element using sketches are researched in great detail by Metwally et al [37].

The seminal work by Flajolet and Martin [38] is the first method of creating sketches using the *hashing function*[6]. This is famously known as *FM sketches*, named after its creator. It is explained in chapter 4. The authors in [38] also proposed a method called as PCSA[7] algorithm, which employs a mechanism of using multiple sketches and averaging

[3]In a group of nodes(also know as cluster), the node which has all the information about all the nodes in that group is known as *cluster head.*

[4]In a tree structure, the node which has no nodes on top of it, but has leaf nodes, is referred to as *root nodes*

[5]In a gossiping network with no addressing mechanism, where nodes keep broadcasting its values continuously, any node will receive, the same value more than once. Hence, the issue of duplicity arises.

[6]A function which maps the data of variable length to a data of fixed length

[7]Probabilistic counting for stochastic averaging

the result of estimates of each sketch, to improve the result. The improved algorithm, *LogLog* [39] achieves reduction in required memory resources over PCSA. The optimized version of *LogLog* is *Super-LogLog*, which optimizes the memory usage to achieve even more reduction in estimate error.

The problem with the *PCSA, LogLog* and *Super-LogLog* algorithms is that, even though these approaches are able of counting distinct elements with really good accuracy, the memory requirement for them, is in order of few hundred KB[8] [37]. This makes them unsuitable for use in WSN[9].

Extrema Propagation *Extrema propagation* [26], is an algorithm based on sketches, which has been employed in a sensor network for aggregating data and can be used for estimating the network size, N. The approach works as follows: Each node x maintains a vector vec_x of k random numbers. The random numbers are generated using a random number generator having the exponential distribution with parameter 1. In every round, each node x propagates its vector vec_x to its neighbours. After receiving a message from node y, which contains vec_y, node x calculates the point wise minimum of both the vectors, vec_x and vec_y. The estimate, N_{est} of the network of size N, is calculated using the equation 3.8.

$$N_{est} = \frac{k-1}{\sum_{i=1}^{k} vec_x[i]} \tag{3.8}$$

The authors have achieved the results with 4 percent error, by maintaining the value of k as 2400, thereby reducing the message size to 1500 bytes [26]. This is a great reduction over *Log-Log* and *Super-LogLog* approaches which require message size in the order of a few hundreds of KB [39].

Static Bucket This algorithm falls under the category of algorithms that make use of the sketching technique. Thus *Static Bucket* inherits the properties of being *duplicate insensitive* and capable of estimating the network size quickly, compared to other approaches. *Static Bucket* has also a unique capability of estimating the network size within the neighbourhood of k hops, thus, this property makes it very special. The message overhead incurred and the memory requirements are both limited by a same factor, which is number of *buckets* and are in the order of few bytes.

[8]Kilo Bytes
[9]*MyriaNed*

Probabilistic Counting or Sampling

The algorithm under this section make use of some *sampling* technique [40, 41] for estimating the size of the network or some *probabilistic* [18] estimator based on certain observations. Here, we discuss a few algorithms which can be used in estimating the network size based on this approach.

Sample and Collide This algorithm by Massoulie et al.[40] works on the principle of random walks (described in section 3.3.1) for creating a collection of uniform samples. The method for sampling is as follows: each node x (known as source node) sends a message which contains a timer, set to some pre-defined value T (should be large). Upon receiving the message, node y uniformly chooses a random number, $X \in [0, 1]$ and the new value of T is set according to equation 3.9. After updating T, if $T \leq 0$, then the process stops, else, it is sent to one of the neighbouring nodes, which is chosen uniformly. When the process is stopped, the message is sent back to the source node. The network size N can be estimated using equation 3.10.

$$T \leftarrow T \; - \; log\left(\frac{(1/X)}{d_y}\right) \tag{3.9}$$

d_y is the degree of node y.

$$N \; = \; \frac{C_l^2}{2l} \tag{3.10}$$

C_l is the total number of samples taken for l repetitions. Clearly, the accuracy of the estimate depends largely on the parameter l.

The performance of this algorithm is dependent, on the ability of the node to generate samples with uniform distribution [40].

Capture-Recapture Mane et al[41], proposed an algorithm *Capture- Recapture*, which works on the principle of random walks. Wherein, the source node that wishes to estimate the size of the network, conducts two random walks (capture and recapture). The working of each of this walks is as follows: the source x, sends a message to a randomly selected neighbour node y, containing a timer set to T, which is a pre-defined value. Upon receiving, node y decrements the timer value T by 1 and transmits this message to another neighbour which is randomly chosen. This process continues till the timer, $T = 0$ or the message reaches the source node. From the start of the walk till its end, the identifier of each and every node that message traverses through is stored in the message. The network size, N, is estimated according to the equation 3.11

$$N \; = \; \frac{((n_1 + 1) \; \times \; (n_2 + 1))}{(n_{12} + 1)} \tag{3.11}$$

n_1 signifies number of nodes that were captured in first walk. n_2 signifies number of nodes that were captured in second walk. n_{12} signifies the number of nodes that were captured in first walk and recaptured again in second walk.

Apart from the issues inherited because of the random walks (mentioned in section 3.3.1), this approach requires a closed network, that is a node should not leave or join the network, while the process of conducting random walks is in progress. Another serious issue, this algorithm fails to address, is the lack of the ability of the nodes to detect the failure of successful transmission of messages (because of the collision) to another node, which is a very common issue an WSN.

MinTopK *MinTopK*[18] can be classified as probabilistic counting. The strength of the algorithm lies in its simplicity of functioning. The message overhead incurred by this algorithm is of order of few bytes[10] and the memory requirements are also very less, which is limited by the list size K and size of each identifier. Further, the encouraging results presented by the authors in [18] adds to its simplicity.

3.4 Choice of algorithms

Table 3.1 presents the summary of, the approaches and the requirements they satisfy. As we can see that *MinTopK* ans *Static Bucket* satisfy all the requirements. Therefore, we chose *MinTopK* [18] and *Static Bucket* [19] for the analysis. They are described in detail in chapter 4.

[10] 4 bytes in our case

Table 3.1: Summary: Requirements satisfaction of the approaches

	Requirements											
Approaches	Structural independence	Distributed	Gossip	Extra Hardware requirement	Nodes estimate simultaneously	Degree of nodes	Collision detection	Multiple phases calculation	Memory efficient	Trigger independent	Continuous	Message ≤ 26 bytes
Epidemic Protocol	✓	✓	✓	✗	✓	✗	✗	–	✗	✓	–	✓
Structural/ Hierarchical approach Network	✗	✓	–	✗	✓	✗	✗	–	✓	✗	–	–
Randomized Reports	✓	✓	✓	✗	✗	✗	✗	–	✓	✗	✗	–
Random Tours	✓	✓	✓	✗	✗	✓	✗	–	✓	✗	✗	✓
Push-Pull protocol	✓	✓	✓	✗	✗	✗	✓	–	✓	✗	✗	✓
Extrema Propagation	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✗
Static Bucket	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
Sample and Collide	✓	✓	✓	✗	✗	✗	✗	–	✓	✗	✗	–
Capture-Recapture	✓	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	–
MinTopK	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓

Chapter 4

Selected Algorithms and Performance Metrics

4.1 Static Bucket

In this section we explain the working of *Static Bucket* [19] algorithm. We first start with explaining how Flajolet-Martin (FM) sketches are used for counting the distinct elements in a stream of data, gradually moving, to how it can be used to estimate the number of nodes in a network. Finally, we explain, how the *Static Bucket* algorithm, as proposed in [19] makes use of the technique of FM sketches in calculating the network size within k hops by introducing a concept of timestamps.

4.1.1 Estimation from FM Sketch

Here, we consider that each node has a unique node id denoted by n_{id}. To count the number of nodes, a simple idea would be to store all the distinct n_{id}'s. Suppose that each node store all the received n_{id} in a set S. A simple count of the number elements in S would tell us the number of the nodes in the network. But, this approach would require a node to maintain a big list which will keep on increasing, as we increase the size of the network. This results in a need of huge memory requirements, which is already a scarce resource in WSN. An efficient and powerful algorithm which estimates the number of the distinct elements in a set S was presented in [42]. This technique, known as Flajolet-Martin (FM) sketch can aggregate long data streams, taking care of the duplicity of the values. Each data item is hashed by a hash function as shown in equation 4.1.

$$h_g(item) = value \tag{4.1}$$

The hashing function used, is a geometric hashing function(h_g) with the parameter p=$\frac{1}{2}$. Equation 4.2 shows that using such hashing function, the probability of obtaining a value, is equal to the inverse of 2 to the power of that value. The processed data set is denoted with P_d and p_n denotes the n^{th} data item in P_d. The FM sketch of P_d, that is, FM(P_d) with i^{th} bit is calculated as shown in the equation 4.3, where m,n $\geq$ 1.

$$\mathbb{P}\left(h_g(p_n) = m\right) = 2^{-m} \tag{4.2}$$

$$FM(P_d)_i = \begin{cases} 1 & \text{if } \exists p_n \in P_d \text{ such that } \; h_g(p_n) = i \\ 0 & \text{else} \end{cases} \tag{4.3}$$

$L_i(P_d)$ denotes the lowest index of an FM sketch, FM(P_d) that has the value 0. The estimator in the equation 4.4 gives the approximate number of the elements, with the error magnitude within one bit. The value of $\phi \approx 0.775351$. The detailed proof for the same is in [38].

$$\text{Estimator} \;=\; \frac{1}{\phi} 2^{L_i(P_d)} \tag{4.4}$$

4.1.2 Network Size Estimation

The FM sketch technique described in section 4.1.1 is further adapted to estimate the number of the nodes in the network. The following adoption works as follows.

As we described in the section 4.1.1, n_{id} denotes an id generated by a node. Let $(n_{id})^t_x$ $\in \mathbb{N}$ denote a node-id generated by the node x at time instant t. These id's are hashed using the geometric hash function described in section 4.1.1. Let $X \in \mathbb{N}$ be the output of the hash function.

$$h(n_{id})^t_x \;=\; X^t_x \tag{4.5}$$

If, A_{rr} is a set which stores a pair of (n_{id}, t), then the i^{th} element of the FM sketch of A_{rr} becomes,

$$FM(A_{rr})_i = \begin{cases} 1 & \text{if } \exists(x,t) \in A_{rr} \text{ such that } \; X_x^t = i \\ 0 & \text{else} \end{cases} \tag{4.6}$$

4.1.3 Static Bucket: The algorithm

The FM sketch method described in the section 4.1.2 is further modified by the authors in [19], to adapt, so that it can be used to measure the network size within k-hops. For this very purpose, the authors of *Static Bucket* introduced the concept of *timestamps*, $T_s \in \mathbb{N}$. As a result, the array is now capable of storing an integer value, instead of a bit. This integer is referred to as *buckets*.

The concept of the time stamps works in the following fashion. Each bucket stores a value i.e. time stamp in the interval [0, T_s-1]. After each round (passage of time), the timestamps are reduced by 1. This means, the higher the value of the buckets the more recent/fresh the data is. Each node carries an array of the timestamps, AT_s (each of which is contained in a bucket). The array of the time stamps for a node x at time instant t is denoted by $(AT_s)_x^t$. Each element of the array of time stamps, (AT_s), is a positive integer representing the time stamps. This integer value is then mapped to a bit value (either 0 or 1). After mapping all the integer values from the array (AT_s) to bits, a new array is created, let this be denoted by $(AT_s)[a]$, where $a \in \mathbb{N}$. The ith bit of $(AT_s)[a]$ is calculated as shown in equation 4.7.

$$(AT_s[k])_i = \begin{cases} 1 & \text{if } AT_s\,[i] \;\geq\; (T_s - k) \\ 0 & \text{else} \end{cases} \tag{4.7}$$

The algorithm is as follows.

- Initialization (time, t=0): Each node x sets all the buckets to value 0, except the bucket in which X_x^0 falls. In which case that bucket is set to $T_s - 1$
- $\forall$ time, $t \geq 1$
  - Broadcast: At any given instant t, each node x maintains an array $(AT_s)_x^{t-1}$ from previous rounds[1]. It reduces the present array of timestamps in the

[1]each round is a unit of time t

following way

$$M_x^t := \big(\ (AT_s)_x^{t-1}\ -\ 1\big)\ \ \vee\ \ 0$$

The node x, then refreshes its corresponding bucket to the value $T_s - 1$. In short, $M_x^t(X_x^0) := T_s - 1$. The node x then broadcasts the message M_x^t and the current array is updated to $(AT_s)_x^t = M_x^t$.

– Receiving: At a given time instant t when node x receives the message M_y^t from node y. Node x then updates its array of buckets $(AT_s)_x^t$ in the following manner

$$(AT_s)_x^t\ =\ (AT_s)_x^t\ \ \vee\ \ M_y^t$$

- Estimation: As in the case of equation 4.4, at any given instant t , $(L_i)_x^t[k]$ denotes the lowest index having a value 0, in the bitvector $(AT_s)_x^t[k]$, then the estimate for the k-hop network is given in the equation 4.8.

$$\text{K-hop Estimate}\ =\ \frac{1}{\phi}2^{L_i[k]} \tag{4.8}$$

This is how the *Static Bucket* algorithm works, the detailed proof of the estimates and the various other formulae is presented in [19]. By making use of the technique of time stamping, it can work in the scenario where a network split or network merge happens.

4.2 MinTopK

In this section we explain the working of the *MinTopK* [18] algorithm. First, we explain the required basis for the algorithm, then we describe the proper functioning of the algorithm.

