THESIS

TOP-DOWN CLUSTERING BASED SELF-ORGANIZATION OF COLLABORATIVE WIRELESS SENSOR NETWORKS

Submitted by

H. M. N. Dilum Bandara

Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall, 2008

COLORADO STATE UNIVERSITY

July 21, 2008

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY H. M. N. DILUM BANDARA ENTITLED "TOP-DOWN CLUSTERING BASED SELF-ORGANIZATION OF COLLABORATIVE WIRELESS SENSOR NETWORKS" BE ACCEPTED AS FULLFILING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Committee Member – Dr. Daniel F. Massey

Committee Member – Dr. V. Chandrasekar

Adviser – Dr. Anura P. Jayasumana

Department Head – Dr. Anthony A. Maciejewski

ABSTRACT OF THESIS

TOP-DOWN CLUSTERING BASED SELF-ORGANIZATION OF COLLABORATIVE WIRELESS SENSOR NETWORKS

Recent advances in Wireless Sensor Network (WSN) technology are enabling the deployment of large-scale and collaborative sensor networks. Energy efficient operation, channel contention, latency, management, and security of such networks are complex and critical issues that have to be addressed with large-scale WSN deployments. Collaborative sensor networks further require dynamic grouping of nodes observing similar events and communication within such groups or across different groups. Cluster based organization of large sensor networks is the key for many techniques that addresses these issues. A backbone network in the form of a cluster tree can further enhance upper layer functions such as routing, broadcasting, and in-network query processing.

A configurable cluster and cluster tree formation algorithm is presented that is independent of network topology and does not require a-priori neighborhood information, location awareness, or time synchronization. Configurable parameters of the algorithm can be used to form cluster trees with desirable properties such as controlled breadth and depth, uniform cluster size, and more circular clusters. Message complexity of the algorithm grows linearly with the number of nodes in the network, therefore algorithm scales well into large networks. Two-step, post cluster optimization phase is proposed to

increase the connectivity of the network and to further reduce the depth of the cluster tree. Simulation based analysis shows that the algorithm forms more circular and uniform clusters, cluster tree with lower depth, and more importantly forms a more ordered structure in the network. Closeness of clusters to hexagonal packing is evaluated. The structure imposed by the algorithm makes it applicable to broad classes of applications.

The proposed cluster tree based routing strategy facilitates both node-to-sink and node-to-node communication. Hierarchical addresses that reflect the parent-child relationship of cluster heads is used to route data along the cluster tree. Utilization of cross-links among neighboring cluster heads and a circular path within the network approximately doubles the capacity of the network. Under ideal conditions, this approach guarantees delivery of events/queries and has a lower overhead compared to routing strategies such as rumor routing and ant routing. The cluster tree formed by our algorithm is used to identify and form Virtual Sensor networks (VSNs), an emerging concept that supports resource efficient collaborative WSNs. Our implementation of VSN is able to deliver unicast, multicast, and broadcast traffic among nodes observing similar events, efficiently. Efficacy of the VSN based approach is evaluated by simulating a subsurface chemical plume monitoring system. The algorithm is further extended to support the formation of a secure backbone that can enable secure upper layer functions and dynamic distribution of cryptographic keys, among nodes and users of collaborative sensor networks.

H. M. N. Dilum Bandara
Electrical and Computer Engineering Department
Colorado State University
Fort Collins, CO 80523
Fall, 2008

ACKNOWLEDGMENTS

I would like to acknowledge and extend my heartfelt gratitude to many people for their support and encouragement that has made the completion of this thesis possible.

It has been the most wonderful experience of my life as a scholar to work under the guidance of my adviser, Prof. Anura Jayasumana. I am very much thankful to him for believing in my abilities, inspiration, sound advice, patience, and invaluable support in every aspect.

Special thanks must go to Dr. Indrajit Ray for his guidance and sound advice. I also would like to thank Prof. Chandrasekhar, Dr. Massey, and other lectures from the Electrical and Computer Engineering, and Computer Science departments. The extensive support from Prof. Tissa Illangasekare and Kevin Barnhart from Colorado School of Mines was immensely helpful.

I am grateful to Tarun, Lee, Ram, Supriya, Saket, Raghunandan, and Dulanjalie, colleagues of the Computer Networking Research Laboratory (CNRL), for their guidance, support, and invaluable thoughts. I am indebted to my parents and my loving wife for their continuous support and encouragement. I also would like to thank peers and every individual that I may have not mentioned above that helped me in various ways.

This work is supported in part by the grant from Environmental Sciences Division, Army Research Office (AMSRD-ARL-RO-EV).

To my parents and my loving wife Sudeshini

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Recent advances in wireless communications and miniature, low power, and low cost sensors are enabling the deployment of large-scale and/or collaborative Wireless Sensor Networks (WSNs). These networks enhance the perception of our surrounding by sensing the physical world around us at a far greater temporal and spatial granularity than have been hitherto possible. Numerous WSN systems are being proposed and implemented leading to novel applications in areas such as habitat monitoring [58], eldercare, smart neighborhood [29, 62], disaster response, surveillance [69], and battlefield intelligence [55].

Section 1.1 presents the factors that motivated the project. Contributions of the thesis are presented in Section 1.2. Section 1.3 provides a brief outline of the rest of the thesis.

1.1 Motivation

Sensor networks are composed of large number of densely deployed sensor nodes that are positioned either inside the phenomenon or very close to it. In most cases, these nodes may be randomly deployed. Self-organization capabilities and corporation among sensors are essential characteristics of these randomly deployed networks [3]. Energy

efficient operation, channel contention, latency, and management of such networks are complex and critical issues that have to be addressed with large-scale WSN deployments. In contrast to early sensor networks that were dedicated to a certain application, collaborative networks that perform different tasks and deployed in the same geographical region are emerging. e-SENSE [29] and U-City [62] are two such projects that enable a smart neighborhood. Better resource efficiency can be achieved by allowing these multiple networks to collaborate with each other [40] with many users accessing different portions of the network. Privacy and dynamic key distribution are some of the unique requirements of such collaborative networks.

Cluster based organization of large sensor networks is the key for many techniques that address these issues [68]. In general, the network is decomposed into a set of administrative entities called *clusters*, with each cluster formed by grouping a set of nearby nodes. Each cluster is managed by a designated node called the *Cluster Head* (CH). With many solutions based on clustering, the nodes within a cluster communicate only with their CH. As a result, member nodes can use a lower transmission power to reach the CH. This increases network lifetime, reduces collisions, and enables spatial reuse of the communication channel [67]. The CHs are responsible for coordinating both inter-cluster and intra-cluster communication. Communication among CHs can be via either single or multi-hops. Clustering reduce the power consumption of the overall network while increasing the network lifetime [68]. Number of messages that flow through the network can be further reduced by aggregating data within a cluster [33, 67]. Applications that span large sensor fields and/or support data aggregation are prime candidates for cluster-based configuration. Clustering is particularly useful for logically

separating multiple sensor applications that perform different tasks and deployed in the same physical area [40].

Many clustering solutions have been proposed in literature some of which will be discussed in Chapter 2. Solutions such as LEACH [33] and HEED [67] increase the network lifetime by frequently alternating the role of a CH among different nodes and by aggregating data. However, these solutions assume that each CH is capable of directly communicating with the base station. This may not be possible in a geographically large network where the base station is more than a hop away.

A backbone network that arranges CHs in the form of a *cluster tree* can be used to forward data from individual clusters to the base station or to facilitate inter-cluster communication. Cluster trees are useful in delivering unicast, multicast, broadcast traffic [65], for data fusion, and for in-network query processing. Performance of such upper layer functions depend on the number of hops between nodes and the base station. As the hop count increases, both the latency and the energy to forward a message increase. For many large-scale applications, it is desirable to have a cluster tree with a lower depth. Though several hierarchical clustering solutions are being proposed [9, 47] they are either not scalable or do not guarantee good connectivity as the networks become larger [20]. These solutions do not provide any mechanism to form and manage collaborative WSNs.

Virtual Sensor Networks (VSNs) is an immerging concept that supports collaborative, resource efficient, and multipurpose sensor networks that may involve dynamically varying subset of sensors and users [40]. Realization of VSNs requires protocol support for formation, usage, adaptation, and maintenance of subset of sensors collaborating on a specific task(s). Hence, there is still a need for a clustering solution

that facilitates most of the aforementioned characteristics of large and collaborative WSNs. It is important to build a solution that imposes some predictable structure on the network and is independent of network topology, neighborhood information, location awareness, time synchronization, etc. The solution need to be scalable and should facilitate the self-organization, management, and security requirements of VSNs and other collaborative WSNs.

1.2 Contributions

Imposing some structure within the network to effectively achieve the application objectives is an attractive option for the self-organization of large-scale WSNs. Cluster based organization and arranging clusters in form of a tree simplifies many higher-level functions and distributed application deployment. Security imposes additional restrictions that need to be satisfied in collaborative WSNs. However, these properties are harder to achieve in resource constrained WSNs.

The thesis presents Generic Top-down Cluster and cluster tree formation (GTC) algorithm, a configurable algorithm that is capable of achieving most of the desirable properties. A hybrid approach that combines local and neighbor information and controllability of the top-down approach is exploited to achieve desired cluster and tree characteristics. The algorithm is independent of network topology and does not require a-priori neighborhood information, location awareness, or time synchronization. The algorithm has a message complexity of $O(n)$, where n is the number of nodes in the network, hence scales well for large networks. Parameters in the algorithm allow cluster and tree characteristics to be changed, e.g., to achieve uniform and circular clusters

and/or cluster trees with controlled breadth and depth. Simple Hierarchical Clustering (SHC), a special case of GTC, is similar to the IEEE 802.15.4 cluster tree [38]. Another special case, Hop-ahead Hierarchical Clustering (HHC) is presented that produces more circular and uniform clusters, and cluster trees with lower depth. Two-step, post cluster optimization phase is also proposed that improve the connectivity of the network and reduce the depth of the cluster tree.

Simulation based analysis shows that the algorithm forms more circular and uniform clusters and cluster trees with lower depth. Based on the cluster tree, the GTC algorithm forms a more ordered structure in the network and has a bounded distance between any parent and child CH. Our analysis shows that properties of HHC are comparable with hexagonal packing, particularly for low-density networks. The HHC forms more circular clusters than [9] and [25]. Receiver Signal Strength Indicator (RSSI) based HHC forms even more uniform clusters and a cluster tree with lower depth. For similar overhead, HHC forms both clusters and a cluster tree while [17] only forms set of clusters. The proposed optimization phases further increase the connectivity of the network and optimize the cluster tree.

The cluster tree formed by the HHC scheme of the GTC algorithm is used to facilitates both node-to-sink and node-to-node communication. Hierarchical address structure that reflects the parent-child relationship among CHs is designed. Such hierarchical addresses greatly simplify routing. In addition, CHs need to store only the routing entries related to their parent and child CHs. Cross-links among neighboring CHs and a circular path within the network is formed to further enhance the capacity of the network. These optimizations allow selection of multiple paths to a given destination

without being tied to the cluster tree. Hierarchical addresses are useful in this case to determine the shortest paths to a given destination. These optimizations approximately double the capacity of the network. Optimum position of the circular path is determined analytically.

The cluster tree formed with HHC scheme is used to identify and form VSNs. Nodes observing the same phenomenon send a message towards the root of the cluster tree. These messages form a *virtual tree* rooted at the root node. This virtual tree can be used to efficiently deliver unicast, multicast, and broadcast traffic among nodes observing the same phenomenon. This approach is more suitable for large and collaborative sensor networks because it grantees delivery of events and has a lower overhead compared to approaches such as Rumor Routing [14], Zonal Rumor Routing [10] and Ant Routing [35]. Localized and distributed phenomenon based simulations are utilized to determine the feasibility of this approach. A subsurface chemical plume monitoring system is simulated to further analyze the efficacy of the VSN based approach.

The algorithm is further extended to support the formation of a secure backbone that can facilitate secure upper layer functions and dynamic distribution of network-wide or group-wide cryptographic keys in collaborative sensor networks. The extended GTC algorithm is independent of the key pre-distribution scheme. Simulation based analysis shows that algorithm retains most of its desirable cluster and cluster tree characteristics, while building the secure backbone. Our analysis also suggests that hierarchical WSNs are more vulnerable to node capture than non-hierarchical networks.

1.3 Outline

Rest of the thesis is organized as follows. Following chapter describes current work related to clustering, routing, and key distribution in WSNs. Chapter 3 describes the desirable characteristics of a cluster and cluster tree formation algorithm, our network model, and problem statement. The GTC algorithm and its performance analysis are presented in Chapter 4. Further optimizations to the algorithm, post cluster optimization phase, and extensive performance analysis are presented in Chapter 5. The hierarchical addressing scheme and three routing strategies are presented in Chapter 6. Chapter 7 presents the mechanism used to identify and form VSNs and it is followed by the chapter on secure backbone formation. Finally, concluding remarks and future work are presented in Chapter 9. The appendices provide detailed explanation of the simulator and its source code.

Chapter 2

BACKGROUND AND RELATED WORK

Clustering, routing, and security have been among the key research areas in wireless sensor networks. In contrast to early sensor networks that were dedicated to a certain application, collaborative networks that perform different tasks and deployed in the same geographical region are emerging. These collaborative networks require either adaptation of existing technologies or new inventions.

The chapter provides a brief description of existing work that motivated or comparable with the ideas presented in the thesis. Section 2.1 describes the work related to clustering in WSNs. Cluster tree formation approaches are described in Section 2.2. WSN routing and security solutions are presented in Section 2.3 and 2.4, respectively. Brief introduction to collaborative sensor networks is presented in Section 2.5.

2.1 Clustering In Wireless Sensor Networks

Energy efficient operation, channel contention, latency, and management are complex and critical issues that have to be addressed with large-scale WSN deployments. In large-scale sensor networks, faraway nodes have to depend on large number of intermediate nodes to forward their data or have to use high transmission power. Former approach increases the latency and power consumption of the entire network while later

approach increases the potential for collisions and significantly increases the power consumption of nodes that are faraway. Many solutions and algorithms for overcoming these problems depend on decomposing the network into number of administrative entities called *clusters* [9, 17, 25, 28, 33, 44, 62, 67]. The structure imposed by clustering makes it somewhat easier to manage the problems introduced by the complexity of large-scale sensor networks. In general, the nearby nodes in a network are grouped into set of clusters, with each cluster managed by a *Cluster Head* (CH). In many solutions, the nodes within a cluster communicate only with their CH. Communication among CHs can be via either single or multiple hops. The CHs are responsible for coordinating both inter-cluster and intra-cluster communication. Applications that span a large sensor field such as earthquake monitoring and applications that support data aggregation such as microclimate and habitat monitoring are candidates for clustering. Clustering is particularly useful for logically separating multiple applications that perform different tasks and that are deployed in the same physical area [40].

Clustering based solutions have their own pros and cons [63, 67-68]. Clusters can reduce the power consumption of a WSN, therefore increase the lifetime of the network [68]. Nodes within a cluster need only to communicate with its CH where by allowing each node to reduce its communication range [33, 67]. This allows the spatial reuse of communication channel while reducing collisions. Number of messages that flow through the network can be further reduced by aggregating data [33]. However, forming and maintaining clusters is a complex task and the associated communication messages may add considerable overhead. The rotation of the role of becoming a CH (e.g., to balance

the workload) and handling node dynamic such as new, moving, or deteriorating nodes are among other issues that need to be addressed.

Cluster formation can be either distributed or centralized. A key challenge in both of these approaches is the selection of the best set of CHs. The CHs can be selected based on parameters such as node ID [44], node degree [21], residual energy [67], or probabilistically [9, 33]. Lowest ID clustering [44], Distributed Clustering Algorithm (DCA) [12], and Max-Min d-clustering [4] are solutions that are relatively simple to implement, yet not directly applicable to WSNs because they are not energy aware. LEACH [33] and HEED [67] are two distributed cluster formation solutions that achieve longer network lifetime by probabilistically selecting CHs based on residual energy of nodes and data aggregation. LEACH does not actually measure the residual energy of a node instead assumes uniform energy consumption for all the CHs. Because of this assumption, it does not guarantee good distribution of CHs. Some of these problems are addressed in [66-67]. LEACH-C [33] proposes a centralized solution that further enhances the network lifetime. Overhead of localized decision based distributed clustering solutions such as [9, 17, 33, 67] are lower compared to centralized solutions such as LEACH-C. However, lack of global knowledge limits the possibility of forming optimum set of clusters (maximum spatial coverage with least number of clusters) in distributed solutions.

A hybrid scheme that combines local and neighbor information can form better clusters with lower overhead. FLOC [25] and ACE [17] are two such approaches that form more uniform and circular clusters than the probabilistic approaches. The FLCO (Fast, LOcal Clustering service) makes use of the dual-band wireless radio model. A CH

can reliably communicate with the nodes that are in its inner-band (*i-band*) and unreliably with the nodes in its outer-band (*o-band*). A CH forms a *solid-disk* cluster by connecting all the nodes that are within its *i-band*. Nodes that are outside the *i-band* of any CH later join the closest CH, if it is within the *o-band* of that CH. FLOC forms none overlapping and approximately equal size clusters. In ACE (Algorithm for Cluster Establishment), CHs are selected using an iterative process based on neighborhood information. ACE clusters are more circular and has properties closer to hexagonal packing. However, iterative messages significantly increase the overhead of ACE. All the aforementioned solutions assume that each CH is capable of directly communicating with the base station. This may not be possible in a geographically large network where the base station is beyond the maximum transmission range of a node.

2.2 Hierarchical Wireless Sensor Networks

A backbone network that arranges CHs in the form of a *cluster tree* can be used to forward data from individual clusters to the base station or to facilitate inter-cluster communication. Cluster trees are useful in delivering unicast, multicast, and broadcast traffic [65], for data fusion, for in-network query processing, etc. Cluster trees can be formed using either bottom-up or top-down approach.

In a bottom-up approach, the individual clusters are formed independently and later combined together to form a higher-level structure such as a cluster tree. A bottom-up, Probabilistic Hierarchical Clustering (PHC) solution is proposed in [9]. Each node has different probabilities of becoming a CH at different levels of the hierarchy. At the lowest level (*level 1*), nodes probabilistically form their own clusters within a multi-hop

neighborhood. These CHs are then combined together and form the next level (*level 2*) of the hierarchy. Then another set of CHs from *level 2* is probabilistically selected to represent *level 3*. This process continues until the desired number of levels is formed. The hierarchy is formed by connecting CHs in level *i* to a CH in level *i + 1*. Data is aggregated at *level 1* and passed to *level 2*, then from *2* to *3*, and the highest level forward the data to the base station. Threshold sensitive Energy Efficient sensor Network (TEEN) protocol extends LEACH to form a similar hierarchy [47]. As we go up the hierarchy, the distance between CHs of adjacent levels increases [20]. Therefore, ensuring connectivity among these clusters in a geographically large network is not straightforward as well.

In top-down approach, a designated *root node* first forms its own cluster. It then selects some of its neighbors to form their own clusters, which in turn cause some of their neighbors to form the next level of clusters. This process continues until the entire sensor field is covered. The cluster tree is formed by keeping track of parent and child CH relationship.

The IEEE 802.15.4 standard [38] proposes a top-down cluster tree formation approach. The Personal Area Network (PAN) coordinator, a fully functional device capable of providing synchronization services, etc. and identified as the principal coordinator of the network, forms the first cluster by choosing an unused PAN ID and broadcasts beacon frames to neighboring nodes. A node receiving a beacon may request to join the cluster at the PAN coordinator. If the PAN coordinator permits the node to join, it adds the new node to its *neighbor list* as a child node and the newly joined node adds the PAN coordinator as its parent node to its own neighbor list. It then keeps forwarding beacon frames from the PAN coordinator and other nodes may then join the

cluster at this node. As seen in Figure 2.1 (extracted from [38]), nodes 1, 3, 4, 6, and 11 join the PAN coordinator with node ID 0. Then those nodes keep propagating beacon frames allowing nodes 2, 7, 9, 10, 22, etc. to join the cluster. Once a predefined application or network requirement such as number of nodes in a cluster or maximum hop count is met, the PAN coordinator may instruct a node to become a coordinator of a new cluster. This process continues until all the nodes are covered. The lines in Figure 2.1 represent only the parent child relationship of nodes and convey no information on the physical shape of the cluster tree. It is unlikely that any of the clusters will be circular in shape, except perhaps cluster one (PAN ID 1). The ZigBee standard proposes a different implementation for clustering 802.15.4 networks based on the Motorola cluster tree algorithm [46]. It first forms individual clusters and later combine them to form a cluster tree based on the top-down approach. ACP [28] is another top-down cluster

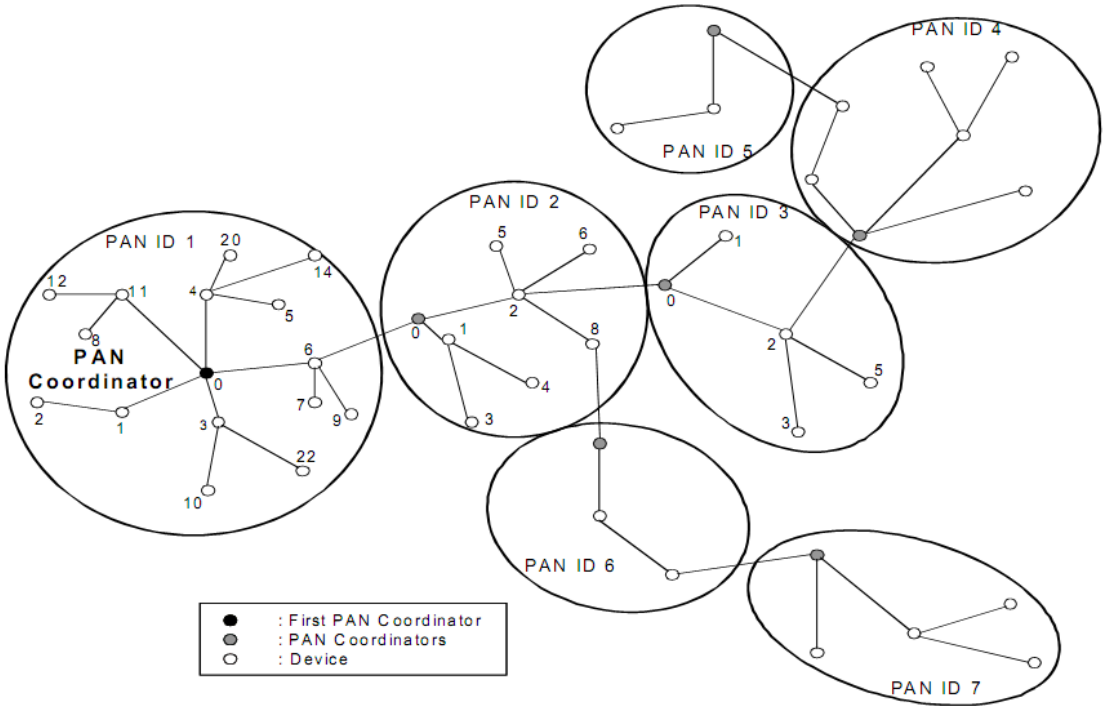


Figure 2.1 – IEEE 802.15.4 cluster tree [38].

formation approach that depends on device location information. ACP produces a large number of overlapping clusters. However, it does not form a cluster tree.

The bottom-up approach, although conceptually appears to be relatively simple, involves considerable communication overhead while building the cluster tree and provides very little or no control on depth and breadth of the tree formed. The top-down approach provides better control while forming clusters and the cluster tree. For example, hierarchical addresses can be assigned to clusters while the cluster tree is being formed, which greatly simplify routing and only require CHs to keep track of its parent and child CHs. Such an approach can also control the number of nodes in a cluster, breadth, and depth of the cluster tree. However, uncontrolled top-down approaches such as the basic scheme given in the IEEE 802.15.4 standard, result in undesirable cluster and tree characteristics such as large variations in cluster size and distance to leaf nodes [5]. The IEEE 802.15.4 standard however is quite flexible and does not prevent one from deploying alternative clustering approaches.

2.3 Wireless Sensor Network Routing Protocols

Resource constrains, data-centric routing, many-to-one communication pattern, redundant data, and inability to build a global addressing scheme make routing in sensor networks challenging and different. Routing protocols are highly influenced by data-centric nature of WSNs and data delivery models can be continuous, event-driven, query-driven, or hybrid [1]. Data-centric routing requires naming schemes that reflect the attributes of the phenomenon rather than addressing individual nodes. For example, in most conventional WSN applications users are more interested in queries such as “which

areas are having temperature over 30 ^{o}C" [3]. However, things are somewhat different with large and collaborative sensor networks. Such networks require some structure within the network and tend to communicate across different nodes, networks (many-to-many communication strategy), and users. WSN routing protocols can be broadly classified as data-centric, hierarchical, and location based [1].

Flooding is the simplest routing approach that broadcasts events within the entire network. Though this approach is simple to implement and requires no prior route setup, it generates significantly large number of messages and energy is wasted due to implosion, i.e., caused by duplicate message send to the same node. Gossiping is a controlled form of flooding where an event is forwarded only to a randomly selected node instead of a broadcast.

In Directed Diffusion, one of the key WSN routing protocols, the sink floods the network with attribute-based queries [39]. As the query propagates through the network, routing path to the sink is established. All receiving nodes cache the query and later respond through pre-established paths if they observe matching event(s). On-demand data delivery increases the energy efficiency of Directed Diffusion and does not require keeping track of global network topology. However, this approach is less scalable and overhead of flooding will dominate if new queries are frequent. Similar query flooding mechanism that significantly increases the network lifetime is presented in [52]. It uses a path selection mechanism that is energy aware. Event data is forwarded through different paths so that it balances the energy consumption of intermediate nodes.

Sensor Protocols for Information via Negotiation (SPIN) is an event driven protocol [41]. A node observing a certain event broadcasts an advertisement of the event

using meta-data. If any of its neighbors are interested in the event they will request the actual data. Actual data is send only to those interested neighbors. Those neighbors then send a new advertisement to their neighbors indicating the availability of data. Data will be forwarded further, if those neighbors are interested. This approach is efficient if data generation is infrequent. SPIN does not need to manage any network topology. However, SPIN data advertisement mechanism does not guarantee delivery of an event to all the interested nodes. If intermediate nodes are not interested in the event, data will not be forwarded any further. Data advertisement and request messages add unnecessary overhead, if many nodes are anyway interested in an event. Though these approaches work for well-defined cases, they are not capable of supporting requirements of collaborative sensor networks.

Rumor Routing [14] is another class of data-centric routing protocol that makes use of agents to propagate both events and queries. It is a hybrid scheme that makes use of constrained event and query flooding. An agent is generated when an event occurs. These agents spread rumors about events across the network using long-lived packets. As the agent is forwarded, path to the event is setup and intermediate nodes cache the event details. A node querying an event generates another agent. These querying agents travel through the network and try to discover a node that knows about the event. If such a node is met, data stored in that node is used to figure out a path to the event. Rumor routing achieves significant energy saving over data and event flooding. If events are frequent, overhead of agents becomes dominant. Zonal Rumor Routing (ZRR) [10] is an extension of Rumor Routing. In ZRR, the network is decomposed into a set of zones and agents are randomly forwarded from one zone to another. This approach improves the number of

successfully delivered queries and reduces the energy consumption. Both Rumor Routing and ZRR can be used to determine a set of nodes observing similar events in collaborative sensor networks. However, neither of these solutions guarantees that two nodes observing similar events will meet each other. Successful delivery depends on the lifetime of agents and significantly high *TTL* (Time To Live) values are required to achieve an acceptable success rate.

Ant routing proposes a mechanism to discover and maintain paths in ad-hoc networks [35]. When a node wants to find and/or maintain a path to a destination, it sends ants (similar to agents) searching for its destination. Ants collect path information as they travel. When an ant reaches the destination, another ant is generated and it carries path information back to the source. Over time, ants travel through different paths and try to discover better paths. This approach is somewhat applicable for collaborative sensor networks that require node-to-node or network-to-network communication. Like Rumor Routing, this approach does not guarantees path discovery and requires long-lived ants.

Most of the hierarchical routing solutions are based on hierarchical clustering solutions. LEACH [33] defines a simple hierarchical routing strategy where data is aggregated at cluster heads and send directly to the base station using long-range communication. TEEN (Threshold sensitive Energy Efficient sensor Network) [47] extends the two level hierarchy of LEACH to multiple levels. The routing scheme proposed in [9] is identical to TEEN. Both these solutions aggregate data at multiple levels before forwarding to the base station. Hierarchy becomes a bottleneck as messages are forced to follow the hierarchy. Nodes along the hierarchy die much faster and as a result, these routing solutions have much lower capacity (i.e., network lifetime).

A multi-layer architecture that increases the network capacity is presented in [36]. Sensor nodes are arranged in a multi-layer architecture based on a binary tree. Nearby nodes that belongs to the same level in the binary tree then form cross-links within themselves (similar to a de Bruijn graph). These cross-links reduce the load on the hierarchy and significantly increase the capacity of the network. A binary addressing scheme is also proposed that simplifies routing across cross-links. However, authors do not provide any explanation on how such a binary tree can be built and addresses can be assigned. Use of a binary tree makes the solution less scalable for large-scale WSNs.

Geographic routing makes use of node location information (absolute or relative) when forwarding data. A message is always forwarded to a node that is closer to the destination. Though many geographic routing solutions exist, as they are beyond the scope of the thesis those will not be discussed. Each routing solution has its own advantages and disadvantages and only applicable for certain types of applications. None of the previously mentioned solutions is directly applicable for large-scale and collaborative WSNs. Hence, it is important to develop routing protocols that facilitates the desired properties of large and collaborative WSNs.

2.4 Key Distribution In Wireless Sensor Networks

Security is a prime concern in large-scale WSNs used for collaborative and mission critical applications. Due to resource constrains securing WSNs are not straightforward and traditional security techniques used in wired and wireless networks are not directly applicable [49]. In addition, unlike traditional networks, sensor nodes are often deployed in inaccessible and inhospitable areas, presenting the added risk of

physical attacks such as node capture and physical tampering [15, 18, 49]. The wireless nature of communication further aggravates the problem because attackers can easily intercept, fabricate, or jam traffic. Lack of prior knowledge about the network topology further complicates the design and verification of secure protocols [15, 18, 49]. Strong and efficient distribution of cryptographic keys is the first step towards achieving these objectives, on top of which may secure protocols can be implemented.

Though dynamic key assignment based on public key infrastructure is popular in wired networks, it is not suitable for WSNs due to the complexity of implementation and computational cost. A fully distributed key generation/distribution approach for heterogeneous sensor networks based on Elliptic Curve Cryptography (ECC) is proposed in [27]. Each sensor node is preloaded with a set of private keys, while nodes with enhanced capabilities store public keys of all the other nodes. Nodes with enhanced capabilities act as Cluster Heads (CHs) and form a backbone network. CHs assign symmetric keys to child nodes. This approach requires nodes with enhanced capabilities to be tamper proof, all nodes to be location aware, and make use of geographic routing.

Key assignment through a trusted base station is another approach. A hierarchical key generation and diffusion algorithm based on parent-child relationship of nodes is presented in [53]. A similar approach based on hierarchical CHs is presented in [15]. Use of CHs reduces the depth of the hierarchy compared to [53]. Both of these solutions require a large number of control messages to be transmitted among nodes requesting keys, CHs, and the base station. The security of the system depends on the physical security of the base station. In most cases, the base stations are also deployed in the same

inaccessible or inhospitable area as the sensor nodes which intern does not provide any added security. The base station also becomes a single point of failure.

Key pre-distribution is currently the most attractive solution for key distribution in WSNs due to its lower computational cost and communication overhead. A randomized key pre-distribution scheme is proposed in [31]. Initially each node is loaded with a fixed number of randomly selected keys from a large key pool. After deployment, nodes try to discover common key(s) between their one-hop neighbors. If two neighbors do not share a common key, they establish a secure path through a third neighbor. Another scheme that achieves much higher resilience against small-scale node capture is proposed in [19]. Though these approaches are relatively simple to implement, the probability of sharing at least one common key between two neighbors is considerably lower. Therefore, most of the nodes have to rely on a third node to establish a common key. Ability to share at least one key with its neighbors is referred as the *local connectivity*. These approaches are not directly applicable in hierarchical WSNs because cluster membership is meaningless, if a node does not directly share a key with the CH.

A more deterministic scheme based on combinatorial design is presented in [42]. Local connectivity of this approach is also lower. Another solution based on Random Block Merging in Combinatorial Design (RBMCD) is proposed in [16]. It combines the desirable properties of [31] and [42] and ensures one or more common keys between any two nodes. Combinatorial design has a compact and efficient algebraic description and can use a group ID to identify a set of keys [42]. This yields a simple algorithm for shared-key discovery, in which very little data needs to be transmitted between two nodes trying to discover a key. Due to ease of key generation, pre-distribution, shared key

discovery, low computational cost, and low communication overhead combinatorial approaches are much more attractive for hierarchical WSNs.

Another set of key pre-distribution schemes makes use of deployment knowledge to effectively assign keys. An extended random key pre-distribution scheme based on deployment knowledge is proposed in [26]. Nodes are deployed from different positions in the network based on a 2-D Gaussian distribution. Each node selects a set of random keys from a key pool. Each key pool shares a certain fraction of keys with its horizontal, vertical, and diagonal neighbors. Key pools that are disjoint do not share any common keys. Sharing across key pools significantly increase the local connectivity. This scheme has a better resilience against localized node capture. A deployment knowledge based combinatorial scheme is also proposed in [54]. Such schemes are desirable for hierarchical WSNs due to higher local connectivity and resilience.

Most of the dynamic and key pre-distribution algorithms focus on resilience against node compromise rather than on connectivity. Lower local connectivity is acceptable in most small-scale sensor networks because such networks can still achieve higher global connectivity through their neighbors. However, local connectivity is much more important in hierarchical sensor networks as parent-child connectivity is meaningless if nodes do not share a key. Therefore, key pre-distribution schemes such as [16] and [26] are more desirable in hierarchical networks. In terms of network lifetime and capacity what matters is the cluster and cluster tree performance. Hence, underlying key distribution scheme should have a minimum impact on network performance and should not considerably alter the cluster and cluster tree characteristics.

2.5 Collaborative Wireless Sensor Networks

In contrast to early sensor networks that were dedicated to a specific application, collaborative networks that perform different tasks and deployed in the same geographical region are emerging. e-SENSE [29] and U-City [62] are two such projects. These networks combine several heterogeneous sensor networks that are deployed in the same geographical region. Better resource efficiency can be achieved by allowing such multiple networks to collaborate and share resources with each other [40]. Certain sensing applications may also involve dynamically varying subset of sensor nodes [40] and/or users [29, 62].

Virtual Sensor Networks (VSNs) is an emerging concept that supports such collaborative, resource efficient, and multipurpose sensor networks that may involve dynamically varying subset of sensors and users [40]. VSNs are useful in three major classes of applications. Firstly, VSNs are useful in geographically distributed applications, e.g., monitoring rockslides and animal crossing within a mountainous terrain. Different types of devices that detect these phenomena can relay each other for data transfer without having to deploy separate networks (Figure 2.2). Secondly, VSNs are useful in logically separating multipurpose sensor networks, e.g., smart neighborhood systems with multifunctional sensor nodes. Thirdly, VSNs can be used to enhance the efficiency of systems that track dynamic phenomena such as subsurface chemical plumes that migrate, split, or merge [6]. Such networks may involve dynamically varying subsets of sensors, e.g., as a plume migrates nodes that monitors the plume changes.

A VSN can be formed by providing logical connectivity among these collaborative sensors. Nodes can be grouped into different VSNs based on the

phenomenon they track (e.g., rockslides vs. animal crossing) or the task they perform. VSNs are expected to provide the protocol support for formation, usage, adaptation, and maintenance of subset of sensors collaborating on a specific task(s). It is also important to handle the phenomenons that may migrate, merge, or split. VSNs should make efficient use of intermediate nodes, networks, or other VSNs to deliver messages across members of a VSN (Figure 2.2).

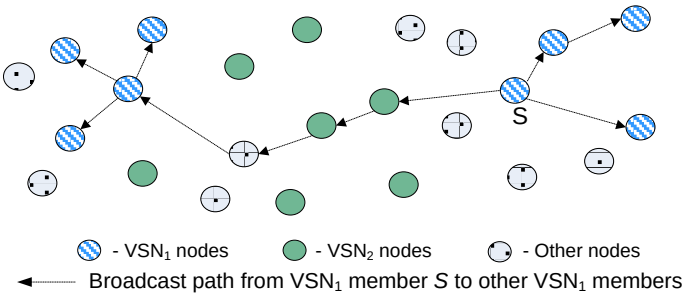


Figure 2.2 – Two geographically overlapped VSNs. Redrawn from [40].

This new concept opens up many new research directions. It is necessary to build algorithms and protocols that support the formation and maintenance of VSNs on resource constrained WSNs. Realization of VSNs require some structure within the sensor filed, many-to-many routing, and implementation of many VSN management functions. Formation of some structure within the sensor field can greatly simplify VSN formation, management, and communication. In contrast to some conventional WSNs that make use of many-to-one communication model (i.e., node-to-sink), VSNs require communication within and across VSNs (i.e., many-to-many, Figure 2.2). VSN management functions should be able to get new nodes into a VSN, remove nodes from a VSN, detect multiple VSNs, allow them to communicate with each other, etc.

2.6 Summary

It is clear from the literature survey that there is a gap between existing solutions and requirements of large and collaborative WSNs. Most of the cluster and cluster tree formation solutions try to balance the energy consumption of the network. However, majority of these solutions do not look into other important cluster and cluster tree characteristics such as minimum overlap, uniform clusters, cluster tree with lower depth, etc. Better connectivity, simplified routing, hierarchical addressing, detecting nodes performing similar tasks, and security are among the prime requirements of VSNs. The goal of the thesis is to come up with a set of solutions that makes VSN a reality.

Chapter 3

PROBLEM FORMULATION

Large and collaborative wireless sensor networks pose numerous challenges and provide many opportunities to come up with novel solutions. However, achieving all the desirable characteristics within a single algorithm/solution is not trivial in resource constrained WSNs. Therefore, we propose a compound solution. Before presenting the solution, it is necessary to define the scope of desirable characteristics of different solutions, the environment under it will operate, and the boundary of the problem that is being explored. Section 3.1 presents a detailed description of desirable characteristics of clusters, cluster tree, routing, and secure backbone formation. The network model is described in Section 3.2 and the problem statement is defined in Section 3.3.

3.1 Desirable Characteristics Of The Solution

Different solutions may desire different characteristics. A balanced approach that combines all the characteristics is necessary to achieve the overall goal of collaborative WSNs. Section 3.1.1 describes the characteristics of clusters and cluster trees. Characteristics of WSN routing and secure backbone design are described in Sections 3.1.2 and 3.1.3 respectively.

3.1.1 Desirable Characteristics Of Clusters And Cluster Trees

Several attributes make a specific cluster and cluster tree formation solution more appropriate for a given application. Such attributes include node connectivity, overlap among clusters, cluster size, overhead of forming/managing clusters and cluster tree, and latency.