4.2.1 Explanation

MinTopK works on the principle of probabilistic counting by means of maintaining a list of the top K values at each node, a list which each node contains. Each node x in a network of N nodes generates its own Node Id, $(n_{id})_x$. The generated Node Ids should be in the interval [0,1) as explained in equation 4.9 and should follow the following properties

$$[0,1) = \{x \in \mathbb{R} \mid 0 \leq x < 1\} \tag{4.9}$$

1. Uniqueness: This property simply means that in a network of N nodes, the n_{id} generated by each node should be different from the n_{id} generated by the other nodes. Equation 4.10 formalizes this property.

$$\forall\ ((n_{id})x\ \in N)\ \ \exists\ \ (\ (n_{id})y \in N \mid (n_{id})x \neq (n_{id})y\ \ ) \tag{4.10}$$

2. Uniformity: This property simply means that in a network of N nodes, the randomly generated Node Id's should follow a uniform distribution amongst themselves. In a network of N nodes, let the Node Id's generated by nodes n_1,n_2, n_3, ..., n_n be as $(n_{id})_1$, $(n_{id})_2$, $(n_{id})_3$, ..., $(n_{id})_n$, then the generated Node Id's should be uniformly distributed with respect to each other.

The principle for estimating the network size, stems from the fact that, since the Node Id's are uniformly distributed and each node contains top K values, where s is the smallest of the top K values, then if the interval [s,1) contains top K values, the network size can be estimated by equation 4.11.

$$Estimator\ =\ \frac{K}{(1-s)} \tag{4.11}$$

As soon as the node is turned on, it will create a list of size K with all the elements as 0, termed as an empty list. Then it generates its own Node Id and stores it in the list. To store a list of top K values each node has to follow a procedure. At the time of the initialization every node prepares a message, which consists of its own generated Node Id and transmits this message in the first round. After that each node which receives a message will store the receiving Node Id, in the message by replacing 0 from the list. This process will continue till the list is filled with all non-zero values. After that, for each value that is received in the message, the receiving value is compared with the smallest element s, from the list and at any given instant, the top K values (in order of magnitude) of the node-ids are stored at a node. After, some time all the nodes in the network converge to the top K values of the generated node-ids. The algorithm precisely works as mentioned in section 4.2.2.

When the list is full, s denotes the smallest element, of the list of size K, in such case, at any given instant t, the estimate of the network size can be done simply by the formulae $\frac{K}{(1-s)}$. The case, when the list is not full, the total number of non zero elements gives the network size.

4.2.2 MinTopK: The Algorithm

The *MinTopK* is as follows,

- Initialization (time, t=0): Each node x creates a list of K elements (here K is the size of the list) and puts the value of all the elements of the list as 0. It then generates its own node id, $(n_{id})_x$ and stores it as the first element of the list.
- $\forall$ time, $t \geq 1$
  - Broadcast: At any given instant t, each node selects a non-zero element from the list and transmits it.
  - Receiving: Whenever a node receives a message from a node y, then if the list of not full yet, then it stores $(n_{id})_y$ in the list. If the list if full, then it compares the value of $(n_{id})_y$ with s, the smallest element of the list. If $(n_{id})_y > s$, then s is replaced with $(n_{id})_y$ else, the received value is discarded.
- Estimation: At any given instant t, if the smallest element s=0, then the network size is the total number of the non-zero element in the list. If the list if full, the estimate is given by the estimator,

$$\frac{K}{(1-s)}$$

It is apparent from the working of *MinTopK* that it does not support network partitioning because the nodes calculate the estimate based on the Node Id's generated by the nodes and once they are in the list, there is no mechanism to discard the Node id's of the nodes which have left the network.

4.3 Characteristics of the algorithms

In this section, we briefly mention the characteristics of both the algorithms. It is summarized in table 4.1

	MinTopK	Static Bucket
Type of the algorithm	Probabilistic Counting	FM Sketches
Message Overhead	$O(sizeof(n_{id}))$	$O(N * sizeofeachbucket)$
Memory Required	$O(K * sizeof(n_{id}))$	$O(N * sizeofeachbucket)$
Parameter	List Size, K	Number of Buckets N
Speciality	Simplicity	Can estimate the neighbourhood size in K hops

Table 4.1: Characteristics of the algorithms

4.4 Performance metrics

In this section we describe the metrics that we have used to measure the performance of the two algorithms.

We first define a term which is necessary to understand the following description. *Stable Estimate:* Stable estimate is the estimate that each algorithm provides for a given network size. This estimate remains stable and does not change further, unless there is a change in the network size.

4.4.1 Steady state error (E_{ss})

Steady state error (E_{ss}), is defined as the difference between the network size that is estimated by an algorithm once it has reached a steady state and that of the actual size of the network. It is expressed in percentage.

An algorithm is said to be converged, when a stable network size is estimated by all the nodes in the network. After an algorithm has converged, the estimates of the nodes in the network do not change further and hence it is said to be in steady state. Let N_{est} be the network size, as estimated by an algorithm and N be the actual size of the network. Then the E_{ss} obtained as shown in the equation 4.12.

$$E_{ss} = \left| \frac{N_{est} - N}{N} \right| * 100 \tag{4.12}$$

4.4.2 Standard Deviation (σ)

While conducting experiments, for the same set of parameters and topology, we conduct various runs, for each experiment, to get a more accurate estimate performance of an algorithm. The simulation is run in such a way that configurations are changed for each run, therefore, we obtain different stable estimates for each of these runs. For evaluating the exact performance of the algorithms in terms of error, it is also necessary to consider the standard deviation (σ) between the stable estimates obtained for each run.

Let, r_1,r_2,r_3,...,r_n $\in n$, be the total number of runs and x_1,x_2,...,x_n be the stable estimates observed in each of these runs. Here, n is the number of times the runs for each experiment were conducted. We obtain the mean value (μ) according to the equation 4.14 and the standard deviation between the stable estimates obtained in each run is calculated as shown in the equation 4.13.

$$\sigma = \sqrt{\left[\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2\right]} \tag{4.13}$$

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.14}$$

4.4.3 Total Delay (T_d)

To evaluate the performance of the algorithm, it is necessary to know, how long does each algorithm takes to reach a stable estimate (also known as steady state), under a particular scenario with a particular network size. This can be expressed in terms of the time required for a network to converge to the stable estimate, known as the convergence time for the stable estimate. This convergence time for the stable estimate is expressed in terms of the total delay (T_d)

Total Delay T_d is defined as the time duration, from the instant the algorithm is instantiated till the time instant, when all the nodes in the network estimate the same estimated value (also referred as stable estimate) is known as the total delay, T_d.

Let at time t=0, the algorithm is instantiated in a network of N nodes, then the first time instance at which all the N nodes in the network give same estimate (which is stable unless there is an actual change in the size of the network), is known as the total delay T_d of the algorithm for the network (under a particular topology and scenario).

4.4.4 Rise Time (T_r)

During the course of the run of an algorithm, from the moment, the algorithm starts until the time instant, at which a stable estimate is reached in a whole network, that is, a network is converged, a variety of the estimates are observed in the network, for each algorithm. The number of the different estimates observed vary because of the way each algorithm works (explained in section 4.1 and 4.2)

The time instant when each of these estimates was first recorded in the network, is known as the rise time, T_r of that estimate. Rise time helps us to understand the stability of the network during the course of its run.

During the course of the run of the algorithm, if a network estimate, N_{Est} was observed *first time* at the time instant t, then, this time instant t is known as the rise time,T_r of the estimate N_{Est}.

4.4.5 Settling Time(T_s)

As mentioned in the previous section, that, during the course of run of an algorithm, many values are observed till the network reaches a stable state. Settling time T_s for each estimate is defined as the time difference between the instant when an estimate value was first observed in a network and the instant when the maximum number nodes have that estimate. In any case the maximum nodes estimating a same value cannot more than the network size.

Settling time allows us to study the behaviour of the algorithm in detail, by letting us know, how an algorithm works in the transient time of its run. By evaluating the algorithms on this parameter, we can comment upon the stability of the algorithms during its transient time.

Let an estimate value, Est_1 be observed by the node n_1 at the time instant t_1. Now, let t_2 be the time instant when the maximum number of nodes estimates the value Est_1. The settling time, T_s for the estimate Est_1 is given by the formula 4.15

$$t_s \;=\; t_2 \;-\; t_1 \tag{4.15}$$

Therefore, we evaluate the performance of both the algorithms based on these parameters and this is done in the chapter 5.

Chapter 5

Experiments

In this chapter, we present our evaluation of both the algorithms under different scenarios and parameters. We first give a brief overview of our approach in terms of scenarios. We then illustrate and reason the choice of our scenarios. Next, we present the results of our experiments that we conducted in the form of the performance metrics defined in chapter 4. We then analysed the results and reason their behaviour. Finally, we conclude this chapter by summarizing our results.

5.1 Experimental setup

5.1.1 Simulator

We use MyriaSim[23] as a simulator for our experiments. It is a simulator which is based on

a. OMNeT++[24],an open source modular, component based simulation library and framework.

b. MiXim[20],which provides support as a simulation framework for wireless and mobile networks using the simulation framework and engine of OMNeT++.

MyriaSim has been extended to operate and work like MyriaNed. Its performance mimics that of MyriaNed, including the working of gMAC as explained in chapter 2. Even the hardware components like radio which are used in the MyriaNed have been modelled and incorporated in MyriaSim. Provisions are incorporated to take into considerations even the physical behaviour like clock offsets. It has been made for and used in the experiments conducted in [19].

The data is logged per frame (unit of time) basis. The data is logged in the format of Node Id, an estimate of the network size by that node and time duration elapsed. Data processing, analysis and plotting is done using Matlab [43]. The scenarios III and IV are generated using BonnMotion [44], a widely used tool for generating scenarios that can be used with various simulators.

5.1.2 Simulation Parameters

Clocks

In [19] the author defined a parameter of allowable clock drift, which denotes the maximum allowed differences in the clock frequency offset that may result in drifting of clocks leading to de-synchronization.

Since, we are working on the application layer and not bothered by the synchronization issue. We decide to keep this parameter as a default value of clock drift $C_d = \pm$ 20 parts per million(ppm).

Transmission Power

In the context of the simulator the transmission power of the nodes is in direct proportion with the transmission range, that is, if the transmission power is increased then the transmission range increases too. Increase in transmission power directly affects the diameter of the network. Since, most of the static topologies that we use are in the shape of the either a matrix or grid (where the nodes are placed $20m$ apart), therefore, to investigate the differences in two, we chose to keep this to a default value of $T_xPower = 0.5\ mW$

Round Duration

The frequency of operation is set to make the duration of each round (described in chapter 3.) to approximately 1 *secs.*.

Random number generator

// write about the RNG that is being used here. The function used for generating the random numbers for *MinTopK* is the *dblrand()* and for *Static Bucket* is *intrand()*. Both these functions are the inbuilt functions of the simulator.

5.1.3 Network Topology and scenario

For a proper analysis of the algorithms we decide to choose our scenarios with a combination of different topology, activity and mobility scenarios. The scenarios used are described as follows:

1. **Scenario I:** Matrix Topology - Here all the nodes are arranged in the form of a Matrix,which are separated by a distance of $20m$. In this scenario, all the nodes are active for the entire duration of the simulation. It is explained in detail in section 5.2.1.

2. **Scenario II:** Grid Topology - Here nodes are arranged in the form of the grid, usually in a rectangular shape. Again, all the nodes are separated by a distance of $20m$. The activity of the nodes remains same as in the previous scenarios. It is explained in detail in section 5.2.2.

3. **Scenario III:** Mesh Topology - Here all the nodes are scattered in a predefined area in a random way, forming a mesh like structure and network of all the nodes. The nodes are active for the complete duration of the simulation. It is explained in detail in section 5.2.3.

4. **Scenario IV:** Mobility Scenario - Here all the nodes are moving in a predefined way. The movement of the nodes is based on a specific mobility pattern that we chose the nodes to have. It is explained in detail in section 5.2.4.

5. **Scenario V:** Network split/merge - Here in this scenario a huge network of 256 nodes are split/merged into two/one network by deactivating/activating a small portion of the network. The modes in this scenario are stationary.

For the experimental phase, we have categorized the network into various sizes. The network sizes are broadly classified in 3 categories, they are:

1. **Small Network:** A network of 64 nodes.

2. **Big Network:** A network of 256 nodes.

3. **Very Large Network:** A network of 1024 nodes.

5.1.4 Application variants

We investigate two applications namely *MinTopK* and *Static Bucket.* There are different parameters also, which differ for each algorithm. They are as follows

- **MinTopK:** The value of 'K', which signifies the size of the list of identifiers, that each node store (as explained in chapter 4).
- **StaticBucket:** The number of Buckets that each node stores and sends in a packet (as explained in chapter 4).

5.1.5 Performance metrics and terms used

The different performance metrics that we use for the evaluation purpose are as follows (explained in detail in chapter 4):

- Rise Time (T_r)
- Settling Time(T_s)
- Standard Deviation(σ) of the final estimate of each run.
- Steady state error(E_{ss}) in percent
- Total Delay (T_d)

Throughout this chapter, few terms have been used, they are as follows:

Stabilized Network: After the start of the algorithm when all the nodes in the network give a same estimate and that estimate does not change further (unless there is a change in a network), then the network is called as stabilized network. This is sometimes referred to as the converged network

Transient time/phase: The time duration from the start of the algorithm in a network till the time all the nodes in the network reach a stable estimate, that is when all the nodes in a network give same estimate. This time duration is known as transient time or transient phase.

5.2 Experiments, results comparison and analysis

In this section we first describe the scenarios, then we present the results for each scenario with different parameters and finally we conclude it with the analysis of the obtained results in terms of the performance metrics.

The results presented are the averaged results of many runs[1], as done in [19]. In each

[1] 10 runs for scenario 1 and 5 runs all other scenarios

run we get a different estimate value, therefore we present our results by averaging the results from different runs. This helps us in understanding the average case performance of the algorithms.

5.2.1 Scenario *I*: Matrix Topology

The main objective of this experiment is aimed at evaluating the performance of both the algorithms, *MinTopK* and *Static Bucket* in a matrix Topology. The reason behind conducting this experiment is to evaluate the performance of the two algorithms under a simple topology based on the performance metrics defined in the chapter 4.

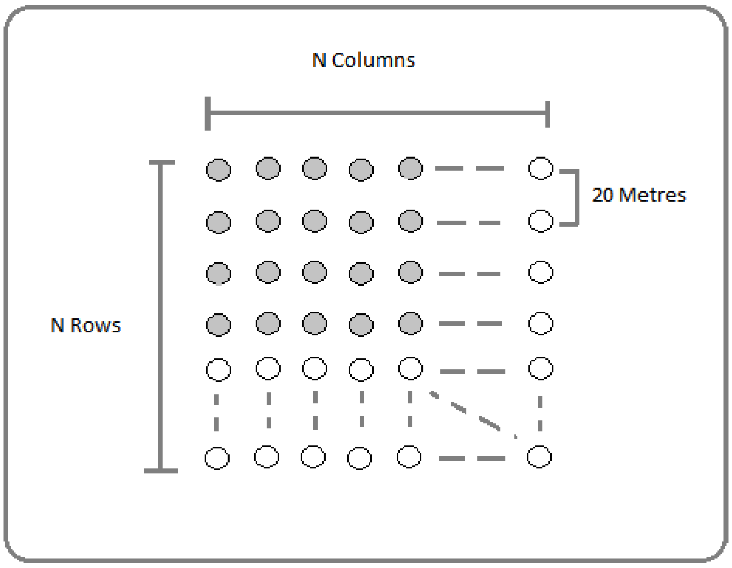


Figure 5.1: A Typical Matrix Topology

Testbed

A typical Matrix topology is as shown in the figure 5.1. Nodes are arranged in equal number of rows and columns and each node is at a distance of 20m from another node. In this scenario each node can see just one neighbour in each direction. Here, for conducting the experiments in this topology, we are using the network which is arranged in the form of a square matrix (as used in [19]) of size 8, 16 and 32 with a network size of 64, 256 and 1024 nodes respectively.

Parameters of application and Initial Results

The parameters used and the results obtained in terms of two important metrics, Estimation Error (E_{ss}) and Total Delay(T_d), in a network size of 64, 256 and 1024 nodes for different parameters of each algorithm is presented. The result for *MinTopK* is tabulated in table 5.1 and *Static Bucket* is tabulated in table 5.2.

Table 5.1: Performance of MinTopK in various network sizes in Matrix Topology

Parameters	**Matrix** $8 * 8$				
List Size(K)	$K = 4$	$K = 8$	$K = 16$	$K = 24$	$K = 32$
Estimate Error (E_{ss}) in Percent	42.2	9.4	9.4	9.4	1.5
Total Delay Time (T_d) in Secs.	28	63	108	149	176
Standard Deviation (σ)	52.0580	18.3328	11.08	13.7919	7.6217
Parameters	**Matrix** $16 * 16$				
List Size(K)	$K = 8$	$K = 16$	$K = 24$	$K = 32$	$K = 64$
Estimate Error (E_{ss}) in Percent	7.42	5.47	7.42	7.03	9.38
Total Delay Time (T_d) in Secs.	86	317	238	293	574
Standard Deviation(σ)	103.6774	45.6438	39.3065	30.7376	17.37
Parameters	**Matrix** $32 * 32$				
List Size(K)	$K = 8$	$K = 16$	$K = 32$		
Estimate Error (E_{ss}) in Percent	5.76	17.38	15.03	$N.A$	
Total Delay Time (T_d) in Secs.	159	243	550		
Standard Deviation (σ)	274.4264	303.7876	236.2947		

MinTopK The results obtained for the MinTopK algorithm are tabulated in the table 5.1.

- Increase in the value of 'K', results in an increase of the total Delay (T_d). (Here,'K' denotes the size of the list used in *MinTopK* and T_d is Total Delay time that the whole network takes to converge to the same estimates). This is because the bigger the size of the list(K), the more time it will take for all the nodes in the network to converge to the top K values because a big list takes more numbers to be stored in each list and this takes more time.

- We have to make a trade-off between the desired accuracy which increases as the size of the list increases (in terms of standard deviation(σ)) and the Total Delay (T_d) by choosing the proper value of 'K'. For example in a topology of Matrix 8∗8, when $K = 4$ the network converges really fast signified by T_d but the error in estimating the network size is also high signified by E_{ss}. Where as, when K=32 the error rate is low but the T_d is very high.

- Also, the value of K and estimate error E_{ss} don't have a direct relation with different network size. This means, that the variable K and the network size are not relational.

- One more interesting point is that the results achieved in our case are different from that of the original paper [18]. This is because of the following factors:
  - Random Number Generator (RNG): The RNG that we have used is the centralized one, whereas the authors have not specified the one that they have used. This factor has an impact on the outcome as *MinTopK* requires the generated number to be uniformly distributed, failing which the error will increase.
  - WSN platform: Some metrics like the total delay, rise time etc. completely depends on the property of the underlying WSN platform used. We have use *MyriaNed*, the authors have not specified the one that they used.
- Another important thing to note is that theoretically even though with increasing size of the 'K' the E_{ss} is expected to drop, in our case, although it is decreasing usually, but it increases in certain cases (still within a certain limit) like when K is changed from 32 to 64 for the network size of 256 nodes. This is compensated by the drop in standard deviation between the runs. That means when the list size (K) is increasing the standard deviation between the stable estimates of the run is reduced compared to the small size of the list.
- Even though in some cases by varying the size of K we get same(or higher) error rate, the standard deviation(σ) reduces, thus reflecting the benefit of using higher values of K.
- For the network size of 1024 nodes, we did experiments with the values of K= 8,16 and 32 only. It is mainly done due to the fact that higher values of K would take even more time to settle, which is not suitable for practical applications and due to heavy computational requirements.

Static Bucket The authors in [19] chose to keep 24 buckets, because the authors were experimenting with network size of 4096 nodes. For the network of bigger size, the number of buckets has a direct impact on the performance of the algorithm, which is evident from the working of the algorithm [19](as explained in chapter 4). Since, we are investigating the performance in network sizes of 64 and 256 nodes which is way smaller, the reduction in number of buckets does not affect the performance in our case. This is shown in the table 5.2. We can see that, as we vary the number of the buckets denoted by N, it does not have much impact on the metrics, E_{ss}, T_d and standard deviation(σ) for the network sizes of 64 and 256 nodes. Therefore, for the course of our experiments, we decided to use the number of buckets as 12, denoted by the parameter 'N'.

Table 5.2: Performance of Static Bucket in various network sizes in Matrix Topology

Parameters	**Matrix** $8 * 8$		
Number OF Buckets (N)	$N = 12$	$N = 18$	$N = 24$
Estimate Error (E_{ss}) in Percent	15.63	15.63	15.63
Total Delay Time (T_d) in sec.	9	10	10
Standard Deviation (σ)	30.8279	30.8279	30.8279
Parameters	**Matrix** $16 * 16$		
Number OF Buckets (N)	$N = 12$	$N = 18$	$N = 24$
Estimate Error (E_{ss}) in Percent	25.78	25.78	25.78
Total Delay Time(T_d) in sec.	18	18	19
Standard Deviation (σ)	121.2228	121.2228	121.2228
Parameters	**Matrix** $32 * 32$		
Number OF Buckets (N)	$N = 12$	*N.A*	
Estimate Error (E_{ss}) in Percent	29.03		
Total Delay Time(T_d) in sec.s	33		
Standard Deviation (σ)	723.7241		

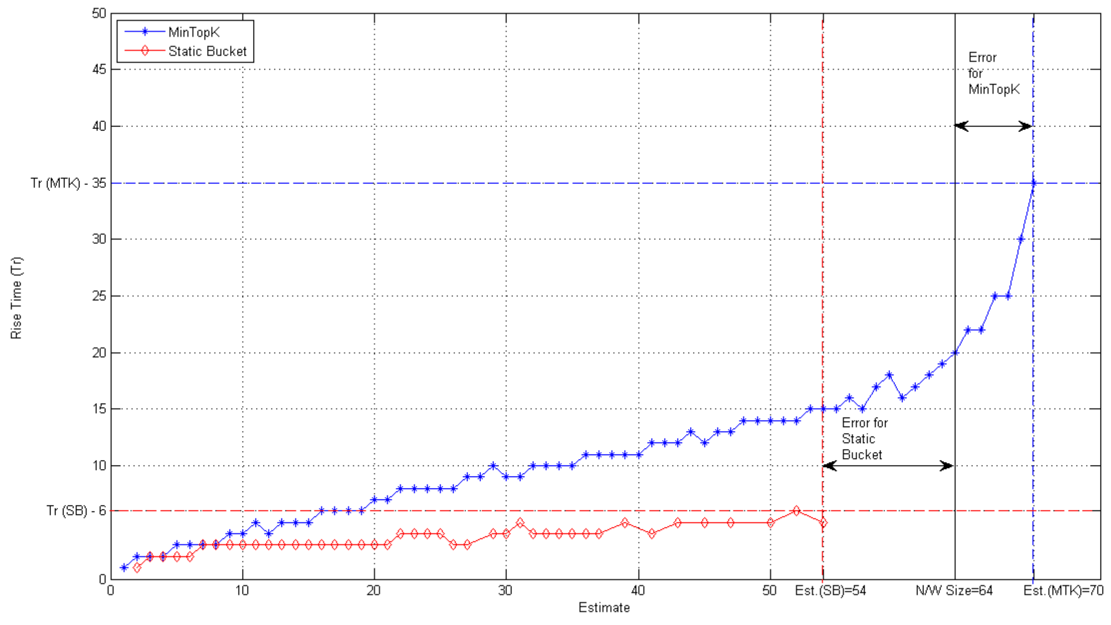
Results Analysis

In this section we will compare and contrast both the algorithms in terms of the performance metrics described in chapter 4. The results are tabulated in 5.3.

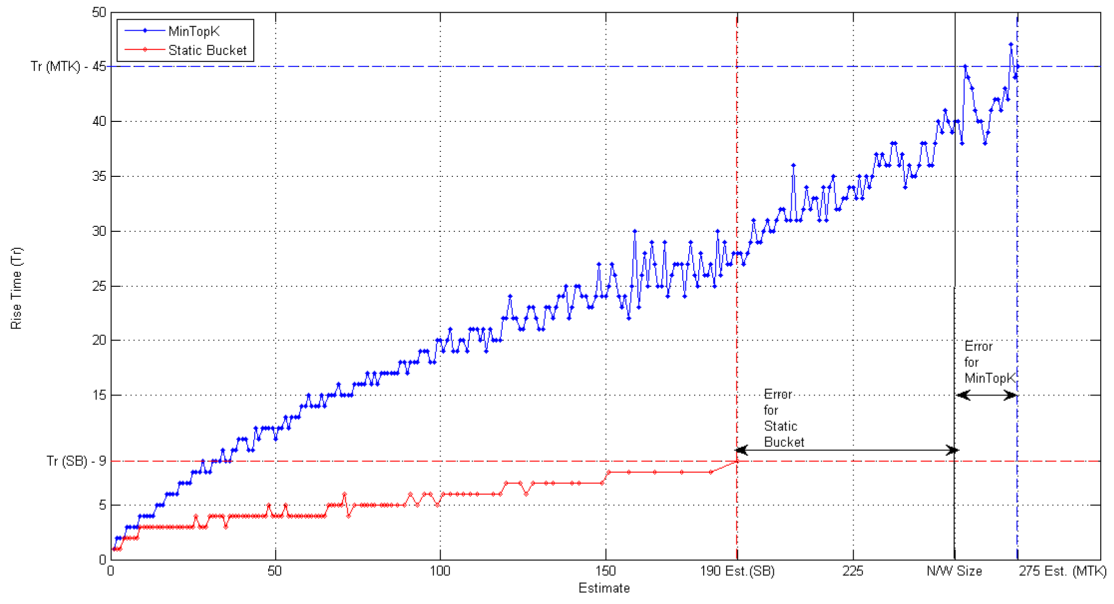
Table 5.3: Scenario I: Results

Parameters	*MinTopK K =8*		*Static Bucket*	
Network Size	64 nodes	256 nodes	64 nodes	256 nodes
Estimate Error (E_{ss}) in percent	9.375	7.42	15.63	25.78
Rise Time (T_r) in sec.	35	45	6	9
Settling Time (T_s) in sec.	28	41	4	9
Standard Deviation	18.3328	103.6774	30.8279	121.2228
Total Delay (T_d) in sec.	63	86	9	18

The Figure 5.2 shows the plot of Rise Time (T_r) (Y-axis) Vs. Estimate (X-axis) for both the algorithms, henceforth denoted as Rise-Time plot. Every point on this graph represents a unique estimate that is observed for each algorithm during the course of its run and the corresponding time at which the estimate is observed. For example: suppose there is a point in the plot having co-ordinates as (X, Y) then it signifies that there is a node in a network which estimates X at time instant Y. (Rise Time(T_r) is described in detail in chapter 4). In this plot we are mainly interested in the rise time (T_r) of the final estimate that the network observes after it is stabilized. The graph also represents the estimation error of each algorithm and the difference from the actual network size on the x-axis. On the Y-axis it represents the rise time(T_r) of the final stable estimate



(a) Small Network



(b) Big Network

Figure 5.2: Rise-Time(T_r) Vs Estimate plot for Matrix Topology

for both algorithms. Again, the rise time is the time instant at which the corresponding estimate was first observed in the network. An important point to note here is that the final point observed in the figure represents the estimate which was ultimately observed by all the nodes in the network after the network stabilizes. Henceforth referred to as the stable estimate. The figure 5.2a shows the plot for the network of 64 nodes and figure 5.2b shows the one for the network of a 256 nodes. There are a few points to observe.

1. The T_r for *MinTopK* is 35 sec. for 64 nodes network and 45 sec. for 256 node network which is higher than that for *Static Bucket* which has a T_r of 6 and 9 sec. for respective network sizes.