Clusters and the cluster tree must ensure connectivity of all the nodes in the sensor field. Random deployment of nodes creates dense and sparse regions within the sensor field. Random node placement does not ensure connectivity of all the nodes, even if nodes are densely deployed. To ensure connectivity, a node must have sufficient transmission power to reach at least one of its neighbors. By allowing a node to tune its transmission power, connectivity of the network can be increased and overall energy consumption can be reduced. Though most probabilistic and completely distributed clustering solutions such as [9, 25, 33, 67] ensure that each node belongs to a cluster, these clusters may be isolated in a geographically large network. Therefore, a cluster formation solution should ensure some bounds on the distance between CHs.

It is important to cover a given sensor field with minimum number of clusters. “Hexagonal clusters have the highest coverage area and can maintain coverage with the least number of clusters” [64]. Overlapping clusters add redundancy [64] and increase the intra-cluster signal contention [25]. A node may belong only to a single cluster yet this decision is application dependent. It is also important to reduce the overlap among clusters. Such none overlapping hexagonal or circular clusters allow better load balancing within clusters, guaranteed upper bound on the number of clusters and depth of the cluster tree, and generate a predictable network topology [25]. Having predictable

topology, even on a randomly deployed network, facilitates intelligent routing solutions without being tied to the cluster tree. Most of the simple antennas in sensor motes are omnidirectional, hence it makes sense to place the CH in the middle of the cluster that allows maximum special coverage. “Aggregation is more useful when the CH is in the middle of the cluster and capable of receiving readings from all the directions” [25]. Reduced number of clusters tends to reduce the breadth and/or depth of the cluster tree. Therefore, it is important for a given clustering solution to form clusters with minimum or no overlap.

Selection of best set of CHs is the key for achieving these desirable characteristics. However, selecting such a set of CHs is not trivial if nodes do not convey any location information. Figure 3.1 illustrates three ways that child CHs can be selected in top-down clustering. Clusters overlap if next child CH is selected from nodes that are within one-hop from the parent CH (Figure 3.1(a)). This scheme is similar to the IEEE 802.15.4 cluster tree [38]. A better choice would be to propagate the new CH selection message beyond the parent cluster through an intermediate node (*X*) and then select the child CH from a node that is 2-hops away (Figure 3.1(b)). The intermediate node *X*, will be in the region of both parent and child clusters. Clusters still overlap because *X* can only belong to either parent or child clusters. This overlap can be ignored, as it is small. However, due to random node placement, it may not be possible to find a node *X* that is at the edge of the parent cluster. Overlapped region will increase if *X* is closer to the parent CH. Another alternative is to forward the cluster formation message by 3-hops through two intermediate nodes *X* and *Y* (Figure 3.1(c)). Though parent and child clusters do not overlap anymore, an open region is created. If there are any nodes within the open

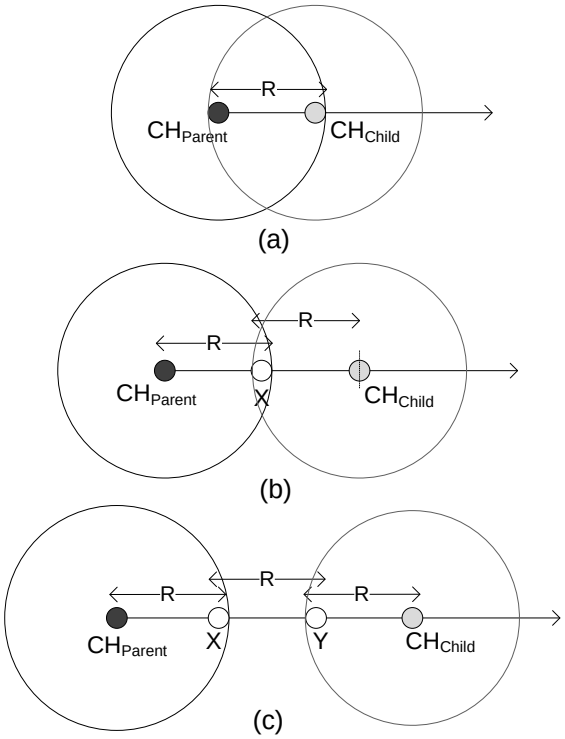


Figure 3.1 – Different ways of selecting next child cluster head in top-down clustering. *R* – Transmission range of a node.

region, those need to be covered by another cluster. The open region either expands or shrinks depending on the actual positions of nodes *X* and *Y*. Hence to minimize overlaps and open regions, the distance (*d*) between parent and child CHs should be selected such that, *2-hops < d < 3-hops*. Ideally, it needs to be just above 2-hops, as it uniformly covers the sensor filed. This sort of a CH selection is possible only with top-down approach.

It is also important that workload of each cluster is balanced. Having similar number of cluster members balance the workload within clusters. Circular or hexagonal clusters ensure uniform cluster size. However, the optimum cluster size is application dependent. Small clusters can be formed by having a lower communication range and connecting only one-hop neighbors. Multi-hop clusters or use of high transmission power

can form larger clusters. Though multi-hop clusters are less attractive, those may be required in certain applications [9].

Cluster formation overhead of localized decision based distributed clustering solutions such as [9, 12, 25, 33, 67] are lower compared to centralized solutions such as LEACH-C [33]. However, lack of global knowledge about the network hinders the ability of these distributed solutions to form optimum set of clusters (i.e., maximum spatial coverage with least number of clusters). Though centralized solutions form better clusters, their overhead is significant. A hybrid approach that combines a node's and its neighbors' information is a better compromise. Such a scheme can produce better clusters with a lower overhead [17, 25]. Therefore, a good solution should form spatially distributed set of clusters with minimum overhead.

Cluster trees are useful in delivering unicast, multicast, broadcast traffic, for data fusion, and for in-network query processing. Performance of such upper layer functions depend on the number of hops between a node and the base station. As the number of hops increases both the latency and energy to forward a message increases. For many latency bound applications such as earthquake monitoring and surveillance [69], it is desirable to have a cluster tree with lower depth. Alternatively, a habitat [58] or microclimate monitoring application may favor a long tree as far as it can perform more aggregation and compression within the network. The location of the root node is another parameter that affects the breadth and depth of the tree. If the root node is placed in the middle of the sensor filed cluster tree can span into all directions. This produces a cluster tree with a lower depth and higher breadth. If it is placed at the edge of the sensor filed, it will not be able to span into all directions therefore a cluster tree with lower breadth and

higher depth will be formed. Collaborative WSNs may require formation of multiple cluster trees within the same sensor field. Another parameter that governs the shape of the cluster tree is the number of child CHs generated by a parent CH. Number of child CHs reflect the branching factor of the cluster tree. Number of clusters required to cover a given sensor field is somewhat constant hence higher branching factor reduce the depth of the cluster tree. However, the branching factor cannot be arbitrary increased.

Cluster heads consume more energy than their cluster members. Therefore, nodes playing the role of a CH may be changed from time to time to distribute the workload. Whenever a CH is changed, the cluster tree needs to be updated. Clusters and the cluster tree should also be capable of handing node dynamics (new, moving, or deteriorating nodes) particularly in sensor/actor [2] and collaborative WSNs. Frequent re-clustering and tree formation is not desirable due to the high overhead therefore clusters and the cluster tree should be capable of tolerating certain network changes.

Achieving all these properties within a single cluster and cluster tree formation algorithm is not trivial. Hence, it is important to identify and achieve at least the key characteristics for a given application scenario.

3.1.2 Desirable Characteristics Of Routing Protocols

Routing protocols for WSNs need to be energy efficient and should maximize the lifetime of the network. Though node-to-sink (many-to-one) communication pattern dominates in conventional WSNs, collaborative sensor networks may require node-to-sink, node-to-node, or VSN-to-VSN communication. Some sort of a addressing scheme is required to facilitate such a communication model. These addresses need to be shorter.

Shorter addresses reduce the size of a packet header hence reduce energy consumption. Addresses need to be assigned on the fly as clusters/VSNs are formed and the process should incur no/minimum overhead.

Successful delivery of messages is also an important property in most WSNs. Though routing protocols such as Rumor Routing [14], Ant Routing [35], and Directed Diffusion [39] enhance the network lifetime, probability of successful message delivery is much lower. This is not desirable in mission critical or dynamic applications and realization of VSNs requires reliable delivery of events and control messages.

Forwarding through multiple nodes reduces the energy consumption and increase the network lifetime. However, multi-hop forwarding is not desirable in large networks due to latency, packet loss, and energy consumption. Hierarchy based routing overcomes some of these issues. Nodes closer to the base station have to deliver more traffic therefore tend to run out of energy much faster. Hence hierarchy based approaches does not balance the energy consumption of nodes. Nevertheless, this may be the only solution for networks that depend on many-to-one communication model, where top of the hierarchy is the base station. The networks that make use of many-to-many communication model may be able to figure out better paths than being tied to the hierarchy. Therefore, a good hierarchical routing solution should maximize the network lifetime by exploring these alternative paths.

3.1.3 Desirable Properties Of Secure Backbones

A Secure backbone is required to facilitate secure upper layer functions, dynamic key distribution, and re-keying in collaborative and mission critical WSNs. Such a

backbone can be built using the cluster tree. Each parent and child CH pair needs to share a separate cryptographic key between them. Though dynamic key assignment between parent and child CHs seems to be attractive, it is neither secure nor energy efficient. Key pre-distribution is a better alternative that is secure and efficient. However, if two nodes do not share a common key, their cluster membership is meaningless and this can significantly affect cluster and cluster tree characteristics. Therefore, a key pre-distribution scheme should ensure better connectivity. Formation of a secure backbone needs to have minimum impact on the network and the cluster and cluster tree formation algorithm should retain most of its desirable properties.

3.2 Network Model

Following properties are assumed about the network and sensor nodes, which are common in most of the WSN research problems:

a) The sensor network is expected to be geographically large and consisting of thousands of sensors.
b) Nodes are randomly placed on a $L \times W$ grid with a given probability p.
c) All nodes are static and location unaware.
d) Nodes are homogeneous, have a fixed transmission power and equally significant.
e) No time synchronization or prior-network topology awareness is assumed.
f) Root node is in the middle of the sensor field.
g) The circular communication model is used for signal propagation.
h) Unless otherwise stated, the free space propagation model is used and no noise is assumed.
i) Single-hop communication model is assumed for both inter-cluster and intra-cluster communication, unless otherwise stated.

3.3 Problem Statement

Future large-scale and collaborative WSNs will require some structure within the network to achieve the application objectives effectively. Therefore, the first task is to identify a cluster and cluster tree formation algorithm that is configurable and scales well for large networks under aforementioned assumptions. The algorithm should be capable of achieving most of the desirable cluster and cluster tree characteristics while being customized to a specific application. To make the algorithm controllable, the controllability of top-down approach should be exploited. Ability to form a secure backbone under pre-distributed keys would be an added advantage. Collaborative WSNs should have the ability to communicate with the base station as well as within the network. Building a routing scheme on top of the cluster tree that can facilitate these requirements is the second task. An addressing scheme has to be developed to facilitate communication within the network. Routing scheme should explore alternative paths between two nodes that wish to communicate without being tied to the cluster tree. All these tasks should lead to the main goal of enabling VSNs. The overall solution should provide protocol support for formation, usage, adaptation, and maintenance of VSNs.

Chapter 4

CLUSTER AND CLUSTER TREE FORMATION

The Generic Top-down Cluster and cluster tree (GTC) formation algorithm, a configurable algorithm that is capable of achieving most of the desirable cluster and cluster tree characteristics is presented. Configurable parameters in the algorithm allow selection of different characteristics, e.g., more uniform and circular clusters, cluster trees with control breadth and depth, etc. Simple Hierarchical Clustering (SHC), a special case of GTC, is similar to the IEEE 802.15.4 cluster tree. Another special case, Hop-ahead Hierarchical Clustering (HHC) produces significantly better clustering solutions.

Section 4.1 presents the GTC algorithm. Section 4.2 describes how desirable cluster and cluster tree properties can be achieved by controlling parameters of the algorithm. Message complexity analysis of the algorithm is presented in Section 4.3. Finally, performance analysis is presented in Section 4.4.

4.1 Generic Top-Down Cluster And Cluster Tree Formation Algorithm

The GTC algorithm is shown in Figure 4.1. The *root node* initiates the cluster formation by executing the ***Form_Cluster*** function. The root node can be one of the sensor nodes or it can be a resourceful base station. All other nodes execute the ***Join_Cluster*** function and listen for a cluster formation broadcast. Root node sends a

```
Form_Cluster(NID_CH, CID_CH, delay, n_CCHs, hops_max, TTL_max, depth)
 1  Wait(delay)
 2  TTL ← TTL_max
 3  Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
 4  ack_list ← Receive_ACK(NID_child, hops, P_1, P_2, timeout_ACK)
 5  IF(ack_list = NULL)
 6      Join_Cluster()
 7  FOR i = 1 TO n_CCHs
 8      CCH_i ← Select_Candidate_CHs(ack_list)
 9      CID_i ← Select_Next_CID(i)
10      delay_i ← Select_Delay(i)
11      depth_i ← depth + 1
12      Request_Form_Cluster(CCH_i, CID_i, delay_i, n_CCHs, hops_max, TTL_max, depth_i)

Join_cluster()
13  Listen_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
14  TTL ← TTL – 1
15  hops ← TTL_max - TTL
16  IF(hops ≤ hops_max AND my_CID = 0)
17      my_CID ← CID_CH
18      my_CH ← NID_CH
19      my_depth ← depth + 1
20      Send_ACK(my_NID, hops, P_1, P_2)
21  IF(TTL > 0)
22      Forward_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
23      IF(hops ≤ hops_max)
24          Exit()
25  ELSE
26      Send_ACK(my_NID, hops, P_1, P_2)
27      IF(Listen_Form_Cluster(CCH, CID, delay, n_CCHs, hops_max, TTL_max, depth,
    timeout_CCH) = TRUE)
28          Form_Cluster(my_NID, CID, delay, n_CCHs, hops_max, TTL_max, depth)
29          Exit()
30  Join_cluster()
```

Figure 4.1 – Generic top-down cluster and cluster tree formation algorithm.

cluster formation broadcast (***Broadcast_Cluster***) indicating its node ID (NID_{CH}), Cluster ID (CID_{CH}), maximum hops to a cluster member from the CH ($hops_{max}$), number of hops to forward the broadcast (TTL_{max}), and its *depth* in the cluster tree. A node hearing this broadcast will join the cluster if it is not already a member of another cluster (*my_CID = 0*) and within $hops_{max}$. Each node keeps track of the neighbor that sends or forwards the

broadcast. When a node joins the cluster, it sets its cluster parameters such as node ID, cluster ID, and its depth in the cluster tree (lines 17-19 of Figure 4.1). An acknowledgment (ACK) is then send to the corresponding CH (line 20) indicating its own node ID (*my_NID*), its distance to the CH (*hops*), and set of properties of the node (p_1, p_2) such as the residual energy and node degree. The CH receiving this ACK adds the node to its acknowledged list (*ack_list*). After sending the ACK, the child node forwards the cluster formation broadcast using ***Forward_Broadcast_Cluster*** function, given that the *TTL* has not expired (*TTL > 0*).

Nodes that are not within $hops_{max}$ do not join the cluster, instead forward the broadcast, if *TTL* is still valid. These intermediate nodes do not need to send any ACKs, therefore reduce the overhead of the algorithm. A broadcast from a particular CH is forwarded only once by a receiving node. This ensures that broadcasts are forwarded outward from the CH. If *TTL* is expires after receiving a broadcast (*TTL = 0*), the node that received the broadcast is capable of being selected as a new CH. Such a node is called a Candidate Cluster Head (CCH). At this stage, the node is either at the edge of the cluster (if $TTL = hops_{max}$) or outside the cluster (if $TTL > hops_{max}$). When a node determines that it is going to be a CCH, it indicates that to the corresponding CH by sending an ACK (line 26). The node then waits for a cluster formation request (***Listen_Cluster_Formation***) from the corresponding CH. If such a request does not arrive before the timeout ($timeout_{CCH}$) node reruns the ***Join_Cluster*** function.

In the mean time, the corresponding CH keeps receiving acknowledgments until the ***Receive_ACK*** function timeouts ($timeout_{ACK}$). After the timeout, the ***Select_Candidate_CHs*** function then selects n_{CCHs} nodes as CCHs from the *ack_list*.

Finally, a request (***Request_Form_Cluster***) is send to each CCH asking them to form their own clusters. A new cluster ID (CID_i), a hold up time ($delay_i$) before forming its own clusters, and other relevant parameters are send to each selected CCH. Upon receiving the request, those selected CCHs form their own clusters by executing the ***Form_Cluster*** function. If a CCH is not able to attach any child nodes, the related branch in the cluster tree is not expanded to the next level. The algorithm continues until all the possible branches are expanded. The cluster tree is rooted at the root node and formed by each CH keeping track of its own parent and child CHs.

4.2 Achieving Desirable Characteristics

The solution generated by the algorithm depends on the implementation of functions such as ***Select_Next_CID***, ***Select_Delay***, and selection of parameters such as $hops_{max}$, TTL_{max}, $delay_i$, and n_{CCHs}. By controlling these parameters and implementation of functions, a wide range of solutions can be obtained.

The coverage of a cluster is determined by $hops_{max}$. Multi-hop clusters can be formed if $hops_{max} > 1$. The distance between parent and child CHs can be controlled by changing TTL_{max}. If $TTL_{max} = hops_{max}$, any cluster member can be selected as a CCH. Single-hop cluster formation under this condition is similar to Figure 3.1(a). In an optimum case, those CCHs need to be selected from nodes that are at the edge of the parent cluster. We name this approach as Simple Hierarchical Clustering (SHC). Figure 4.2 illustrates a conceptual case where the sensor field is optimally covered by selecting three CCHs at each level. To make the diagram simple, only CCHs are indicated and one branch is expanded into several levels. It is sufficient to select three CCHs, if those are

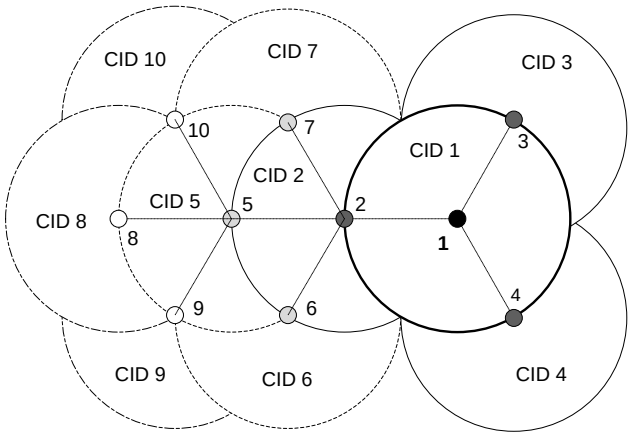


Figure 4.2 – Physical shape of ideal SHC clusters. Scattered lines indicate the parent-child relationship among CHs. $hops_{max} = TTL_{max} = 1$.

separated physically as widely as possible. Let n_i denote node i and c_i denote cluster i. The root node (n_1) forms the cluster c_1 by connecting all one-hop neighbors. Then it request three of its neighbors (n_2, n_3, and n_4), which are at the edge of the cluster to form their own clusters. These clusters are called *level 1* clusters as they belongs to the first level of the cluster tree. The root node belongs to *level 0*. Then n_2 request three of its neighbors (n_5, n_6, and n_7) to form clusters c_5, c_6, and c_7. Even in this conceptual case, the shapes of clusters are not circular except for the first cluster. Note that c_6 and c_7 are smaller than c_5. Clusters c_9 and c_{10} are even smaller. Therefore, as the depth of the tree increases many overlapping child clusters are formed which reduces the effective cluster size. Lower circularity also increases the number of clusters that are required to cover the entire sensor field. This approach is similar to the IEEE 802.15.4 cluster tree [38]. For single-hop SHC, line 26 in Figure 4.1 is redundant, hence needs to be removed.

Another alternative would be to select TTL_{max} such that $TTL_{max} > hops_{max}$. This allows nodes that are several hops away from the edge of the parent cluster to be selected as CCHs. Nodes within $hops_{max}$ join the cluster while other nodes keep forwarding the

cluster formation broadcast until *TTL* expires. A clustering solution similar to Figure 3.1(b) can be achieved by selecting TTL_{max} such that:

$$TTL_{max} = 2 \times hops_{max} \tag{4.1}$$

As discussed in Section 3.1.1 these clusters somewhat overlap with the parent CH. However, the overlapped region is much smaller compared to the SHC. The best option is the 3-hop forwarding where TTL_{max} is selected such that:

$$TTL_{max} = 2 \times hops_{max} + 1 \tag{4.2}$$

We name this multi-hop forwarding approach as Hop-ahead Hierarchical Clustering (HHC). Figure 4.3 illustrates how the sensor field can be optimally covered with single-hop HHC. The root node (n_1) sends a cluster formation broadcast and all the one-hop neighbors join the cluster (c_1). The broadcast is then forwarded to the next set of neighbors and the process continues until *TTL* expires. For example, after hearing the broadcast from n_1, n_2 joins the cluster. The broadcast is then forwarded to n_3 and from n_3 to n_4. Finally, n_4 is 3-hops away from the CH and is a candidate to be selected as a new CH. The root node needs to select several such nodes as CHs of new clusters. In Figure

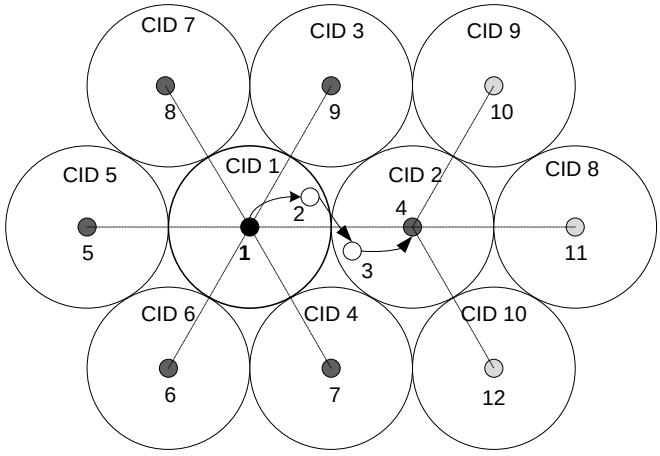


Figure 4.3 – Physical shape of ideal HHC clusters. Scattered lines indicate the parent-child relationship among CHs. $hops_{max} = 1$, $TTL_{max} = 3$.

4.3, that root node has selected six nodes (n_4 to n_9) as child CHs that are in different directions of the sensor field. For all the other levels it is sufficient that each parent CH selects up to three nodes as child CHs, e.g., only c_8, c_9, and c_{10} are formed by c_4. The HHC forms larger clusters, more circular clusters, and has a better distribution of CHs.

At each CH, several nodes (n_{CCH}) are chosen as CCHs of the next level using the ***Select_Candidate_CHs*** function. Those CCHs are selected from nodes in the *ack_list* that are furthest away from the CH ($hops = TTL_{max}$). The implementation of the ***Select_Candidate_CHs*** function depends on the availability of certain data such as residual energy of a node, node degree, location information, or cryptographic key identifiers. When a node sends an ACK to the CH such data is send using parameters p_1 and p_2. Optimum set of CCHs can be selected if node location information is available otherwise, nodes can be selected randomly. However, it is possible to select physically nearby nodes when CCHs are randomly selected. Therefore, some of the selected CCHs will not be able to form their own clusters. This affects the breadth and depth of the cluster tree. To build a cluster tree with higher breadth and lower depth, n_{CCH} needs to be somewhat higher. Therefore, the selection of n_{CCH} should be application dependent. Better load balancing can be achieved by selecting CCHs based on higher residual energy or lower node degree [67]. Dense clusters can be built by selecting nodes with higher node degree. If the cluster setup phase is cycled like in [33, 67] these parameters pay a key role in selecting different CHs at different rounds.

The ***Select_Next_CID*** function assigns cluster IDs to newly formed clusters. The function can be implemented only at the root node or at each and every parent CH. In the former case, each parent CHs has to request a set of CIDs for all the selected CCH. This

approach generates significant number of control messages and does not scale up. The overhead can be reduced by delegating the task to respective parent CHs. A parent CH can assign CIDs based on either NID of the CCH (if NIDs are unique) or hierarchically. In hierarchical ID assignment, parent CH can derive the child's CID based on its own hierarchical CID and the child's branch number in the cluster tree. Such hierarchical addresses are useful in hierarchical routing. Design of the hierarchical addressing scheme is described in Section 6.1.1.

Many collisions occur when multiple CCHs try to form clusters at the same time. Due to these collisions, certain nodes may not hear a cluster formation broadcast from any of the CCHs. This reduces the network lifetime and node connectivity. Cluster formation broadcasts from different CCHs are time multiplexed to reduce collisions. The *Wait* function delays the cluster formation broadcast generated by a CCH. The $delay_i$ for each CCH is determined by the ***Select_Delay*** function. Furthermore, different $delay_i$ values can be used to control the shape of the cluster tree. By assigning appropriate delays, the algorithm can be used for breadth-first, depth-first, or hybrid cluster tree formation.

For breadth-first tree formation, delay should ensure that cluster formation of level i completes before the start of level $i + 1$. Figure 4.4 illustrates starting time of different clusters at different levels of the cluster tree based on breadth first tree formation. Let c_i be the cluster number, t_{CH} be the time that takes to form a cluster, t_{CCH} be the delay between two CCHs of the same cluster, and n be the number of CCHs. Assume that cluster c_0 starts forming its cluster at time $t = 0$. It will complete its cluster formation by $t = t_{CH}$. After c_0 completes the cluster formation, its first child (c_1) can start

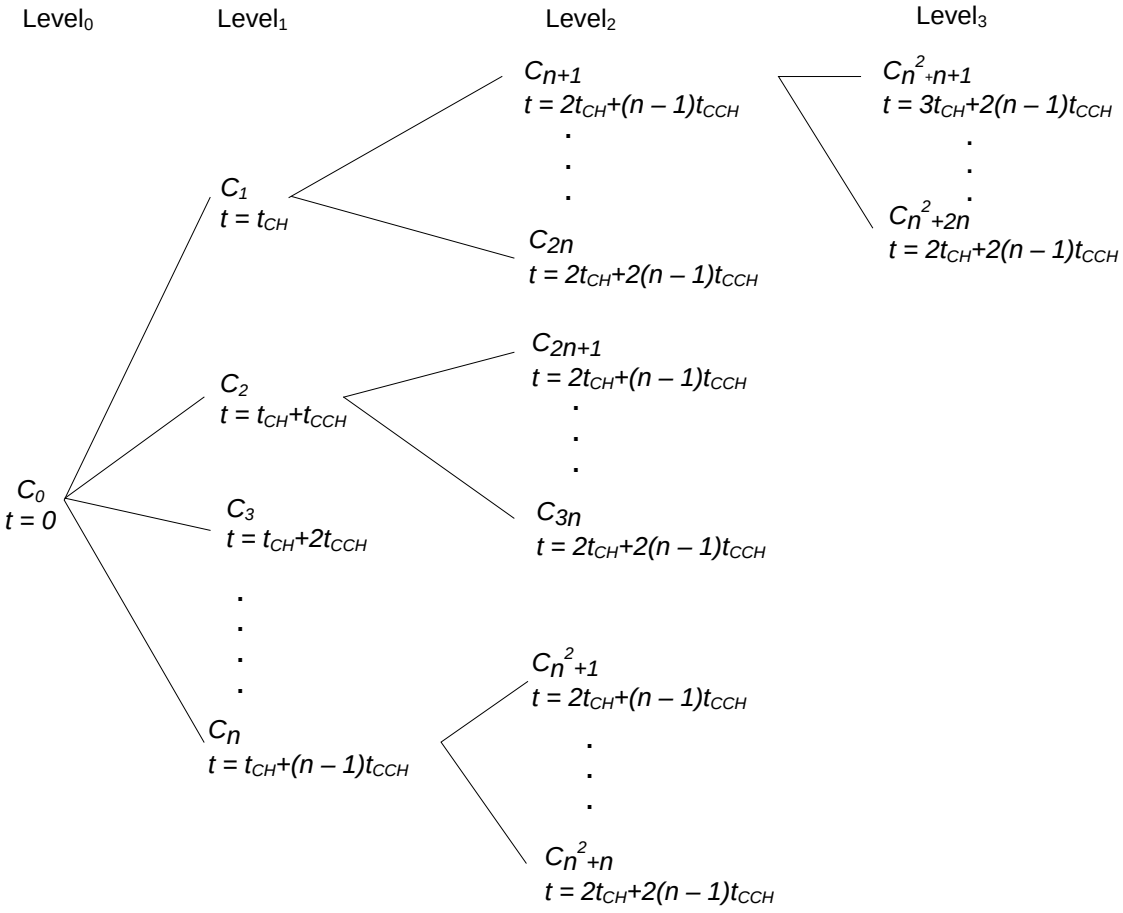


Figure 4.4 – Breadth-first cluster tree formation – starting time of different clusters. c_i – cluster number, t_{CH} – time to form a cluster, t_{CCH} – delay between two CCHs of the same cluster, n – number of child CHs.

the cluster formation. Hence the starting time of c_1 is $t = t_{CH}$. Then the second child can initiate its cluster formation after t_{CCH}, hence $t = t_{CH} + t_{CCH}$. Similarly n-th child cluster (c_n) starts at $t = t_{CH} + (n - 1)t_{CCH}$. Level *2* clusters can only initiate when all the level *1* clusters are completely formed. Therefore, starting time of the first level *2* cluster (c_{n+1}) is $t = t_{CH} + t_{CH} + (n - 1)t_{CCH} = 2t_{CH} + (n - 1)t_{CCH}$. At the same time, all the other child clusters of c_0 can request their first child cluster to initiate the cluster formation. It is unlikely that these clusters will bump into each other causing collisions as they are geographically distributed. This approach reduces the time that it takes to form all the

clusters. The *n*-th child clusters of $c_1, c_2, c_3, \ldots, c_n$ can start their cluster formation at $t = 2t_{CH} + (n-1)t_{CCH} + (n-1)t_{CCH} = 2t_{CH} + 2(n-1)t_{CCH}$. Similarly level *3* clusters can start the cluster formation when all the *n*-th clusters of level *2* complete their cluster formation. Hence their stating time is $t = 3t_{CH} + 2(n-1)t_{CCH}$. Based on Figure 4.4 the starting time of level *i* can be derived as:

$$t_i = dt_{CH} + (d-1)(n_{CCHs} - 1)t_{CCH} \quad (4.3)$$

where *d* is depth of level *i* (i.e., $d = i$), t_{CH} is the time that takes to form a cluster, t_{CCH} is the delay between two CCHs of the same cluster, and n_{CCHs} is the number of CCHs (i.e., branching factor of the tree). The delay between two CCHs of the same cluster should be selected such that $t_{CCH} \geq t_{CH}$.

For depth-first tree formation, delay should allow a branch to complete its expansion before start of another branch. This approach takes more time and forms a deeper cluster tree. The ***Select_Delay*** function decides on a suitable $delay_i$ based on the desired shape of the cluster tree and informs the CCHs using the ***Request_Form_Cluster*** function. In HHC, if a CCH hears a cluster formation broadcast from another neighboring CH, while it is delaying its cluster formation, it joins the new cluster and does not form its own cluster. This further reduces the overlapping clusters. Note that this is just a delay, not an exact time; therefore the algorithm does not require synchronization of clocks among sensor nodes.

Alternatively, the ***Wait*** function can be replaced with a ***Delay*** function. The ***Delay*** function will be part of the ***Form_Cluster*** function and it waits some time ($delay_i$) before sending the ***Request_Form_Cluster*** message to a selected CCH. When such a message is send, the CCH immediately forms its own cluster (no $delay_i$ will be send to the CCH in

this case). This approach provides more control over the previous one. Based on the information provided by the child clusters that are already formed the parent CH can dynamically decide on a suitable CCH. For this scheme to work, each new child CH has to inform the parent CH about the newly acquired cluster members. This prevents the issue of a CCH being overtaken by another cluster and provides better spatial coverage. However, extra control messages add some overhead, hence this function is not implemented.

The tree that defines the node-to-node, node-to-CH, and CH-to-CH connectivity is called the *physical tree*. Cluster tree that combines only the CHs is called the *logical tree* and it defines the CH-to-CH connectivity. WSNs that use the same power level for both intra-cluster and inter-cluster communication need to rely on *gateway nodes* [46] to forward messages from one CH to another. In such cases, the physical tree defines node-to-CH, CH-to-gateway, and gateway-to-gateway connectivity while cluster tree defines only the CH-to-CH connectivity. Networks that use the same power level for both inter-cluster and intra-cluster communication have different physical and logical trees. In certain cases [9, 33, 67] nodes may use high power for inter-cluster communication. In such networks physical and the logical tree are identical because no gateway nodes are involved. Though use of high power reduces latency, it increases the energy consumption. Use of same or different power levels for inter-cluster communication is independent of the GTC algorithm. In GTC, the maximum distance between any parent CH and its child CHs is $R \times TTL_{max}$, where R is the transmission range. Therefore, GTC has a bounded CH-to-CH distance compared to [9, 33, 67]. Though the process is slower, it is acceptable in long loved WSNs where network lifetime ranges from days to year.

4.3 Message Complexity Of The Algorithm

We analyze the message complexity of the single-hop cluster formation. Let us assume that *n* sensor nodes are distributed in an area *A* with a uniform node density λ, where $\lambda = n/A$.

In single-hop SHC, each CH broadcasts one cluster formation message and all the nodes within the communication range (*R*) join the cluster. The broadcast will not be forward any further. Then each child node sends an acknowledgement (ACK) back to the CH. As there are altogether $\lambda\pi R^2$*-1* nodes in the circular region, other than the CH, $\lambda\pi R^2$*-1* ACKs will be generated. Then the CH sends three more messages asking three of those nodes to form their own clusters. Total number of control messages per cluster can be calculated as follows:

$$\begin{aligned} &\textit{Number of broadcasts} && = 1 \\ &\textit{Number of ACKs} && = \lambda\pi R^2 - 1 \\ &\textit{Number of cluster formation requests} && = 3 \\ &\therefore \textit{Total number of control messages per cluster} && = 1 + \lambda\pi R^2 - 1 + 3 \\ & && = 3 + \lambda\pi R^2 \end{aligned} \quad (4.4)$$

To cover the sensor filed $A/k\pi R^2$ clusters are required; *k* is a factor that defines the level of overlapp among clusters, $0 < k \leq 1$.

$$\therefore \textit{Total number of control messages} = (3 + \lambda\pi R^2) \times \frac{A}{k\pi R^2} \quad (4.5)$$

Replacing λ with *n/A* in Equation 4.5:

$$\textit{Total number of control messages} = \frac{n}{k} + \frac{3A}{k\pi R^2} \quad (4.6)$$

Therefore, the message complexity is *O(n)*.

In HHC, each CH broadcasts one cluster formation message and all the 1-hop neighbors join the cluster. The broadcast is then forward by two more hops, first to 2-hop

neighbors and then from 2-hop to 3-hop neighbors. Total number of broadcasts can be calculated as follows:

$$\begin{aligned}
&\textit{Number of broadcasts by root node} && = 1\\
&\textit{Number of broadcasts by 1-hop neighbors} && = \lambda\pi R^2 - 1\\
&\textit{Number of broadcasts by 2-hop neighbors} && = \lambda\pi(2R)^2 - \lambda\pi R^2\\
&\therefore \textit{Total number of broadcasts per cluster} && = 1 + \lambda\pi R^2 - 1 + \lambda\pi(2R)^2 - \lambda\pi R^2\\
& && = 4\lambda\pi R^2 \qquad (4.7)
\end{aligned}$$

Node within 1-hop and 3-hops will send ACKs back to the CH. Assume that 2-hop and 1-hop neighbors do not aggregate these ACKs therefore ACKs are forwarded as individual messages. 2-hop nodes do not need to send any ACKs. Therefore:

$$\begin{aligned}
&\textit{Number of ACKs by 1-hop neighbors} && = \lambda\pi R^2 - 1\\
&\textit{Number of ACKs by 3-hop neighbors} && = \lambda\pi(3R)^2 - \lambda(2R)^2\\
& && = 5\lambda\pi 3R\\
&\textit{Number of hops that 3-hop ACKs get forwarded} && = 2\\
&\therefore \textit{Total number of ACKs by 3-hop neighbors} && = (1 + 2)5\lambda\pi R\\
&\therefore \textit{Total number of ACKs per cluster} && = 15\lambda\pi R + \lambda\pi R^2 - 1\\
& && = 16\lambda\pi R - 1 \qquad (4.8)\\
&\textit{Number of cluster formation requests} && = 6\\
&\textit{Number of hops cluster formation requests travel} && = 3\\
&\textit{Total number of cluster formation requests} && = 6 \times 3\\
&\therefore \textit{Total number of control messages per cluster} && = 4\lambda\pi R^2 + 16\lambda\pi R - 1 + 18\\
& && = 17 + 20\lambda\pi R \qquad (4.9)\\
&\therefore \textit{Total number of control messages} && = (17 + 20\lambda\pi R^2) \times \frac{A}{k\pi R^2}
\end{aligned}$$

Replacing λ with n/A in Equation 4.9:

$$\textit{Total number of control message} = \frac{17A}{k\pi R^2} + \frac{20n}{k} \qquad (4.10)$$

Still the message complexity is $O(n)$. Therefore, the message complexity of both SHC and HHC is proportional to the number of nodes in the network. It can also be seen that number of control messages are proportional to the area of the sensor field. Some of the

ACKs in HHC can be reduced by piggybacking ACKs before being forwarded by the 2-hop and 1-hop neighbors, however complexity is still *O(n)*.

4.4 Performance Analysis

The characteristics of clusters and the cluster tree are evaluated using simulations. *n* nodes are randomly placed on a *101×101* square grid with a given probability *p* (e.g., 0.25, 0.5, 0.75, and 1.0). Grid spacing is 10 units. Simulation results are based on 150 sample runs and over 95% confidence level is observed for most of the parameter combinations. CCHs are randomly selected. The number of CCHs are selected such that; $n_{CCHs} = 3$ for SHC and $n_{CCHs} = 6$ for HHC. Except where noted, the simulation results presented use 5000 nodes and the cluster tree is formed using the breadth-first tree formation approach. The root node is placed in the middle of the sensor field. Cluster characteristics are compared with FLOC [25] and PHC [9]. Specific implementation details of the simulator are presented in Appendix A.

4.4.1 Metrics

Following metrics are used to analyze cluster and cluster tree characteristics.

Circularity

Measure how circular a given clusters is. It reflects the ratio between the actual number of nodes that are in the cluster and total number of nodes that are within the communication range of the CH.

$$Circularity = \frac{1}{m} \sum_{i=1}^{m} \frac{no\ of\ \ nodes\ in\ cluster\ i}{no\ of\ \ nodes\ in\ the\ range\ of\ \ CH_i} \times 100 \qquad (4.11)$$

where *m* is the number of clusters in the network. If a cluster can attract all the neighbors in a single-hop or multi-hop neighborhood, its circularity is 100%. In multi-hop clusters there can be nodes that does not belong to the cluster though they are in the range of the cluster. This occurs when there are no intermediate nodes to forward the cluster formation broadcast. For example, node *X* in Figure 4.5 is not in the range of any node that can forward the cluster formation message. Therefore, such nodes are not considered to be in the range of the CH. The ratio (*L/A*) between circumference (*L*) and area (*A*) of a cluster is an alternative circularity metric. A given cluster is circular if the ratio is closer to *2/R*. However, this measure has two issues. Firstly, circular clusters cannot be formed at the border of the sensor field. In this case the measure of *L/A* will reflect these clusters as non-circular. However, in reality these clusters have attracted the maximum number of nodes that they can attract. Equation 4.11 takes this into account and assigns 100% to such clusters, allowing us to discard boarder effect. Secondly, certain clustering schemes allows CH of a different cluster to reside inside another cluster, in such cases *L/A*

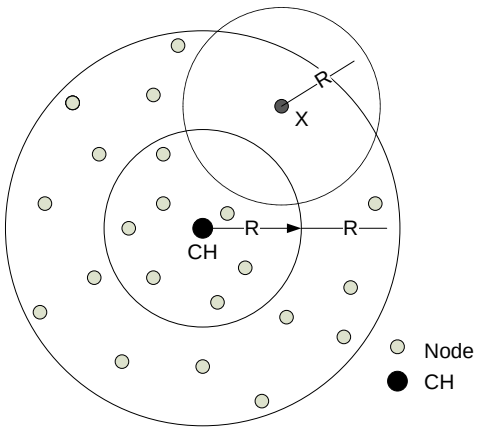


Figure 4.5 – A node that is disconnected in a 2-hop cluster. *R* – Transmission range of a node.

measure is inappropriate. Based on Equation 4.11 circularity of a hexagonal cluster can be derived as flows:

$$\begin{aligned} \textit{Circularity of a hexagonal cluster} &= \frac{\textit{area of a hexagon} \times \lambda}{\textit{area of a circle} \times \lambda} \times 100 \\ &= \frac{6 \times \frac{\sqrt{3}R^2}{4} \lambda}{\lambda \pi R^2} \times 100 \\ &= \frac{3\sqrt{3}}{2\pi} \times 100 \\ &= 82.69\% \end{aligned}$$

where λ is the node density and R is the transmission range.

Number of Clusters

Is the total number of clusters that are required to cover a given sensor field. This is equivalent to number of CHs.

Clusters Size

Is the number of nodes in a cluster including the CH.

Node/CH depth

It is the depth of a node/CH in either the logical (i.e., cluster tree) or the physical tree. Depth of the root node is zero. Depth of a child node that is i hops away from its CH is $depth_{CH} + i$.

4.4.2 Cluster Characteristics

4.4.2.1 Single-hop Clusters

Figure 4.6 shows the physical shape of clusters formed by SHC and HHC schemes. In SHC (Figure 4.6(a)), it can be seen that only the first cluster has approximately a circular shape while the shape of other clusters vary widely. It illustrates

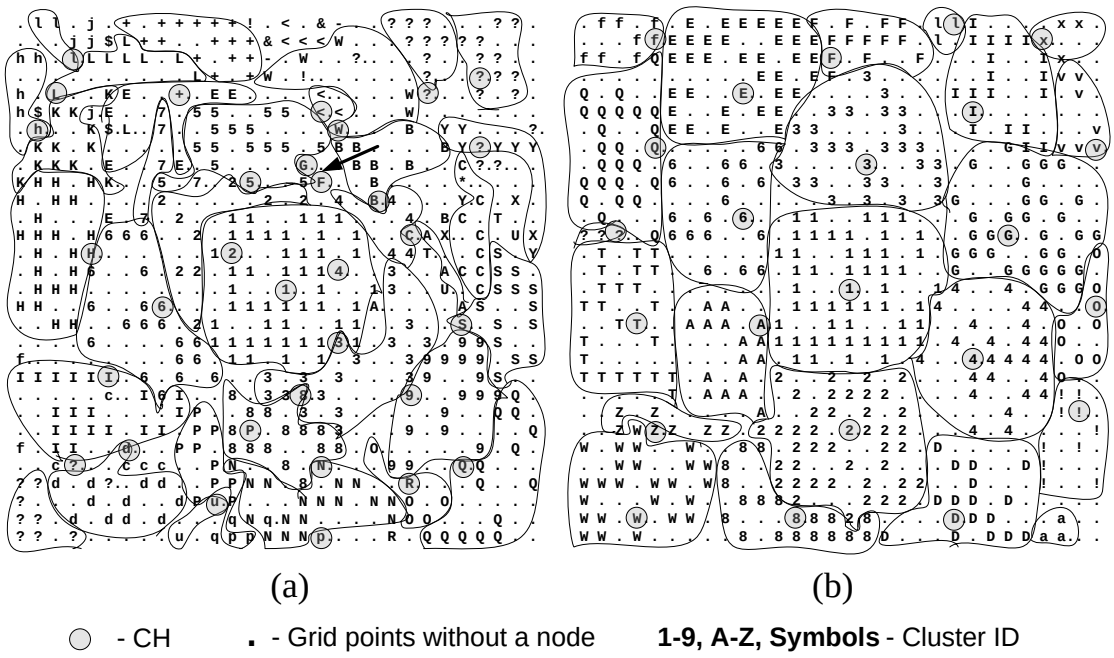


Figure 4.6 – Physical shape of single-hop clusters: (a) SHC clusters, (b) HHC clusters, Grid size = *30×30*, *n = 450*, *R = 30*.

the fact that the practical results differ widely from the conceptual scenario. Most of those CHs are selected from nodes that are at the edge of the parent cluster, e.g., CH of c_3 is curved into c_1. It is possible that some of the child CHs reside inside the parent cluster. For example, CHs of c_2 and c_4 are inside c_1. This problem cannot be prevented as candidate CHs are selected randomly and each node within the cluster has equal probability of being selected as a CCH. Note that clusters c_G and c_F (indicated by the arrow) do not have any child nodes. Those clusters initially had child nodes but those child nodes were later converted to CHs of clusters c_W and $c_<$. Alternatively, HHC clusters in Figure. 4.6(b) are much larger and somewhat circular. It is not possible to ensure that a CH will always be in the middle of a cluster because CCH selection is based on the hop count rather than the geometric distance. Clusters at the border of the sensor field are not circular because there are no more child nodes to be attached. Figure 4.7

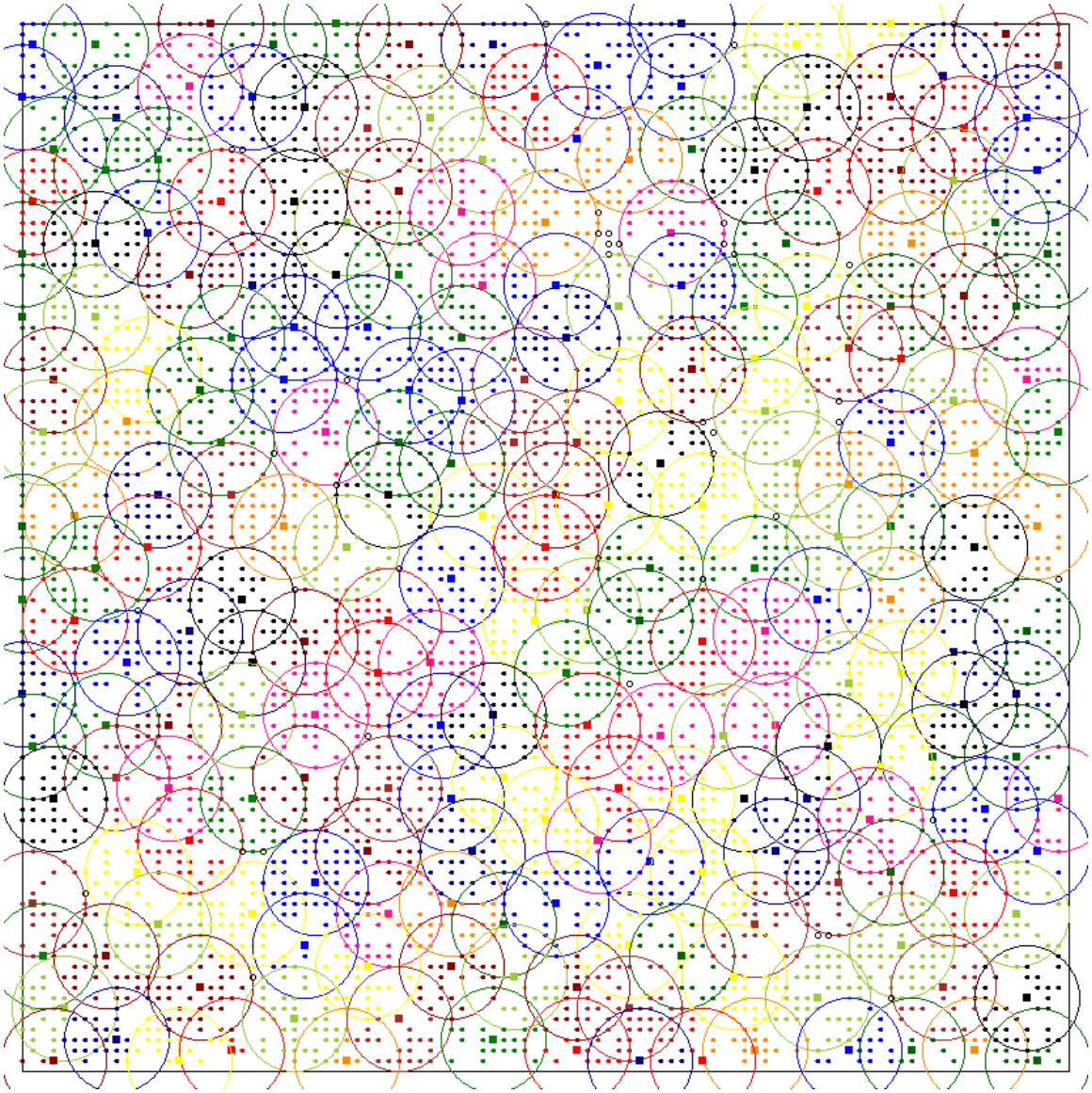


Figure 4.7 – Coverage map of HHC clusters. Grid – *100×100*, *n = 5000*, *R = 50*. Shaded circles - sensor nodes, Shaded squares - CHs, Circles - nodes without a cluster.

shows the coverage map for a much larger network. Overlap among clusters is somewhat higher as *R = 50*.

Figure 4.8 illustrates the circularity of single-hop clusters and their corresponding standard deviations (STD). HHC clusters have a much higher circularity than SHC. Ideal hexagonal clusters have the highest circularity. Ability to push CCHs further away from

the parent CH reduces the overlap among HHC clusters. However, there can be several clusters that significantly overlap with each other, e.g., c_8 and c_Z in Figure 4.6(b). Such clusters increase the STD of HHC. Almost all the SHC clusters overlap with each other hence have a lower STD. It can also be seen that the circularity reduces as the transmission range (R) increases. When R is higher, large number of nodes within 2-hops and 3-hops are capable of being selected as CCHs. When CCHs are randomly selected from such a large set of nodes, it is possible to select certain nodes that are closer to the CH, further away from the CH, or overlap within each other's transmission range. Though formation of two clusters within R is prevented (by assigning appropriate *delay* values) clusters can still overlap, if their CHs are within $2R$. Holes are created in the network when furthest away, i.e., closer to $3R$, nodes are selected as CCHs (Figure 4.3). These holes become much larger as R increases and new clusters are required to cover them up. The newly formed clusters will overlap with the existing clusters therefore reduces the overall circularity. All these factors reduce the circularity with increasing transmission range.

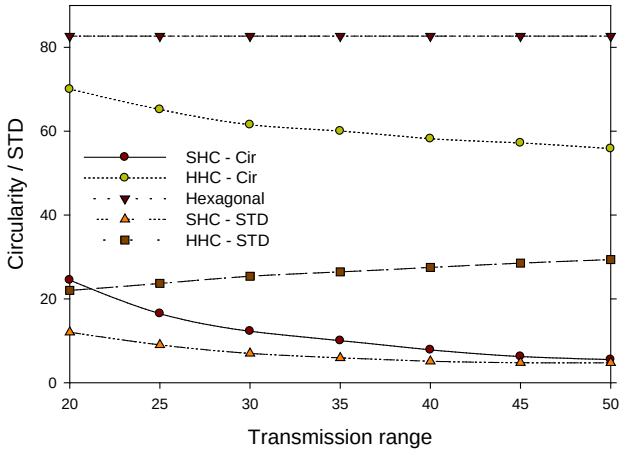


Figure 4.8 – Circularity of single-hop clusters.

Figure 4.9 shows the circularity of single-hop HHC clusters for different network densities, i.e., number of nodes. The circularity of the clusters reduces as the density increases. The number of nodes within 2-hops and 3-hops increases as the density increases. This behavior is similar to the case of increasing transmission range. Selection of CCHs among such a large set of nodes is not optimum. Due to the high density, even the small open regions that are created need to be covered by a cluster. Alternatively, it is not possible to effectively cover the sensor filed when the network is too sparse and transmission range is too low (e.g., HHC cannot effectively cover the sensor field when *R = 20* and *N = 2500*, hence no data is presented in Figure 4.9). It can be concluded that circularity of HHC clusters are closer to hexagonal clusters particularly for lower transmission ranges and node densities.

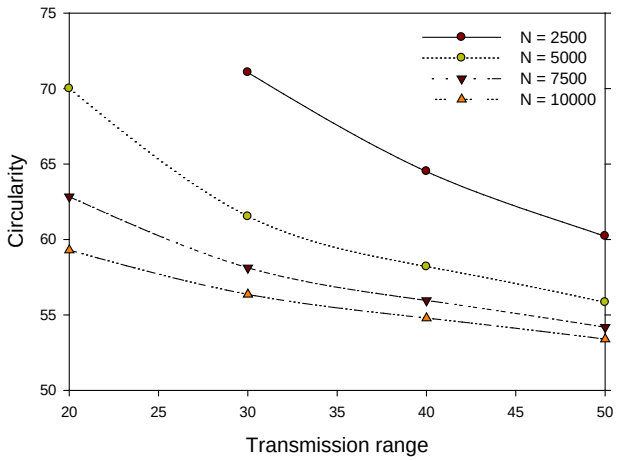


Figure 4.9 – Circularity of single-hop HHC clusters for different network densities.

Figure 4.10 compares the circularity of cluster formed by SHC, HHC, and FLOC [25]. Refer Appendix A for specific implementation details of FLOC. The HHC clusters have the highest circularity. FLOC clusters are 100% circular within the *i-band*; however, they overlap within the *o-band*, which reduces the overall circularity of a cluster.

Neighboring clusters in FLOC coordinate among each other through child nodes, to minimize overlap among clusters. This produces clusters with similar size and circularity hence STD of FLOC is somewhat lower than HHC. For all three algorithms, circularity decrease with increasing *R*. It was further observed that circularity of FLOC clusters also reduce as the network density increase.

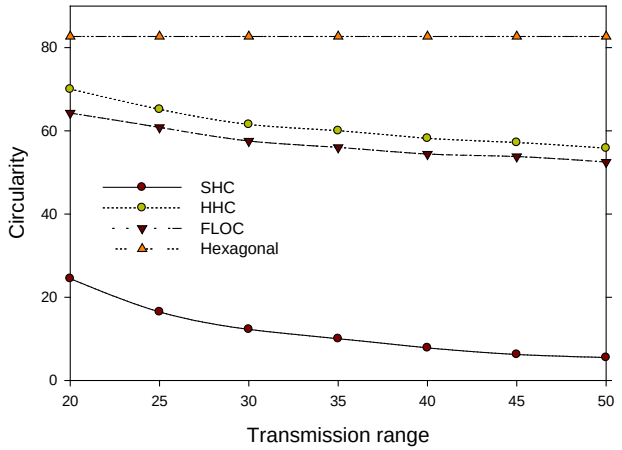


Figure 4.10 – Circularity of clusters formed by SHC, HHC, and FLOC.

Number of clusters and CHs produced by each algorithm is shown in Figure 4.11. Number of clusters required to cover a given sensor field depends on both circularity and area of a cluster. Due to higher circularity HHC produces the lowest number of clusters. As *R* increases, circularity of a cluster reduces while area increases (proportional to R^2). However, increase in area is dominant therefore number of clusters required to cover a given sensor field reduces as *R* increase. It is also observed that number of clusters produced by each algorithm somewhat increases with the network density. As the network becomes dense circularity somewhat reduces (Figure 4.9) therefore more clusters are required to cover a given sensor field.

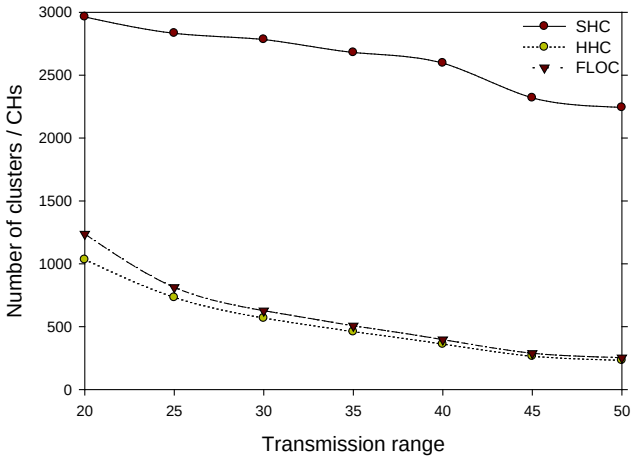


Figure 4.11 – Number of clusters and CHs.

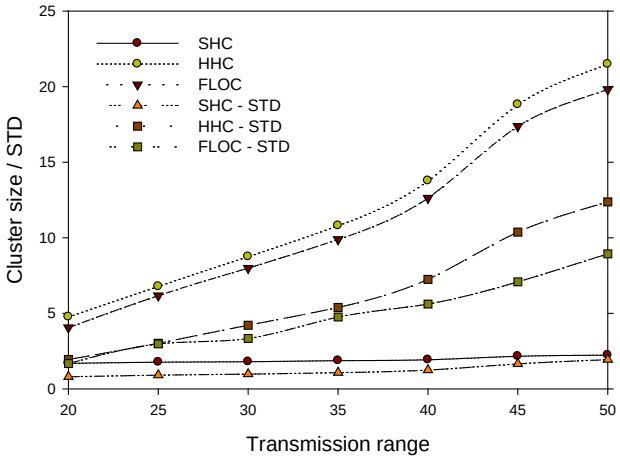


Figure 4.12 – Number of nodes in a cluster.

Number of nodes per cluster is shown in Figure 4.12. HHC produces much larger clusters while SHC clusters are much smaller. Lower overlap among clusters allows them to attract most of the nodes in its neighborhood. Therefore, HHC forms bigger clusters than SHC and FLOC. As the R increases cluster size rapidly increases except for SHC. The increase in SHC is less significant because low circularity dominates the cluster size over the increase in area. It is also observed that cluster size increases linearly with the node density. Smaller clusters that are formed at the edge of the sensor field increase the

STD of both HHC and FLOC. STD of HHC cluster size is higher due to the slightly varying circularity of HHC clusters. Figure 4.13 shows the distribution of cluster size. From Figure 4.11, 4.12, and 4.13 it is clear that HHC produces a smaller number of large clusters while SHC produces a larger number of small clusters.

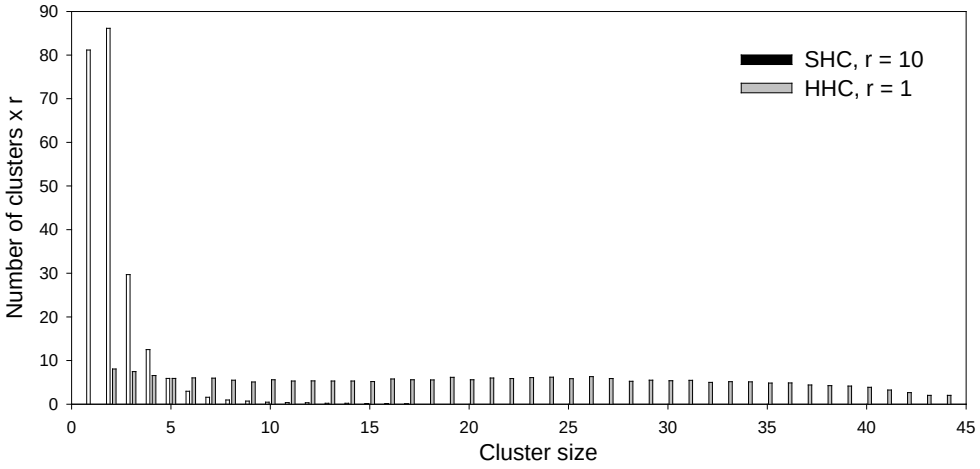


Figure 4.13 – Distribution of cluster size. *R = 50, $hops_{max}$ = 1*.

4.4.2.2 Multi-hop Clusters

When *R = 20* and 5000 nodes are placed on our *101×101* square grid, a PHC [9] cluster needs to be 9-hops. Such a large number of hops within a single cluster is not desirable due to high overhead and latency. To obtain comparable results the network size was reduced to 2500 nodes and $hops_{max}$ is selected such that $hops_{max} = 3$ when *R = 20* otherwise $hops_{max} = 2$. Appropriate TTL_{max} values are selected for SHC and HHC. Refer Appendix A for specific implementation details of PHC.

Figure 4.14 illustrates the circularity of multi-hop clusters. HHC has the highest circularity and PHC has the lowest. Circularity of multi-hop HHC is lower compared to the single-hop case. This behavior can be explained as follows. As $hops_{max}$ increases,

TTL_{max} needs to be increased. When TTL_{max} increases, number of CCHs significantly increases. As explained earlier a large set of CCHs may result in non-optimum set of clusters. Selection of CCHs also depends on how the cluster formation message is propagated. In multi-hop networks, a message traveling through a longer path may reach a node earlier than a message traveling through a shorter path. If a node relatively closer to the CH first receives a message with an expired TTL, i.e., that travelled through a longer path, it will assume that it is a CCH and sends an ACK back to the CH. If the CH selects such a node as a child CH, the parent and child clusters may overlap. These two issues reduce the circularity of multi-hop clusters. Later issue can be easily overcome by allowing each node to wait sometime before sending the ACK. In multi-hop SHC, cluster formation messages are forwarded several hops hence it is similar to the case in Figure 3.1(b). Circularity of SHC clusters significantly increased as CCHs are selected from nodes that are further away from the CH. PHC does not prevent the formation of two or more neighboring nodes within the same neighborhood. This is very likely to occur due to the probabilistic selection of CHs. As a result, overlap among clusters significantly

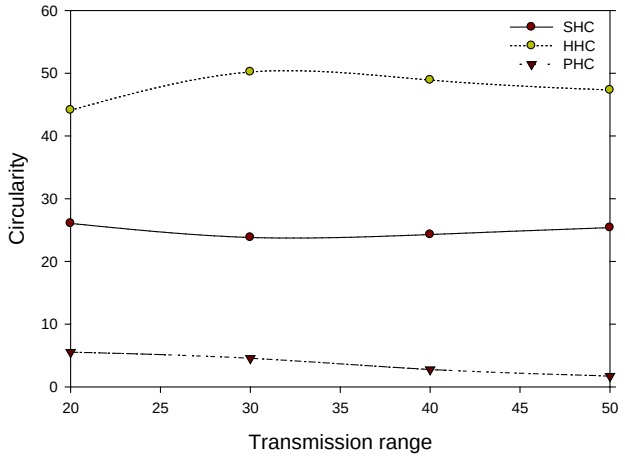


Figure 4.14 – Circularity of multi-hop clusters. $n = 2500$, $hops_{max} = 3$ when $R = 20$, $hops_{max} = 2$ otherwise.

increase. Sudden increase in circularity at $R = 30$ is due to the change from 3-hop to 2-hop clusters. Generally, multi-hop clusters form lower number of bigger clusters however their properties are not optimum.

4.4.3 Cluster Tree Characteristics

Figure 4.15 shows the distribution of nodes in the cluster tree. Breadth first spanning tree approach is used to form the cluster tree. The cluster trees formed by the HHC scheme are much shorter than the ones formed by SHC scheme. Because parent-child CHs in HHC are geographically distributed cluster tree has a higher branching factor as a result depth of the cluster tree reduces. As R increases, clusters become much larger and fewer clusters are required to cover the sensor filed. Therefore, the depth of the tree reduces with increasing R. Similar behavior is observed for multi-hop clusters (Figure 4.16). Figure 4.17 shows the cluster tree that corresponds to the coverage map shown in Figure 4.7. The root node is placed in the middle of the network and has four child CHs. Those child CHs then further expand their tree until the entire network is fully covered. Links among CHs may overlap with each other particularly for higher transmission ranges. This behavior is not so apparent in networks with lower R (Figure 4.18). As R increases more and more open regions are formed which needs to be covered by another cluster. These clusters reduce circularity and could increase collisions during inter-cluster communication. If the root node happens to be at an edge of the sensor field breadth of the cluster tree reduces while depth of the cluster tree increases.

Compared to [9, 33, 67], the distance between any parent-child CH pair in GTC has a bounded distance of $R \times TTL_{max}$. Although fully distributed clustering approaches

can form individual clusters, their connectivity is not guaranteed, particularly in sparse networks. It was observed that 1-5% of the nodes in HHC are disconnected from rest of the network, which is quite high. Though GTC algorithm does reduce collisions within two clusters, it does not prevent/reduce collisions within a cluster as it being formed. To effectively achieve desirable cluster and cluster tree properties these issues needs to be addressed.

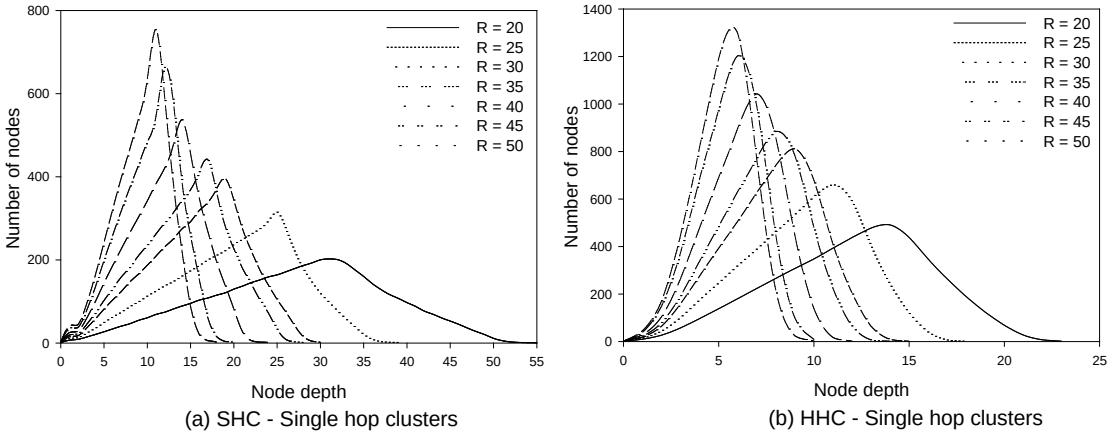


Figure 4.15 – Distribution of nodes along the cluster tree – single-hop clusters. Breadth-first tree formation, $n_{CCHs} = 3$ for SHC, $n_{CCHs} = 6$ for HHC.

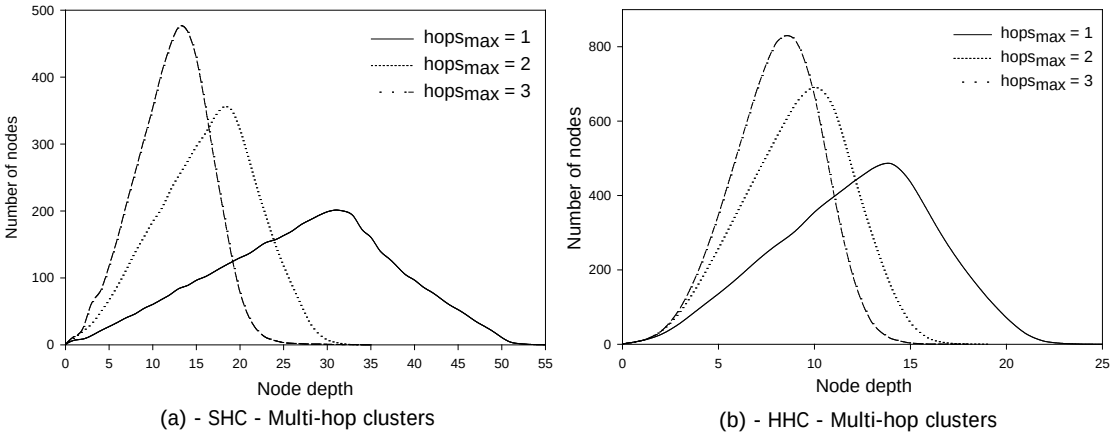


Figure 4.16 – Distribution of nodes along the cluster tree – multi-hop clusters. Breadth-first tree formation, $R = 20$, $n_{CCHs} = 3$ for SHC, $n_{CCHs} = 6$ for HHC.

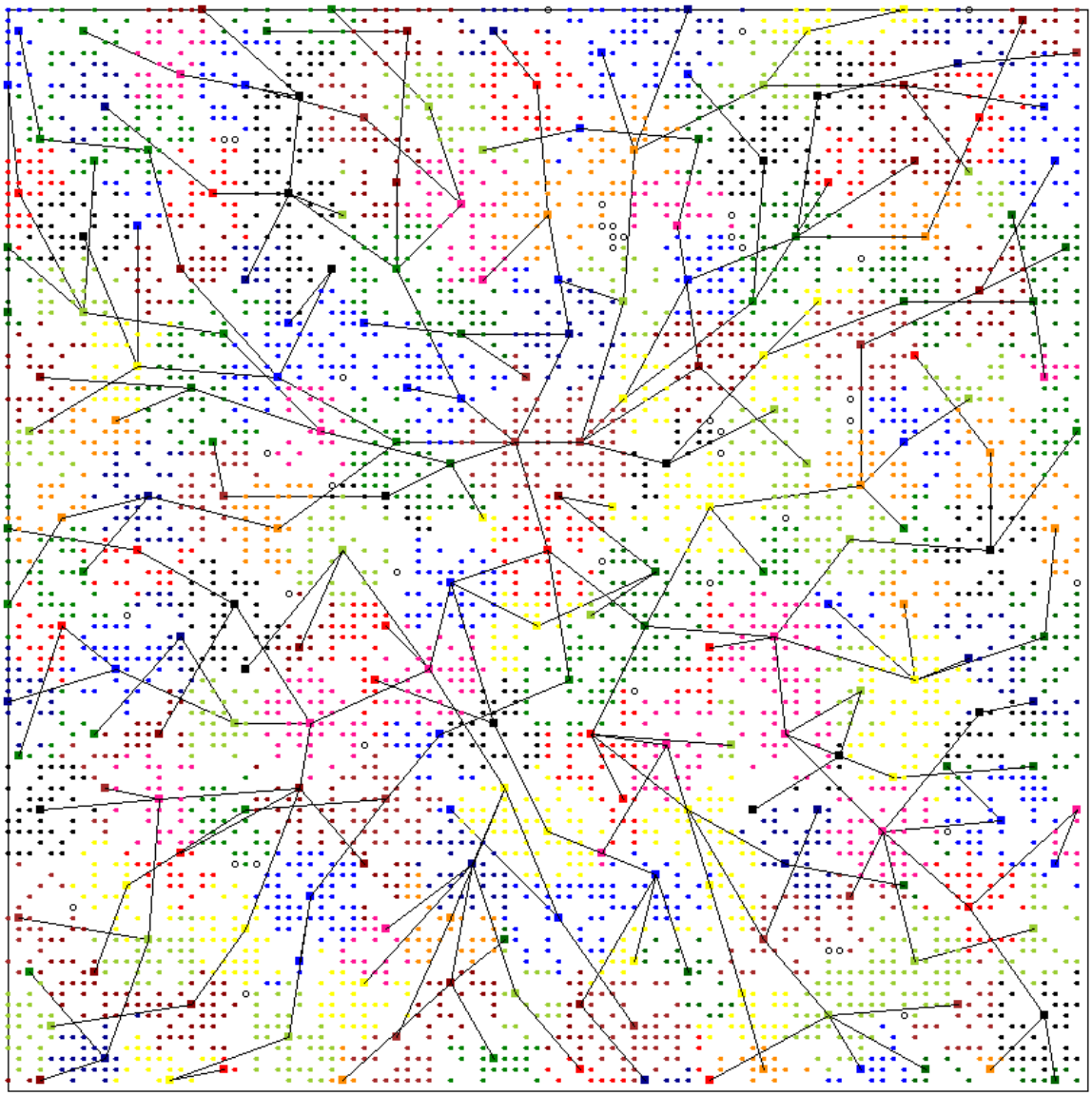


Figure 4.17 – Physical shape of HHC cluster tree – High transmission range. Grid – *100×100*, *n = 5000*, *R = 50*, root node in the middle, depth-first tree formation. Shaded circles - sensor nodes, Shaded squares - CHs, Circles - nodes without a cluster.

4.5 Summary

The chapter presented a detailed analysis of the generic top-down cluster and cluster tree formation algorithm. Algorithm is independent of network topology and does not require a-priori neighborhood information, location awareness, or time synchronization.

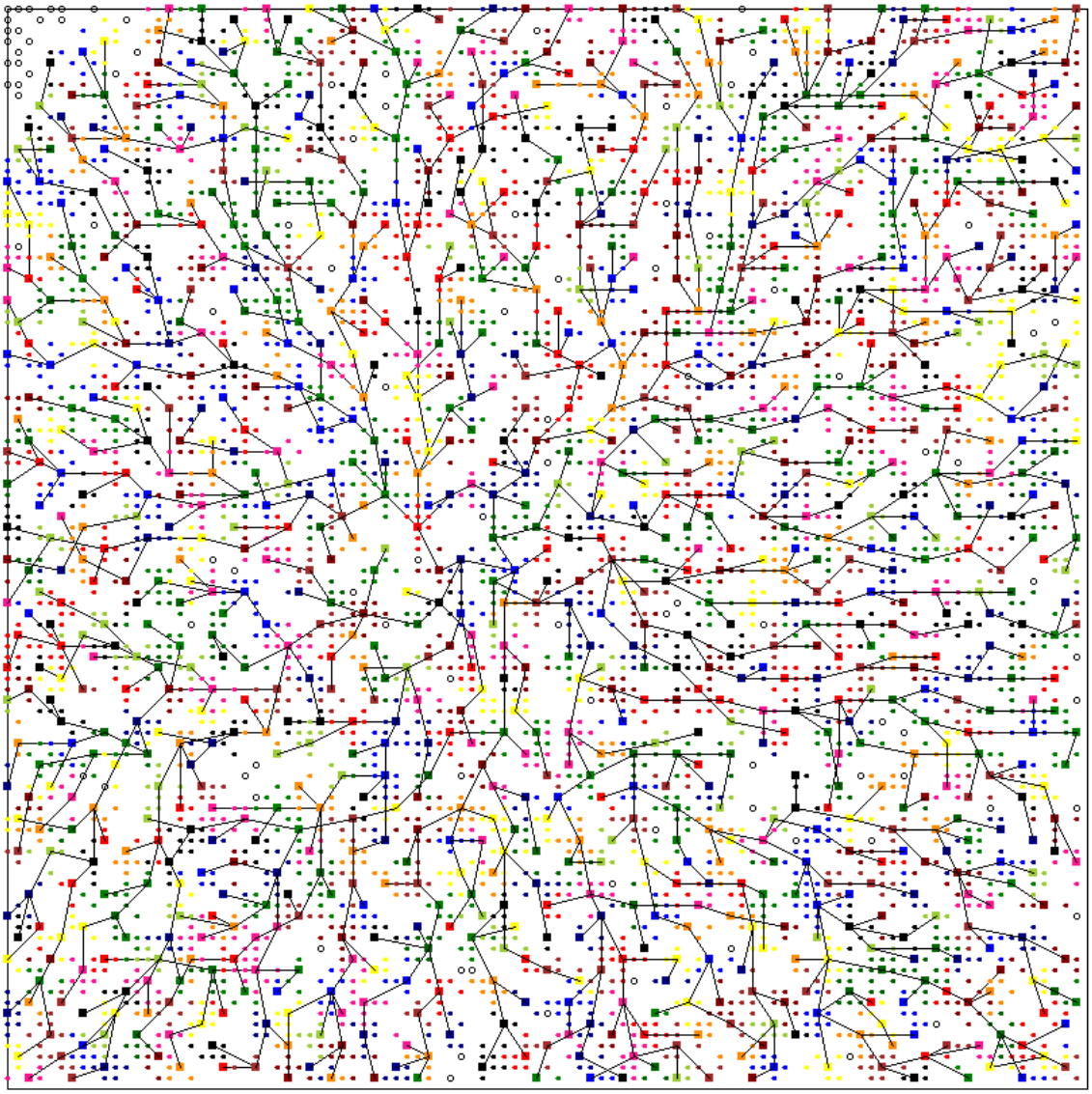


Figure 4.18 – Physical shape of HHC cluster tree – Low transmission range. Grid – *100×100*, *n = 5000*, *R = 20*, root node in the middle, depth-first tree formation. Shaded circles - sensor nodes, Shaded squares - CHs, Circles - nodes without a cluster.

SHC, variant of GTC, is similar to the IEEE 802.15.4 cluster tree. However, its properties are not desirable for most of the WSN applications. HHC, another variant of GTC, is proposed that forms more uniform and circular clusters and a cluster tree with lower depth. The chapter also analyzed how selection of different parameters and implementation of functions achieve desirable cluster and cluster tree characteristics. It is

important to further reduce collisions within clusters, number of control messages, and disconnected nodes. Analysis of the algorithm under more realistic simulation environment is also important. Next chapter address these issues.

Chapter 5

EXTENDED TOP-DOWN CLUSTER AND CLUSTER TREE FORMATION ALGORITHM

The HHC scheme of the GTC algorithm forms more circular and uniform clusters and produce cluster trees with lower depth. Although the GTC algorithm reduces collisions among different clusters, it does not prevent collisions within a cluster while it is being formed. It was realized that 1-5% of the nodes in HHC do not belong to a cluster. Because sensor nodes are location unaware, CCHs had to be selected randomly. However, most sensor nodes are capable of proving Receiver Signal Strength Indicator (RSSI) measurements. RSSI values can be used to estimate distance between two nodes hence can be utilized to select better set of CCHs. The chapter addresses the issues in HHC and extends the algorithm to make use of RSSI.

Extensions to the GTC algorithm are presented in Section 5.1. Section 5.2 presents the RSSI based CH selection. The two-step cluster and cluster tree optimization phase is presented in Section 5.3. Section 5.4 provides an analytical model that predicts the depth of the cluster tree. Finally, an extensive performance analysis of the algorithm is presented based on a more realistic simulation environment.