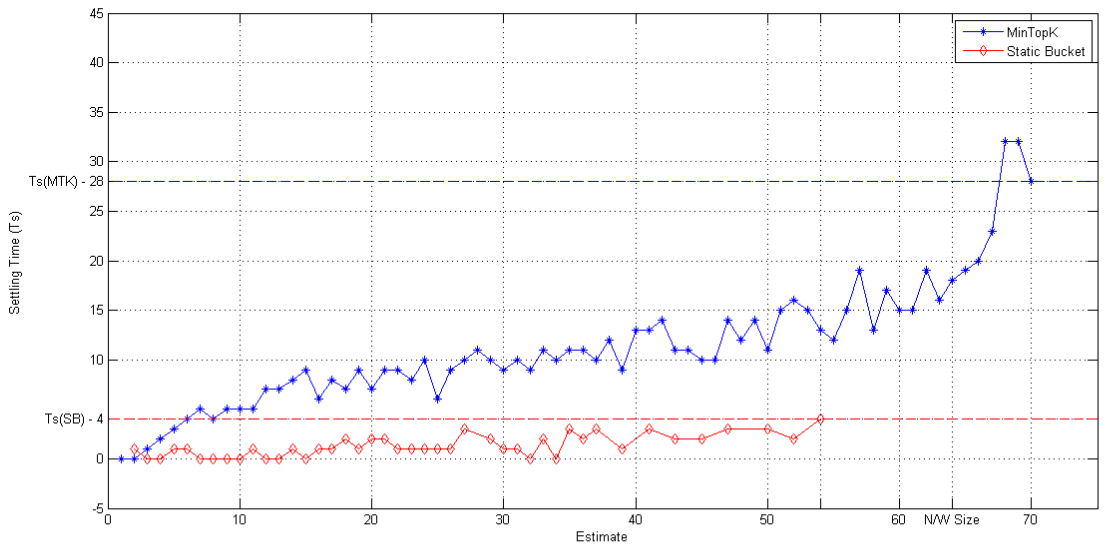
2. The estimate error is less in case of *MinTopK* than *StaticBucket*.

3. Even though the error (E_{ss}) is higher in *Static Bucket*, the number of different estimates observed from the moment the algorithm starts to the moment the algorithm reaches a stable estimate (This time is referred as transient time), are more in case of *MinTopK* algorithm than *Static Bucket*. This signifies that the degree or the number of different estimates observed is higher in case of *MinTopK* than *Static Bucket* algorithm. It implies that the *Static Bucket* has a smooth transition to stable value than *MinTopK*.

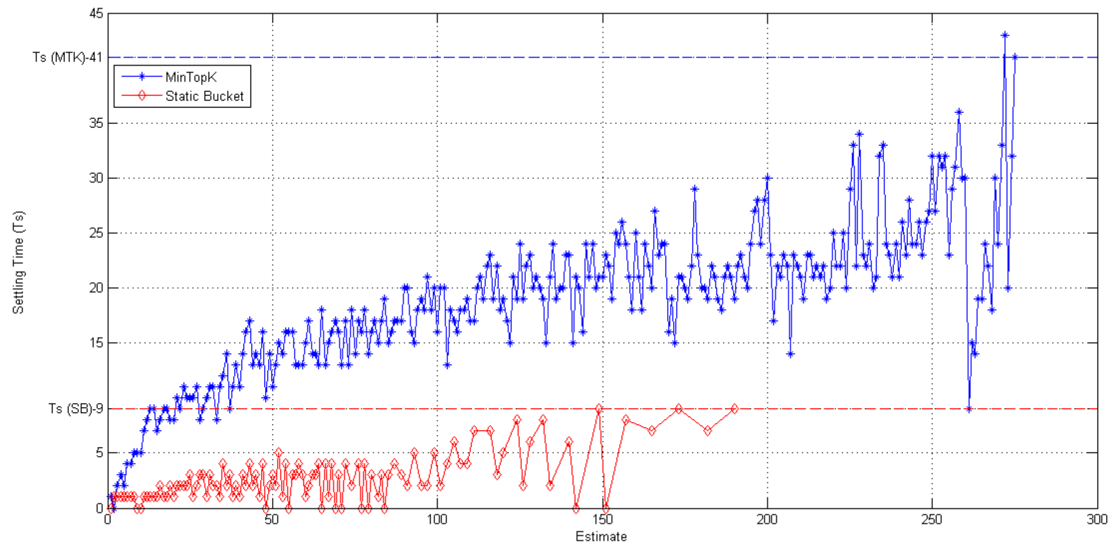
Figure 5.3 shows the plot of Settling Time(T_s)(Y-axis) Vs Estimate (X-axis) for both the algorithms. Every point on this plot represents the time required for the corresponding estimate to settle. In short, it represents the difference between the time instants from when the corresponding estimate value was first observed and when it was last seen in the network. It is noteworthy that the estimate represented by the final point was ultimately observed by all the nodes in the network whereas this is not the case with other estimate values in the plot. Settling time is described in detail in chapter 4. On X-axis it represents the estimate values and on Y-axis it represents the settling time. The T_s for the last estimate also known as stable estimate is marked on the y-axis for both the algorithms. This is of prime importance since, the Figure 5.3a shows the plot for the network of 64 nodes and the figure 5.3b shows the one for the network of 256 nodes. These are the observations from the plot.

1. The settling time (T_s) for *MinTopK* is 28 sec. for 64 nodes network and 41 sec. for 256 nodes network which is higher than that for *Static Bucket* which has a T_s of 4 and 9 sec. for respective network sizes.

2. During the transient time of the algorithms, the degree of variability is very high in case of *MinTopK* than *Static Bucket*. Because, as the number of nodes increase in a network, then the total number of nodes having different top K values also increase. As a result, many nodes have different value in the list as the smallest element, therefore resulting in varied estimate. The reason we see huge spikes towards the end is because, as time progresses nodes in the network start getting



(a) Small Network



(b) Big Network

Figure 5.3: Settling-Time(T_s) Vs Estimate plot for Matrix Topology

closer to the top K values, therefore the number of nodes estimating same value start increasing. As a result, a particular estimate is observed in a network for longer time, thus resulting in huge values of T_s towards the end and hence the spikes towards the end of the graph.

5.2.2 Scenario II: Grid Topology

The main objective of this experiment is aimed at evaluating the performance of both algorithms in a rectangular grid like topology. The motivation behind this is to evaluate the performance in a network with a large diameter. The diameter of a network in a broad concept is referred as the total width of the network in terms of the hops. For example if the network is arranged in rectangular form with nodes that are equidistant from each other, then the number of nodes on the long side is known as diameter of the network.

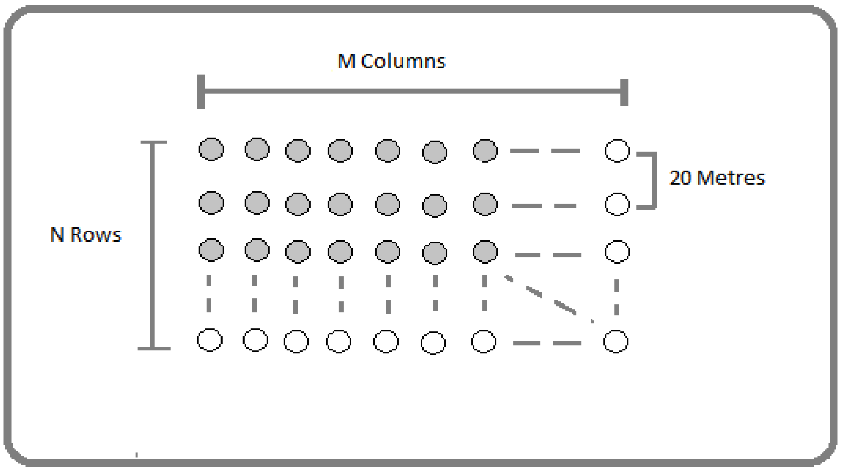


Figure 5.4: A Typical Grid Topology

Testbed

A typical Grid topology is shown in figure 5.4. Nodes are arranged in a rectangular form of N*M with N as the number of rows and M as the number of columns [19]. Here also the nodes are separated by a distance of $20m$. In this experiment we are using two network sizes. One is a small network of 64 nodes arranged in form 2∗32 having a diameter of 32 and another with 256 nodes arranged in form 4∗64 with a diameter of 64.

Table 5.4: Performance of MinTopK in various network sizes in Grid network

Parameters	**Grid** $2 * 32$		
List Size (K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error (E_{ss}) in Percent	9.4	9.4	1.56
Total Delay Time (T_d) in sec.	173	382	797
Standard Deviation (σ)	18.3328	11.08	7.6217
Parameters	**Grid** $4 * 64$		
List Size (K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error (E_{ss}) in Percent	4.3	3.51	8.2
Total Delay Time (T_d) in sec.	255	498	895
Standard Deviation(σ)	103.6774	45.6438	30.7376

Parameter of application and results

The parameter variable for *MinTopK* is K as 8, 16 and 32 and for *Static Bucket* is N=12. The result of the experiment with different list and network sizes for the *MinTopK* is tabulated in the table 5.4 in terms of the performance metrics of error, delay time and standard deviation.

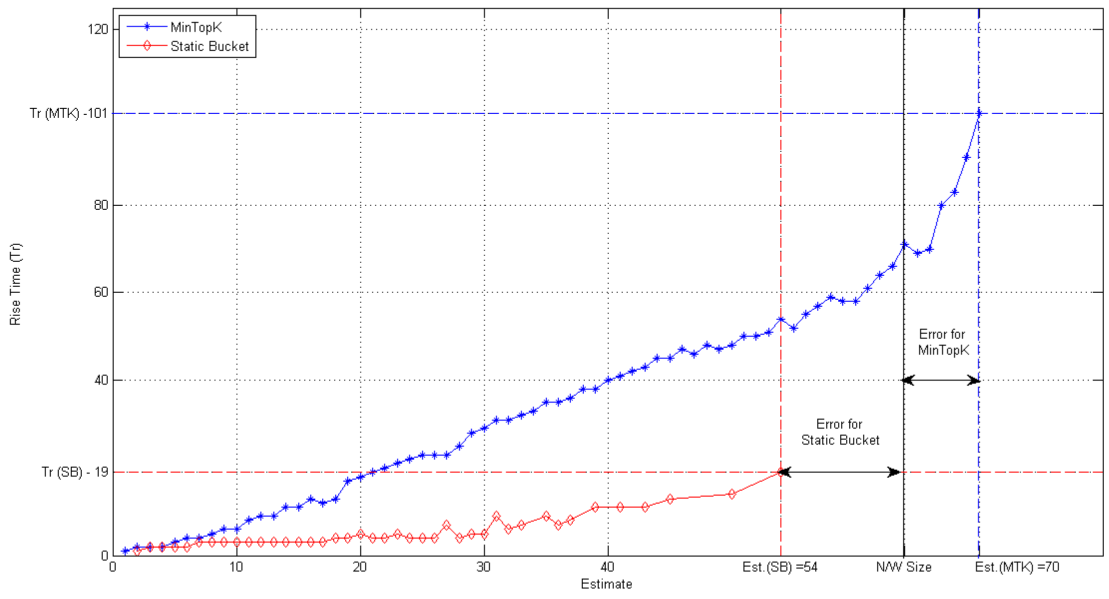
Table 5.5: Scenario II: Results

Parameters	*MinTopK* K =8		*Static Bucket*	
Network Size	2∗32	4∗64	2∗32	4∗64
Estimate Error (E_{ss}) in percent	9.375	4.3	15.63	25.78
Rise Time (T_r) in sec.	101	139	19	24
Settling Time (T_s) in sec.	72	116	18	31
Standard Deviation	18.3328	103.6774	30.8279	121.2228
Total Delay (T_d) in sec.	173	255	37	55

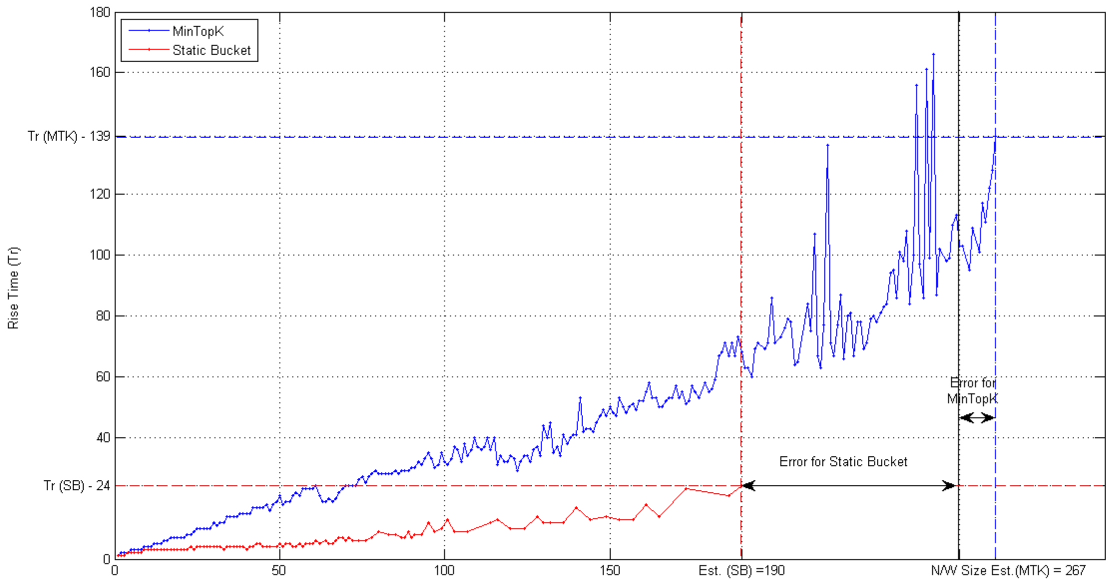
Result Analysis

The results are tabulated in the table 5.5. The value of the list is kept as K=8. We compare the performance in terms of performance metrics described in chapter 4.

The figure 5.5 shows the rise time plot for both network sizes. Figure 5.5a for small network and 5.5b for the big network. The plot is described in detail in the previous section. The observations are as follows.



(a) Small Network



(b) Big Network

Figure 5.5: Rise-Time(T_r) Vs Estimate plot for Grid Topology

1. The rise time (T_r)for a 64 node network is 101 seconds for *MinTopK* and 19 seconds *Static Bucket* and for a 256 node network it is 139 and 24 seconds for respective algorithms.

2. As expected, the rise time (T_r) is higher than that of matrix topology, this is due to the larger diameter of the network.