5.1 Extended GTC Algorithm

The extended GTC algorithm is shown in Figure 5.1. Lines 22 and 27 are the only new additions to the algorithm from Figure 4.1. By appropriately selecting different parameters and implementation of functions, the extended algorithm can implement SHC,

```
Form_Cluster(NID_CH, CID_CH, delay, n_CCHs, hops_max, TTL_max, depth)
 1  Wait(delay)
 2  TTL ← TTL_max
 3  Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
 4  ack_list ← Receive_ACK(NID_child, hops, P_1, P_2, timeout_ACK)
 5  IF(ack_list = NULL)
 6        Join_Cluster()
 7  FOR i = 1 TO n_CCHs
 8        CCH_i ← Select_Candidate_CHs(ack_list)
 9        CID_i ← Select_Next_CID(i)
10        delay_i ← Select_Delay(i)
11        depth_i ← depth + 1
12        Request_Form_Cluster(CCH_i, CID_i, delay_i, n_CCHs, hops_max, TTL_max, depth_i)

Join_cluster()
13  Listen_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
14  TTL ← TTL – 1
15  hops ← TTL_max - TTL
16  IF(hops ≤ hops_max AND my_CID = 0)
17        my_CID ← CID_CH
18        my_CH ← NID_CH
19        my_depth ← depth + 1
20        Send_ACK(my_NID, hops, P_1, P_2)
21  IF(TTL > 0)
22        Wait(Random(time_backoff))
23        Forward_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth)
24        IF(hops ≤ hops_max)
25              Exit()
26  ELSE
27        IF(Wait_Listen_Neighbors(Random(time_backoff)) = FALSE)
28              Send_ACK(my_NID, hops, P_1, P_2)
29              IF(Listen_Form_Cluster(CCH, CID, delay, n_CCHs, hops_max, TTL_max, depth,
    timeout_CCH) = TRUE)
30                    Form_Cluster(my_NID, CID, delay, n_CCHs, hops_max, TTL_max, depth)
31                    Exit()
32  Join_cluster()
```

Figure 5.1 – Extended generic top-down cluster and cluster tree formation algorithm.

HHC, and RSSI based clustering. Only the new additions/changes to the previous algorithm are described here.

The root node initiates cluster formation process by executing the ***Form_Cluster*** function. It sends a cluster formation broadcast using the ***Broadcast_Cluster*** function and announces its presence to its neighbors. All other nodes execute the ***Join_Cluster*** function and wait for a cluster formation broadcast. A node joins the cluster if it is not already a member of another cluster and within $hops_{max}$. Then an acknowledgment (ACK) is send to indicate the node's interest to become a member of the cluster. After sending the ACK, the node waits some random back-off time based on $time_{backoff}$ (line 22) and then forwards the broadcast using the ***Forward_Broadcast_Cluster*** function, if *TTL* is still valid. Random back-off time reduces the probability of two nodes broadcasting at the same time therefore reduce the collisions.

Nodes that are not within $hops_{max}$ do not join the cluster instead forward the broadcast until *TTL* expires. To reduce collisions even these broadcasts are randomly delayed. Intermediate nodes do not need to send any ACKs. If *TTL* is expired, the receiving node is capable of being selected as a new CH (i.e., CCH). By listening to the transmissions from neighbors, the possibility of selecting two nearby nodes as CCHs can be reduced. Therefore, each candidate node waits some time before sending an ACK back to the corresponding CH (***Wait_Listen_Neighbors*** function). Waiting time is random and selected based on $time_{backoff}$. While waiting, nodes keeps listening to the channel and try to detect any ACKs send by their neighbors, to the same CH. If such an ACK is detected, (function returns *TRUE*) the node gives up its candidacy to be a CH and waits for new cluster formation broadcast, i.e., reruns the ***Join_Cluster*** function. If no

ACKs are detected by the time the function timeouts, it becomes a CCH and sends an ACK. Child CHs are chosen from these spatially distributed set of CCHs therefore generate better set of clusters. Such a set of child CHs increase the breadth and reduces the depth of the cluster tree. Reduced number of ACKs further reduces the overhead of the GTC algorithm.

5.2 RSSI Based Cluster Head Selection

Node location information is essential to select a precise set of child CHs. However, “due to constrain on cost, size, energy consumption, and implementation environment (e.g., GPS is costly, and not accessible indoors) most sensors are location unaware” [48]. Many localization techniques have been proposed to determine the location of such sensors [48]. Received signal strength based techniques, based on the Received Signal Strength Indicator (RSSI) available in most wireless devices, can be used to estimate the distance between two nodes. RSSI is attractive because it has minimum impact on hardware, power consumption, size, and cost of sensors. However, reliability of RSSI as a distance measure is debatable [34, 45, 57]. RSSI measurements in new radios such as CC2420 [59] seem to be stable over time particularly if nodes are raised above ground [34, 45]. It has also been shown that RSSI decreases exponentially with the distance [34]. Therefore, RSSI is a suitable metric to determine relative distances among nodes. By utilizing RSSI, it is possible to push the cluster formation message further away from the parent CH. If the cluster formation message is pushed by 2-hops clusters will slightly overlap (Figure 3.1(b)) and if it is pushed up to 3-hops it will create

a hole (Figure 3.1(c)). A better alternative would be to push the message up to 2-hops and then select the closest node, i.e., with the highest RSSI, as the child CH.

Forming non-overlapping set of clusters with minimum number of uncovered nodes is difficult without RSSI. A heuristic for such a case when nodes are uniformly distributed in the network is as follows. Let us select a random node to forward the cluster formation broadcast. For a uniformly distributed network, this node is equally likely to be within or outside a circle of radius *r* (Figure 5.2 (a)). Then:

$$\begin{aligned} &\textit{Probability of being inside } r = \textit{Probability of being inside } (R - r) \\ &\pi r^2 \lambda = (\pi R^2 - \pi r^2)\lambda \\ &r^2 = R^2 - r^2 \\ &r = R/\sqrt{2} \end{aligned} \tag{5.1}$$

where λ is the node density and *R* is the transmission range of a node.

Let us use this information to separate parent and child CHs by 3-hops. The cluster formation message is forwarded over 3-hops before its final destination, which becomes the next CH. In the ideal case, it will be at a distance of $3r = 3R/\sqrt{2}, \approx 2.1R$. Therefore,

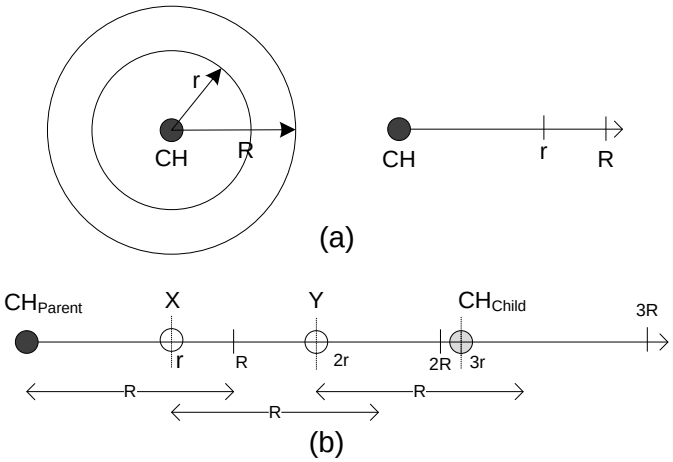


Figure 5.2 – Propagation of cluster formation broadcast: (a) – Location of the randomly selected node. (b) – Distance between parent and child CHs when cluster formation message is forwarded by 3-hops. *R* - Transmission range of a node. *r* – Distance to a randomly selected node from the CH.

the new child CH is just more than 2-hops away from the parent CH. This explains the behavior of our HHC scheme. However, due to the fact that the three nodes do not lie on a straight line, the results based on this heuristic will be far from optimal.

The RSSI, when available, can be used to estimate the distance to a node that sends a message. Signal strength reduces with the distance, accordingly the RSSI value reduces. Cluster formation process can make use of RSSI values (that corresponds to the cluster formation broadcasts) to enhance the properties of clusters. To make use of RSSI, lines 22 and 27 of the algorithm need to be modified as:

22 Wait(Random(RSSI + $time_{backoff}$))

*27 **IF**(Wait_Listen_Neighbors(RSSI + Random($time_{backoff}$)) = **FALSE**)*

Now the waiting time depends on two factors, the *RSSI* value of the received broadcast and the random back-off time. A node with a lower *RSSI* value gets a higher priority and forwards the broadcast before a node with a higher *RSSI* value. This allows nodes that are further away from the CH (lower *RSSI*) to forward cluster formation broadcast first. Similarly, in line 27, nodes further away from the CH get higher priority in responding back as candidate CHs because the waiting time is proportional to *RSSI*. Nearer nodes will respond only if they do not hear an ACK from a further away neighbor. This increases the probability of selecting further away nodes as CCHs. However, it is not desirable to push the CCHs too far as it creates holes in the network.

Pushing the cluster formation message to the maximum distance during the first 2-hops and then selecting the next nearest node (within 2-hops and 3-hops) would be a better alternative. In that case lines 22 and 27 need to be changed as:

22 Wait(Random(RSSI + $time_{backoff}$))

*27 **IF**(Wait_Listen_Neighbors(1/RSSI + Random($time_{backoff}$)) = **FALSE**)*

Nearest nodes first send ACKs to the CH while other nodes respond only if they do not hear one of its neighbors sending an ACK. This approach produces more uniform clusters, reduces the number of disconnected nodes, and reduces the depth of the cluster tree. We name RSSI based SHC and HHC as RSHC and RHHC respectively.

5.3 Cluster And Cluster Tree Optimization Phase

The HHC pushes CCHs further away from the parent CH producing more circular clusters. However, circular clusters can generate open regions in the network (e.g. holes generated by three adjacent circles, Figure 4.3). A node may not belong to a cluster because; it is in such an open region, disconnected from rest of the nodes, or unable to hear a cluster formation broadcast due to collisions. Because of these reasons, it was realized that around 1–5% of the nodes in HHC were not into a cluster.

An optimization phase that executes after the cluster and tree formation phase can allow most of these unconnected nodes to join an existing cluster. If not, such nodes can form their own clusters and later join the cluster tree. The algorithm is shown in Figure 5.3 can be used to optimize these nodes. When a node realizes that it is not in a cluster after some predefined time it executes the ***Join_Existing_Cluster*** function. It then listens to the channel (***Listen_For_Cluster*** function) and tries to detect a neighboring CH. The node may respond to a periodic beacon from a CH or to a message send by the CH to one of its child nodes. If such a message is detected (function returns *TRUE*), the node sends an ACK to the corresponding CH and joins the cluster. If no such message is heard before the timeout, the node forms its own cluster by executing the ***Form_Cluster*** function of the GTC algorithm. Each such node waits some time, based on

Random(backoff_CH), before advertising itself as a CH (line 7). This random delay reduces collisions and the possibility of forming multiple clusters in the same neighborhood. Cluster formation broadcast is not propagated beyond the new cluster and no CCHs will be selected (i.e., $hops_{max} = 1$, $TTL_{max} = 1$, and $n_{CCHs} = 0$). Depth is set to infinity because these new clusters are not yet part of the cluster tree. The same algorithm can also be used to add new nodes to an existing network.

```
Join_Existing_Cluster()
1  IF(Listen_For_Cluster(NID_CH, CID, depth, timeout_listen_CH) = TRUE)
2        my_CID ← CID
3        my_CH ← NID_CH
4        my_depth ← depth + 1
5        Send_ACK(NID_child, hops, p1, p2)
6  ELSE
7        Form_Cluster(NID_child, NULL, Random(backoff_CH), 0, 1, 1, ∞)
```

Figure 5.3 – Algorithm that handles non-cluster members.

```
Broadcast_CH_Presence(NID_CH, CID, depth, TTL)

Listen_Optimize_Tree()
1   Lsiten_Broadcast_CH_Presence(NID_CH, CID, depth)
2  IF(my_depth > depth + 1)
3        my_CID ← CID
4        my_CH ← NID_CH
5        my_depth ← depth + 1
6        opt_msg_send ←FALSE
7  IF(opt_msg_send = FALSE)
8        Broadcast_CH_Presence(NID_child, my_CID, my_ depth, TTL)
9         opt_msg_send ←TRUE
```

Figure 5.4 – Cluster tree optimization algorithm.

Depth and breadth of the cluster tree depends on how the cluster formation broadcast was forwarded and which nodes were randomly selected as CCHs. Collisions also affect the shape of the cluster tree. Cluster tree can be further improved by

exchanging another set of messages between CHs. The algorithm shown in Figure 5.4 can be used to further improve the cluster tree. Newly formed clusters from the previous optimization phase can also join the cluster tree during this phase. Therefore, in order execution of the node optimization and the cluster tree optimization phases are important. However, it is possible to execute only the cluster tree optimization phase.

After the cluster formation phase all the CHs (except the root node) executes the ***Listen_Optimize_Tree*** function and try to upgrade their membership in the cluster tree. The root node initiates the tree optimization phase by indicating its presence to neighboring CHs (***Broadcast_CH_Presence*** function). When the broadcast is received, each neighboring CH compares its current depth with what was heard from the neighbor. If the new depth is lower, it selects the broadcasting CH as its new parent and reorganizes its cluster tree membership. When such a change occurs, it may also need to inform its cluster members (not shown in Figure 5.4) however this depending on the application scenario. This new information may not useful for certain CHs that are already having the same depth. However, these CHs have to send their own optimization broadcasts at least once. This ensures that each CH gets at least one optimization message hence gets an opportunity to upgrade the cluster tree membership. In future if a CH hears another message with even lower depth, it may again reorganize its cluster tree membership. Subsequently, it has to reorganize all its child CHs (lines 6 and 7).

Cluster tree optimization phase does not need to be a completely separate task. CHs may continue with their regular tasks (relaying, aggregating, etc.) and can deal with such a broadcast as a special message. The tree optimization algorithm is independent of the inter-cluster communication mechanism. The broadcasts may travel single or multiple

hops depending on the CH-to-CH communication model. For multi-hop transmissions, i.e., low-power, *TTL* should be selected such that $TTL = TTL_{max}$. For single-hop transmissions, i.e., high-power, $TTL = 1$. In multi-hop case, the optimization broadcasts are propagated similar to the cluster formation messages. The number of CHs that can upgrade their cluster tree membership with multi-hop forwarding is limited. Single-hop broadcasts have to use high-power therefore can directly communicate with many neighboring CHs. Nodes that were not reachable during the cluster formation phase (due to lack of intermediate nodes, Figure 4.5) can now be reached within a single-hop. Such CHs can significantly improve their cluster tree membership. As a result, breadth of the cluster tree increases and consequently the depth reduces. Use of high transmission power considerably reduces the total number of broadcasts but consume much more energy. These optimization phases are not applicable for SHC because CCHs are selected from nodes that are already within the parent cluster.

5.4 Depth Of The Cluster Tree

Breadth/depth of the cluster tree and number of clusters in the network depend on the shape and size of the sensor filed. It also depends on the position of the root node and transmission power utilized by a node. Therefore, it is important to predict the breadth and depth of the cluster tree. Following analytical model can be used to predict the depth of a cluster tree formed in a circular sensor field.

Let us make use of hexagonal packing that initiates from the root node (Figure 5.5). The root node is placed in the middle of the sensor field and its depth is assumed zero. Let R be the radius of the sensor field and r be the transmission range of a node.

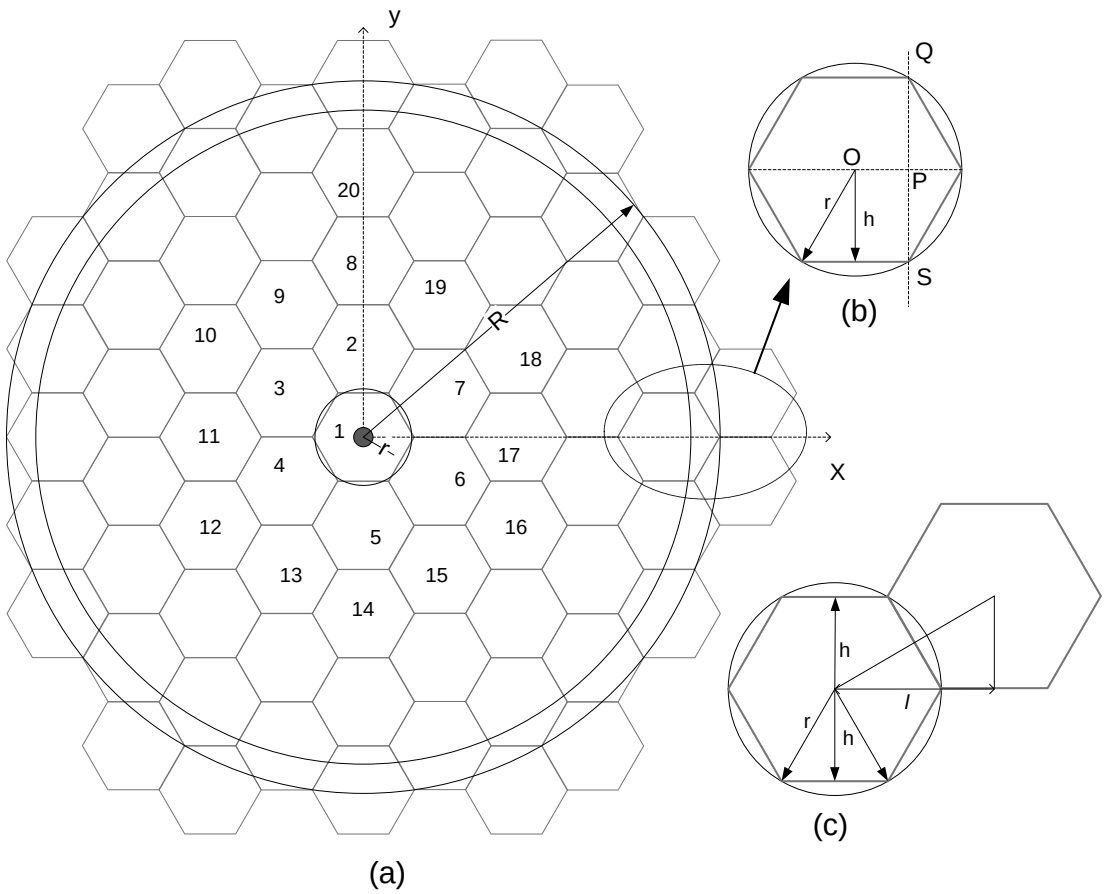


Figure 5.5 – Ideal hexagonal packing. *R* – Radius of the sensor field, *r* – transmission range of a sensor node.

Let us first analyze the cluster formation along Y-axis.

Distance to edge of the sensor filed from root node = R

Distance to edge of the sensor filed from the edge of 1st cluster = $R - h$

where *2h* is the height of a hexagon (Figure 5.5(c)) and $h = \sqrt{3}r/2$

∴Number of clusters between edge of 1st cluster & edge of sensor filed $= \left\lceil \frac{R-h}{2h} \right\rceil$ (5.2)

If we assume each cluster is a child of another (e.g. c_2 is a child of c_1, c_8 is a child of c_2, c_{20} is a child of c_8, etc.), the number of clusters along Y-axis indicates the depth of the cluster tree. In reality, clusters will never form along the same axis; therefore, analysis along Y-axis provides only a lower bound. Analysis along X-axis provides the upper

bound. Radius of the sensor filed determines the number of clusters formed along X-axis. Consider inset (b) of Figure 5.5, which shows a border case analysis. If the sensor filed is beyond the line *QS* drawn through point *P* (*R > distance to P from the root node*) additional level of clusters needs to be formed. Otherwise, no additional level of clusters is required. The depth of the cluster tree along X-axis is analyzed under these two conditions.

$$OP = r/2$$

$$l \text{ (distance between two CHs along X-axis)} = \sqrt{(2h)^2 - h^2} = \sqrt{3}h$$

$$\text{If edge of sensor field} \le P = (R \% \sqrt{3}h) \le OP = (R \% \sqrt{3}h) \le r/2$$

$$\therefore \text{ depth} = \left\lfloor \frac{R}{\sqrt{3}h} \right\rfloor \quad (5.3)$$

$$\text{If edge of sensor field} > P = (R \% \sqrt{3}h) > r/2$$

$$\therefore \text{ depth} = \left\lfloor \frac{R}{\sqrt{3}h} \right\rfloor + 1 \quad (5.4)$$

Therefore, from Equations 5.2, 5.3, and 5.4 and replacing *h* with *r*:

$$depth_{min} = \left\lceil \frac{R}{\sqrt{3}r} - \frac{1}{2} \right\rceil \quad (5.5)$$

$$depth_{max} = \begin{cases} \left\lfloor \frac{2R}{3r} \right\rfloor & if\left(R\% \frac{3r}{2} \right) > \frac{r}{2} \\ \left\lfloor \frac{2R}{3r} \right\rfloor + 1 & else \end{cases} \quad (5.6)$$

Practical cluster formation is significantly different to the idea case. However, Equations 5.5 and 5.6 can be used to determine the bounds of the cluster tree. If the size of the sensor field and desired depth of the cluster tree is known, above equations can be used to back calculate the desired transmission power.

From Figure 5.5 it can be seen that number of clusters in each level increase linearly (1, 6, 12, 18, 24, …). Only six clusters need to be formed at *level 1*. Therefore, it is sufficient for the root node to select six child CHs. In *level 2*, 12 child CHs need to be selected by six parent CHs. Hence only two child CHs need to be select by each *level 2* CH. This ratio reduces as the number of levels (depths) increases. Therefore, it is sufficient to select fewer number of child CHs as the depth increases. However, in practice some of the parent CH may not be able to form all the necessary child clusters. This reduces branching factor and increases depth of the cluster tree. Therefore, we select three CCHs at each level except at the root node where we select six CCHs.

5.5 Performance Analysis

The characteristics of clusters and the cluster tree are evaluated using simulations that are more extensive. Nodes are randomly placed on a circular region with a radius of *500m*. The sensor filed is embedded within a *201×201* grid and the grid spacing is *5m*. The root node is placed in the middle of the sensor field and single-hop clusters are formed using the breadth-first tree formation approach. Six CCHs are selected for the first level and three for all the other levels. It is assumed that a node does not successfully hear a broadcast if it is within the collision region of two concurrent broadcasts, therefore cannot join either of the clusters. Free space propagation model is used for signal propagation and path loss exponent is set to 2.2 [51]. The results are based on 100 sample runs (20 random networks × 5 samples per network). Results are compared with hexagonal packing. Circular region is considered to make the comparison with hexagonal

packing easier. Specific implementation details of the simulator are presented in Appendix A. Following acronyms are used to identify different clustering mechanisms:

- Hexagonal – Clusters based on hexagonal packing
- HHC-Opt – HHC with two-step optimization phase
- RHHC – RSSI based HHC
- RHHC-2 – RSSI based HHC with $TTL_{max} = 2$
- RHHC-min– RSSI based HHC, where the cluster formation broadcast is pushed to the maximum limit during the first 2-hops and then select the nearest nodes during the third hop.
- RSHC – RSSI based SHC

5.5.1 Cluster Characteristics

Figure 5.6 illustrates the circularity of clusters. The HHC has the highest circularity and it is followed by RHHC. Availability of RSSI values enhance the properties of SHC clusters however, the improvements are not significant. Because SHC has much lower performance, it will not be considered in future comparisons.

Figure 5.7 compare the circularity of HHC and RSSI based HHC. Circularity of HHC, RHHC, and RHHC-min are similar. Slight differences in circularity can be explained as follows. HHC has the highest circularity while 2-hop RHHC has the lowest circularity. RHHC-min clusters are uniformly distributed because the CCHs are selected from nodes that are just above 2-hops away from the parent CH. Uniform coverage increase the overlap among clusters to a certain extent. Therefore, circularity of RHHC-min is lower than HHC. RHHC generates holes in the network as cluster formation

messages are pushed to the limit based on RSSI. These holes need to be covered by new set of clusters therefore overall circularity reduces. Instead of forwarding the cluster formation message by 3-hops we also tried RSSI based HHC with 2-hop forwarding (RHHC-2). Though it can push child CHs to the maximum limit within 2-hops, circularity is lower as clusters can still overlap (Figure 3.1(b)).

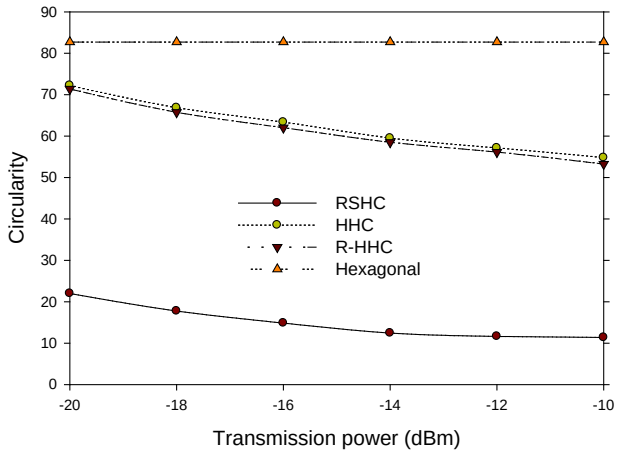


Figure 5.6 – Circularity of clusters.

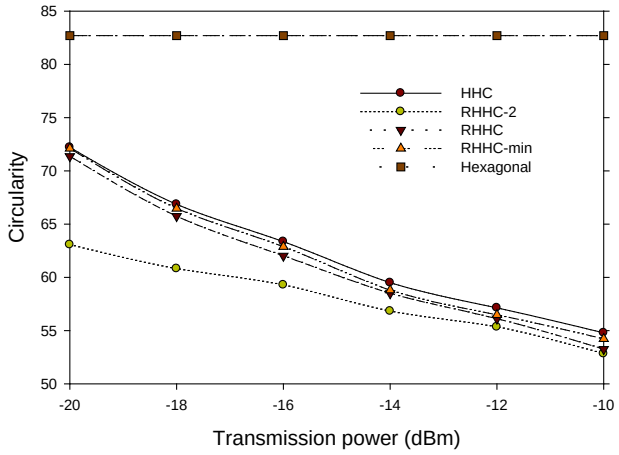


Figure 5.7 – Circularity of HHC and RSSI based HHC clusters.

Due to several factors, circularity reduces as the transmission power (P_T) increases. When P_T is higher, many nodes are capable of being selected as CCHs. The

Wait_Listen_Neighbors function reduces the number of ACKs and prevents two nodes that are within each other's communication range (*R*) from sending an ACK. However, if those nodes are still within *2R* their clusters will overlap. This is harder to prevent because CCHs are randomly selected. High P_T also increases collisions (even with random waiting) which affect the forwarding of cluster formation broadcast. Due to collisions, some of the nodes may not be able to hear a cluster formation broadcast. As a result, those nodes cannot join a cluster even though they are in the range of a CH.

Figure 5.8 illustrates the variation of circularity with node density. When the network is sparse, clusters are more circular. Circularity reduces with the increasing node density. This behavior is similar to the case of increasing P_T. As there are many nodes in a region even the smaller open regions needs to be covered. When the network is sparse circularity of clusters formed by HHC, RHHC, and RHHC-min is better than hexagonal packing. Therefore, it can be concluded that circularity of HHC and RSSI based HHC is comparable with hexagonal clusters particularly for lower P_T and node densities.

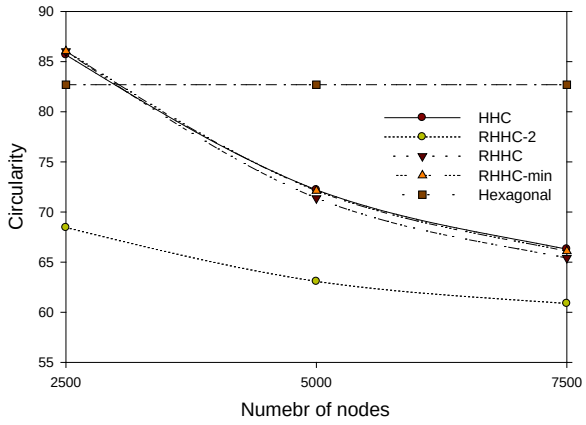


Figure 5.8 – Circularity of cluster for different network densities. *R = -20dBm*.

Though use of RSSI improves the cluster and cluster tree characteristics, RSSI values are not that reliable because of the noise. Figure 5.9 shows how circularity is

affected by noise. Circularities of all the clustering mechanisms drop approximately by 15%. Reliability of RSSI reduces with the distance due to the noise [32, 43, 55]. We model the same behavior in our simulator (Section A.3). Therefore, nodes that are closer to a transmitting node not only indicate higher RSSI values but also those values are less affected by noise. In RHHC-min, CCHs are selected from nodes having higher RSSI values in the last hop. Therefore, CCH selection in RHHC-min is less affected by noise. This enables the selection of better set of CCHs; therefore, circularity of RHHC-min is higher than the other two solutions. Standard deviation of RHHC-min was also lower which further confirm this behavior. Other two solutions are much more affected by unreliability of RSSI that increase with the distance. Variation seen in Figure 5.9 is due to the fact that nodes are placed on a grid. This issue can be overcome by selecting a more granulized grid. It was further observed that some of the cluster members were actually outside the transmission range of the CH. This behavior did not significantly increase the number of clusters or reduce the cluster size. However, it significantly affected the measurement of circularity.

The two-step, optimization phase somewhat reduces the average circularity of the clusters (Figure 5.10). During the first stage, nodes without a cluster either join an existing cluster or form their own clusters. Addition of disconnected nodes to an existing cluster somewhat increases the circularity. The new clusters formed by disconnected nodes are smaller and very likely to overlap with most of the existing clusters. Therefore, overall circularity of the network reduces. These new clusters also increase the standard deviation, as their circularity is much lower than the rest of the network.

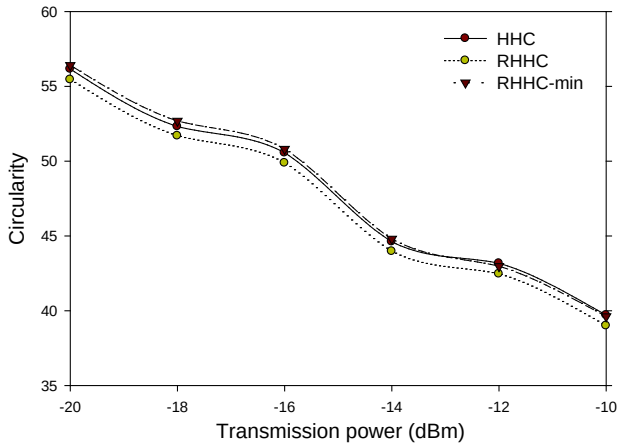


Figure 5.9 – Circularity of clusters under uncertainties in signal strength. Random noise with zero-mean and standard deviation of *-6dBm*.

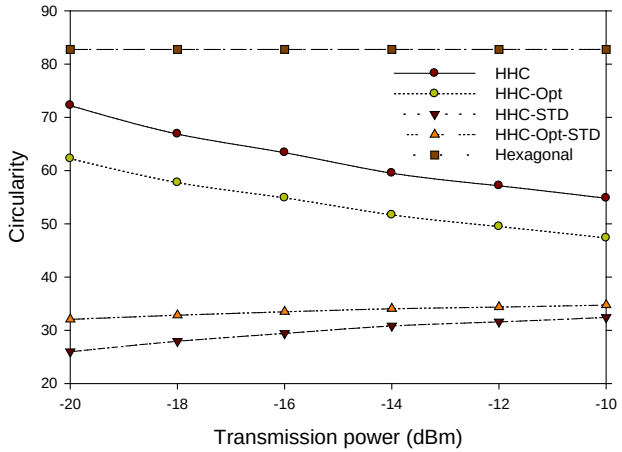


Figure 5.10 – Reduction in circularity due to the optimization phase.

Number of clusters produced by each solution is shown in Figure 5.11. HHC and RHHC produce similar number of clusters. RHHC-min produces higher number of clusters because its circularity is lower and requires more clusters to uniformly cover the sensor filed. Results are not significantly different from hexagonal packing. As P_T increases, area covered by a cluster increases therefore the number of clusters required to cover a sensor field reduces. It was also observed that uncertainties in signal strength somewhat increase the number of clusters produced by each solution.

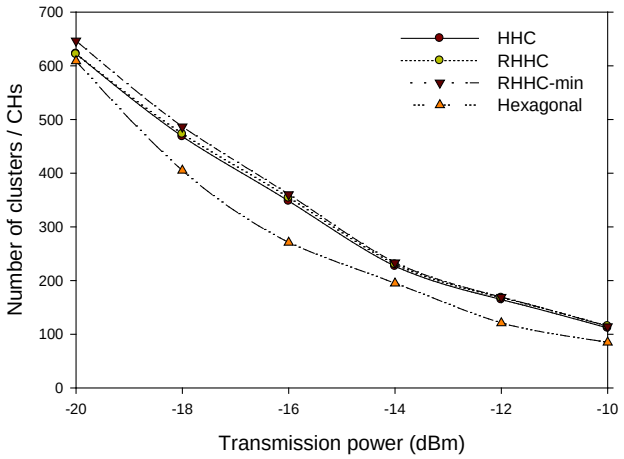


Figure 5.11 – Number of clusters and cluster heads.

Number of clusters formed by networks of different densities is shown in Figure 5.12. When 2500 nodes are placed in the network, circularity of HHC and RHHC-min is higher than hexagonal packing (Figure 5.8). As a result, HHC and RHHC-min form lesser number of clusters. Sparse networks also include some open regions that do not need to be covered by clusters. Therefore, the number of clusters required to cover a sensor field further reduces. As the density increases, even the smaller open regions need to be covered by a cluster. Therefore, the number of clusters increases with density. New set of clusters are formed during the cluster and cluster tree optimization phase. Therefore, HHC-Opt generates more clusters than HHC (Figure 5.13).

Cluster size distribution is shown in Figure 5.14. HHC has a slightly higher cluster size than RHHC and RHHC-min. For the same P_T , HHC produces lower number of clusters therefore has the highest cluster size. It was also observed that RHHC-min has a relatively lower standard deviation. This further suggests that RHHC-min forms more uniform clusters. These cluster sizes are comparable with the hexagonal clustering particularly for lower P_T values. Higher P_T values form much larger clusters due to the

increased coverage area. Reduction in circularity with increasing P_T significantly reduces the cluster size. Cluster size also increases with the node density. However, due to extensive overhead on CHs such larger clusters may not be desirable. As discussed earlier, cluster and tree optimization phase generate several new clusters. Most of these new clusters include only one or two nodes. Hence, the optimization phase reduces the average cluster size.

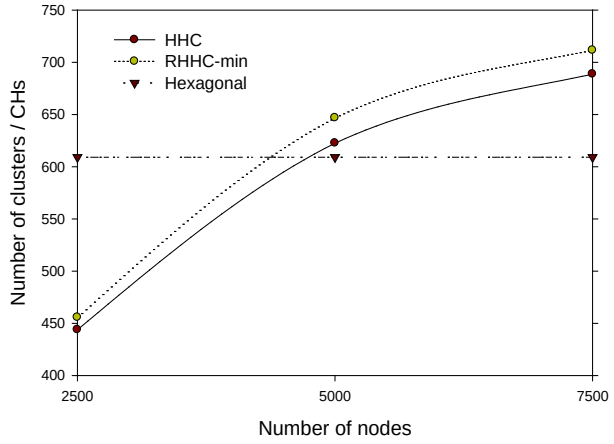


Figure 5.12 – Number of clusters produced by networks of different sizes.

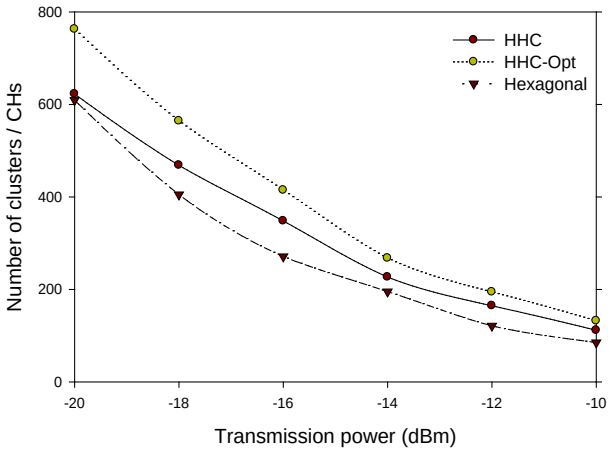


Figure 5.13 – Number of clusters and cluster heads produced by the optimization phase.

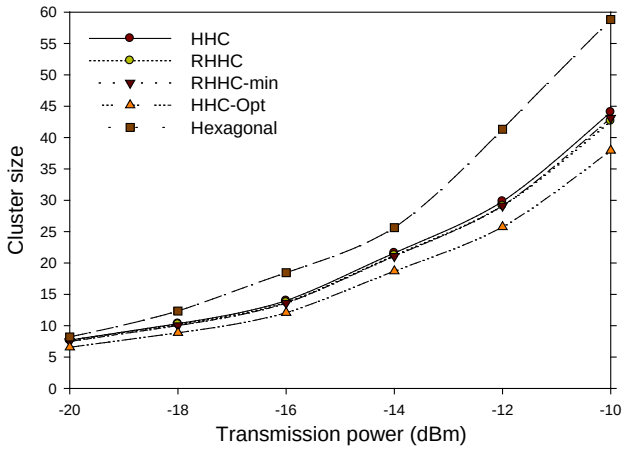


Figure 5.14 – Cluster size distribution.

Figure 5.15 show the variation in cluster size due to varying signal strength. When the signal is noisy, cluster size somewhat reduces. This effect is prominent in higher P_T values. As P_T increases clusters become larger as well as reliability of the signal strength reduces. This affects the circularity of clusters (Figure 5.9) therefore reduce the cluster size. Though reduction in cluster size due to noise is less significant, circularity reduces by 15% (Figure 5.9), which seems to be contradicting. It was later realized that this was due to a limitation in our circularity metric. Due to noise, even nodes beyond the transmission range of a CH may join a cluster during cluster formation phase. These nodes are also considered when determining the average cluster size. However, they are eliminated when calculating circularity based on Equation 4.11, where the transmission range is calculated assuming no noise. Being able to hear a cluster formation broadcast due to varying signal levels does not guarantee such a node can always communicate with the CH. Therefore, it is reasonable to discard such nodes when calculating circularity.

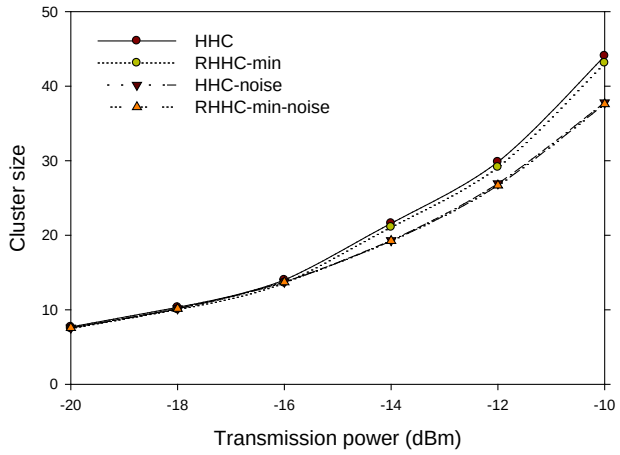


Figure 5.15 – Cluster size under uncertainties in signal strength. Random noise with zero-mean and standard deviation of *-6dBm*.