3. We observe for *MinTopK* in figure 5.5b that the rise time of the final estimate (stable estimate) is lesser than the rise time for some other incorrect estimate (the spikes). This is because a certain node in a network can converge to the top K values and get the stable estimate faster than the other nodes. Thus resulting in lesser rise time (compared to other estimates).

4. For *MinTopK* we observe spikes in the rise time for some estimates values. These spikes indicate that there are certain nodes in the network, which estimate the lower values (than the stable estimate value) even after the stable estimate has been observed by some node in the network. Therefore, the nodes which converge later have a smallest id in its list smaller than the actual s of the top K values of the whole network. Thus resulting in a lesser estimate with high rise time (T_r).

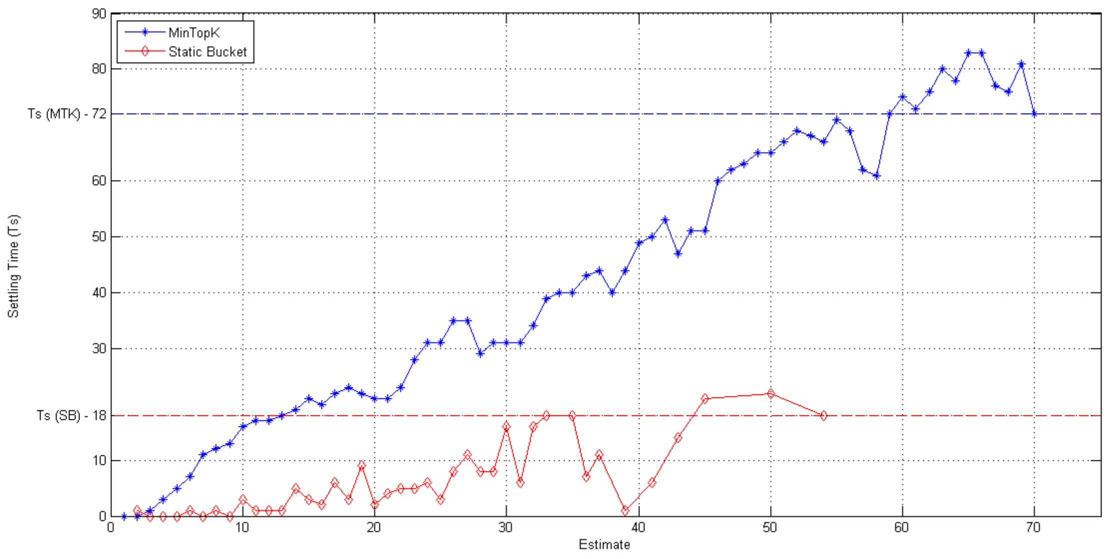
The figure 5.6 shows the settling time plot for both network sizes in this scenario. Figure 5.6a for small network and 5.6b for the big network. The observations are as follows.

1. The settling time of *MinTopK* for a network of 64 nodes is 72 sec. and a network of 256 nodes is 116 sec. For *Static Bucket* it is 18 and 31 respectively.

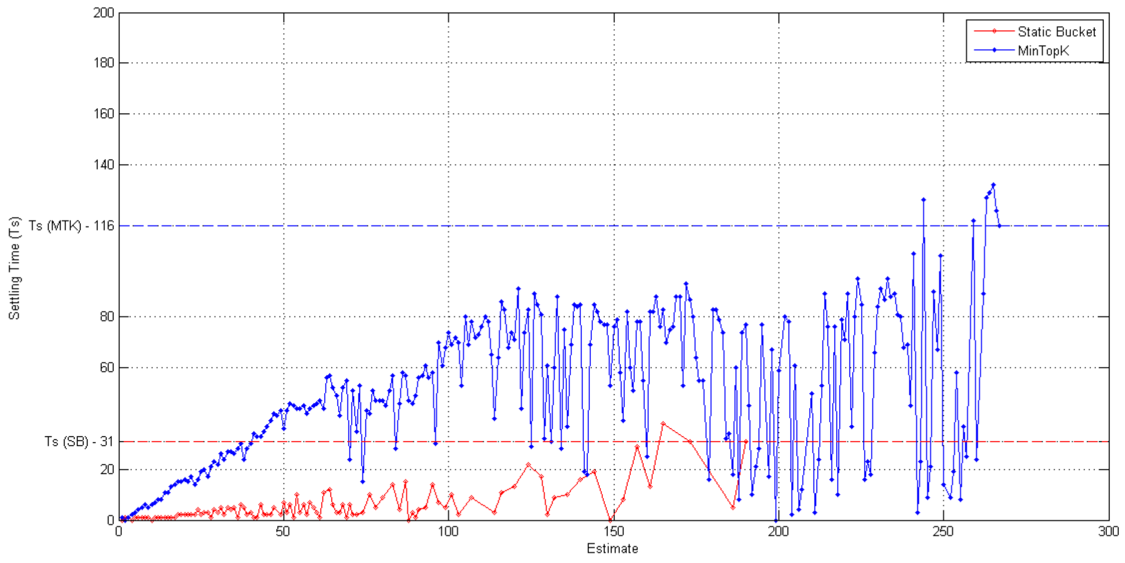
2. For *MinTopk* we observe a lot of variation in Settling plot shown in figure 5.6b compared to the one in the matrix topology for the same network size, shown in the figure 5.3b. The reason behind this is that due to the increased diameter of the network, at any given instant there are many nodes and they all have different numbers and hence different top K values. As a result there are different estimates and therefore varied settling times (T_s).

5.2.3 Scenario III: Mesh Network

The main objective of this experiment is aimed at evaluating the performance of both the algorithms in terms of defined metrics in a mesh like structure, where all the nodes connect through other nodes and form a complete network and the nodes are placed at random in a predefined area. Also, the density of the nodes is increased compared to the other scenarios. Density is defined as the total number of nodes in a square metre of area. This is done because in the actual set-up of the nodes in the physical world all the nodes are more likely to be used in this set-up.

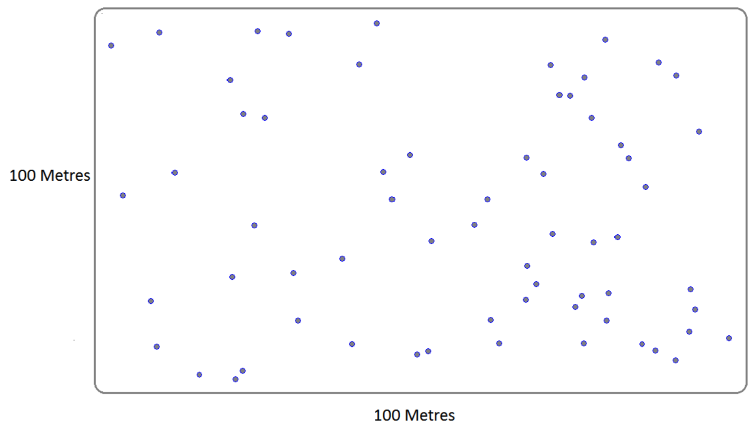


(a) Small Network

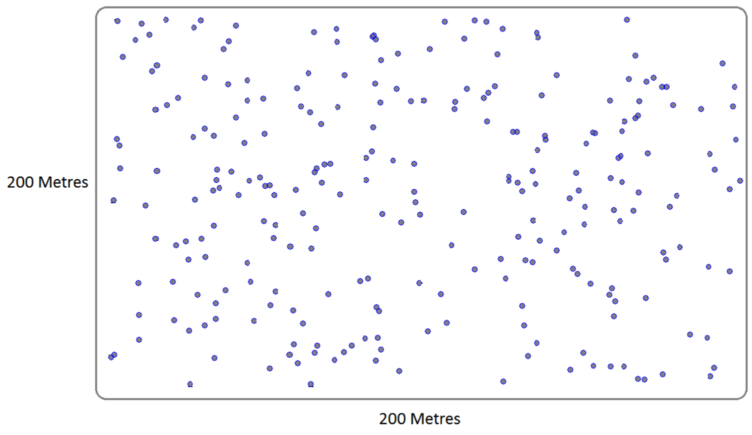


(b) Big Network

Figure 5.6: Settling Time (T_s) Vs Estimate plot for Grid Topology



(a) Small Network



(b) Big Network

Figure 5.7: Layout of nodes in Mesh Topology

Testbed

A typical mesh topology is shown in the figure 5.7. The scenario is generated using Bonnmotion [44], a tool used for scenario generation. For the purpose of our experiments we are using the network sizes of 64 nodes for which the layout is shown in the figure 5.7a and 256 nodes for which the generated layout is shown in the figure 5.7b.In this layout the nodes connect through each other to form a network. This network is also a dense network compared to the previous scenarios for the same number of nodes. Compared to the Matrix topology, in this scenario for a 64 node network, the node density is increased from 1 node/($306m^2$) to 1 node/($156.25m^2$) and for a 256 node network it it changes from 1 node/($351m^2$) to 1 node/($156.25m^2$).

Table 5.6: Performance of MinTopK in various networks sizes in Mesh network

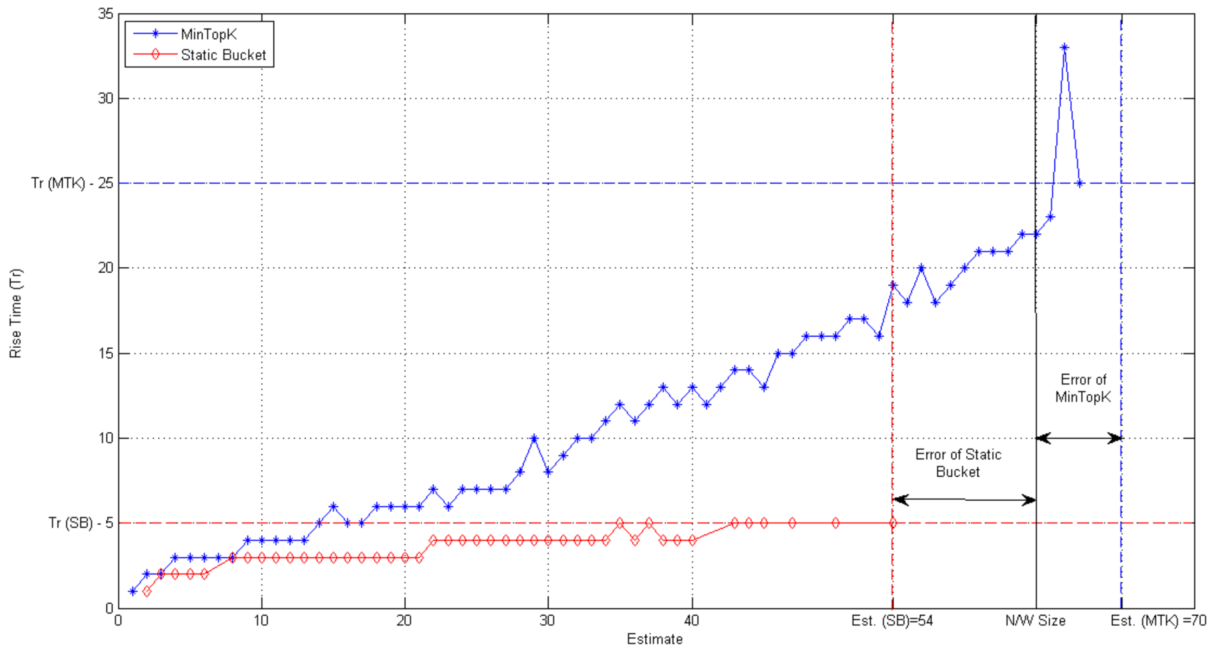
Parameters	**Mesh** 64 **nodes**		
List Size (K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error(E_{ss}) in Percent	4.68	3.12	3.12
Total Delay Time (T_d) in sec.	43	114	155
Standard Deviation (σ)	20.02	8.8634	6.8293
Parameters	**Mesh** 256 **nodes**		
List Size(K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error (E_{ss}) in Percent	4.3	3.51	8.2
Total Delay Time (T_d) in sec.	106	193	298
Standard Deviation(σ)	91.9034	52.0438	26.5586

Parameter of application and results

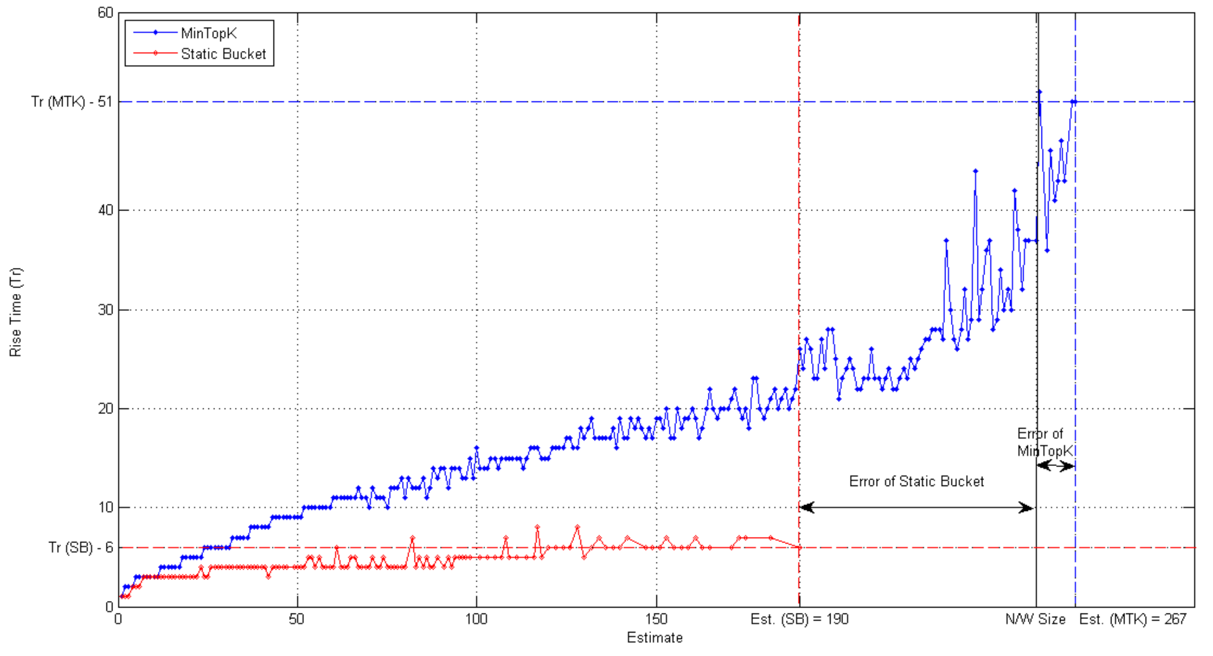
The parameter variable for *MinTopK* is the list size, K as 8,16 and 32. For *Static Bucket* the number of buckets denoted by N is kept at 12. The results of the experiment for different parameters for *MinTopK* in this scenario are tabulated in table 5.6 in terms of the performance metrics of error, delay time and standard deviation.