Number of nodes that are not in a cluster is shown in Figure 5.16. Node connectivity increases with the increasing P_T, therefore number of nodes without a cluster reduces. Only a few nodes in RHHC-min is without a cluster. This further suggests that RHHC-min uniformly cover the sensor field. RHHC has the highest number of disconnected nodes due to the open regions that it creates. Even for higher P_T values, more than 1% of the nodes are disconnected, in all the schemes. However, it can be seen that the cluster and cluster tree optimization phase is able to connect almost all the nodes in the network. It was also observed that the number of disconnected nodes reduces with increasing network density.

Figure 5.17 shows the total number of control messages (cluster formation broadcast and ACKs) produced by each scheme. RHHC-min produces the least number of control messages while the optimization phase produces the most. HHC and RHHC have similar overhead. Relatively lower number of ACKs from CCHs was observed in RHHC-min. A node with a higher RSSI immediately sends an ACK preventing most of its neighbors from sending their ACKs. This allows selection of geographically

distributed set of CCHs. If requested by the parent CH these CCHs are guaranteed to form a cluster. Therefore, the breadth of the cluster tree increases (Figure 5.18) and ACKs generated by CCHs are not wasted. Number of control messages increases somewhat linearly with the number of nodes in the network. This confirms that message complexity of the algorithm is *O(n)*, where *n* is the number of nodes in the network. Table 5.1 shows the control message overhead per node. RHHC-min has the lowest overhead while HHC-Opt have the highest. Overhead of HHC-Opt not only includes cluster formation overhead but also includes cluster and tree optimization overheads. However, the overhead of all these schemes are lower compared to the overhead of ACE [17]. For lower overhead HHC and RSSI based HHC forms both clusters and cluster tree while only clusters are formed in ACE.

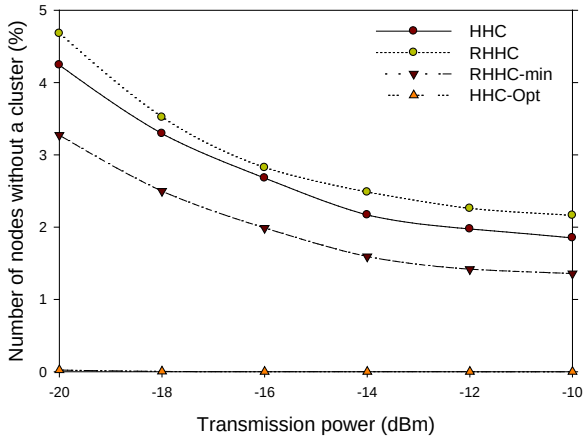


Figure 5.16 – Number of nodes not in a cluster.

It can be concluded that RHHC-min forms the best set of clusters. It ensures optimum coverage, uniform clusters, and has a low overhead. Its performance is due to its ability to push the child CHs just above 2-hops from the parent CH. Performance of

HHC is acceptable event without RSSI. The two-step, optimization phase increases the connectivity however introduce additional overhead to the GTC algorithm.

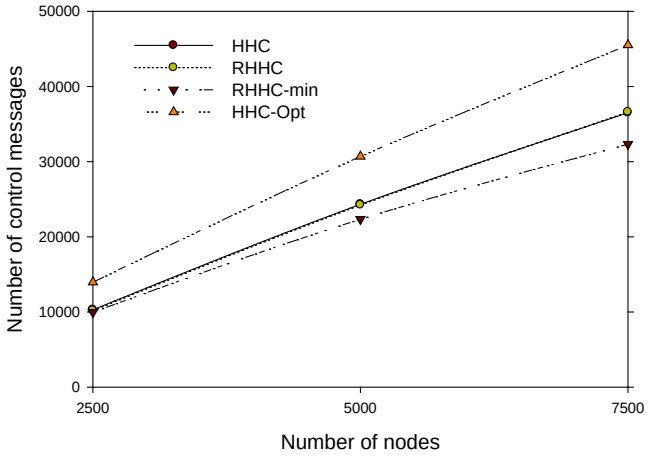


Figure 5.17 – Number of control messages. P_T = *-20dBm*.

Table 5.1 – Number of control messages per node. *R = -20dBm*, HHC-Opt uses low power to propagate cluster optimization messages.

Scheme	2500	5000	7500
HHC	4.12	4.86	4.87
RHHC	4.08	4.84	4.88
RHHC-min	3.98	4.47	4.31
HHC-opt	5.58	6.13	6.07

5.5.2 Cluster Tree Characteristics

Figure 5.18 shows the distribution of CHs in the cluster tree. RHHC-min forms a shorter tree than the other solutions. Most of the selected CCHs in RHHC-min form a cluster; therefore, it has a relatively higher branching factor. Higher branching factor reduces the depth of the cluster tree. Branching factor and initial CH distribution of RHHC-min is closer to hexagonal packing. Such a lower depth and high breadth cluster tree is desirable for most WSN applications. Table 5.2 compares the simulation results

with the depth predicted by Equations 5.5 and 5.6. For lower P_T empirical depth is within the minimum and maximum depth predicted by our model. Cluster properties are comparable with hexagonal packing for lower transmission power levels, hence indicates that our model based on hexagonal packing is valid for circular clusters. The model needs to be extended to predict the depth of the cluster tree when clusters are less circular.

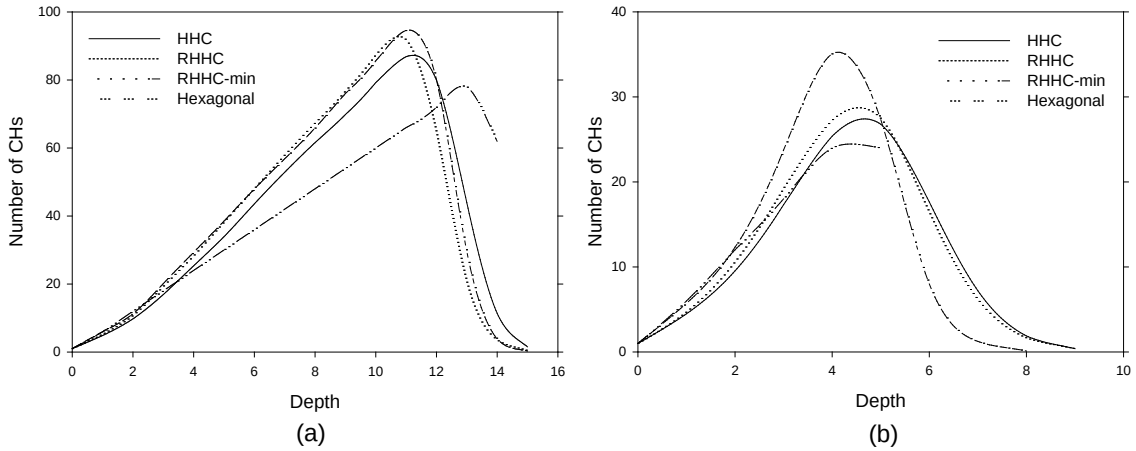


Figure 5.18 – Distribution of CHs at different levels of the cluster tree: (a) P_T = *-20dBm*, (b) P_T = *-10dBm*

Table 5.2 – Comparison of theoretical and empirical depth of the cluster tree.

Transmission Power	**Scheme**	**Depth – Theoretical**	**Depth – Empirical**
-20dBm	HHC RHHC RHHC-min HHC-Opt	13-15	15 14 14 14
-10dBm	HHC RHHC RHHC-min HHC-Opt	4-5	8 8 7 4

Figure 5.19 shows the improvement on cluster tree due to the two-step cluster and cluster tree optimization phase. The same power level that was used to build clusters is utilized during the tree optimization phase. Higher P_T values improve the cluster tree significantly (Figure 5.19(b)). As P_T increases, more and more CHs can hear the cluster

optimization broadcast allowing them to join a parent CH with a lower depth. This significantly increases the branching factor of a parent CH (rapid increase in number of CHs in Figure 5.19(b)). Workload of a CH increases if it has many child clusters hence too many child clusters are not desirable as well. Figure 5.20 shows the physical shape of a cluster tree formed by one of the data samples. For simplicity only CHs and nodes without a cluster is indicated. Figure 5.21 shows the same cluster tree after the optimization phase. The new cluster tree is more structured than the original one. Note that nodes that were not in a cluster are now connected to an existing cluster or have formed their own clusters. Though optimization phase incur some overhead and diminish cluster properties such as circularity and cluster size it significantly improves the branching factor, reduce the depth of leaf nodes, and average depth of a node. Such cluster trees are important in latency bound applications.

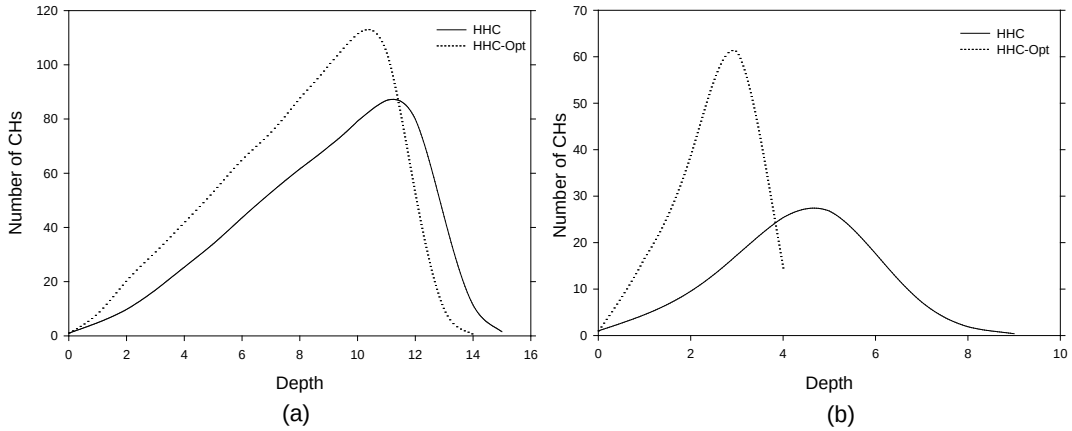


Figure 5.19 – Cluster tree improvement with optimization phase: (a) P_T = *-20dBm*, (b) P_T = *-10dBm*.

Figure 5.22 shows the ability of the HHC clustering scheme to form a clustered network even in a sensor field with a large open region. Such open regions can occur if nodes are not placed in a particular region, if all the nodes placed in the region fails

(unlikely), or due to barrier like a concrete wall. Because of the open region, two of the child CHs of the root node in Figure 5.22 is unable to extend their branches. Nevertheless, other branches of the tree were able to go around the open region and cover the entire sensor field.

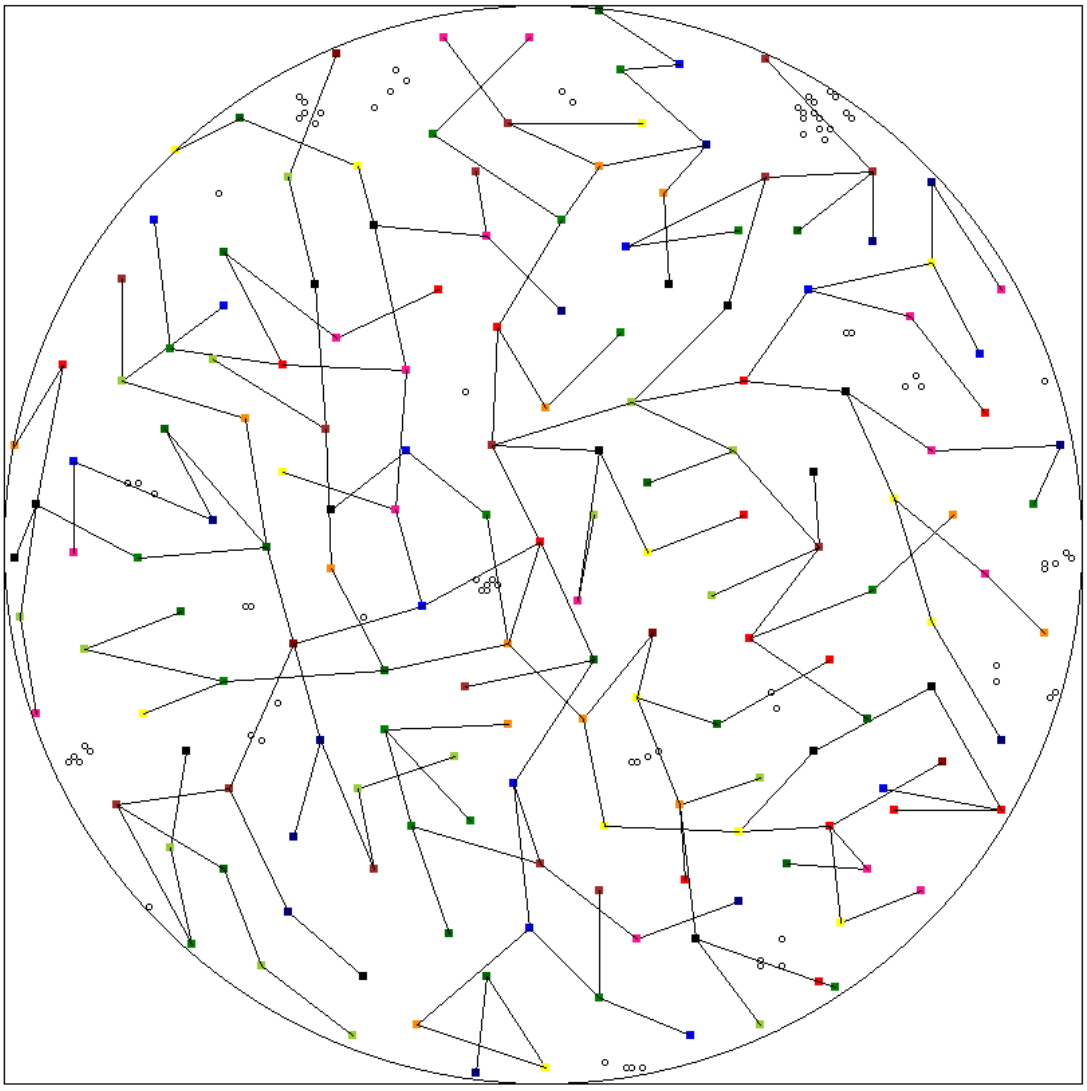


Figure 5.20 – Physical shape of the cluster tree before the optimization phase. HHC cluster formation approach. P_T = *-12dBm*. Shaded circles – sensor nodes, Shaded squares – CHs, Circles – nodes without a cluster.

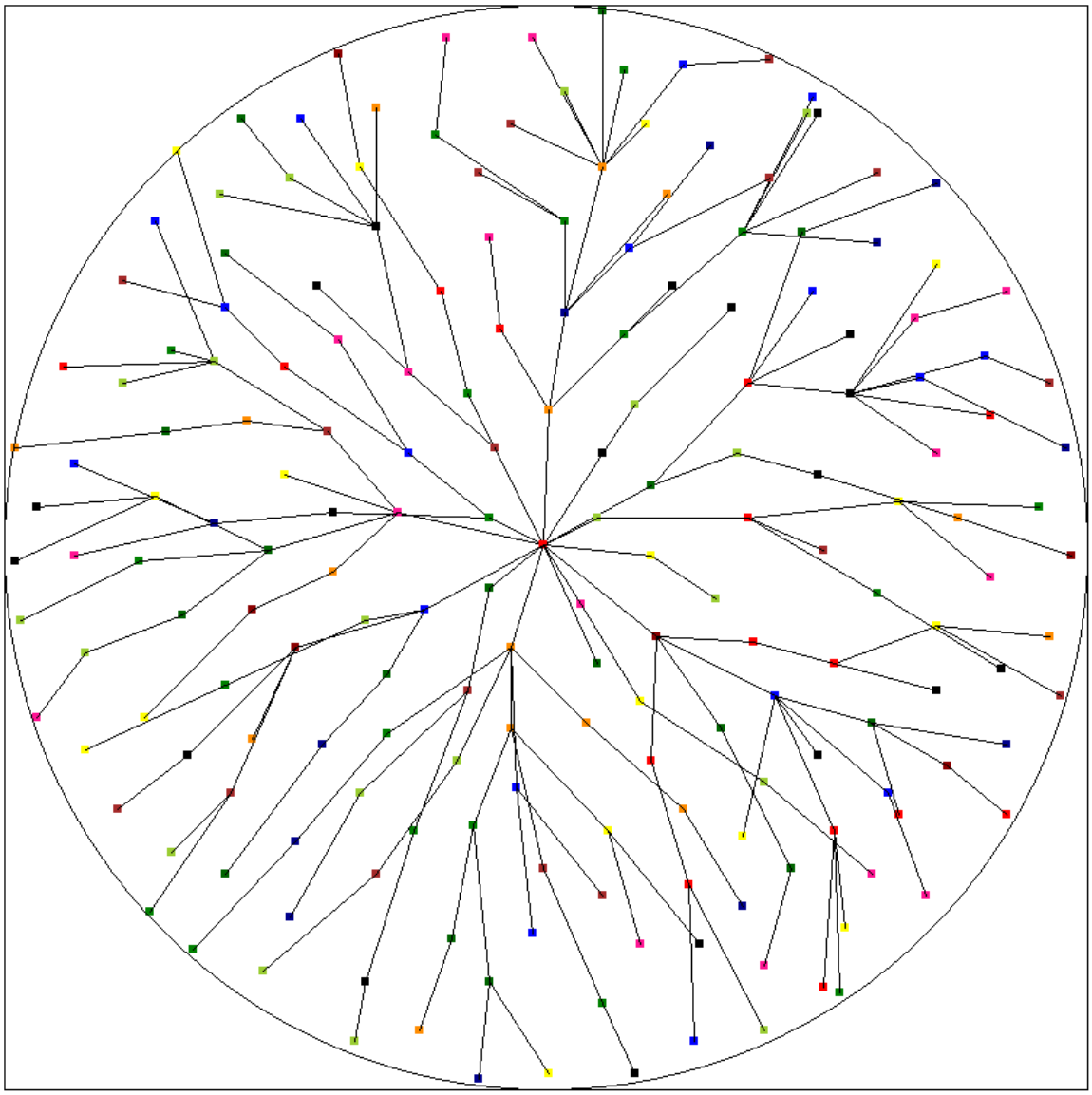


Figure 5.21 – Physical shape of the cluster tree after the optimization phase. HHC cluster formation approach. P_T = *-12dBm*. Same P_T for cluster tree optimization phase. Shaded circles – sensor nodes, Shaded squares – CHs, Circles – nodes without a cluster.

Figure 5.23 shows the distribution of clusters in the sensor field. Different colors indicate the depth of a cluster in the cluster tree. It can be seen that the clusters of the same depth tend to arrange in a structure somewhat similar to a ring. However, the ring may not be fully connected. This information may be useful in sensor localization or when a circular path needs to be formed within the network. To determine any possibility

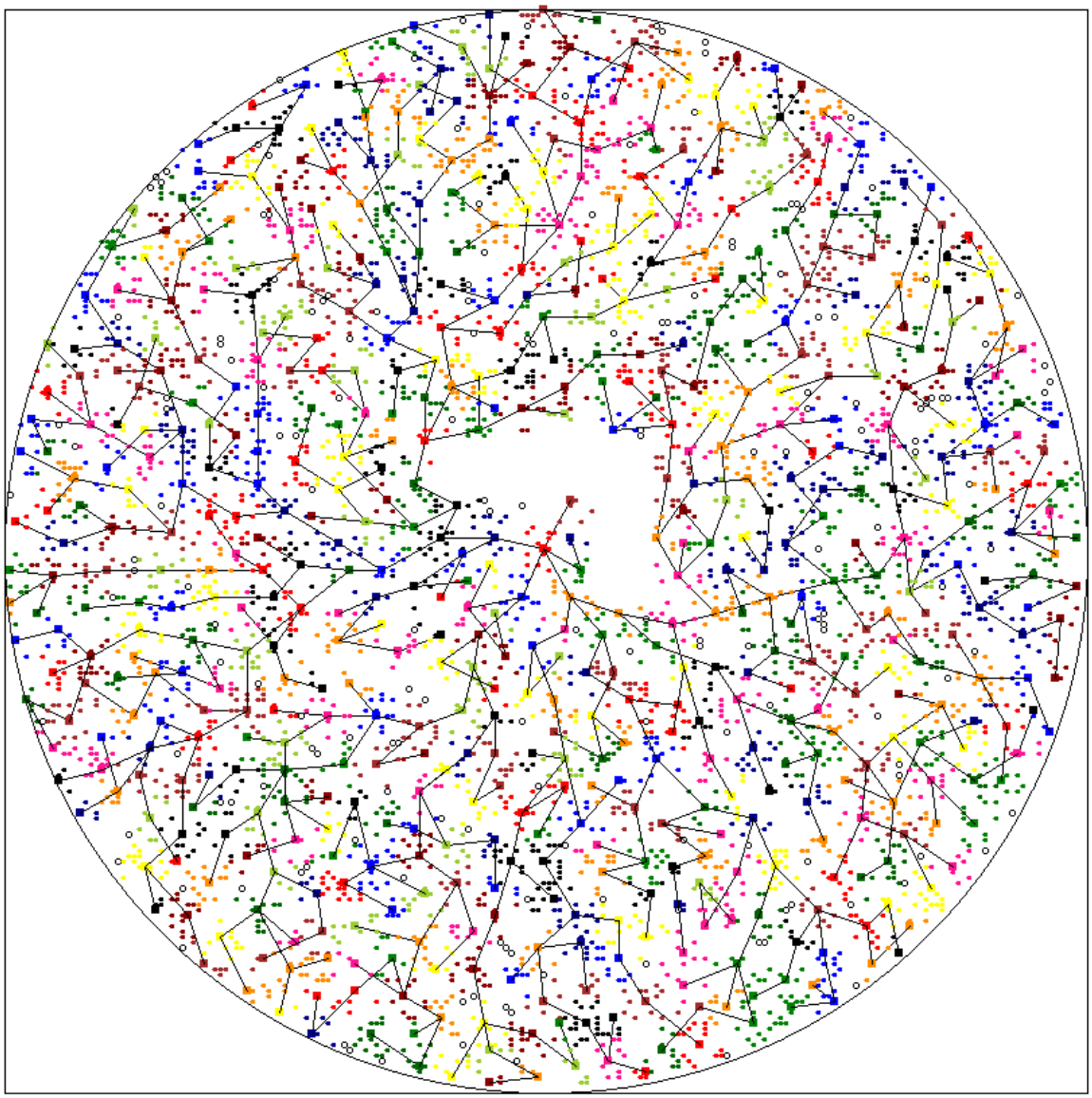


Figure 5.22 – HHC cluster formation in a network with an open region. P_T = *-20dBm*. L shaped open region (*200m×50m + 50m×150m*). Shaded circles – sensor nodes, Shaded squares – CHs, Circles – nodes without a cluster.

of localizing clusters, distance to each CH from the root node is plotted in Figure 5.24. Based on the figure localization accuracy is around ±*50m*. Availability of RSSI increases the accuracy to ±*35m* and cluster optimization phase increases it up to ±*25m*. Even ±*25m* accuracy is not adequate for a localization technique therefore the scheme is not suitable.

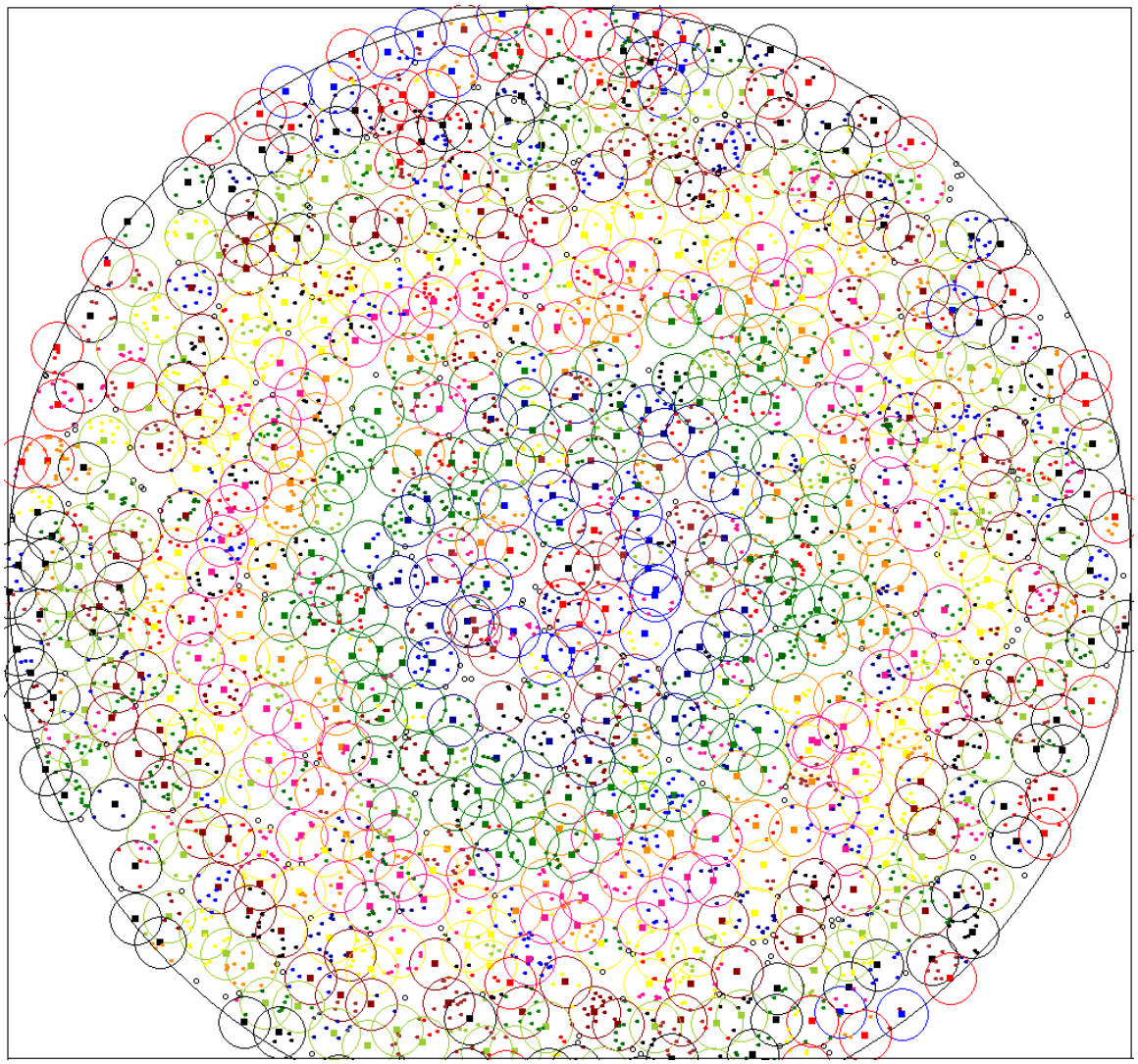


Figure 5.23 – Distribution of cluster in the sensor field. P_T = *-20dBm*. Colors indicate the depth of a cluster in the cluster tree. Shaded circles – sensor nodes, Shaded squares – CHs, Smaller circles – nodes without a cluster, Larger circles – Coverage area of a CH.

The ability to form a partial ring is exploited in one of our routing schemes, which will be described in the next chapter.

5.6 Summary

The GTC algorithm is further extended to reduce collisions and enable the selection of a better set of CCHs. RHHC-min scheme that pushes the cluster formation

message to the maximum distance during the first 2-hops and then selects a nearest node out performs the other solutions. It uniformly covers the sensor field and produces a lower depth tree. Our HHC scheme, without any RSSI values, also performs well. The two-step, optimization phase reduces the number of disconnected nodes and improves the cluster tree. Such an improved cluster tree would be beneficial in long-lived WSNs even though the optimization overhead is somewhat high. In the next chapter, we make use of the cluster tree produced by HHC scheme to deliver messages in large and collaborative sensor networks.

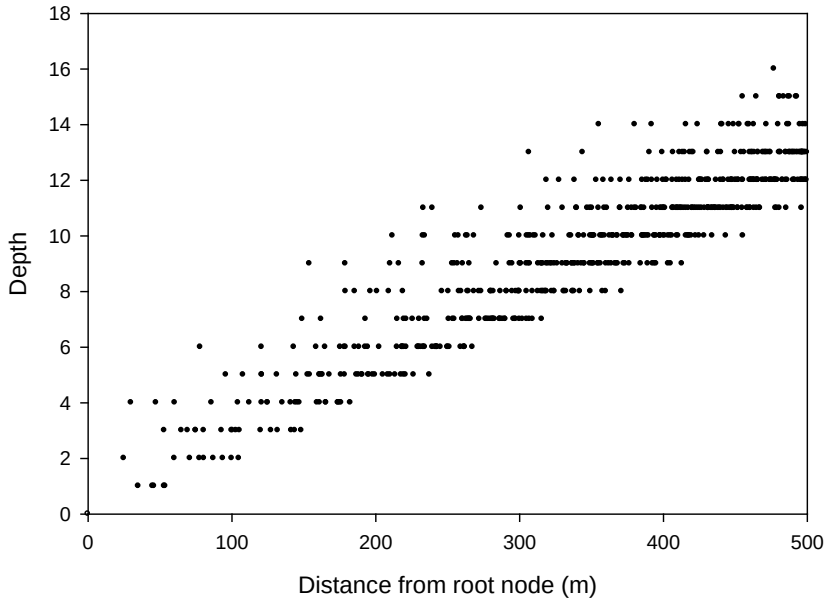


Figure 5.24 – Distance vs. depth in the cluster tree. P_T = *-20dBm*, HHC cluster formation. Data is based on a single sample.

Chapter 6

ROUTING

Routing protocols are highly influenced by data-centric nature of wireless sensor networks. Attributes based naming and many-to-one, e.g., node-to-sink, communication model are two of the key attributes of conventional WSNs that are dedicated to a specific task. In addition to those attributes collaborative WSNs may require to facilitate many-to-many (CH-to-CH or VSN-to-VSN) communication model and a logical addressing scheme. A cluster tree can be used to facilitate unicast, multicast, and broadcast traffic in such networks. A logical addressing scheme that reflects the hierarchical relationship of parent and child CHs can be used to determine appropriate routing paths. The chapter presents a hierarchical addressing scheme and three routing mechanisms that are developed on top of the cluster tree formed with the HHC scheme of the GTC algorithm.

Section 6.1 introduces the concept of cluster tree based routing and hierarchical addressing. A routing mechanism that makes use of cross-links within the cluster tree to enhance the network capacity is presented in Section 6.2. Section 6.3 presents another routing mechanism that makes use of a circular path within the network. The optimum placement of the circular path is derived using an analytical model. Performance analysis is presented in Section 6.4.

6.1 Cluster Tree Based Routing

Collaborative WSNs require communicating with the base station as well as within the network. The cluster tree formed with the HHC scheme is used to facilitate these communication models. It is assumed that two CHs that wish to communicate with each other at least know the destination address, through some other mechanism. If the communication mode is node-to-sink, nodes anyway know address of the sink hence this constrain is not necessary. Next chapter presents a mechanism to identify addresses of CHs that are in the same or different VSNs.

Figure 6.1 shows a hypothetical cluster tree formed with ideal HHC clustering. Assume that the root node is either the sink or capable of forwarding messages to the sink. Under this condition, events/messages generated by any of the nodes in the network can be easily delivered to the sink through the cluster tree. For example, consider a message that originates from one of the members of *P*'s cluster. The cluster member first

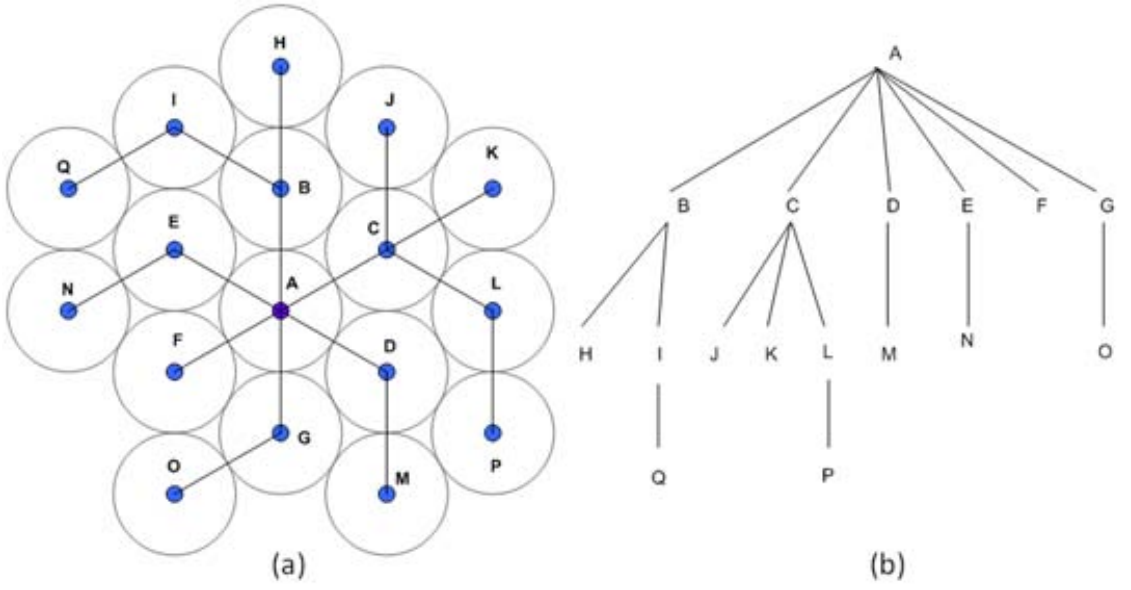


Figure 6.1 – A hypothetical cluster tree formed with HHC clustering. *A*-*Q* – CHs, *A* – root node, lines indicate the parent-child relationship among CHs.

forwards the message to *P*. Then the message will be forwarded *L*, which is the parent CH of *P*. The message will be further forwarded to *C* and from *C* to *A*, which is the final destination of the message.

However, this approach will not work if nodes or CHs in different branches of the cluster tree want to communicate. For example, assume that *J* wants to communicate with *P*. The message is first forwarded to *C*, as it is the parent CH of *J*. *C* only knows about its parent CH *A* and its child CHs *J*, *K*, and *L*. Then *C* will forward the message to *A* because it does not know anything about *P*. However, even CH *A* does not have any information about *P*. Therefore, *A* has to either drop the packet or try to send it through a randomly selected branch. This problem can be overcome by each CH keeping track of all its descendants. If *C* was aware of *P*, it could have directly forward the message to *L*, which will then forward the message to *P*. As we go up the hierarchy, more and more data about descendant CHs needs to be stored. The root node has to store data about the entire network; therefore, this approach is not scalable. A logical addressing scheme that reflects the parent-child relationship among CHs can overcome this issue.

6.1.1 Hierarchical Addressing

A hierarchical address can be assigned to a child CH based on the address of the parent CH and child's branch number. Such an addressing scheme for the previously discussed cluster tree is shown in Figure 6.2. Branch numbers of a child CH are determined based on the order that CCHs are selected. The first CCH that is selected to form a cluster is considered to be in branch *0*, the second one is in branch 1, and *n*-th CCHs is in branch *n – 1*. Branch numbers are always relative to the parent CH. If one of

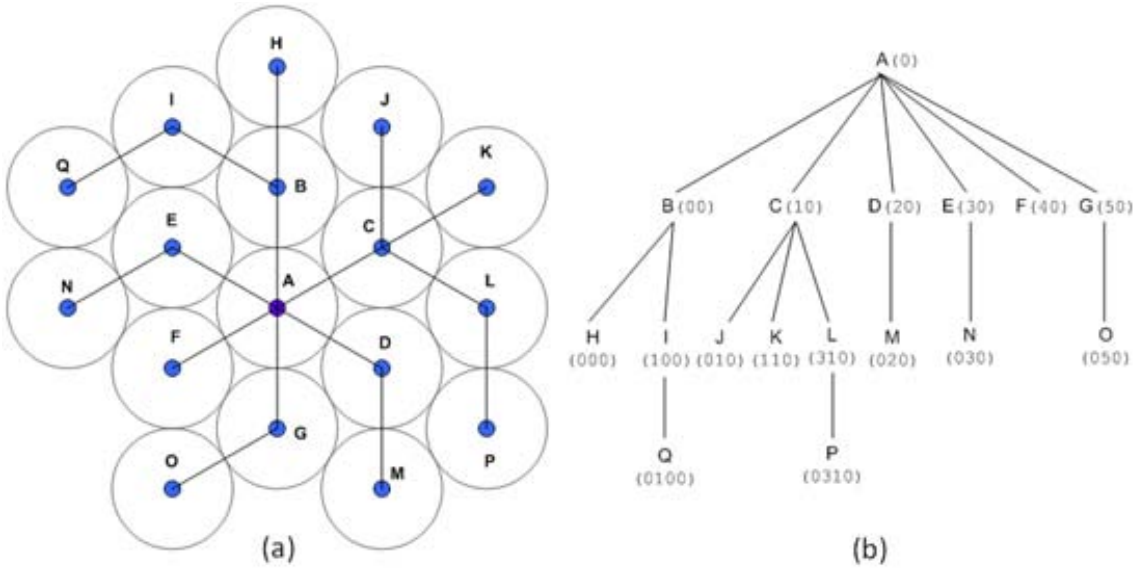


Figure 6.2 – A hypothetical cluster tree labeled with hierarchical addresses.

the select CCHs is unable to form its own cluster, particular branch is assumed to be discontinued. The root node (*A*) initiates the cluster and tree formation process and does not have any parent CHs, hence its Cluster ID (CID) is assumed to be *0*. *B* is the first child CH of *A*; therefore, its address is *00*. Then the second child *C* is assigned address *10*. Root node selects up to six CCHs. *G* is the last child CH of the root node therefore gets the address *50*. When *B* assigns addresses to its child CHs it merges its hierarchical address *00* with the child's branch number. *H* is the first child CH of *B* hence assigned the address *000*. Address *100* is assigned to *I* as it is the second child CH. *H* does not have any child CHs therefore the branch related to *H* does not span any further. *Q* is the one and only child of *I* hence assigned the address *0100*. The third child CH of *C* was unable to form a cluster; therefore, address *210* is not assigned to any child CH. *L*, which is the fourth child CH, is assigned the address *310*. These hierarchical addresses are calculated by the parent CH and send to the selected CCHs using the ***Request_Form_Cluster*** function. If a parent CH realizes that, some of the selected CCHs

were unable to form a cluster it may reuse the hierarchical addresses assigned to those nodes. For simplicity, we assume that addresses are not reused.