Table 5.7: Scenario III: Results

Parameters	*MinTopK* K =8		*Static Bucket*	
Network Size	64 Nodes	256 Nodes	64 Nodes	64 Nodes
Estimate Error (E_{ss}) in percent	4.6875	4.3	15.63	25.78
Rise Time (T_r) in sec.	25	51	5	6
Settling Time (T_s) in sec.	18	55	5	7
Standard Deviation	20.02	91.9034	30.8279	121.2228
Total Delay (T_d) in sec.	43	106	10	13



(a) Small Network



(b) Big Network

Figure 5.8: Rise-Time(T_r) Vs Estimate plot for Mesh Topology

Result Analysis

The results are tabulated in the table 5.7. The value of the list is kept at K=8. We compare the performance in terms of the performance metrics described in chapter 4.

The figure 5.8 shows the rise time plot for both the algorithm and network sizes. Figure 5.8a for the small network and 5.8b for the big network. The technique for plotting remains the same as in case of scenario I. The observations are as follows.

1. The rise time (T_r) for a 64 node network for *MinTopK* is 25 seconds and for *Static Bucket* is 5 seconds and for a 256 node network it is 51 and 6 seconds for the respective algorithms.

2. The important point to note here is that the rise time varies for both the algorithms and network sizes. Compared to *scenario I*, for a small network, the rise time is reduced for both the algorithms where as for a big network the rise time for *MinTopK* is increased while for *Static Bucket* it decreases. This reduction in rise time is due to the increase in the density of nodes. This simply means that denser the network, the faster the convergence to stable estimate.

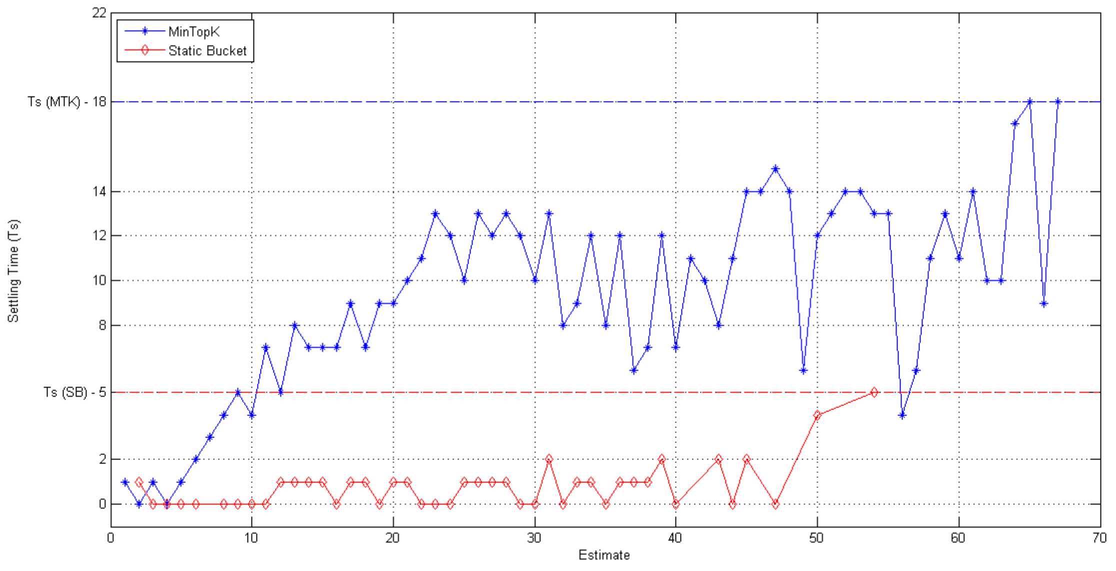
3. All the other properties of the graph are explained in the above section.

The figure 5.9 shows the settling time plot for both the network sizes in this scenario. 5.9a for a small network and 5.9b for the big network. The observations are as follows.

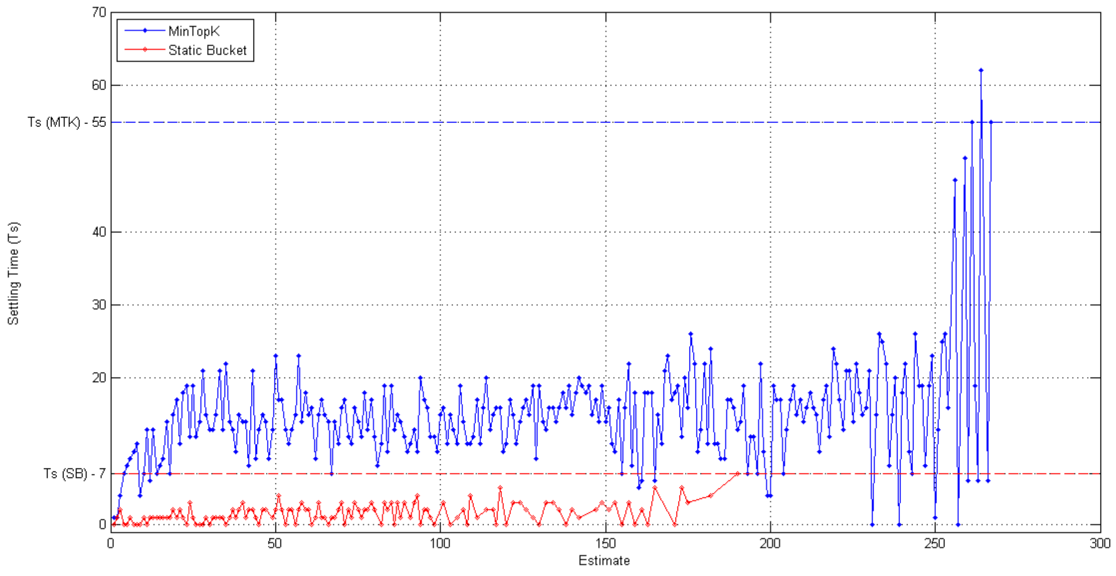
1. The settling time of *MinTopK* for a network of 64 nodes is 18 seconds and for a network of 256 nodes is 55 seconds. For *Static Bucket* it is 5 and 7 seconds for the respective network sizes.

2. When we compare the settling time plot for mesh topology shown in figure 5.9 with that of matrix topology shown in the figure 5.3. We can see the difference in the pattern of the plots. In case of Matrix Topology, the plot is more smooth where as in mesh topology it is fluctuating a lot. This is because, the variety of estimates are observed faster in the mesh topology due to the high density of the nodes in this topology.

3. Another thing to observe is that for a big network the plot for *MinTopK* spikes all of a sudden. This means that the settling time of the high estimates rises abruptly. This is because of the mesh structure some nodes had a degree of high order and they were able to transfer the id's faster thus many nodes were estimating almost in the same range. But overall it took a long time for all the nodes in the network to have same top K values, thus resulting in the increased settling time (T_s) for network.



(a) Small Network



(b) Big Network

Figure 5.9: Settling Time(T_s) Vs Estimate plot for Mesh Topology

5.2.4 Scenario IV: Mobility

The main objective of this experiment is aimed at evaluating the performance of both the algorithms in mobile nodes, using the performance metrics used above. Here nodes travel using a mobility technique, it also takes into effect the scenario where the number of neighbours of nodes keeps on changing continuously. Also, this scenario actually mimics the use of the MyriaNed in social ad-hoc scenarios where, nodes gather and perform an activity in a predefined area.

Testbed

A typical Mobility scenario is motivated from the mesh scenario mentioned in the previous section. For a network of 64 and 256 nodes we have kept the area as 100*100 and 200*200 metres respectively. The mobility imparted to the nodes is modelled from the *Random waypoint* mobility model using the Bonnmotion [44] tool. The nodes keep moving within the predefined area with the speeds in the interval (0.5, 1.5) m/s^2. The speed is chosen because the average walking speed for humans is around 1.25 m/s^2 [45]. Therefore we decide to choose the speed in this range. Furthermore it is more likely that the these speeds are used in social ad-hoc scenario.

Table 5.8: Performance of MinTopK in various network sizes in Mobility Scenario

Parameters	**Mobility 64 nodes**		
List Size (K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error(E_{ss}) in Percent	4.68	3.12	3.12
Total Delay Time (T_d) in sec.	75	87	155
Standard Deviation (σ)	20.02	8.8634	6.8293
Parameters	**Mobility 256 nodes**		
List Size(K)	$K = 8$	$K = 16$	$K = 32$
Estimate Error (E_{ss}) in Percent	4.3	3.51	8.2
Total Delay Time (T_d) in sec.	77	136	269
Standard Deviation(σ)	91.9034	52.0438	26.5586

Parameter of application and results

The parameter varied for *MinTopK* is list size K, which is kept as 8, 16 and 32. For *Static Bucket* the bucket the number of buckets is kept at N=12. The result of the experiment for a chosen list size of *MinTopK* is tabulated in table 5.8 in terms of error, delay time and standard deviation.

Table 5.9: Scenario IV: Results

Parameters	*MinTopK* K =8		*Static Bucket*	
Network Size	64 Nodes	256 Nodes	64 Nodes	64 Nodes
Estimate Error (E_{ss}) in percent	4.6875	4.3	15.63	25.78
Rise Time (T_r) in sec.	27	35	6	8
Settling Time (T_s) in sec.	48	42	2	6
Standard Deviation	20.02	91.9034	30.8279	121.2228
Total Delay (T_d) in sec.	75	77	8	14

Result Analysis

The results are tabulated in the table 5.9. The value of the list is kept at K=8. We compare the performance in terms of the various performance metrics described previously.

The figure 5.10 shows the rise time plot for both the algorithms. Figure 5.10a shows the one for the small network and 5.10b for the big network. The plot is explained in the scenario I. The observations are as follows.

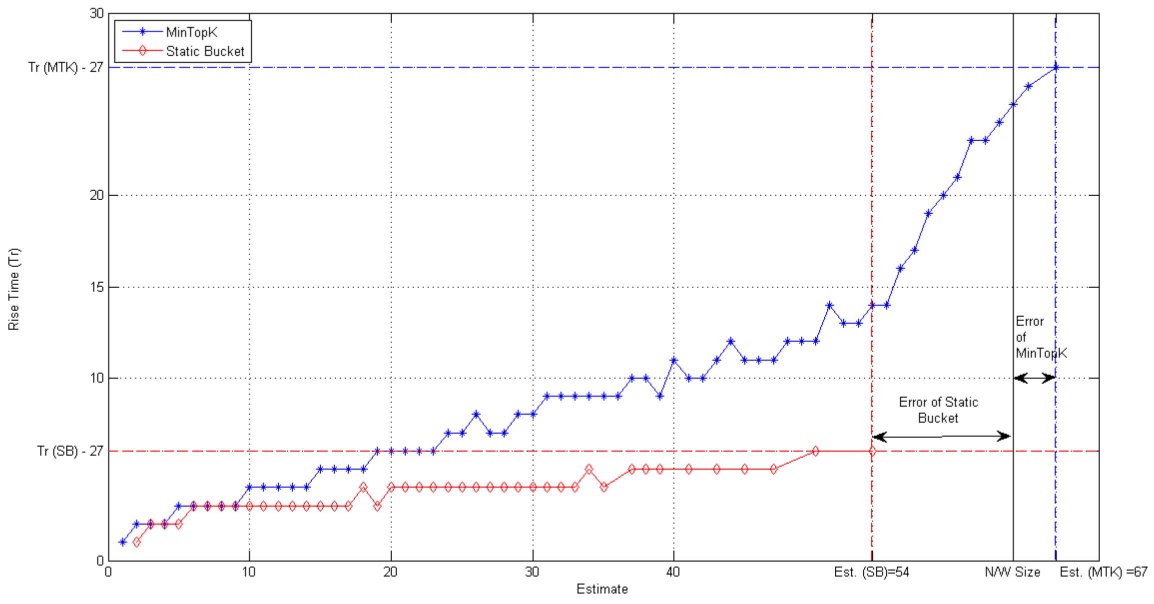
1. The rise time (T_r) for a 64 node network for *MinTopK* is 25 seconds and for *Static Bucket* is 6 seconds and for 256 node network it is 35 and 8 seconds for the respective algorithms.

2. A point to note here is that for *MinTopK*, for small network, the rise time remains almost the same, where as for a big network the rise time is reduced drastically. This is due to the mobility of the nodes and hence constantly changing neighbours of the nodes, few nodes in the network will have the top K values faster than other nodes, thus resulting in lower rise time (T_r). Whereas for *Static Bucket*, the rise time remains almost the same. This results in the conclusion that mobility impacts the rise time of the algorithms, more so for big network size in case of *MinTopK*.