For routing purposes, each CH needs to keep track of the hierarchical addresses of its parent and child CHs. Root node only needs to keep track of its immediate child nodes. Therefore, hierarchical addresses significantly reduce the number of routing entries that needs to be stored in a node. Given a hierarchical address of a destination, the entire path to the destination can be reconstructed. In practice, only the next hop needs to be determined by a CH. The pseudo code given in Figure 6.3 can be used to determine the next hop. For simplicity let us assume a hierarchical address to be an array of digits with the Least Significant Digit (LSD) indicating the branch number of the root node, which is always *0*. The input variable *current* indicates the hierarchical address of the CH that is trying to discover the next hop and *destination* is the hierarchical address of the destination CH.

```
Next_Hop(current, destination)
 1  IF(current = destination)
 2        Return current
 3  min_length ← Min(Size(current), Size(destination))
 4  FOR i = 0 TO min_lenth -1
 5        IF(current[i] ≠ destination[i]
 6              Break
 7  IF(i  < Size(current))
 8        Return parent_CH
 9  ELSE IF(i = Size(current))
10        Return destination[i]
```

Figure 6.3 – Pseudo code to determine next hop.

For example, consider a case where CH *L* wants to communicate with CH *M*. *L*'s address is *310* while *M*'s address is *020*. As the source and destination addresses are different, individual digits of the two addresses need to be compared to determine the

next hop. Both *L* and *M* has three digits hence *min_length = 3*. Each digit is compared starting from the LSD (line 4). When compared, LSD of both address are zero. Then the next least significant digit is compared. In this case it is not a match (*1 ≠ 2*) hence the *for* loop terminates. Digits that match indicate the common meeting point of the two branches, for *L* and *M* it is the root node. By the time the *for* loop terminates the variable *i* indicates the number of digits that matched. For *L* and *M*, *i = 1*. If the number of digits that match is less than the size of the current address (*i < Size(current)*) then the common meeting point is above the current CH. Therefore, the message needs to be forwarded to the parent CH (line 8). For *L* and *M*, (i = *1 < 3*), therefore the message will be forwarded to the parent CH *C*. Similarly, *C* will compare its address with the address of *L*. It will determine that its parent CH is the best node to forward the message therefore sends it to *A*. When the root node tries to determine the next hop, it will realize that it is the only common point, i.e., *i = 1*. When number of matching digits are same as the size of the current address, i.e., *i = Size(current)*, the next hop should be one of the child CHs. This assumption is valid given that current and destination addresses are different, i.e., lines 3 to 10 never executes if condition in line 1 is satisfied. The next digit after the common portion of the destination address (digit *i*) indicates the branch number of the child CH. When *0* and *020* is compared, common portion of the address is *0*. Therefore, the child CH's branch number is *2*, which corresponds to CH *D*. Consequently, the message will be forwarded from *A* to *D*. At *D*, addresses *20* and *020* are compared. Addresses are identical up to *20* hence next hop should be the *0*-th branch. Therefore, the message is forwarded from *D* to *M*, which is the destination.

As shown in Figure 6.2(b) hierarchical addresses do not need to be of the same size. Therefore, the length of a hierarchical address can be fixed or variable. Fixed size addresses are easy to deal with however waste memory if useful portion of the address is small. Alternatively, variable length addresses reduce the size of an address however increase the decoding complexity. For example, a cluster tree with a maximum depth of five needs 6-digit hierarchical addresses, given that root node's address is *0*. However, level *1* CHs need only 2-digits to represent their address. Therefore, 4-digits are wasted. Such large addresses increase the size of a packet header therefore can significantly affect the performance of WSNs. Wireless sensor nodes consume significant amount of energy even to send a single bit. However, thousands of instructions can be executed for the same energy. Therefore, it is worthwhile to use variable length addresses although it somewhat increases the complexity of the addressing scheme.

Figure 6.4 illustrates the design of our variable length hierarchical addressing scheme. The two-part address includes a variable length hierarchical address and a fixed length depth. Given the depth, the size of the address portion can be determined. The depth of the root node and its address is always zero. Therefore, the root node can be indicated by an address with $depth = 0$ (Figure 6.4(a)) and no address portion required as it is always zero. A d-bit depth, allows us to address a cluster tree with a maximum depth of $2^d - 1$. Number of bits required to represent a branch of the cluster tree (i.e., digit) depends on the branching factor b. Hence, $log_2 b$ bits are required to represent a branch. Figure 6.4(c) shows the format of a level *1* hierarchical address with a branching factor of *4*. The format of a level *2* hierarchical address is shown in Figure 6.4(d). CHs that are closer to the root node have shorter addresses while CHs that are further away have

longer addresses. As we discussed in Section 5.4 branching factor reduces as we go down the cluster tree. This information can be utilized to further reduce the size of a hierarchical address, i.e., as *b* reduces $log_2 b$ reduce. A cluster tree with a maximum depth of *d* and breadth of *b* therefore has a maximum address portion of $log_2 b(2^d-1)$ bits. Therefore, the size of an address vary from *d* to $log_2 b(2^d-1) + d$ bits.

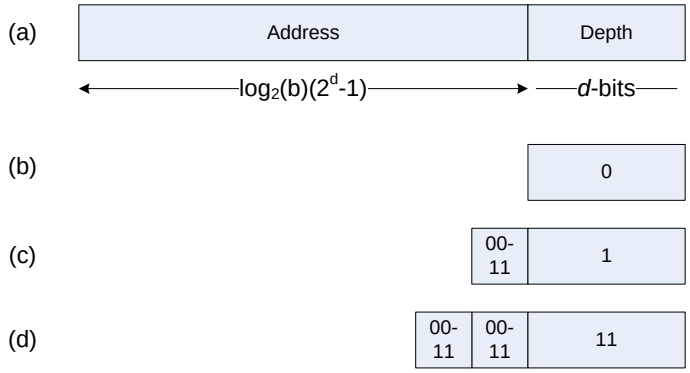


Figure 6.4 – Variable length hierarchical addresses: (a) – two-part address, (b) – address of the root node, (c) – address of a level 1 CH, (d) – address of a level 2 CH. *d* – depth and *b* – branching factor.

6.1.2 Addressless Routing

It is not essential to have a logical addressing scheme to communicate across different nodes within a cluster. Depending on the application scenario, it may be possible to select other alternative approaches. Figure 6.5 shows a hypothetical cluster tree that connects heterogeneous devices.

In case of fire, smoke detectors or manual fire alarm controls need to communicate with the fire alarms. However, they may not know the addresses of fire alarms or anything about the branches that include those alarms. It is possible to facilitate node-to-node communication within this network without any addressing scheme or parent CHs keeping track of all the descendants. If each parent CH can keep track of the

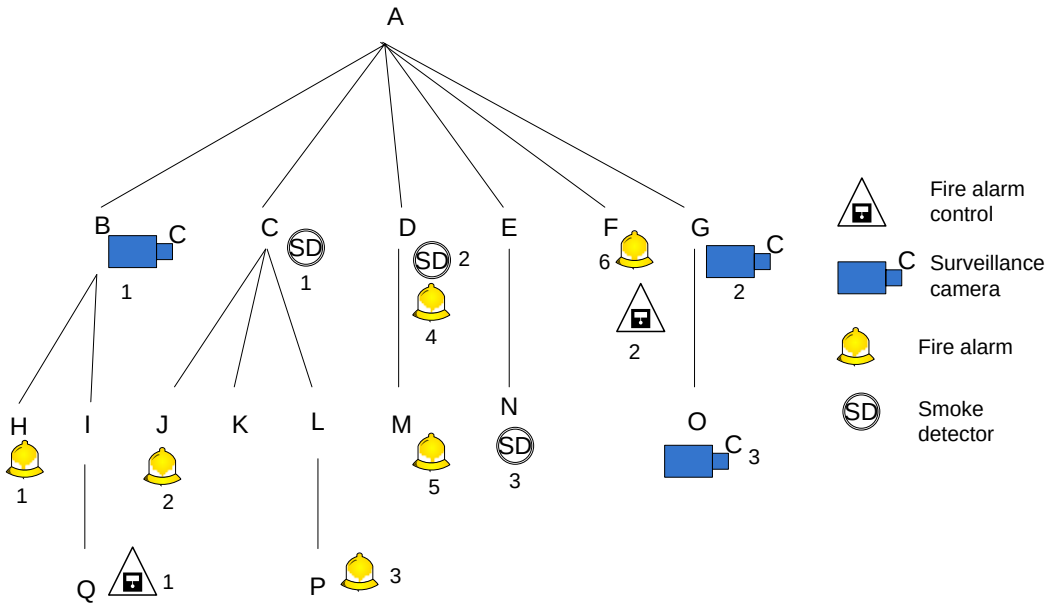


Figure 6.5 – A cluster tree that connects heterogeneous devices.

device types (camera, alarm, smoke detector, etc.) of their descendants, such a communication model can be facilitated. Whenever a device comes up, it informs its parent CH about the device type. For example, CH *P* informs CH *L* that it has an alarm. CH *L* caches this information and informs CH *C* about an alarm (just saying one of my descendants has an alarm). *C* also caches this information and informs *A* that it knows about an alarm and a smoke detector. When *J* informs *C* about its alarm, *C* caches that information. However, it will not again inform *A* about an alarm because it has already done so. Similarly, *D* will get to know about the alarm at *M*. *D* will further forward message about an alarm and a smoke detector to *A*. This scheme is efficient because only information about devices types and related branches are cached at each node.

Let us see how this cluster tree is useful in delivering events to actuators. Whenever a device detects an event, it will send a message towards the root node. If the smoke detector at cluster *D* detects a fire, it will send a message towards the root node. *D*

will also send a copy of the message to *M* as it knows about the alarm at *M*. When the message reaches the root node, it needs to forward the messages to branches with alarms. Hence, a copy of the message will be forwarded to child CHs *B*, *C*, and *F*. When the message reaches *B*, it will realize that its child CH *H* has an alarm therefore forwards the message to *H*. Similarly, the message is forwarded to CHs *J* and *P*.

6.1.3 Relative Branch Number Based Addressing

However, this scheme does not work if smoke detectors want to communicate only with a subset of alarms. Assume that alarms at CHs *H* and *J* are interested in smoke detector at CH *C* while CHs *D*, *F*, *M*, and *P* are interested in smoke detectors at CHs *D* and *N*. The alarms can make use of an event/query propagation mechanism similar to Rumor Routing [14] or path detection mechanism similar to Ant Routing [35], to inform about their interest and subscribe to the respective fire detectors. They can perform a random walk within the cluster tree until required device is reached. Like the previous approach, if parent CHs are aware of the device types of its descendants the random walk can be made more deterministic. In Rumor and Ant routing as the agents/ants travel through the network they accumulate and carry information about all the visited nodes. This list of nodes is called the *visited list*. Consider a network with 16-bit node IDs as proposed in IEEE 802.15.4 standard [38]. An agent that travels in this network will accumulate 100-bytes of extra data when it travels 50-hops (50-hops is typical for long-lived agents in Rumor Routing). This extensive overhead hinders the performance of Rumor Routing. Knowledge about parent-child relationships in cluster tree based networks can significantly reduce this overhead.

An alternative cluster tree-labeling scheme based on branch numbers is shown in Figure 6.6. Each CH assigns a relative branch ID to its parent and child CHs. Parent CH is always given branch ID *0* while child CH branch IDs starts from *1*. Consider a case where alarm *3* at CH *P* is interested in a smoke detector labeled *3* (Figure 6.5). CH *P* does not know that the smoke detector *3* is at CH *N* therefore sends a long-lived agent(s) to figure out a possible path to the third smoke detector. For simplify of the discussion, we will consider only a case where the agent is able to figure out a path to the destination. The agent is sent to the parent CH *L* because it is only the branch related to *P*. When *L* receives the agent, it realizes that it was sent by child CH *P* and adds *P*'s branch ID to the *visited_list*. Then *L* forwards the agent to parent CH *C*. *C* appends the *visited_list* with *L*'s branch ID which is *4*. Then *visited_list = 14*. When the agent reaches the root node, the updated *visited_list* is *142*. The root node then forwards the agent to CH *E*, which is its fourth child. *E* realizes that the agent came from the parent CH A, i.e., root node, therefore append *0* to the *visited_list*, i.e., *visited_list = 1420*. Finally, the agent is

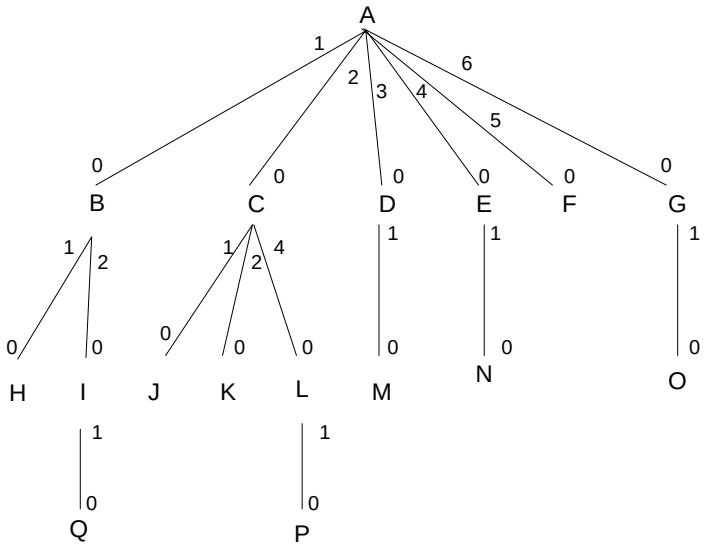


Figure 6.6 – Alternative cluster tree labeling scheme based on branch numbers.

forwarded to *N*, which is the CH with the third smoke detector. The final *visited_list = 14200*. By the time the agent reaches *N*, it has travelled 5-hops.

If the third smoke detector detects a smoke, it will try to inform about the event to the alarm at CH *P* using the *visited_list*. Given the *visited_list* and the hop count, the path to the destination can be reconstructed. First the event message is forwarded to the parent CH *E* because the last visited branch is *0* (i.e., *visited_list = 1420**0***). Event messages carry the *visited_list* with them to determine the path to the destination. At CH *E*, the last visited branch is removed from the *visited_list* therefore the new *visited_list = 1420*. *E* realizes that the last branch in the *visited_list* belongs to its parent CH therefore forwards the event to *A*, i.e., root node. After removing the entry related to *A*, the new *visited_list* is *142*. Then root node figures out that the next branch is its second child CH therefore forwards the event to CH *C*. Similarly, CH *C* removes its entry from the *visited_list* and sends it to its fourth child CH *L*. Finally, the event will be forwarded from *L* to *P*, which is the destination. Though this scheme adds some complexity while constructing and decoding the *visited_list*, it significantly reduces the size of the *visited_list*. For example, our HHC scheme has a maximum branching factor of six therefore each CH has to keep track of seven branches (*6* + parent CH). Only 3-bits are required to uniquely address these branches instead of the 16-bit node IDs hence this addressing scheme has significantly lower overhead.

The same concept can be extended to non-hierarchical WSNs by assigned relative node IDs to neighbors. Instead of *16*-bit addresses, log_2*(no of neighbors)*-bit addresses can be used to identify neighbors of a particular node. Because most WSNs are sparse or moderate log_2*(no of neighbors)* will be much smaller than *16*-bits.

This branch based labeling scheme cannot detect loops. By looking at the *visited_list* it is not possible to identify whether the agent has gone through the same CH creating a loop. Let us assume the agent that search for the third smoke detector was forward to CH *F* by the root node. When the agent reaches *F,* it modifies the *visited_list* to *1420*. CH *F* cannot send the agent any further because it does not have any child nodes hence sends it back to its parent CH. Then the new *visited_list = 14205*. If the root node then forwards the agent to CH *E*, new *visited_list* is *142050*. By looking at this *visited_list* it is not possible to determine that the agent has gone through the root node twice. Similar to Rumor Routing this problem can be prevented by each CH caching the agent information. CHs need to cache agent ID and number of hops the agent took to reach the particular CH for the first time. Given these information loops in the *visited_list* can be removed. Due to time constrains no performance analysis is performed on this addressing scheme.

6.2 Cross-links Based Routing

The root node becomes a single point of failure in hierarchical WSNs. Because it has to deliver most of the traffic to the sink or across different branches, it will die much faster than other CHs in the network. If the root node is not energy constrained its child CHs will become a bottleneck. Child CHs will die faster as they have to share the traffic going through the root node. It has also been proposed to put multiple high power nodes closer to the root node so that they can handle more traffic [60]. Neither of these approaches effectively makes use of the energy available in rest of the nodes/CHs in the network. We propose two routing approaches that make use of the energy available in

other CHs, to a certain extent. The first approach makes use of cross-links within the cluster tree to enhance the network capacity.

Figure 6.7 shows a cluster tree with cross-link among neighboring CHs. In a clustered network neighboring clusters may belongs to different branches of the cluster tree. For example, CHs *H* and *J* belong to completely different branches of the cluster tree. As they are in the same neighborhood, they should be able exchange each other's data and figure out their hierarchical addresses. Such neighboring clusters can form cross-links within the cluster tree. These cross-links can be used to deliver messages across different branches of the cluster tree without going through the root node. For example, consider a case where CH *H* wants to communicate with CH *K*. When the message travels through the cluster tree it will use the path $H \rightarrow B \rightarrow A \rightarrow C \rightarrow K$, which is 4-hops long. If *H* knows that its neighboring CH *J*, is in the same branch as *K* it can use *J* to relay the message. Then the new path will be $H \rightarrow J \rightarrow C \rightarrow K$ and it has a distance of 3-hops. The new path is short and does not go through the root node. Ability to form many cross-links reduces the workload on the root node. Circularity of actual clusters is

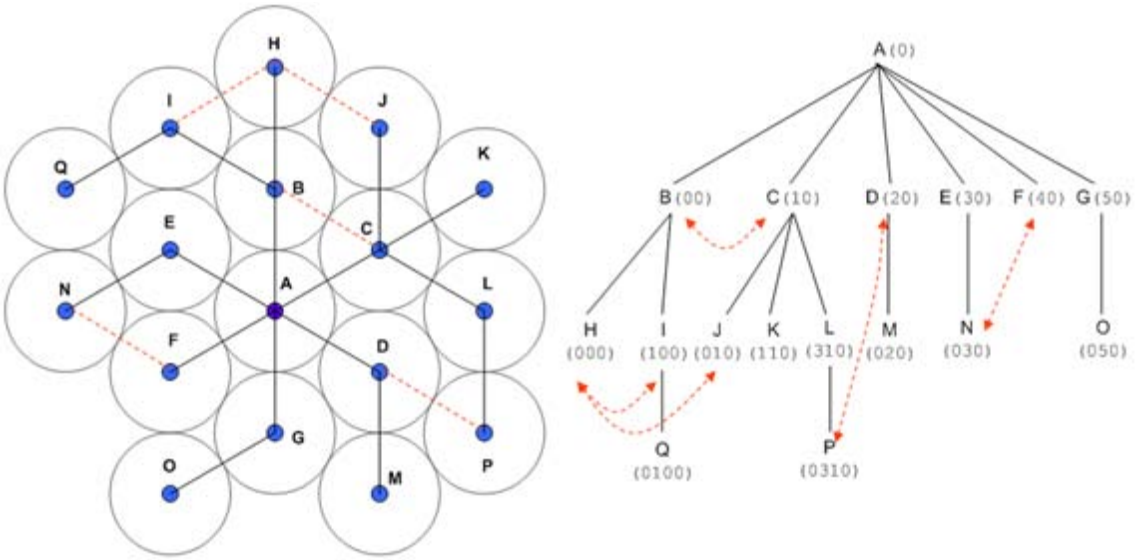


Figure 6.7 – A cluster tree with cross-links among neighboring CHs.

lower; therefore, several clusters are formed in the same neighborhood. This allows many cross-links to be formed within the cluster tree. This scheme utilizes the energy available in CHs that are located at different levels of the cluster tree.

It is not a good option to blindly go through cross-links just because they are available. We can use our hierarchical addressing scheme to determine whether going through one of cross-links or going through the root node is shorter. Given address of CHs *H* (*000*) and *K* (*110*) it can be determined that they meet only at the root node. CH *H* has 2-hops to the root node while *K* also has 2-hops therefore the total distance is 4-hops. If *H* compares *J*'s address (*010*) with *K*'s address (*110*) they meet at a CH with address *10*, i.e., hierarchical address *10* is common in both *J*'s and *K*'s addresses. Therefore, both *J* and *K* have 1-hop to the CH with address *10*. CH *H* also needs to forward the message to *J*, which requires another hop. Altogether, 3-hops are required. Therefore, path through neighboring CH *J* is shorter than going through the root node.

CHs may discover their neighbors actively or passively. In active neighbor identification, each CH sends a broadcast indicating hierarchical cluster address to its neighbors. The CHs may use higher power (single-hop) or low power (multi-hop) transmission to send the broadcast. Our cluster tree optimization phase can also be used to discover neighboring addresses. Instead of specifically sending their hierarchical addresses, CHs may passively listen to the messages send by their neighbors to determine neighbor's hierarchical addresses. Passive listening is slow and requires packet header to include the source address.

6.3 Circular Path Based Routing

It was observed that clusters of the same depth tends to be somewhat localized and lie on a ring like structure (Figure 5.23). This information can be used to build one or more circular paths in the network. These circular paths can be used to forward messages without being tied to the cluster tree. Figure 6.8 shows a cluster tree with a circular path. The path is formed by connecting level 2 CHs that are in the same neighborhood. CHs along the circular path share their hierarchical addresses. Instead of sharing cluster information only with the neighbors, CHs share their addresses with neighbors of neighbors that are at the same depth hence this is an extension of the cross-links based routing. Neighbor address discovery is similar to the route sharing in Routing Information Protocol (RIP).

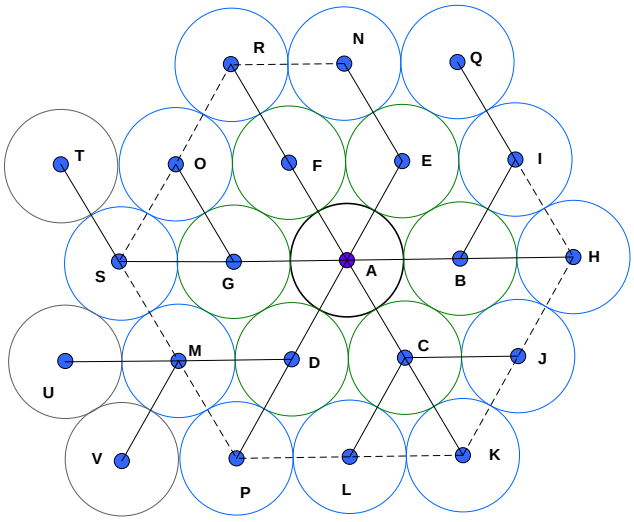


Figure 6.8 – A cluster tree with a circular path. Path is formed by connecting all the level 2 CHs that are in the same neighborhood.

For example, assume that two CHs *U* and *K* want to communicate with each other. A message going from *U* to *K* has to travel 5-hops if it goes through the cluster tree. By utilizing the cluster tree and the circular path, it is possible to send the message

through 4-hops. *U* is not part of the circular path therefore; it uses the cluster tree and forwards the message to its parent CH *M*. Because *M* is part of the circular path, it has the option of selecting either the circular path or the cluster tree. It will take 4-hops to sends the message through the cluster tree. Sending the message through the circular path requires only 3-hops. Therefore, *M* sends the message through the circular path using CHs *P* and *L*. However, to do this *M* has to know about CH *K*'s address through neighboring CHs *P* and *L*. We name this approach as circular path based routing.

It is not useful to share each CHs address with all the other CHs in the circular path. For example, consider two CHs *H* and *P* that are at a depth of two. It requires 4-hops for them to communicate using either the cluster tree or the circular path. Going through the circular path is preferred as it reduces the burden on the root node. Therefore, it is useful for *H* and *P* to know about each other. Consider *P*'s neighbor *M*. It takes 4-hops for *H* and *M* to communicate through the cluster tree while 5-hops are needed to communicate through the circular path. Therefore, this information is not useful and *H*'s address should not be propagated any further. Hierarchical addresses are required to determine the best path and to make sure only useful addresses are forwarded to neighbors. Depending on the position of the circular path, i.e., at which depth this path is formed, the fraction of messages that goes through the cluster tree and the circular path varies. Optimum position of the circular path is determined using an analytical model. The analytical model varies depending on whether we are interested in minimizing energy or maximizing network lifetime. The case of minimizing energy is analyzed first.

If all the nodes use the same transmission power, energy to transmit a message is proportional to the number of hops and number of hops is proportional to the distance.

$$
\begin{aligned}
\textit{Energy to transmit} &= E[\textit{Energy to transmit}] \\
&= E[\textit{Energy per hop} \times \textit{hops}] \\
&= E\left[\textit{energy per hop} \times \frac{\textit{distance}}{\textit{transmission range}}\right] \\
&= \frac{\textit{energy per hop}}{\textit{transmission range}} \times E[\textit{distance}]
\end{aligned}
$$

If the same transmission power level is utilized, both transmission range and energy to send a message is constant across the entire network. Therefore, expected distance travel by a message needs to be minimized to minimize energy.

Figure 6.9 shows two nodes that are trying to communicate in a circular sensor field. The source node is placed at a distance r_1 from the root of the cluster tree and the destination node is placed at r_2. All the messages have to go through the cluster tree. For simplicity, communications that take place without going through the root node is not considered. Therefore, a message travels a distance of $r_1 + r_2$. For the circular region:

$$E[d] = \iiint d(r_1, r_2, \theta)\, p(r_1, r_2, \theta)\, dr_1 dr_2 d_\theta \tag{6.1}$$

where $d(r_1, r_2, \theta) = r_1 + r_2$ (6.2)

To determine $p(r_1, r_2, \theta)$, let us consider a small circle drawn at r with a thickness of Δr

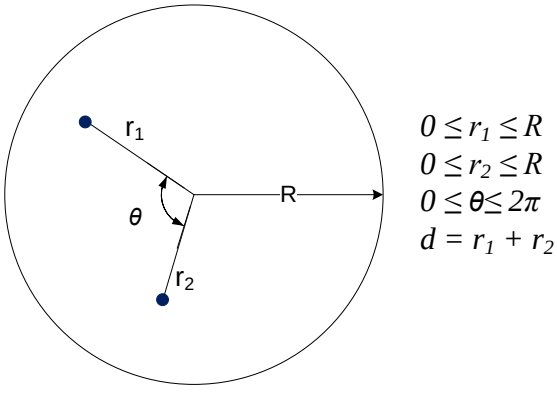


Figure 6.9 – Positions of a source and a destination node trying to communicate through the cluster tree. R – radius of the sensor field, r_1 – distance to source node from the root node, r_2 – distance to destination node from the root node, θ – angle between two nodes.

(Figure 6.10). Assume that nodes are uniformly distributed with a node density of λ.

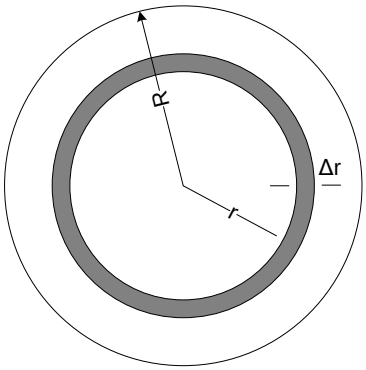


Figure 6.10 – Area covered by a small ring of Δr. R – radius of the sensor field, r – distance to a node from the root node.

$$\begin{aligned}
&\textit{Number of nodes in the sensor field} && = \pi R^2 \lambda \\
&\textit{Number of nodes in the small circle at } r && = 2\pi r \Delta r \lambda \\
&\textit{Probability of finding a node at } r && = \frac{2\pi r \Delta r \lambda}{\pi R^2 \lambda} \\
&P(r) && = \frac{2r}{R^2}\Delta r
\end{aligned}$$

r_1 and r_2 are two independent events and θ is the angle between them. Therefore:

$$\begin{aligned}
P(r_1, r_2, \theta) &= \frac{2r_1}{R^2}\Delta r_1 \times \frac{2r_2}{R^2}\Delta r_2 \times \frac{\theta}{2\pi}\Delta\theta \\
&= \frac{2r_1 r_2 \theta}{\pi R^4}\Delta r_1 \Delta r_2 \Delta\theta
\end{aligned} \tag{6.3}$$

Substituting values for $d(r_1, r_2, \theta)$ and $P(r_1, r_2, \theta)$ in Equation 6.1:

$$\begin{aligned}
E[d] &= \int_0^R \int_0^R \int_0^{2\pi} (r_1 + r_2)\left(\frac{2r_1 r_2}{\pi R^4}\right) d\theta dr_1 dr_2 \\
&= \frac{2}{\pi R^4} \int_0^R \int_0^R \int_0^{2\pi} (r_1 + r_2) r_1 r_2 \; d\theta dr_1 dr_2 \\
&= \frac{4}{3} R
\end{aligned} \tag{6.4}$$

Equation 6.4 gives the expected distance between any two nodes that want to communicate through the cluster tree. Let us extend this analysis to a network that makes use of the cluster tree and the circular path. As shown in Figure 6.11, depending on the

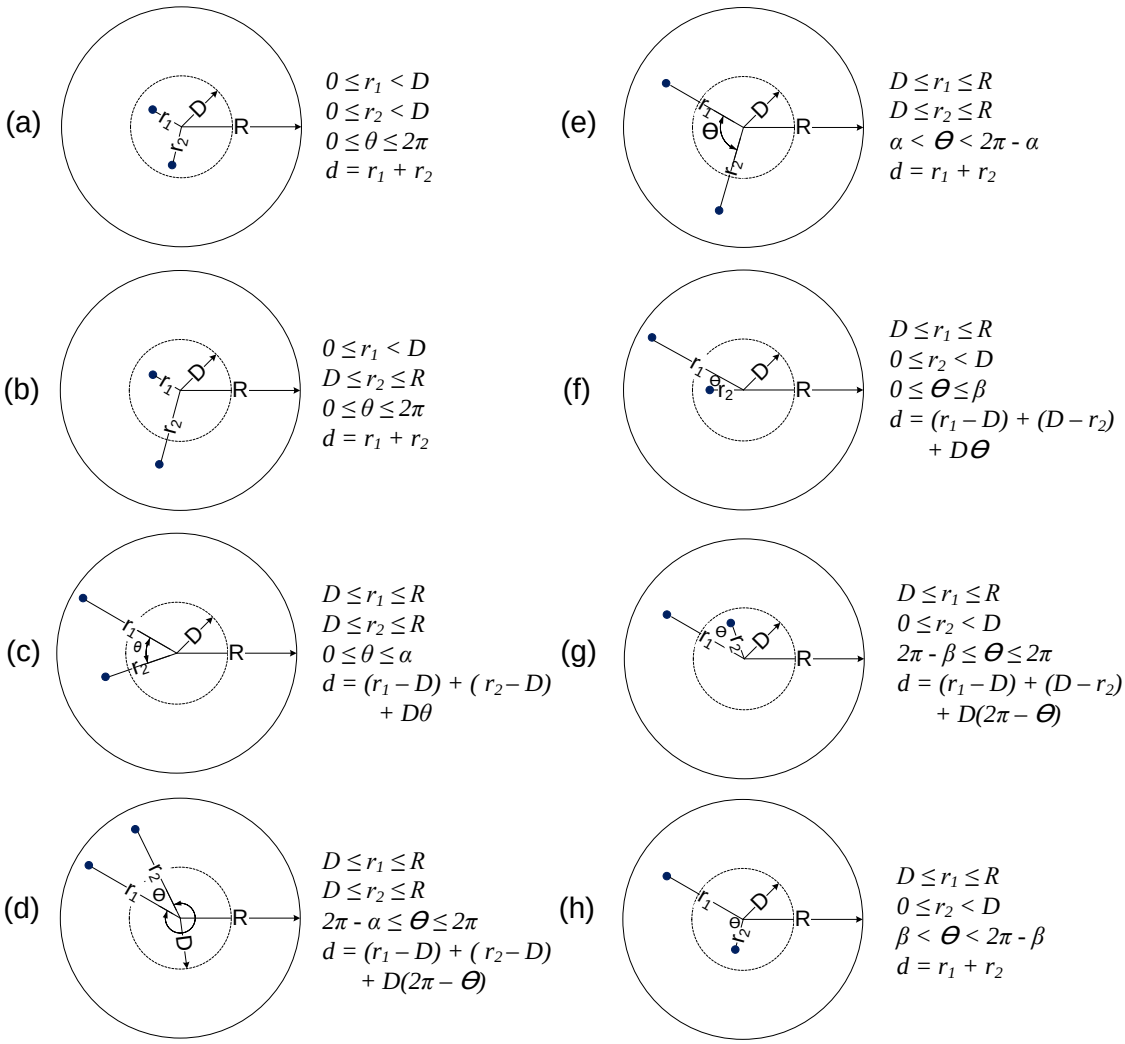


Figure 6.11 – Different positions of a source and a destination node trying to communicate using the cluster tree and the circular path. R – radius of the sensor field, D – radius of the circular path, r_1 – distance to source node from the root node, r_2 – distance to destination node from the root node, θ angle between the two nodes.

positions of the source and destination nodes eight different combinations can occur. If the source node is inside the circular path (Figure 6.11 (a) and (b)), it cannot make use of the circular path therefore have to purely depend on the cluster tree. If either the source or both source and destination nodes are outside the circular path, nodes can make use of both the cluster tree and the circular path. As discussed earlier, message will go through the circular path only if it provides a better path than the cluster tree. If both source and

destination nodes are outside the circular path ($D \le r_1 \le R$, $D \le r_2 \le R$), messages will go through the circular path if they are within a certain angle α (Figure 6.11 (c) and (d)). Otherwise, the messages will be forwarded through the cluster tree as usual (Figure 6.11(e)). If the source node is outside the circular path ($D \le r_1 \le R$) and destination node is inside ($0 \le r_2 < D$), messages will go through the circular path if they are within a certain angle β (Figure 6.11 (f) and (g)). If not, the messages will be forwarded through the cluster tree as usual (Figure 6.11(h)).

Following inequalities can be used to determine the critical angles α and β.

When $r_1, r_2 \ge D$

$$\begin{aligned}
&\textit{Distance through cluster tree} > \textit{Distance through cluster tree} + \textit{Circular path} \\
&r_1 + r_2 > (r_1 - D) + (r_2 - D) + D\alpha \\
&r_1 + r_2 > r_1 + r_2 - 2D + D\alpha \\
&0 > D(\alpha - 2) \\
&2 > \alpha \\
&\therefore \alpha \le 2
\end{aligned} \tag{6.5}$$

When $r_1 \ge D, r_2 < D$

$$\begin{aligned}
&\textit{Distance through cluster tree} > \textit{Distance through cluster tree} + \textit{Circular path} \\
&r_1 + r_2 > (r_1 - D) + (D - r_2) + D\beta \\
&r_1 + r_2 > r_1 - r_2 + D\beta \\
&2r_2 > D\beta \\
&\frac{2r_2}{D} > \beta \\
&\therefore \beta \le \frac{2r_2}{D}
\end{aligned} \tag{6.6}$$

Then the distance function that represents all eight cases can be written as:

$$d=\begin{cases} r_1+r_2 & 0\le r_1<D,\quad 0\le r_2<D,\quad 0\le\theta\le 2\pi & A\\ r_1+r_2 & 0\le r_1<D,\quad D\le r_2\le R,\quad 0\le\theta\le 2\pi & B\\ r_1+r_2 & D\le r_1\le R,\quad D\le r_2\le R,\quad 2<\theta<2\pi-2 & C\\ r_1+r_2 & D\le r_1\le R,\quad 0\le r_2<D,\quad \frac{2r_2}{D}<\theta<2\pi-\frac{2r_2}{D} & D\\ r_1+r_2+D\theta-2D & D\le r_1\le R,\quad D\le r_2\le R,\quad 0\le\theta\le 2 & E\\ r_1+r_2+D(2\pi-\theta)-2D & D\le r_1\le R,\quad D\le r_2\le R,\quad 2\pi-2\le\theta\le 2\pi & F\\ r_1-r_2+D\theta & D\le r_1\le R,\quad 0\le r_2<D,\quad 0\le\theta\le\frac{2r_2}{D} & G\\ r_1-r_2+D(2\pi-\theta) & D\le r_1\le R,\quad 0\le r_2<D,\quad 2\pi-\frac{2r_2}{D}\le\theta\le 2\pi & H \end{cases} \qquad (6.7)$$

By adding individual terms, the excepted distance can be determined.

$$E[d]=E[A]+E[B]+E[C]+E[D]+E[E]+E[F]+E[G]+E[H] \qquad (6.8)$$

Answers to individual terms can be obtained by integrating each term.

$$E[A]\Rightarrow\ \frac{2}{\pi R^4}\int_0^D\int_0^D\int_0^{2\pi}(r_1+r_2)r_1r_2\ d\theta dr_1dr_2=\frac{4D^5}{3R^4}$$

$$E[B]\Rightarrow\ \frac{2}{\pi R^4}\int_D^R\int_0^D\int_0^{2\pi}(r_1+r_2)r_1r_2\ d\theta dr_1dr_2=\frac{2D^2}{3R^4}\left\{R^3+R^2D-2D^3\right\}$$

$$E[C]\Rightarrow\ \frac{2}{\pi R^4}\int_D^R\int_D^R\int_2^{2\pi-2}(r_1+r_2)r_1r_2\ d\theta dr_1dr_2=\frac{4(\pi-2)(R^2-D^2)(R^3-D^3)}{3\pi R^4}$$

$$E[D]\Rightarrow\ \frac{2}{\pi R^4}\int_D^R\int_0^D\int_{\frac{2r_2}{D}}^{2\pi-\frac{2r_2}{D}}(r_1+r_2)r_1r_2\ d\theta dr_1dr_2=\frac{D^2}{9\pi R^4}\left\{2(3\pi-4)R^3+3(2\pi-3)R^2D+(17-12\pi)D^3\right\}$$

$$E[E]\Rightarrow\ \frac{2}{\pi R^4}\int_D^R\int_D^R\int_0^2(r_1+r_2-2D+D\theta)r_1r_2\ d\theta dr_1dr_2=\frac{(R^2-D^2)}{3\pi R^4}\left\{4R^3-3R^2D-D^3\right\}$$

$$E[F]\Rightarrow\ \frac{2}{\pi R^4}\int_D^R\int_D^R\int_{2\pi-2}^{2\pi}\{r_1+r_2-2D+D(2\pi-\theta)\}r_1r_2\ d\theta dr_1dr_2=\frac{(R^2-D^2)}{3\pi R^4}\left\{4R^3-3R^2D-D^3\right\}$$

$$E[G]\Rightarrow\ \frac{2}{\pi R^4}\int_0^D\int_D^R\int_0^{\frac{2r_2}{D}}(r_1-r_2+D\theta)r_1r_2\ d\theta dr_1dr_2=\frac{4(R^3-D^3)D^2}{9\pi R^4}$$

$$E[H]\Rightarrow\ \frac{2}{\pi R^4}\int_0^D\int_D^R\int_{2\pi-\frac{2r_2}{D}}^{2\pi}\{r_1-r_2+D(2\pi-\theta)\}r_1r_2\ d\theta dr_1dr_2=\frac{4(R^3-D^3)D^2}{9\pi R^4}$$

By submitting above answers in Equation 6.8:

$$E[d]=\frac{4R}{3}-\frac{2D}{\pi}+\frac{3D^3}{\pi R^2}-\frac{D^5}{\pi R^4} \qquad (6.9)$$

To find the minimum expected distance:

$$\frac{E[d]}{dD} = -\frac{2}{\pi} + \frac{9D^2}{\pi R^2} - \frac{5D^4}{\pi R^4}$$
$$= \frac{1}{\pi}\left\{-2 + \frac{9D^2}{R^2} - \frac{5D^4}{R^4}\right\}$$
$$= \frac{1}{\pi}\left(-2 + 9k^2 - 5k^4\right) \quad where\ k = D/R \tag{6.10}$$

Roots can be found by equating Equation 6.10 to zero.

$$5k^4 - 9k^2 + 2 = 0$$
$$k = \pm\sqrt{\frac{9-\sqrt{41}}{10}}, \pm\sqrt{\frac{9+\sqrt{41}}{10}}$$
$$k = 0.509$$
$$\therefore D = 0.509R \tag{6.11}$$

The only valid root is $k = 0.509$, therefore $D = 0.509R$. When the circular path is placed at this position, the expected energy consumption of a message will be lower.