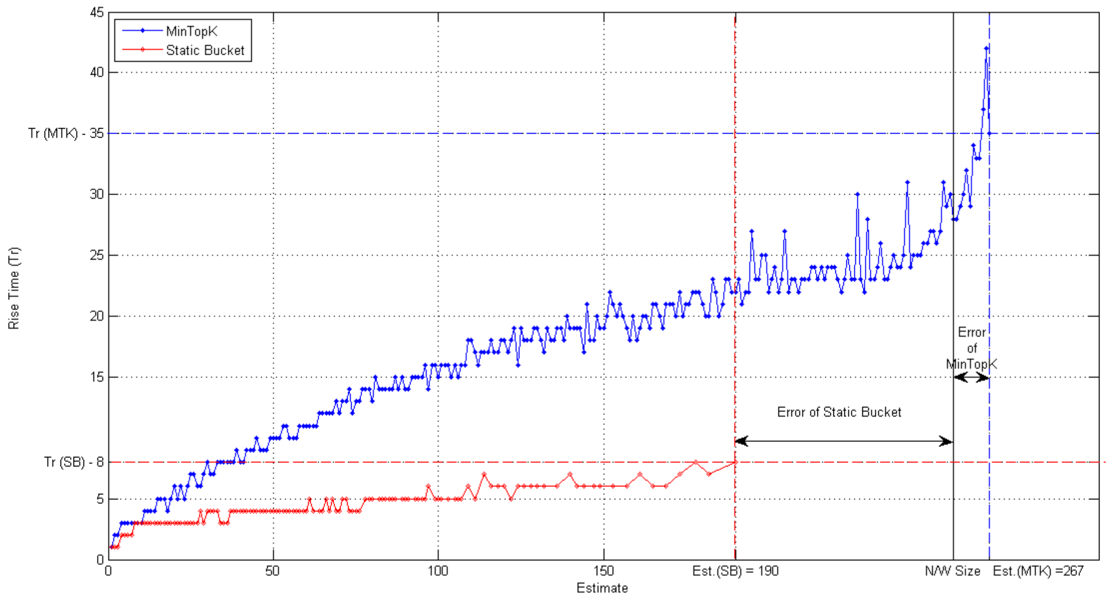
The figure 5.11 shows the settling time plot for both the network sizes in this scenario. Figure 5.11a for the small network and 5.13b for the big network. The observations are as follows.

1. The settling time of *MinTopK* for a network of 64 nodes is 48 seconds and for a network of 256 nodes is 42 seconds. For *Static Bucket* it is 2 and 6 seconds for the respective network sizes.

2. As compared to the mesh scenario 5.3a, the *Static Bucket* performs well in this scenario for the small network size because the number of messages required to converge is very less for it and this is further aided by the mobility. From the figure it can be observed that the settling time is very small for all the estimates



(a) Small Network



(b) Big Network

Figure 5.10: Rise-Time(T_r) Vs Estimate plot for Mobility Scenario

in this scenario. As compared to the mesh topology, for *MinTopK* we observe that for small network size, settling time for the final estimate is very high. This is due to the fact that because of constant mobility of the nodes the time for all the nodes to have top K value increases.

3. For big network, the settling time of the *Static Bucket* remains almost the same as mesh topology, showing that the mobility does not affect the settling time in this case for bigger network sizes. For *MinTopK* we observe that the settling time is reduced marginally than mesh topology, thus indicating that mobility helps this algorithm for bigger network sizes.

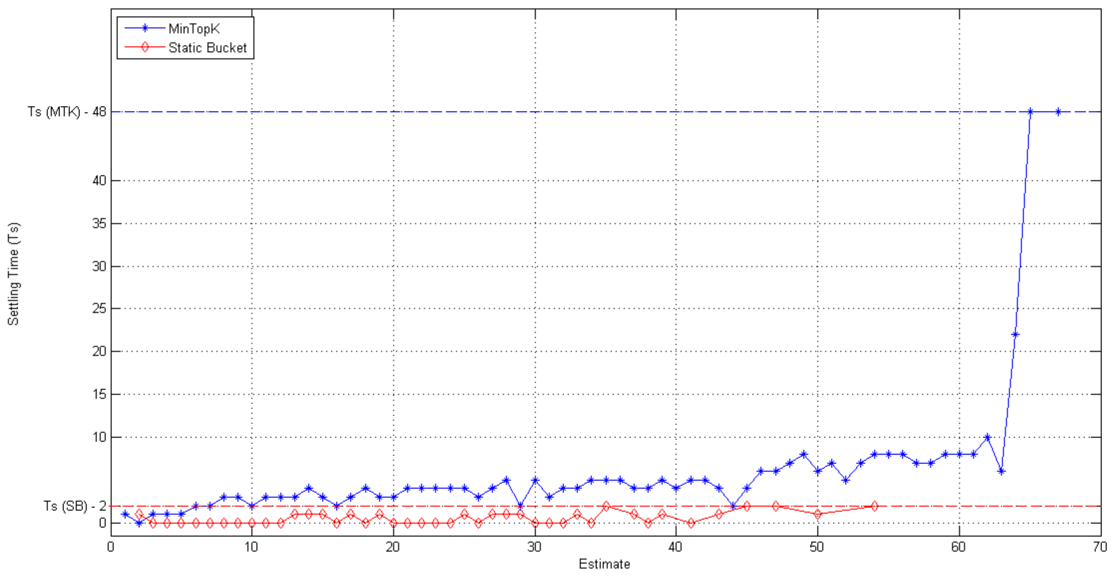
4. For *MinTopK* we can observe a lot of spikes in the big network sizes. This indicates that some lesser (inaccurate) value of estimates has a settling time more than the stable estimate. This is again because of the constant motion of the nodes, some node takes more time to have top K values, thus giving inaccurate results. This is evident by the decrease in settling time compared to mesh topology.

5.2.5 Scenario V: Network split and merge

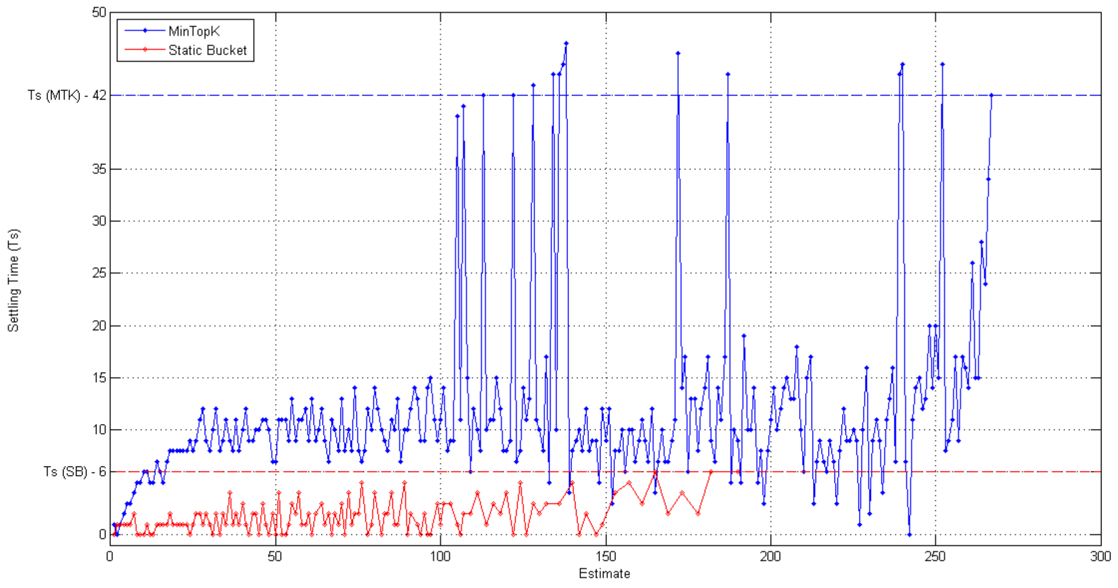
The main objective of this experiment is aimed at observing the behaviour of the algorithms in scenario where network split or merge happens. As explained in chapter 4, *MinTopK* does not support change in a network size whereas *Static Bucket* is able to do so because of the use of timestamps. Therefore, in this section we will focus just on the performance of *Static Bucket*.

Testbed

As there is no inbuilt scenario in a simulator that can be used to demonstrate network splitting or merging, therefore we are creating a scenario that mimics network partitioning/merging by deactivating/activating some nodes in the network. Figure 5.12 depicts the nodes in the same layout as used in the figure 5.4 in section 5.2.2. Here, 256 nodes are arranged in a grid of $4 * 64$. For network splitting we split the network of 256 nodes into a big network of 176 nodes and a small network of 64 nodes by deactivating 16 nodes (represented by hollow circles in the figure 5.13a) in the middle of the network after some time t. Whereas for network merging, we start the middle nodes after time t. As explained in the chapter 4, we are setting the parameter k, the maximum allowable hop count(network diameter) as 75, which is almost the diameter of the network.



(a) Small Network



(b) Big Network

Figure 5.11: Settling Time(T_s) Vs Estimate plot for Mobility Scenario

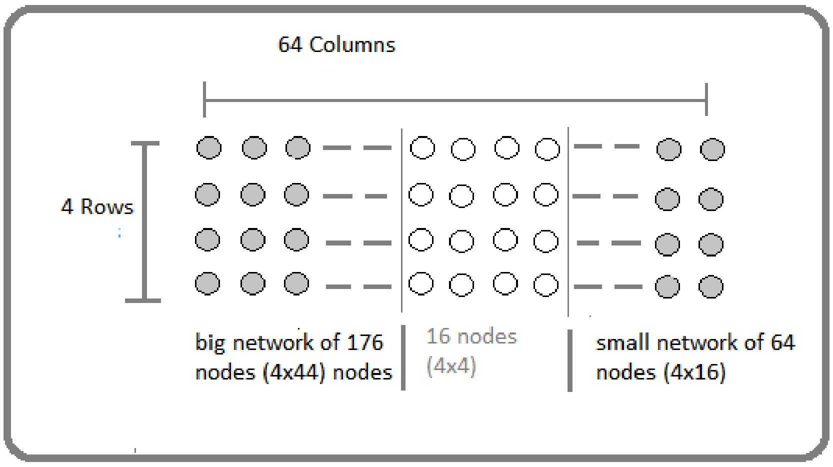

64 Columns
4 Rows
big network of 176 nodes (4x44) nodes
16 nodes (4x4)
small network of 64 nodes (4x16)

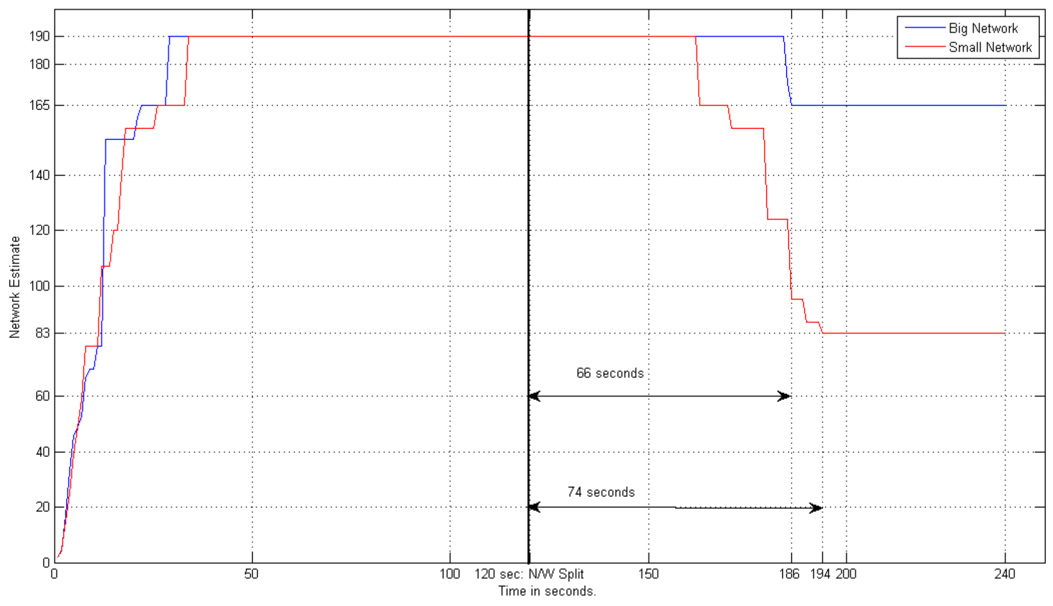
Figure 5.12: Network split/merge scenario

Result Analysis

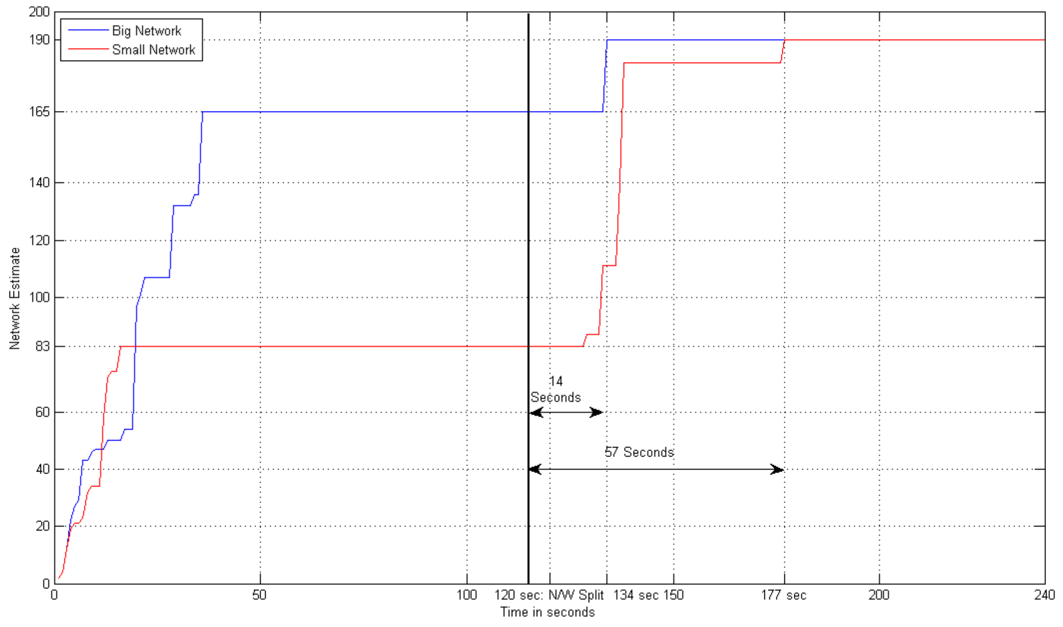
Figure 5.13 shows the Estimate Vs Time plot for the network split and merge scenarios. X-axis represents the time in seconds and the Y-axis represents the network estimate. These are the observations of a node, each from a big network and a small network. The blue line represents the behaviour of a single node from big network and the red line represents the behaviour of a single node from small network. After a change in the network size, the time duration that a node takes to completely estimate the change in the network size, is termed as the *recovery time*.