Reducing energy of a message does not necessarily increase the network lifetime. The capacity of the network depends on the bottleneck node. If the circular path is closer to the root node most of the messages will go through it. Then nodes along the circular path become the bottlenecks. If the circular path is further away from the root node most of the messages will go through the root node making it the weakest point. Therefore, the optimum capacity of the network can be achieved by balancing the workloads of the root node and nodes along the circular path.

P(message going through root node) = P(message going through a node in circular path)

First four terms, *A* to *D*, in Equation 6.7 corresponds to the cases were a message travel only through the root node. Remaining four terms corresponds to the cases were a message travel through the circular path and the cluster tree. Then:

$$P(R)=\begin{cases}\frac{2r_1r_2}{\pi R^4} & 0\le r_1<D,\quad 0\le r_2<D,\quad 0\le\theta\le 2\pi & I\\ \frac{2r_1r_2}{\pi R^4} & 0\le r_1<D,\quad D\le r_2\le R,\quad 0\le\theta\le 2\pi & J\\ \frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad D\le r_2\le R,\quad 2<\theta<2\pi-2 & K\\ \frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad 0\le r_2<D,\quad \frac{2r_2}{D}<\theta<2\pi-\frac{2r_2}{D} & L\end{cases} \tag{6.12}$$

$$P(C)=\begin{cases}\frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad D\le r_2\le R,\quad 0\le\theta\le 2 & M\\ \frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad D\le r_2\le R,\quad 2\pi-2\le\theta\le 2\pi & N\\ \frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad 0\le r_2<D,\quad 0\le\theta\le\frac{2r_2}{D} & O\\ \frac{2r_1r_2}{\pi R^4} & D\le r_1\le R,\quad 0\le r_2<D,\quad 2\pi-\frac{2r_2}{D}\le\theta\le 2\pi & P\end{cases} \tag{6.13}$$

where *P(R)* is the probability of a message going through the root node and *P(C)* is the probability of a message going through the circular path. By adding individual terms, two probabilities can be determined.

$$P(R)=P(I)+P(J)+P(K)+P(L) \tag{6.14}$$

$$P(C)=P(M)+P(N)+P(O)+P(P) \tag{6.15}$$

Answers to individual terms can be obtained by integrating each term.

$$P(I)\Rightarrow\quad \frac{2}{\pi R^4}\int_0^D\int_0^D\int_0^{2\pi} r_1r_2\;\; d\theta dr_1dr_2\;\;=\frac{D^4}{R^4}$$

$$P(J)\Rightarrow\quad \frac{2}{\pi R^4}\int_D^R\int_0^D\int_0^{2\pi} r_1r_2\;\; d\theta dr_1dr_2\;\;=\frac{D^2}{R^4}\left\{R^2-D^2\right\}$$

$$P(K)\Rightarrow\quad \frac{2}{\pi R^4}\int_D^R\int_D^R\int_2^{2\pi-2} r_1r_2\;\; d\theta dr_1dr_2\quad\;\;=\frac{(\pi-2)(R^2-D^2)^2}{\pi R^4}$$

$$P(L)\Rightarrow\quad \frac{2}{\pi R^4}\int_D^R\int_0^D\int_{\frac{2r_2}{D}}^{2\pi-\frac{2r_2}{D}} r_1r_2\;\; d\theta dr_1dr_2=\frac{(3\pi-4)(R^2-D^2)D^2}{3\pi R^4}$$

$$P(M)\Rightarrow\quad \frac{2}{\pi R^4}\int_D^R\int_D^R\int_0^2 r_1r_2\;\; d\theta dr_1dr_2\quad\;\;=\frac{(R^2-D^2)^2}{\pi R^4}$$

$$P(N)\Rightarrow\quad \frac{2}{\pi R^4}\int_D^R\int_D^R\int_{2\pi-2}^{2\pi} r_1r_2\;\; d\theta dr_1dr_2\;\;=\frac{(R^2-D^2)^2}{\pi R^4}$$

$$P(O) \Rightarrow \quad \frac{2}{\pi R^4}\int_0^D \int_D^R \int_0^{\frac{2r_2}{D}} r_1 r_2 \; d\theta dr_1 dr_2 \quad = \frac{2(R^2 - D^2)D^2}{3\pi R^4}$$

$$P(P) \Rightarrow \quad \frac{2}{\pi R^4}\int_0^D \int_D^R \int_{2\pi - \frac{2r_2}{D}}^{2\pi} r_1 r_2 \; d\theta dr_1 dr_2 \quad = \frac{2(R^2 - D^2)D^2}{3\pi R^4}$$

By submitting answers in Equations 6.14 and 6.15:

$$P(R) \; = \; \frac{\pi - 2}{\pi} + \frac{8D^2}{3\pi R^2} - \frac{2D^4}{3\pi R4} \tag{6.16}$$

$$P(C) \; = \; \frac{2}{\pi} - \frac{8D^2}{3\pi R^2} + \frac{2D^4}{3\pi R4} \tag{6.17}$$

Figure 6.12 shows the two probability functions. When *D* is smaller, most of the messages go through the circular path. If the angle between two nodes is greater than the critical angels α and β messages will always go through the root node. Therefore, even when $D \rightarrow 0$, 0.3633 fraction of the messages travel through the root node. When $D \rightarrow R$, all the messages go through the root node. The maximum network lifetime can be achieved if the workload of the root node and workload of a CH along the circular region is equal, i.e., they have equal likelihood of dying. Let us assume that the workload on the circular path is equally divided among all the CHs within the circular path. Then:

Fraction of messages going through circular path	$= P(C)$
E[number of hops each message travel through circular path]	$= h$
Workload on circular path	$= P(C)h$
Workload on a CH that is in the circular path	$= \frac{P(C)h}{m}$

where *h* is the expected number of hops that a message travels through the circular path and *m* is the number of CHs in the circular path. Therefore:

$$P(R) = \frac{P(C)h}{m} \tag{6.18}$$

Equation 6.18 is valid only if $h/m \geq 0.5707$. However, *m* is several times larger than *h*, i.e., more CHs available on the circular path than the expected hop count on the circular

path. Therefore, we cannot find a *D* that satisfies Equation 6.18. Based on empirical data it is possible to find the value of *D* that maximize the network lifetime. This problem may be able to solve using constrained minimization.

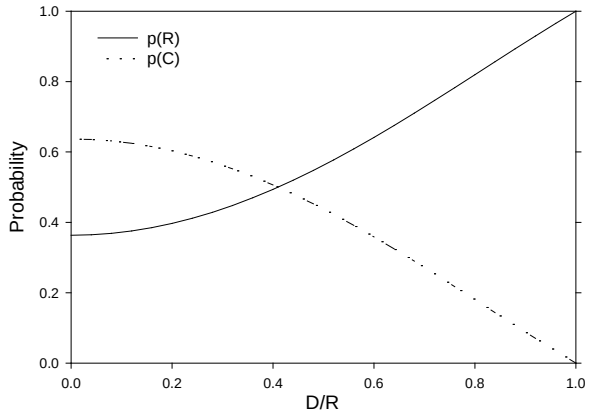


Figure 6.12 – Probability of a message going through the root node or the circular path.

6.4 Performance Analysis

The HHC scheme is used to build the cluster tree. Messages are send from a random source node to a random destination node until the first node dies. Hierarchical address of the destination CH is assumed to be known in advance. P_T is the transmission power used during the cluster formation and *R* is the corresponding transmission range. Inter-cluster communication is single-hop; therefore, CH-to-CH transmission range is *3R*. In cross-links based routing, before data transfer each CH sends a broadcast indicating its hierarchical addresses, so that all the CHs are aware of their neighboring clusters. Results are based on 100 samples. Refer Appendix A for specific implementation details and simulation parameters.

Figure 6.13 shows the number of messages delivered by cluster tree based routing and cross-links based routing. Use of cross-links reduces the workload on the root node

therefore doubles the capacity of the network. In all the data samples, the root node failed first. Generally, for P_T = *-20dBm* a single node can forward ≈*15000* messages. From the figure it can be seen that ≈*16500* messages were forwarded by the cluster tree when P_T = *-20dBm*. This implies that most of the messages were passed through the root node. Network capacity decreases with increasing P_T. Use of higher transmission power significantly drains energy in CHs therefore network capacity rapidly reduces with increasing P_T. This behavior is clear in Figure 6.14 where energy to send a message increases rapidly with increasing P_T.

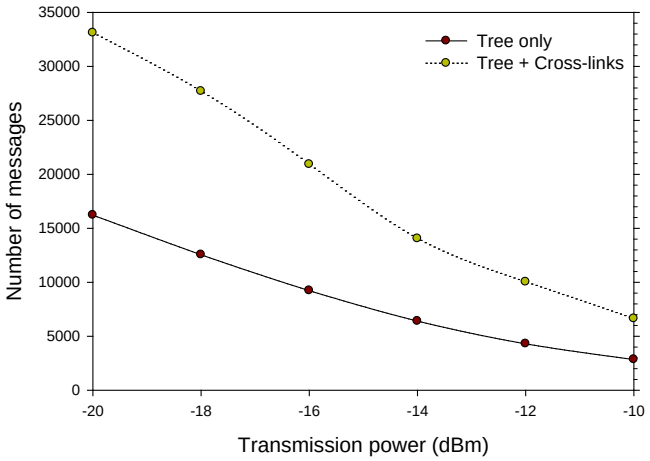


Figure 6.13 – Number of messages delivered.

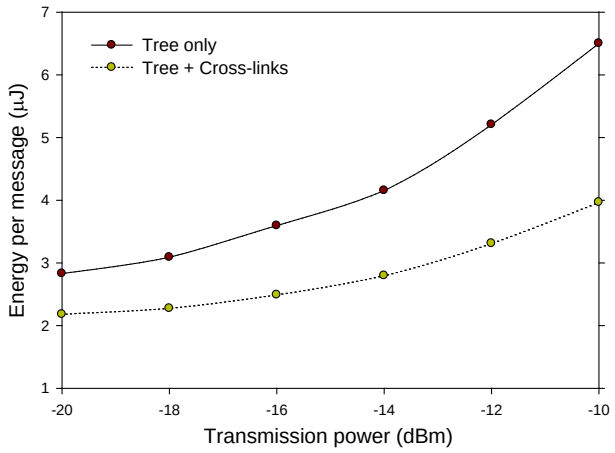


Figure 6.14 – Energy required to send a message.

Figure 6.15 shows the fraction of energy remaining in the entire network after the first message is dropped due to lack of energy. Even through cross-links based routing makes use of more energy, still more than 99% of the energy in the entire network is unutilized. More energy can be utilized if we allow messages to be routed even after the failure of couple of nodes/CHs. After the failure of the root node, majority of the new messages were dropped as they were trying to go through the root node. Though "number of messages delivered until the first message is dropped" is not a good metric, it easily pinpoints the bottleneck in hierarchical WSNs.

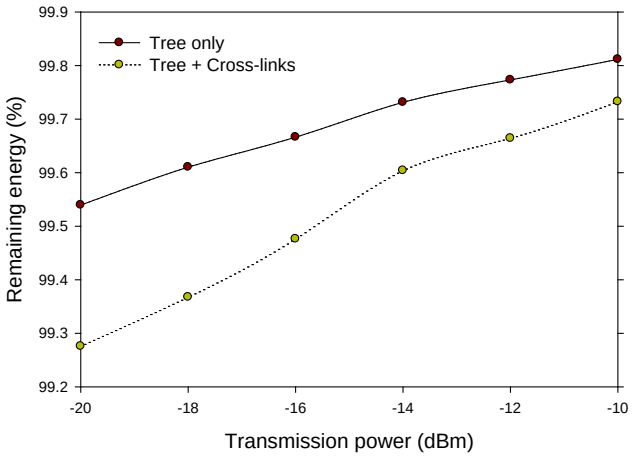


Figure 6.15 – Fraction of energy remaining in the entire network.

To facilitate circular path based routing a ring needs to be formed within the network by connecting CHs of the same depth. In practice, it is not possible to build a complete ring because physical shape of clusters is different and CHs of the same depth may not be in the same neighborhood. For example, though CHs *I* and *N* in Figure 6.8 are at the same depth they cannot form a link because they are not in the same neighborhood (too far apart). However, a circular band can be build by allowing CHs of two adjacent levels to share their addresses. Such a band increases the connectivity of the circular path.

Figure 6.16 shows the energy consumed by circular path based routing. Depth *0* represents the case where no circular path is present. The scenario where circular path is formed between root node and its child CHs is indicated with depth *0-1*. Similarly, *1-2* indicates the case where circular path is formed between CHs at depth 1 and 2. For P_T = *-20dBm*, minimum energy is consumed when the circular path is formed between CHs at depth *5* and *6*. Depth 5 and 6 corresponds to CHs that are between *≈120 – 240m* from the root node (Figure 5.24). Therefore, the average distance is around *180m*. For P_T = *-10dBm* minimum energy is consumed when the circular path is between depth *3* and *4*. This corresponds to an average distance of *270m*. According to Equation 6.11, *D = 0.509R*, therefore ideally the circular path needs to be at *254.5m*. Results are somewhat different to the optimum *D* predicted from the model. CHs are not fully localized therefore we had to build a band instead of a circular path hence results can be different.

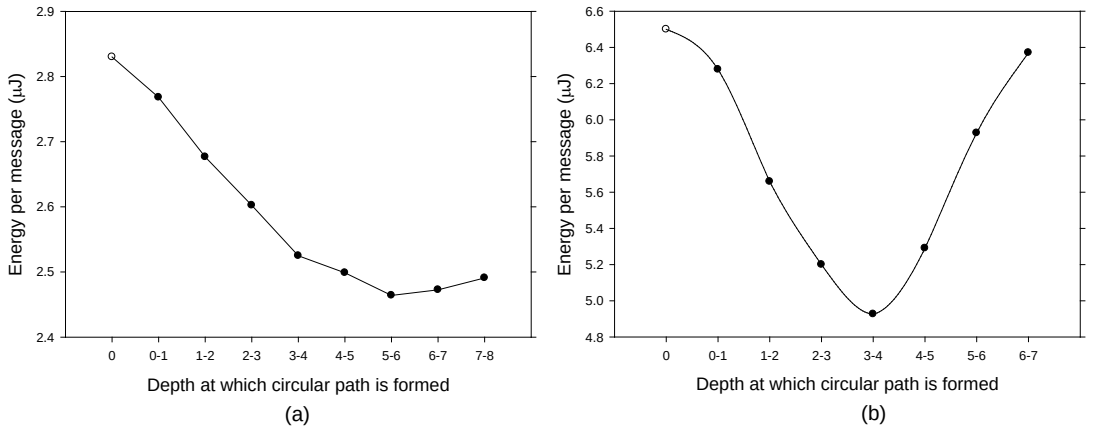


Figure 6.16 – Circular path based routing - energy required to send a message: (a) – P_T = *-20dBm*, (b) – P_T = *-10dBm*.

Figure 6.17 shows the number of successfully delivered messages with circular path based routing. For P_T = *-20dBm*, the peak performance is 2.3 time more than what is delivered by only the cluster tree. For P_T = *-10dBm*, the network is able to deliver 2.5

times more messages. Generally, it was seen that circular path based routing at least double the network capacity. Depth of 3-4 in Figure 6.16(a) corresponds to an average distance of *120m* while depth of *3-4* in figure 6.16(b) corresponds to *270m*. When P_T = *-20dBm*, in 94% of the samples a CH along the circular path died when the circular path was closer to the root node, i.e., $depth \leq 2$. The root node died in all the samples when the circular path was further away from the root node, i.e., $depth \geq 4$. When $2 \leq depth \leq 3$, 25% of the time the root node died. At the optimum point, the root node died 75% of the time and a CH along the circular path died in the remaining 25%. This behavior confirms the analytical model though we were not able to find a specific answer.

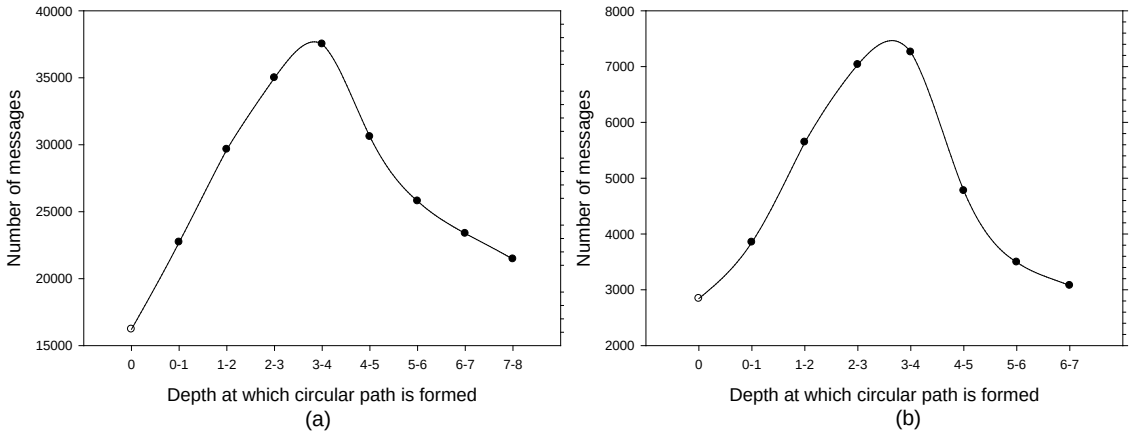


Figure 6.17 – Circular path based routing – number of messages delivered: (a) – P_T = *-20dBm*, (b) – P_T = *-10dBm*.

Figure 6.18 shows the number of messages delivered by all three routing mechanisms. When the circular path is placed between the two optimum depths, circular path based routing delivers the highest number of messages. It also consumes the lowest energy to send a message and is able to utilize energy available in many CHs. Overhead of cross-link and circular path formation were not significant and both schemes consumed similar amount of energy (less than 0.01% of total energy in the network). The

network capacity significantly increases when the thickness of circular path is increased. When P_T = *-20dBm*, a band formed by connecting CHs in depth one to five increase the network capacity by more than three times. It was also realized that number of messages that can be delivered is invariant of the network density.

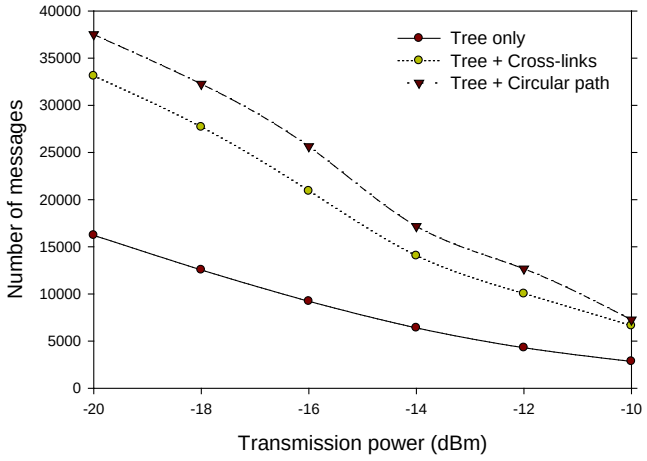


Figure 6.18 – Number of messages delivered by each routing scheme. P_T = *-20dBm*. Circular path based routing values are based on peak performance.

6.5 Summary

A cluster tree based routing scheme and two extensions based on cross-links within the cluster tree and circular path in the network was presented. Both cross-links and circular path at least double the network capacity and has similar overhead in terms of discovering neighbor information. Routing is based on a hierarchical addressing scheme that reflects the parent-child relationship among CHs. These addresses significantly reduce the routing information that needs to be stored at each CH. Cross-links and circular path based routing make use of hierarchical addresses to determine the shortest path to a destination. These routing mechanisms are utilized to form and communicate within VSNs.

Chapter 7

TOWARDS VIRTUAL SENSOR NETWORKS

Virtual Sensor Networks (VSNs) is an emerging concept that supports collaborative, resource efficient, and multipurpose sensor networks that may involve dynamically varying subset of sensors and users. A VSN combines sensors collaborating on a specific task(s) into a logical network. VSNs are expected to provide the protocol support for formation, usage, adaptation, and maintenance of such sensors and/or networks. However, realization of this concept requires design and implementation of many algorithms and protocols. As an initial step towards VSNs, a mechanism to form VSNs by connecting nodes observing the same phenomenon is proposed. We make use of the HHC based cluster tree and routing schemes to effectively form VSNs and communicate across them. We simulate a VSN based close loop system to demonstrate the efficacy of the approach.

Section 7.1 presents the functions that are required to form and manage a VSN. A cluster tree based mechanism to form VSNs is presented in Section 7.2. Inter-VSN and intra-VSN communication models are presented in Section 7.3. Section 7.4 presents a simple analytical model to determine the energy consumption in the network. Finally, Section 7.5 presents the performance analysis.

7.1 Virtual Sensor Network Support Functions

Formation, usage, adaptation, and maintenance of VSNs require implementation of many functions and protocols. These functions and protocols should be able to get new nodes into a VSN, remove nodes from a VSN, detect multiple VSNs, merge VSNs together (e.g., when two chemical plumes merge), split a VSN into multiple VSNs (e.g., when a chemical plume splits), and facilitate communication within and across VSNs.

Self-organization of VSN members is the first step. Whenever a node detects a relevant event for the first time, it should send a VSN formation/discovery message within the network indicating that it is aware of the particular phenomenon and wants to collaborate with similar nodes. The node may join an existing VSN (if there is one) or makes it possible for other nodes that wish to form a VSN, to find it. Therefore, every node that detects a relevant event for the first time executes the following function and informs other nodes about its interest to form/discover a VSN.

Form_Discover_VSN(msg)

The message (*msg*) format should be similar to the following:

```
struct msg:
        source      //Node ID of the source node
        type        //Type of phenomenon, VSN ID
        reading     //Sensor reading(s)
```

where phenomenon *type* indicate a particular VSN.

These messages can be distributed within the network using a random routing scheme such as Rumor Routing [14], Zonal Rumor Routing [10], or Ant Routing [35]. Though these infrastructure-less approaches are relatively simple to implement, they incur significant overhead [10, 14] and do not guarantee that two nodes that detects the same phenomenon are going to identify each other [14]. Alternatively, formation of some

structure within the network can easily deliver these messages. Such an approach can significantly reduce the overhead and it will guarantee that two nodes that detect the same phenomenon are going to meet with each other.

Intermediate nodes that relay VSN formation/discovery messages need to keep track of the following VSN routing data to facilitate communication within members of a VSN:

```
struct VSN_table:
        neighbor        //Sender of the VSN msg
        type            //Type of phenomenon, VSN ID
```

In multifunctional WSNs, a node may belong to multiple VSNs hence may keep track of different VSN types for the same neighbor. If a node no longer detects the phenomenon, it may unsubscribe from the VSN by sending an unsubscribe message to other VSN members:

Unsubscribe_VSN(msg)

Inter-VSN and intra-VSN communication models are application dependant. Unicast messages are required when a message needs to be send to a specific VSN member. For example, if a chemical plume is predicted to be moving towards a certain direction, node(s) in that region need to be informed. Multicast messages are useful in delivering messages to all the members of a VSN. For example, if each node independently calculates the average chemical concentration of a plume, each other's data needs to be shared. When VSNs merge or split, it may be required to inform all the members of existing VSNs, hence broadcast within all the VSNs is also important. Therefore, following functions are required while maintaining VSNs:

Unicast_VSN(destination, type, data)
Muticast_VSN(type, data)
Broadcast_VSN(data)

In addition to these functions, it is required to develop functions that are able to detect multiple VSNs and handle VSN dynamics such as migrating, merging, and splitting VSNs. However, for the time being we only present an algorithm to self-organize VSN members and allow them to communicate with each other.

7.2 Cluster Tree Based Virtual Sensor Network Formation

In VSNs, sensor nodes observing the same phenomenon (i.e., similar events) form a logical network. Formation of such a logical network is somewhat easier if subset of nodes that are expected to collaborative with each other is known in advance. For example, in a smart neighborhood based system all the intruder detection devices and alarms are known beforehand; therefore, locating/connecting them is somewhat easier. However, in certain applications a node's interest on a particular event may vary over time. For example, nodes that are involved in detecting underground chemical plumes may vary due to migrating, merging, and splitting plumes [6, 40]. Therefore, it is important to build this logical network dynamically as nodes get interested in the events.

We form a VSN by connecting nodes observing the same phenomenon through a *virtual tree*. We make use of the cluster tree formed by our HHC scheme to deliver VSN formation/discovery messages and to communicate within and across VSNs. The algorithm given in Figure 7.1 is used to form such a virtual tree. Whenever a node detects a relevant event for the first time it sends a VSN formation/discovery message (using the ***Form_Discover_VSN(msg)*** function) towards the root node of the cluster tree, indicating that it is aware of the phenomenon and wants to collaborate with similar nodes. Intermediate CHs need to keep track of the following information:

```
struct my_data
        cluster_members      //List of cluster members
        n                    //No of VSNs
        VSNs                 //Array of VSN IDs
        m                    //No of VSN routing entries
        VSN_table            //VSN routing table
```

```
Handle_VSN_Message(msg)
    //Initially n = 0, m = 0
 1  IF msg.source ∈ my.cluster_members
 2      IF(msg.type ∉ my_data.VSNs)
 3          my_data.VSNs[n] ← msg.type
 4          n ← n + 1
 5          Forward_To_Parent_CH(msg, my.parent_CH)
 6      my_data.VSN_table[m] ← (msg.source, msg.type)
 7      m ← m + 1
 8  ELSE
 9      IF(msg.type ∉ my_data.VSN_table)
10          Forward_To_Parent_CH(msg, my_data.parent_CH)
11      my_data.VSN_table[m] ← (child_CH, msg.type)
12      m ← m + 1
```

Figure 7.1 – VSN formation algorithm.

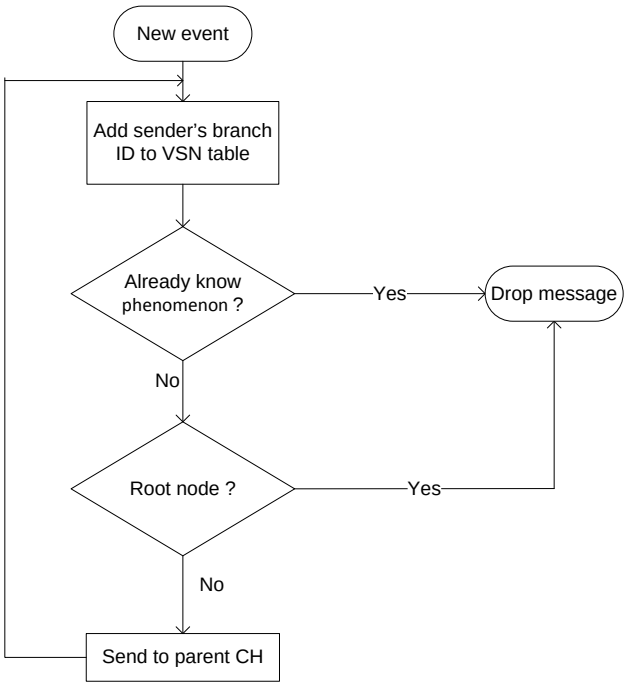


Figure 7.2 – VSN formation steps.

A CH executes the ***Handle_VSN_Message*** function whenever a VSN formation/discovery message is received. If the message is from one of its cluster members, the CH checks its array of VSN (*my_data.VSNs[]*) entries to determine whether it is already aware of the VSN (line 2). If not, the CH marks itself as being part of the VSN (line 3), i.e., its cluster is detecting the phenomenon. Note that, it is possible for a CH to be part of multiple VSNs, e.g., multifunctional sensor nodes. It then forwards the message to its parent CH using the ***Forward_To_Parent_CH*** function. The CH further adds the cluster member to its VSN routing table (line 6), regardless of it being previously aware of the VSN or not. These routing entries are useful when delivering messages to members of a VSN.

If the received message is from one of the child CHs, the parent CH checks its VSN routing table to see whether it is already aware of the VSN (line 9). If not, the message is forwarded to its parent CH so that the parent can also keep track of the VSN (line 10). Similarly, the message is forwarded up to the root node. The virtual tree is formed by all the CHs along the path towards the root node keeping track of the VSN, i.e., by adding the child CH to their VSN routing tables (line 11). Only CHs that are in the phenomenon mark themselves as part of the VSN and other CHs only support their communication.

A hypothetical sensor field that tracks chemical plumes is shown in Figure 7.3. Two chemical plumes are located around clusters *E, F, J, K, N, O*, and *R*. Assume that a node in cluster *J* first detects the plume. It sends a VSN formation/discovery message towards the root node. Firstly, the message is forwarded to its own CH *J*. This is the first time that *J* is receiving this information; therefore, it marks its self as detecting the

phenomenon (indicated by the circle in Figure 7.3(b)). It then forwards the message to its parent CH *H*. *H* caches the information about the new VSN (i.e., add child CH *J* to its VSN routing table) and forwards the message to its parent CH *B*. Finally, the message is forwarded to the root node (*A*). Both the root node and CH *B* also cache the information about the new VSN. Suppose that another node in cluster *K* detects the same phenomenon therefore forwards its message to the root node. It will follow the path *K* → *C* → *A*. *K*, *C*, and *A* cache the information about the VSN. However, the root node is already aware of the phenomenon. Therefore, it does not need to caches the new information. However, the root node keeps track of the branch (i.e., child CH *C*) that the message was received from. Keeping track of such branches enable routing among nodes that tracks the same phenomenon.

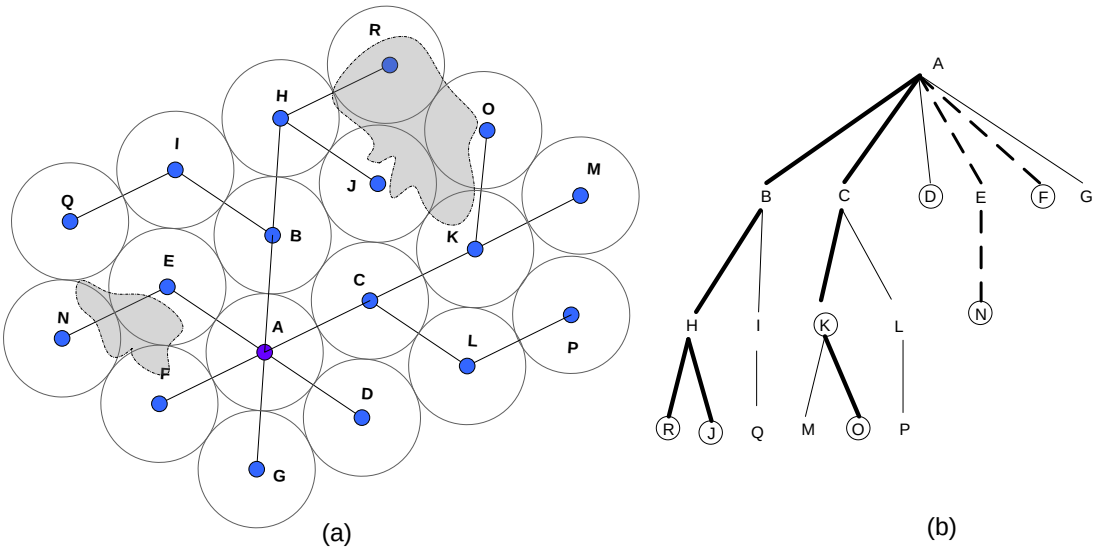


Figure 7.3 – A hypothetical sensor field that tracks chemical plumes: (a) – ideal HHC clusters detecting two chemical plumes, (b) – two virtual trees that connect VSN members.

Meanwhile other nodes in the event region will also detect the plume. When nodes in cluster *R* detect a plume, they will also send messages towards the root node.

When the first message is received from one of the cluster members, CH of *R* also marks itself as detecting the phenomenon. The message is then forwarded to its parent CH *H*. However, *H* is already aware of the phenomenon and has already informed its parent CH *B.* Therefore, *H* will not forward the message any further. Nevertheless, *H* keeps track that *R* is also in the phenomenon. The same occurs when cluster members of *O* detect the plume. Nodes in clusters *J*, *K*, *R*, and *O* may need to communicate with each other to detect migrating, splitting, and merging plumes. Therefore, form a virtual network, hereafter referred as the *virtual tree*, on top of the cluster tree. The virtual tree connects all the CHs that are detecting the same phenomenon (Figure 7.3(b)). These clusters make use of other CHs in the virtual tree to communicate with each other. Similarly, clusters *E*, *F*, and *N* can form another logical tree. These two trees belong to the same VSN if they monitor the same phenomenon. If not, they can be considered as two separate VSNs. Our approach can also enable multiple VSNs to communicate with each other because each virtual tree is guaranteed to meet at the root node.

Sending data about an event towards the root node guarantees that two or more nodes observing the same phenomenon will identify each other. Though random routing algorithms such as Rumor Routing [14], Zonal Rumor Routing [10], and Ant Routing [35] can deliver such messages without any infrastructure, they do not guarantee that two nodes observing similar events will find about each other [14]. Many long lived (i.e., significantly higher *TTL* values) agents are required, even to achieve moderate probability of successful delivery [14]. These agents need to keep track of the visited nodes to reconstruct the routing path therefore size of a message increases as an agent travels [14]. In our approach, a VSN formation/discovery message needs to travel only up

to the root node hence require significantly lower *TTL* than the random routing schemes. Routing path (virtual tree) is constructed by adding entries to each CH's VSN routing table (not to the agent) hence size of a VSN formation/discovery message is fixed. Therefore, our approach has a much lower overhead and it guarantees to form a VSN by connecting all the nodes that observe the phenomenon.

Our scheme works regardless of whether the phenomenon is localized or distributed. Cross-links based routing can be used for intra-VSN communication. For example, clusters *J, K, R,* and *O* are in the same neighborhood therefore they can make use of cross-links within the cluster tree. Our scheme can also enable multiple VSNs to communicate with each other because each virtual tree meets at the root node.

When plumes move, split, or merge certain nodes may not be interested in the phenomenon any more. Such nodes can give-up their VSN membership by sending an unsubscribe message towards the root node. If a CH realizes that none of its cluster members (including child CHs) are interested in the phenomenon, it can request its parent CH not to send any future VSN related messages.

The VSN formation scheme does not need to be event driven. If nodes definitely know that they are going to be part of a VSN, they can inform others about their interest to participate in a VSN as they join a cluster. Though the initial cluster and cluster tree formation phase introduce additional overhead this sort of a VSN formation approach is more appropriate for long-lived WSNs with dynamically varying subset of sensors. Cost of VSN formation and management through the cluster tree is lower.

7.3 Inter-VSN and Intra-VSN Communication

Inter-VSN and intra-VSN communication models are application dependant. VSN communication models may need to support the unicast, multicast, and broadcast functions defined in Section 7.1. While the virtual tree is formed, each CH keeps track of the child CHs (i.e., branches of the cluster tree) that are members of the VSN. This information is useful in delivering multicast and broadcast messages without any addressing scheme. This is similar to the addressless routing scheme in Section 6.1.2.

To facilitate unicast messages within a VSN, CHs needs to know about other CHs that are detecting the same phenomenon (e.g., CHs *E*, *F*, *J*, *K*, *N*, *O*, and *R* in Figure 7.3). Therefore, each CH needs to inform its address to all the member CHs of the VSN. It is not necessary to keep track of individual cluster members as far as a CH can represent all its cluster members. Address sharing can be accomplished in several ways. One of the easiest solutions is to store all the CH addresses at the root node and use it as a lookup table (similar to the DNS). This approach requires extra control messages and the root node becomes a single point of failure. Instead, a CH may send a broadcast within the VSN whenever it forms/discovers a new VSN, allowing other member CHs to cache the new address. However, the new CH does not know any addresses of the other CHs that are already in the VSN. Therefore, it has to request those addresses from a CH along the path towards the root node. If the parent CH is aware of the VSN, it may provide those addresses. If not, the parent CH can request its own parent CH to provide the addresses. If none of the CHs along the path towards the root node is aware of any other CH, the root node can provide those addresses. Similarly, addresses of multiple VSNs can be

shared through the root node. Though broadcasts are costly, it does not add significant overhead if number of CHs in a VSN is small.

7.4 VSN Based Close Loop System

Many sensor network applications allow nodes to use different sampling schedules depending on the presence of the phenomenon, dynamic nature of the phenomenon, and application requirements. For example, concentration of a subsurface chemical plumes can change within several days; however, they tend to migrate very slowly. Therefore, it is desirable for nodes that are already detecting the plume to sample everyday while other nodes to sample in every two/four weeks. Nodes that are in the plume consume more energy while other nodes can save energy by reducing unnecessary sampling. When the plume migrates to a different area, different set of nodes will be frequently active allowing previously active nodes to sleep more and save energy. Such an approach can reduce overall energy consumption of the network. The same idea can even be extended to fast moving phenomenons such as hazardous gases.

Individual nodes may dynamically determine their sampling rate depending on presence of the phenomenon. However, this approach fails if nodes closer to the base station decide to sleep, because they are not detecting the phenomenon. All the messages generated by the nodes that sample faster will be either extensively delayed or dropped. Another alternative is the use of a close loop system where some sort of a data analysis system, prediction system, or a user request group of nodes to change their sampling rate. Such a system need to send many unicast messages asking individual nodes to change their sampling rates as the phenomenon moves, hence incur significant overhead. VSNs

can facilitate both these schemes while overcoming their inherent problems. A node that detects the phenomenon forms a path (a VSN) all the way up to the root node and makes sure all the intermediate CHs are active to deliver its data. As explained in Section 7.1, VSN can easily deliver multicast messages within VSNs. Hence, if a close loop system wants to change sampling rate of set of nodes it can send a VSN wide multicast message.