Figure 5.13a shows the case for network split. In this scenario all the nodes start at time instant t=0 and then, at time t=120 seconds the middle portion of the network which consists of the 16 nodes is deactivated, thus creating two separate networks. We can see that the recovery time of the node from the big network is 66 seconds, whereas for the node from the small network it is 74 seconds. The reason behind this is that the node in the big network has to detect the change from 190 nodes to 165 nodes, whereas the node in the small network has to detect the change from 190 nodes to 83 nodes.

Figure 5.13 shows the case for network merge. In this scenario, the nodes of the big network and small network both start at time t=0 while the middle portion of 16 nodes is kept deactivated. At time t=120 seconds, the middle portion of 16 nodes is activated



(a) Network Split



(b) Network Merge

Figure 5.13: Estimate Vs Time plot for network split and merge

thus effectively merging both the networks to form one big network. The recovery time for the node from big network is 14 seconds whereas for the node from small network is 57 seconds.

The *recovery time* in the network split scenario is more than that of the network merge scenario because, in network split scenario, when the nodes are deactivated, their messages still remain in the network for some time till it eventually dies. This is because of the use of timestamps (as explained in chapter 4).

The recovery time in the case of network merge is lesser than in the case of a network split because, in case of network merge, then messages from the nodes that have been deactivated, still remain in the network because of the timestamps, thus increasing the recovery time.

5.3 Summary

In this section, we summarize the results of the previous section. From the results and the analysis, we conclude the following points.

1. The parameter changes result in the following way

   - For *MinTopK*, as we kept increasing the size of the list, the error kept decreasing in terms of standard deviation between the number of runs. Also, the Total Delay time (T_d) kept increasing with increase in K further resulting in an increase in the rise time(T_r) and settling time(T_s).

   - For *Static Bucket* the decrease in number of buckets from 24 to 12 did not result in any form of performance penalty in terms of performance metrics. This was because of our use of algorithm for smaller network size.

2. The benefits of *MinTopK* are as follows:

   (a) The Error rate is lesser as compared to *Static Bucket* in all the scenarios.

   (b) The standard deviation between the stable estimates of different runs is also lesser in MinTopK Algorithms.

   (c) We can adjust parameter K to different values in *MinTopK* according to the requirement of the application that it is used for.

3. The benefits of *Static Bucket* are as follows:

   (a) The Total Delay (T_d) time of the *Static Bucket* is very less and it is a fraction of *MinTopK*. This ultimately results in smaller rise time (t_r) and settling time (T_r).

(b) Static Bucket has very less fluctuation during the course of its run. That means that from the start of its time to the time of its convergence the estimates recorded increases smoothly, unlike *MinTopK* where the recorded estimates fluctuates a lot until it reaches a stable estimate value.

4. When the diameter of the network is increased as in the case of scenario II, the total delay (T_d) time of both the algorithms increases. This implies that the larger network (the network with big diameter) takes more time to converge and become stable.

5. When the nodes were placed in a mesh like structure with high node density, as in scenario III compared to the scenario I, then for *MinTopK* the total delay (T_d) decreases for both the network sizes. In case of *Static Bucket* also the total delay (T_d) decreases. It can be seen more clearly in a network size of 256 nodes where it reduces from 18 seconds to 13 seconds. This shows that for a network in mesh structure and higher densities of nodes, the total delay decreases, resulting in faster convergence.

6. The introduction of mobility in the network as shown in scenario IV, plays an important role. It affects the convergence time i.e. the total delay(T_d) of the network, compared to the scenario I (Matrix Topology)5.1, 5.2. As observed from the experimental results for mobility scenario, for *MinTopK* 5.8, the total delay (T_d) is reduced in most of the cases. With a network of 256 nodes the improvement is even bigger. Same follows for *Static Bucket* 5.2.

7. The number of different estimates in the course of a transient time (the time from when the algorithm starts to the time it converges to stable estimate) of an algorithm, the variation in the rise time (T_r) and settling time (T_s) of different recorded estimate values increases with the increase in density of the nodes and also with the introduction of the mobility. This happens for both the algorithms. This is very apparent in case of *MinTopK*, because it has a higher time delay(T_d).

Chapter 6

Conclusion and Recommendation for future work

The primary goal of this assignment was to do node counting in a WSN. In the state of the art study, we discussed various methods that can be used to estimate the network size in a completely distributed WSN. Clearly, *MinTopK* and *Static Bucket* stood out, as they satisfied the requirements (section1.3.2) and were able to generate continuous estimate of the current network size. Therefore, we chose to evaluate the performance and do an analysis of these two algorithms in *MyriaNed*, each having a distinct speciality. We have implemented the *MinTopK*[1] algorithm on *MyriaSim*.

We carefully selected the performance metrics (section 4.4) as the basis for the evaluation of these two algorithms. In addition, we investigated the performance under the range of diverse scenarios (section5). As a follow up work of the authors of *Static Bucket*, we evaluated performance with node mobility also.

The uniqueness of our work lies in the fact that the authors of *MinTopK* and *Static Bucket* presented the result just in terms of error. Whereas, we presented the result in terms of steady state error, standard deviation of the runs, total delay and rise and settling times.

The message overhead incurred by *MinTopK* algorithm is 4 bytes while that of the *Static Bucket* is 12 bytes. So, in terms of message overhead *MinTopK* is very good compared to *Static Bucket*. Our simulations demonstrate that *MinTopK* has an lesser error rate than *Static Bucket*. Also, the error in terms of standard deviation of the final estimate, is lesser in case of *MinTopK* than *Static Bucket*. Therefore, in terms of **accuracy**, *MinTopK* performs better than *Static Bucket*. Nevertheless both *MinTopK* and *Static*

[1]The implementation of *Static Bucket* was provided from the authors

Bucket have an steady state error of $\pm$ 30 %.

In terms of total delay, the *Static Bucket* outperforms *MinTopK* by huge margin, as *MinTopK* takes a long time to converge compared to *Static Bucket*, thus providing a huge window for giving inaccurate estimates. Same follows in the case of rise time and delay time.

In addition to this, the results show that the node mobility, in general, adds to the performance of *MinTopK*, thus reducing total delay time.

Thus, *MinTopK* should be used, where higher accuracy is required and *Static Bucket* should be used, where total delay is of prime importance. Also, *MinTopK* is more suitable to the smaller network sizes. Because, it has a lesser steady state error rate and even though the total delay is more, it will not have a great impact on its performance. Whereas, *Static Bucket* is more suitable for the larger network sizes. Because, even though the error rate is bit higher comparatively, the total delay is very less in *Static Bucket* and that plays a greater role in bigger networks. Also, *Static Bucket* is suitable for applications in dynamic networks, where the network size changes frequently.

Recommendation for future work

The performance of *MinTopK* is hugely dependent, on the quality of generated numbers having uniform distribution. As, there is no mechanism to achieve this, the design of such random number generator(RNG) should be the next step.

Similar to the approach of [19], we can further reduce the steady state error of *MinTopK* by maintaining more than one list at a time and using the average of the estimates from all the list. Thus, even though it will add to the message overhead but this method can overcome the imperfections of the random number generator having uniform distribution.

As per [19], the network size estimates can be used by the nodes to adapt to the network density. For example: A node can decide to broadcast less often in case of high network density and since, *MyriaNed* uses TDMA protocols this would help reduce collision in the network.

Also, it is important to test these algorithms on the physical devices. Since, simulations, although essential part for the study of experiments in WSN, they often fail to address factors which can be seen in physical environments.

References

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes.," *IEEE Communications Surveys and Tutorials*, vol. 7, no. 1-4, pp. 72–93, 2005.

[2] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[4] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th international conference on Supercomputing*, ICS '02, (New York, NY, USA), pp. 84–95, ACM, 2002.

[5] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 214–226, ACM, 2004.

[6] R. van de Bovenkamp, F. Kuipers, and P. Van Mieghem, "Gossip-based counting in dynamic networks," in *NETWORKING 2012*, pp. 404–417, Springer, 2012.

[7] P. Anemaet, "Distributed g-mac: A flexible mac protocol for servicing gossip algorithms," *Master's thesis, Technical University of Delft, The Netherlands*, 2008.

[8] I. Demirkol, C. Ersoy, and F. Alagoz, "Mac protocols for wireless sensor networks: a survey," *Communications Magazine, IEEE*, vol. 44, no. 4, pp. 115–121, 2006.

[9] Wikipedia, "Time division multiple access— Wikipedia, the free encyclopedia," 2004. [Online; accessed May-2013].

[10] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 6–28, 2004.

[11] M. Nabi, T. Basten, M. Geilen, M. Blagojevic, and T. Hendriks, "A robust protocol stack for multi-hop wireless body area networks with transmit power adaptation," in *Proceedings of the Fifth International Conference on Body Area Networks*, pp. 77–83, ACM, 2010.

[12] M. Blagojevic, M. Nabi, M. Geilen, T. Basten, T. Hendriks, and M. Steine, "A probabilistic acknowledgment mechanism for wireless sensor networks," in *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pp. 63–72, IEEE, 2011.

[13] F. van der Wateren, "Myrianed development guidelines - the art of developing wsn applications with myrianed," Tech. Rep. Version 1.2, Chess B.V., Haarlem, The Netherlands, October 2012.

[14] M. Dobson, S. Voulgaris, and M. van Steen, "Merging ultra-low duty cycle networks," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 538–549, IEEE, 2011.

[15] Wikipedia, "Clock drift— Wikipedia, the free encyclopedia," 2004. [Online; accessed May-2013].

[16] M. Dobson, S. Voulgaris, and M. van Steen, "Network-level synchronization in decentralized social ad-hoc networks," in *Pervasive Computing and Applications (ICPCA), 2010 5th International Conference on*, pp. 206–212, IEEE, 2010.

[17] Libelium, "50 sensor applications for a smarter world @ONLINE."

[18] J. Evers, D. Kiss, W. Kowalczyk, T. Navilarekallu, M. Renger, L. Sella, V. Timperio, A. Viorel, S. van Wijk, and A.-J. Yzelman, "Node counting in wireless ad-hoc networks," *European Study Group Mathematics with Industry*, p. 49, 2011.

[19] M. Dobson, *Low-power epidemic communication in wireless ad hoc networks.* PhD thesis, Vrije University, 2013.

[20] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. K. Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte, and S. Valentin, "Simulating wireless and mobile networks in omnet++ the mixim vision," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, (ICST, Brussels, Belgium, Belgium), pp. 71:1–71:8, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[21] Wiki, "Wsn platforms @ONLINE."

[22] F. van der Wateren, "Myriacore implementation details - the inside of myriacore

and gmac," Tech. Rep. Version 0.3, Chess B.V., Haarlem, The Netherlands, April 2010.

[23] M. Dobson, "Myriasim: A simulator for myrianed based on omnet++ and mixim."

[24] A. Varga *et al.*, "The omnet++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM2001)*, vol. 9, sn, 2001.

[25] P. Jesus, C. Baquero, and P. S. Almeida, "A survey of distributed data aggregation algorithms," *arXiv preprint arXiv:1110.0725*, 2011.

[26] C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus, "Extrema propagation: Fast distributed estimation of sums and network sizes," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 4, pp. 668–675, 2012.

[27] Z. Alliance, "Zigbee specification," *Document 053474r06, Version*, vol. 1, 2006.

[28] Wikipedia, "German tank problem — Wikipedia, the free encyclopedia," 2004. [Online; accessed 11-July-2013].

[29] R. Ruggles and H. Brodie, "An empirical approach to economic intelligence in world war ii," *Journal of the American Statistical Association*, vol. 42, no. 237, pp. 72–91, 1947.

[30] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi, "In-network aggregation techniques for wireless sensor networks: a survey," *Wireless Communications, IEEE*, vol. 14, no. 2, pp. 70–87, 2007.

[31] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating aggregates on a peer-to-peer network," *submitted for publication*, 2003.

[32] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh, "Peer counting and sampling in overlay networks: random walk methods," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pp. 123–132, ACM, 2006.

[33] A. Montresor, M. Jelasity, and O. Babaoglu, "Robust aggregation protocols for large-scale overlay networks," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 19–28, IEEE, 2004.

[34] M. Jelasity and A. Montresor, "Epidemic-style proactive aggregation in large overlay networks," in *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pp. 102–109, IEEE, 2004.

[35] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate in-

formation," in *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pp. 482–491, IEEE, 2003.

[36] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, (New York, NY, USA), pp. 1–12, ACM, 1987.

[37] A. Metwally, D. Agrawal, and A. E. Abbadi, "Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, (New York, NY, USA), pp. 618–629, ACM, 2008.

[38] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[39] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Algorithms-ESA 2003*, pp. 605–617, Springer, 2003.

[40] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh, "Peer counting and sampling in overlay networks: random walk methods," in *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, (New York, NY, USA), pp. 123–132, ACM, 2006.

[41] S. Mane, S. Mopuru, K. Mehra, and J. Srivastava, "Network size estimation in a peer-to-peer network," *University of Minnesota, MN, Tech. Rep*, pp. 05–030, 2005.

[42] P. Flajolet and G. N. Martin, "Probabilistic counting," in *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pp. 76–82, IEEE, 1983.

[43] MATLAB, *version 7.14.0.739 (R2012a).* Natick, Massachusetts: The MathWorks Inc., 2010.

[44] N. Aschenbruck, R. Ernst, E. Gerhards-Padilla, and M. Schwamborn, "Bonnmotion: a mobility scenario generation and analysis tool," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, p. 51, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[45] Wikipedia, "Walking — Wikipedia, the free encyclopedia," 2004. [Online; accessed 30-June-2013].