We simulate a subsurface chemical plume monitoring system to demonstrate the efficacy of our cluster tree based VSN formation scheme. We build a close loop system by coupling the WSN to a plume model. Design of the system is shown in Figure 7.4. Set of nodes is grouped into a cluster and set of clusters are connected together through a cluster tree. The VSN is formed on top of the cluster tree. The root node acts as a base station and connects rest of the network to the Plume Modeling and Prediction (PMP) system. Depending on their sampling schedule, nodes periodically test chemical concentration of the soil. If the chemical concentration is beyond a certain threshold, the node is considered to be in the plume. Such nodes send the concentration data all the way up to the PMP system and at the same time form/join a VSN. Such nodes will continue to report their concentration values as far are they are substantially different from the previously reported value. Based on the data, PMP will generate a transport model of the plume. Such a model is useful in predicting plume migration patterns and remedial treatment. As explained earlier, the PMP system or a user can request a node or group of nodes to change their sampling rate through the VSN. The VSN changes the active schedule of the intermediate CHs to match the new sampling rate, while forwarding such messages. This ensures that the new data will not be dropped or delayed. Nodes may miss some events if their sampling rate is low than the plume dynamics. Plume predictions can

be used to reduce such misses. If the plume prediction model determines that, a node(s) will be in a plume at time $t + \Delta t$, the PMP system can send an advance request to such a node(s) asking it to be active at the predicted time. We consider only these three types of communications in our simulation; however, depending on the application scenario and use requirements other functions can be incorporated into a VSN.

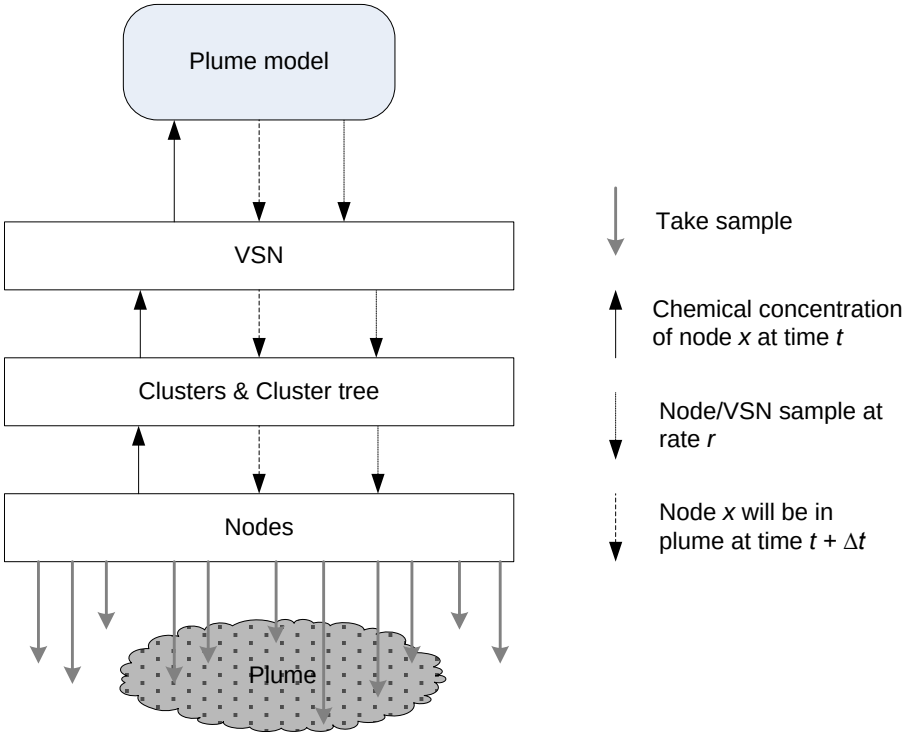


Figure 7.4 – Different layers and their interactions in a VSN based close loop system. Direction of Black arrows indicates direction of data flow.

We further derive an energy model for a VSN based system that delivers data to a base station. Consider a network where all the nodes occasionally come up at the same time (Figure 7.5), i.e., at every $m\Delta t$. Periodic walk-up of the entire network is required due to several reasons. If a new node detects the phenomenon at $m\Delta t$, it is guarantee to send it message all the way up to the root node and join the VSN, because all the intermediate CHs are active at that time. Thereafter, such nodes will increase their

sampling rate. Hence, sometimes it is possible that more nodes are active after a network wide walk-up (e.g., in Figure 7.5 more nodes are active at time Δt after the first $m\Delta t$). In addition, a network wide walk-up is required to inform different sampling schedules and/or predictions to nodes that are not in a VSN. Nodes may use their own sampling schedules within $m\Delta t$ depending on the presence and dynamic nature of the phenomenon. When the phenomenon migrates, certain node may realize they are no longer in the phenomenon hence may reduce their sampling rate. However, those nodes need to make sure they will walk-up at every $m\Delta t$. Although many nodes samples at the same time only a subset of them will generate new messages. New message generation depends on intensity of the event and how much is it different from the previously reported value.

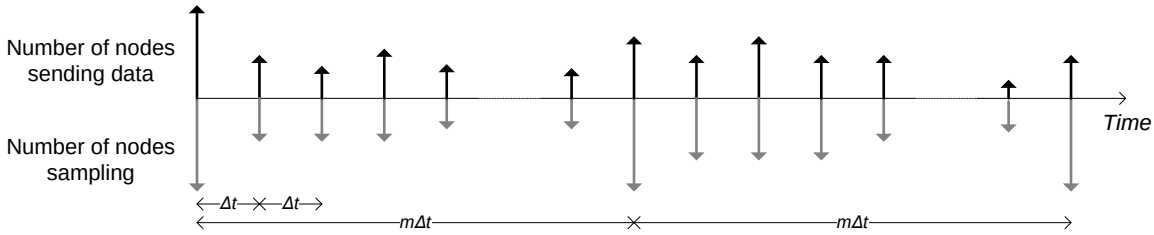


Figure 7.5 – Sampling schedule of different nodes. Length of an arrow is proportional to the number of nodes involved in the particular action.

Energy consumption in the network can be analyzed under three categories energy to stay active and sleep, energy to sample, and energy to communicate. Description of different symbols used in the model is given in Table 7.1.

Energy - Active/Sleep ($E_{act/slep}^{\Delta t_i}$)

The energy consumption is determined by the active (P_a) and sleep (P_s) power consumption of sensor nodes, node's duty cycle, and number of nodes detecting the phenomenon.

Table 7.1 – Symbols used in the energy model

Symbol	Description
Sensor node related parameters	
P_a	Power consume while the node is active
P_s	Power consume while the node is sleeping
P_m	Power consume while sampling
P_d	Power consume while transmitting
n_m	Number of samples
T_m	Time that takes to measure/test a sample
T_c	Time that a child node is active within Δt_i
T_{CH}	Time that a cluster head is active within Δt_i
B	Bandwidth of the node
Network/VSN related parameters	
n	Number of nodes in the network, $n = n_c + n_{CH}$
n_c	Number of child nodes (i.e., cluster members) in the network
n_{CH}	Number of cluster heads in the network
h_i	Node i's depth in the cluster tree (i.e., number of hops to forward a message)
b	Size of a message including the related acknowledgment
Phenomenon related parameters	
$n_{c\_in\_p}^{\Delta t_i}$	Number of child nodes in the phenomenon within Δt_i
$n_{CH\_in\_p}^{\Delta t_i}$	Number of cluster heads in the phenomenon within Δt_i
$n_{CH\_VSN}^{\Delta t_i}$	Number of cluster heads that are not in the phenomenon but in the VSN at Δt_i. They are active to facilitate communication
Δt	Duration of a time step
m	Number of time steps before everyone come up again
α	Changes in concentration level of the phenomenon being tracked– fraction of times a new message is generated, given that the node is already detecting the phenomenon. If concentration changes are rapid $\alpha \to 1$, if slow $\alpha \to 0$.
β	Spatial dynamics of the phenomenon – fraction of times a new message is generated, given that the node samples. If phenomenon is fast moving $\beta \to 1$, if slow moving $\beta \to 0$.

When Δt_i is an integer multiple of $m\Delta t$, every node $(n_c + n_{CH})$ is active for either T_C or T_{CH} seconds. Therefore:

Energy consumed by cluster members while active = $P_a T_c n_c$

Energy consumed by all cluster members while sleeping = $P_s(\Delta t - T_c)n_c$

Energy consumed by CHs while active = $P_a T_{CH} n_{CH}$

Energy consumed by all CHs while sleeping = $P_s(\Delta t - T_{CH})n_{CH}$

$\therefore$ *Total active/sleep energy* $= P_a T_c n_c + P_s(\Delta t - T_c)n_c + P_a T_{CH} n_{CH} + P_s(\Delta t - T_{CH})n_{CH}$

$$= \left(P_a T_c + P_s(\Delta t - T_c)\right)n_c + \left(P_a T_{CH} + P_s(\Delta t - T_{CH})\right)n_{CH} \quad (7.1)$$

At any other Δt_i, $n_{c\_in\_p}^{\Delta t_i}$ cluster members and $(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i})$ CHs are active.

Energy consumed by cluster members while active = $P_a T_c n_{c\_in\_p}^{\Delta t_i}$

Energy consumed by active cluster members while sleeping = $P_s(\Delta t - T_c)n_{c\_in\_p}^{\Delta t_i}$

Energy consumed by CHs while active = $P_a T_{CH}(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i})$

Energy consumed by active CHs while sleeping $= P_s(\Delta t - T_{CH})(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i})$

However, during the same time interval, $(n_c - n_{c\_in\_p}^{\Delta t_i})$ nodes and $(n_{CH} - n_{CH\_in\_p}^{\Delta t_i} - n_{CH\_VSN}^{\Delta t_i})$ CHs were sleeping. Therefore:

Energy consumed by sleeping cluster members = $P_s \Delta t(n_c - n_{c\_in\_p}^{\Delta t_i})$

Energy consumed bysleeping CHs = $P_s \Delta t(n_{CH} - n_{CH\_in\_p}^{\Delta t_i} - n_{CH\_VSN}^{\Delta t_i})$

$\therefore$ *Total active/sleep energy =*

$$\begin{aligned} &P_a T_c n_{c\_in\_p}^{\Delta t_i} + P_s(\Delta t - T_c)n_{c\_in\_p}^{\Delta t_i} + P_a T_{CH}(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i}) \\ &\quad + P_s(\Delta t - T_{CH})(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i}) + P_s \Delta t(n_c - n_{c\_in\_p}^{\Delta t_i}) + P_s \Delta t(n_{CH} - n_{CH\_in\_p}^{\Delta t_i} - n_{CH\_VSN}^{\Delta t_i}) \\ &= \left(P_a T_c + P_s(\Delta t - T_c)\right)n_{c\_in\_p}^{\Delta t_i} + \left(P_a T_{CH} + P_s(\Delta t - T_{CH})\right)(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i}) + \\ &\quad P_s \Delta t(n_c - n_{c\_in\_p}^{\Delta t_i}) + P_s \Delta t(n_{CH} - n_{CH\_in\_p}^{\Delta t_i} - n_{CH\_VSN}^{\Delta t_i}) \end{aligned} \quad (7.2)$$

Generally, $P_s << P_a$ therefore if a node is frequently active, energy to stay active is more dominant. If the node duty cycle is significantly lower, i.e., $T_c << \Delta t - T_c$ sleep power will dominate. If many nodes are in the phenomenon active power dominates.

Energy - Sampling ($E_{test}^{\Delta t_i}$)

Cost of sampling is determined by the power consumed to sense (P_m), time to test a sample (T_m), number of samples (n_m) and number of nodes detecting the phenomenon.

Energy to measure a sample = $P_m T_m$

Energy to measure multiple samples = $P_m T_m n_m$

Every node $(n = n_c + n_{CH})$ samples when Δt_i is an integer multiples of $m\Delta t$.

$\therefore$ *Energy to test samples by all the nodes =* $P_m T_m n_m n$ (7.3)

At any other Δt_i, there $(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})$ nodes and CHs sampling for the phenomenon.

$\therefore$ *Energy to sample if detecting phenomenon=* $P_m T_m n_m (n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})$ (7.4)

Energy – Transmission ($E_d^{\Delta t_i}$ **)**

Cost per message, per hop = $P_d \left(b / B \right)$

Total cost to relay the message by h-hops = $P_d \left(b / B \right) h = P_d bh / B$

When Δt_i is an integer multiple of $m\Delta t$ every node is active. Nodes that are already detecting the phenomenon will generate a new message if the new concentration of the phenomenon is significantly different from the previous reported sample, hence depends on α. Nodes that are not already in the plume generates a new message if they detect the phenomenon hence depends on β. Therefore:

Number of nodes that are already detecting the phenomenon that generate a new message = $(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha$

$\therefore$ *Energy consumed by those nodes to forward the message =* $\sum_{i=0}^{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha} \frac{P_d bh_i}{B}$

Number of nodes that are not already detecting the phenomenon generating a new message = $(n - n_{c\_in\_p}^{\Delta t_i} - n_{CH\_in\_p}^{\Delta t_i})\beta$

$\therefore$ *Energy consumed by those nodes to forward the message =* $\sum_{i=0}^{(n - n_{c\_in\_p}^{\Delta t_i} - n_{CH\_in\_p}^{\Delta t_i})\beta} \frac{P_d bh_i}{B}$

$\therefore$ *Total energy consumed =* $\sum_{i=0}^{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha} \frac{P_d bh_i}{B} + \sum_{i=0}^{(n - n_{c\_in\_p}^{\Delta t_i} - n_{CH\_in\_p}^{\Delta t_i})\beta} \frac{P_d bh_i}{B}$

Can be simplified to following, if average node depth is considered

$$\frac{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha P_d bh_{ave}}{B} + \frac{(n - n_{c\_in\_p}^{\Delta t_i} - n_{CH\_in\_p}^{\Delta t_i})\beta P_d bh_{ave}}{B} \quad (7.5)$$

$(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})$ nodes and CHs sample the phenomenon at any other Δt_i,. Whether a node generates a message or not depends on whether new concentration of the phenomenon is significantly different from the previous sample that generates a new message.

Number of nodes that are already detecting the phenomenon generating a new message = $(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha$

$\therefore$ *Energy consumed by those nodes to forward the message =* $\sum_{i=0}^{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha} \frac{P_d bh_i}{B}$

If average node depth is considered = $\frac{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha P_d bh_i}{B}$ (7.6)

Communication cost will be high if the phenomenon is highly dynamic (i.e., high α and β), message sizes are large, network is very large (i.e., deeper cluster tree) and/or many nodes are in the phenomenon.

Total Energy Consumption

$$E(\Delta t_i) = E_{act/slep}^{\Delta t_i} + E_{test}^{\Delta t_i} + E_d^{\Delta t_i} \quad (7.7)$$

When Δt_i is an integer multiple of $m\Delta t$:

$$E(\Delta t_i) = \left(P_a T_c + P_s(\Delta t - T_c)\right)n_c + \left(P_a T_{CH} + P_s(\Delta t - T_{CH})\right)n_{CH} + P_m T_m n_m n + \frac{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha P_d bh_{ave}}{B} + \frac{(n - n_{c\_in\_p}^{\Delta t_i} - n_{CH\_in\_p}^{\Delta t_i})\beta P_d bh_{ave}}{B} \quad (7.8)$$

At any other Δt_i

$$E(\Delta t_i) = \left(P_a T_c + P_s(\Delta t - T_c)\right)n_{c\_in\_p}^{\Delta t_i} + P_s \Delta t(n_c - n_{c\_in\_p}^{\Delta t_i}) + \left(P_a T_{CH} + P_s(\Delta t - T_{CH})\right)(n_{CH\_in\_p}^{\Delta t_i} + n_{CH\_VSN}^{\Delta t_i}) + P_s \Delta t(n_{CH} - n_{CH\_in\_p}^{\Delta t_i} - n_{CH\_VSN}^{\Delta t_i}) + P_m T_m n_m (n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i}) + \frac{(n_{c\_in\_p}^{\Delta t_i} + n_{CH\_in\_p}^{\Delta t_i})\alpha P_d bh_i}{B} \quad (7.9)$$

7.5 Performance Analysis

7.5.1 VSN Formation

VSNs are formed on top of the cluster tree built with our HHC scheme presented in Section 5.1. Different routing strategies are utilized depending on how events need to be propagated within a VSN. Three different event regions are considered. In the first scenario, the phenomenon is localized within region 1 (Figure 7.6). In the second scenario, the phenomenon is detected in three different regions (Regions 1 to 3). The third scenario randomly distributes the phenomenon around the entire sensor field. Randomly picked nodes from those regions are assumed to detect the event. All the event regions are considered to be parts of the same VSN. Except where noted the simulation results presented assume 500 nodes detect the same event.

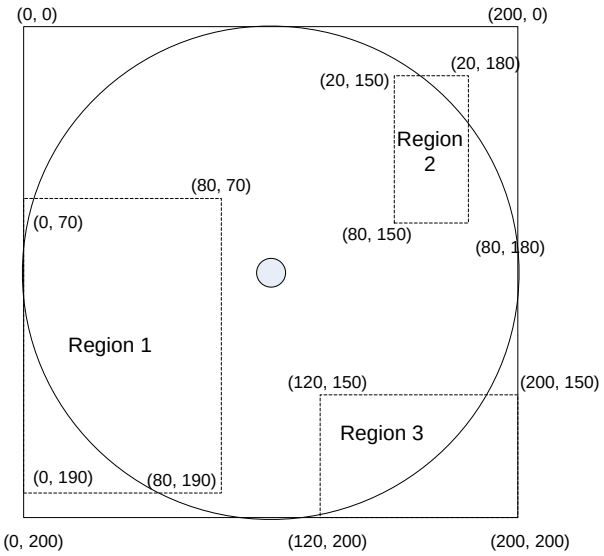


Figure 7.6 – Event regions.

Figure 7.7 and 7.8 shows two virtual cluster trees that are formed in scenarios one and two. In the first scenario, all the nodes that detect the phenomenon are localized hence only a few branches are required to form the virtual tree. Many branches are required

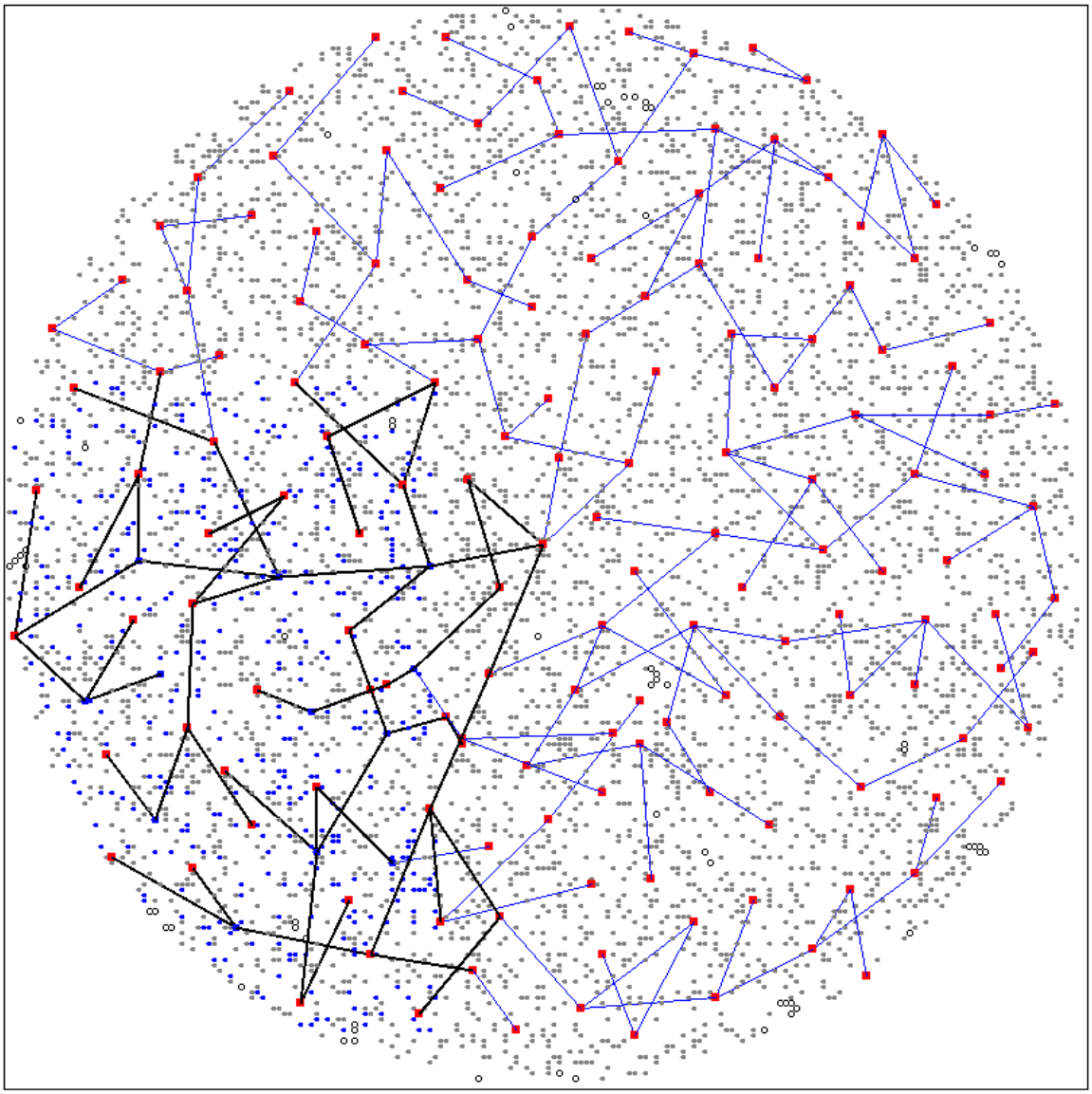


Figure 7.7 – Virtual tree formed by nodes detecting events in a single region. Blue (dark) squares indicate the nodes that detect the event. Dark lines indicate the virtual tree that connects members of the VSN. P_T = *-12dBm*.

to form the virtual tree when the phenomenon is distributed in three regions. Different branches of the cluster tree meet either at the root node or at other CHs.

Figure 7.9 shows the total number of hops travelled by all the VSN formation/discovery messages. First few messages that propagate towards the root node have to travel large number of hops. However, they contribute to the formation of most of

the branches in the virtual tree. When the phenomenon is localized within a single region, most of the new nodes that detect the phenomenon do not need to send their VSN formation/discovery messages all the way up to the root node. They will get to know about other VSN members from CHs that are along its path. Therefore, the number of hops travel by a VSN formation/discovery message reduces as more and more nodes

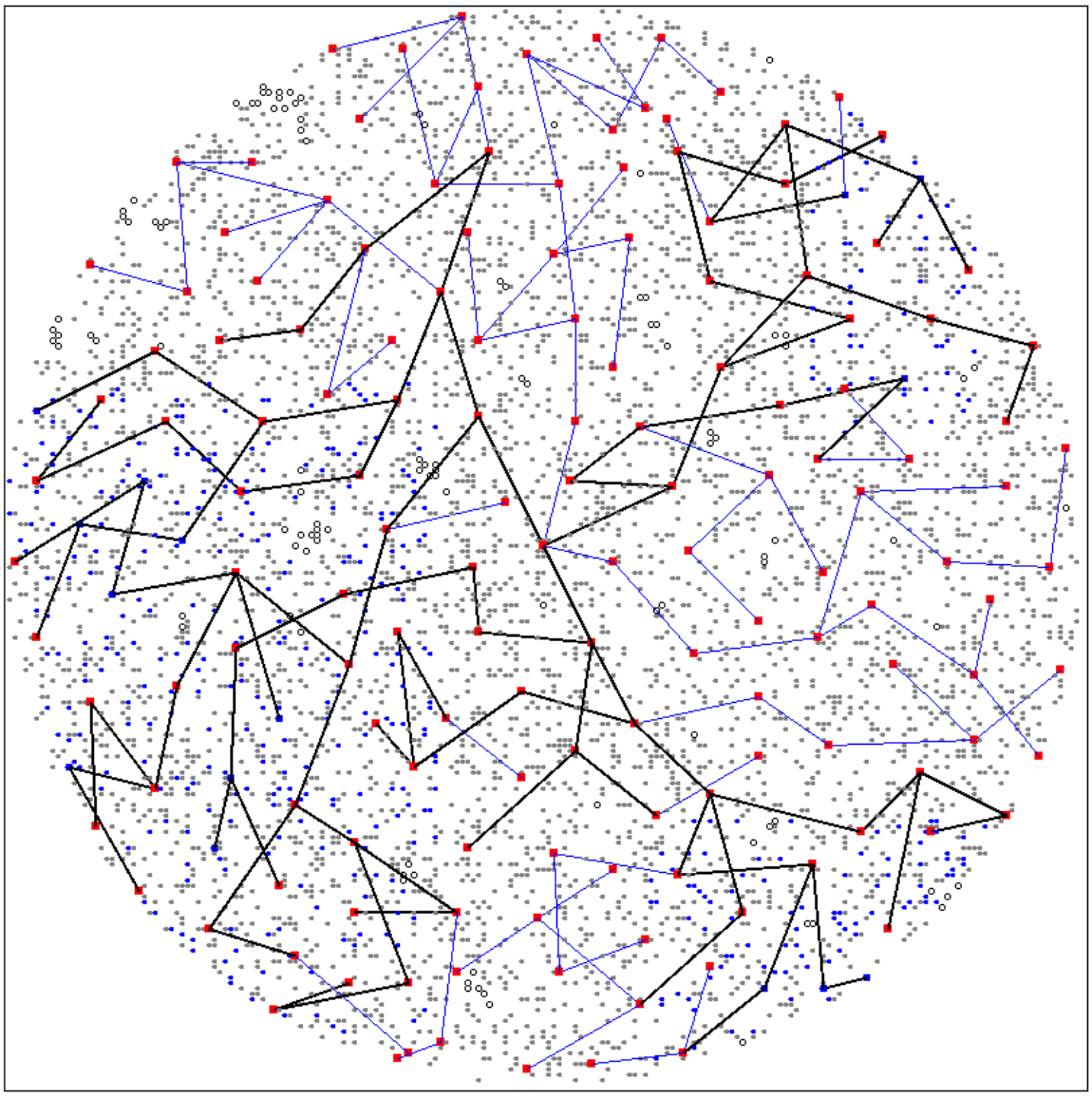


Figure 7.8 – Virtual tree that connects three event regions. Blue (dark) squares indicate the nodes that detect the event. Dark lines indicate the virtual tree that connects members of the VSN. P_T = *-12dBm*.

detect the phenomenon. When the phenomenon is distributed within three regions, not many CHs along the path of the VSN formation/discovery messages are aware of the VSN. Therefore, these messages need to be forwarded several more hops. Most of the paths towards the root node are distinct, when the phenomenon is fully distributed. As a result, these messages need to travel a higher number of hops.

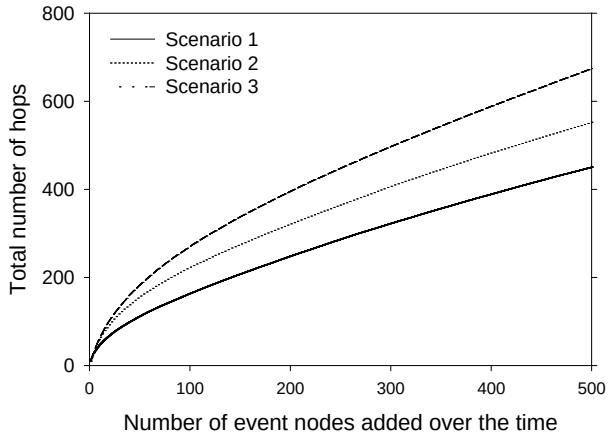


Figure 7.9 – Total number of hops travelled by VSN formation message.

We further simulate Rumor Routing to compare the probability of two VSN members discovering each other. We considered the second scenario (Figure 7.8) and randomly selected node pairs spanning different event regions. Each node then sends a single agent and tries to discover each other. With Rumor Routing, when *TTL* is 350, two nodes were able to figure out each other with a probability of 0.51. The probability increased to 0.84 when *TTL* is 600 and it was further increased to 0.91 when *TTL* is 1000. These values are comparable with the data given in [14]. However, to form VSNs we may need all the nodes to figure out each other therefore actual overhead would be significantly higher. Alternatively, as seen in Figure 7.9, our scheme requires only 553-hops (896-hops if inter-cluster communication is multi-hop) and it guarantees that all the

nodes will identify each other. As realized in Sections 5 and 6 overhead of the HHC cluster tree formation is 4-5 messages per node. Therefore, our approach is much more efficient than Rumor Routing, even with the overhead of cluster tree formation.

7.5.2 Inter VSN Communication

Figure 7.10 shows the number of unicast messages that can be exchanged within the VSN before battery runs out. First scenario delivers the highest number of messages while the third scenario delivers the lowest number of messages. In the first scenario, all the communication occurs within a single region. Therefore, the virtual tree uses only a subset of branches that initiate from the root node. Because events are localized, some of the source and destination nodes lie on the same branch therefore most message do not require going through the root node. The network can deliver more and more messages as the workload of the root node reduces. The virtual tree formed by the third scenario span across most of the branches of the cluster tree. Communication within those distributed VSN members require most of the messages to be relayed through the root node. Therefore, deliver lower number of messages. For the third scenario, capacity of the network is similar to what was observed with cluster tree based routing (Figure 6.13). The second scenario utilizes a subset of the branches of the cluster tree therefore delivers more messages than the third scenario. However, communication across three different regions increase the workload of the root node hence it cannot deliver as many messages as the first scenario. Energy requires to send a message significantly increases with the transmission power (P_T) therefore number of messages that the network can deliver reduces with increasing P_T.

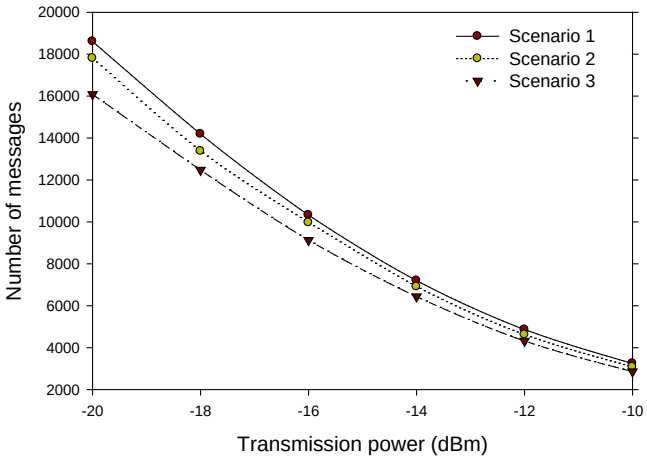


Figure 7.10 – Number of unicast messages.

Figure 7.11 shows the number of multicast messages that can be delivered within the VSN. These can be either data delivery messages or VSN control messages. Multicasts are forwarded to all the VSN members hence it is essentially a broadcast within VSN(s). At each CH, the multicast messages needs to be forwarded to all the branches that are in the virtual tree. For each branch, a separate copy of the message is forwarded. This significantly increases the workload of a CH and drains its energy much faster. Therefore, number of different multicast messages delivered in the network significantly reduces. When the phenomenon is localized within a single region, CHs around that region have to handle many cluster members and child CHs. For each of them, a separate multicast message needs to be send. This increase the workload on those CHs therefore reduces their lifetime. As a result, the first scenario delivers the lowest number of multicast messages. Because many CHs are involved in the third scenario, workload on each CH is lower, i.e., lower number of cluster members per cluster observes the same event. Therefore, the third scenario delivers the highest number of

multicast messages. Due to extensive overhead, the number of multicast messages that were delivered are approximately 1/6 of the unicast messages.

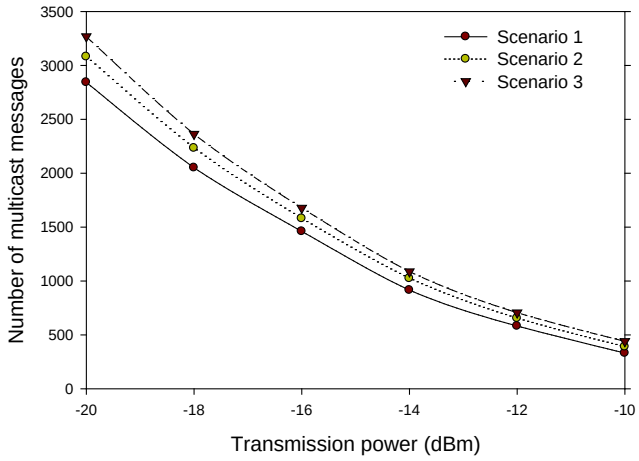
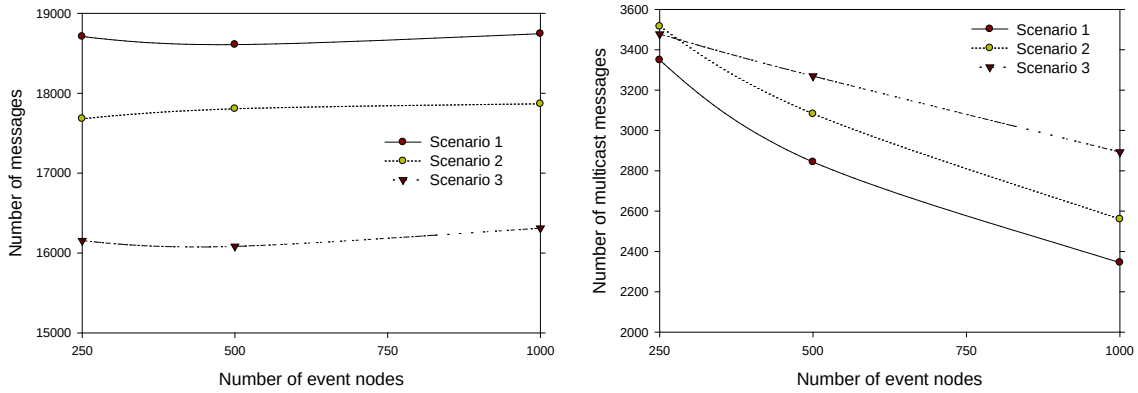


Figure 7.11 – Number of multicast messages.

Number of messages that can be relayed within the VSN does not significantly vary with the number of nodes that are observing the phenomenon (Figure 7.12(a)). When many nodes detect the phenomenon, it is possible to select many source-destination pairs. Although this reduces the workload on individual nodes/CHs the root node is still the bottleneck. Therefore, three scenarios do not vary their network capacity with increasing number of VSN members. However, number of event nodes has a much higher negative impact on the multicast messages (Figure 7.12(b)). With increasing number of VSN members, the multicast messages need to be forwarded to many nodes. This significantly increases the workload on CHs therefore reduce the network capacity. This extensive overhead can be reduced by sending a single broadcast instead of sending multiple messages for each recipient of the multicast message.

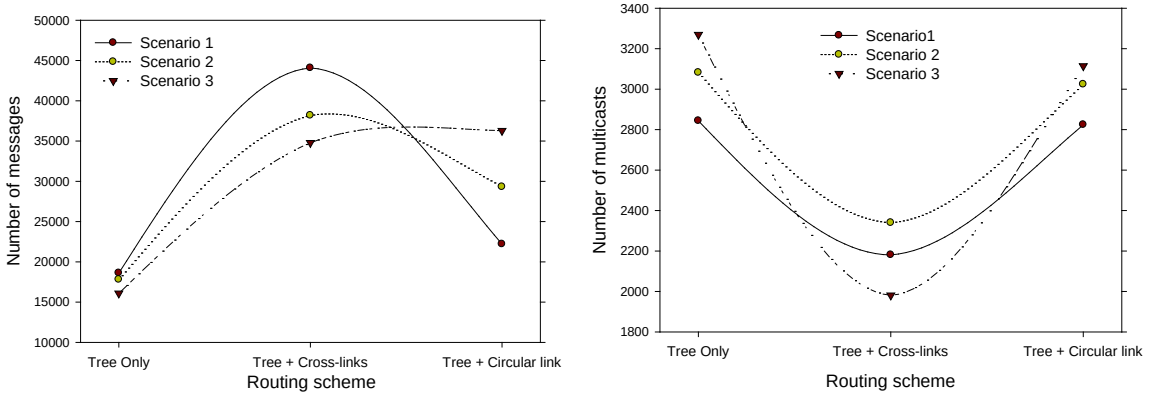
Figure 7.13 illustrate the impact on different routing schemes for unicast and multicast data delivery. These routing techniques are assumed to be available before

nodes form their VSNs. Availability of cross-links allows nodes to select better paths to reach their neighbors without going through the root node. Therefore, cross-links based routing significantly increase the number of unicast messages in all three scenarios. To make use of the circular path, VSN members should be outside of the circular path and needs to be geographically distributed. Therefore, the circular path increases the network capacity of the third scenario where VSN members are distributed within the entire network. However, it does not significantly increase the network capacity of scenarios one and two which have somewhat localized events.



(a) (b)

Figure 7.12 – Variation in number of messages with different number of VSN members. (a) – Unicast messages, (b) – Multicast messages.



(a) (b)

Figure 7.13 – Messages delivered with different routing schemes. (a) – Unicast messages, (b) – Multicast messages.

When cross-links are available during the VSN formation phase CHs always try to use the shortest path through neighbors to reach the root node. This increases the number of branches and CHs in the virtual tree. When a multicast message is send, it has to be forwarded to all these CHs through many branches. This increases the workload on CHs that are part of the virtual tree, as a result number of multicast messages reduce. Having a circular path does not provide any advantage during the VSN formation phase as event messages are forwarded towards the root node. Therefore, both cluster tree and circular path based routing have the same performance.

It is not essential that a designed node (i.e., root node) initiate the GTC algorithm. The root node does not need to be placed in the middle of the sensor field. A node that detects a certain phenomenon can initiate the cluster formation process by itself. Multiple cluster trees will be formed if several such nodes initiate the cluster formation around same time. Each such tree can be considered as a separate VSN and can be given different VSN identifiers. Such two cluster trees are shown in Figure 7.14. For simplicity, only the CHs are shown. New mechanisms need to be developed to facilitate communication within these VSNs.

7.5.3 Close Loop System

The close loop system shown in Figure 7.4 is simulated using synthetic data that simulates the migration pattern of two plumes (Figure 7.15). Refer [8] for specific details of synthetic data generation. 1000 nodes are randomly placed in a 2500m×2000m sensor field. Three simulation scenarios are considered. The first scenario simulates a conventional WSN where all the nodes sample once a day. Second and third scenarios

allow nodes to sample once a day or once in every two weeks depending on the presence of the plume. Cluster members are active for two seconds while CHs are active for 25 seconds to maintain the cluster tree and to diminish issues related to clock skew. In addition, the third scenario couples PMP system to form a close loop. Plume prediction model is executed in every eight weeks and it predicts the migration pattern of the plume for the next eight weeks. Data is collected over three years (1095 days). Results are based on 100 samples. See Section A.5 for specific simulation parameters.

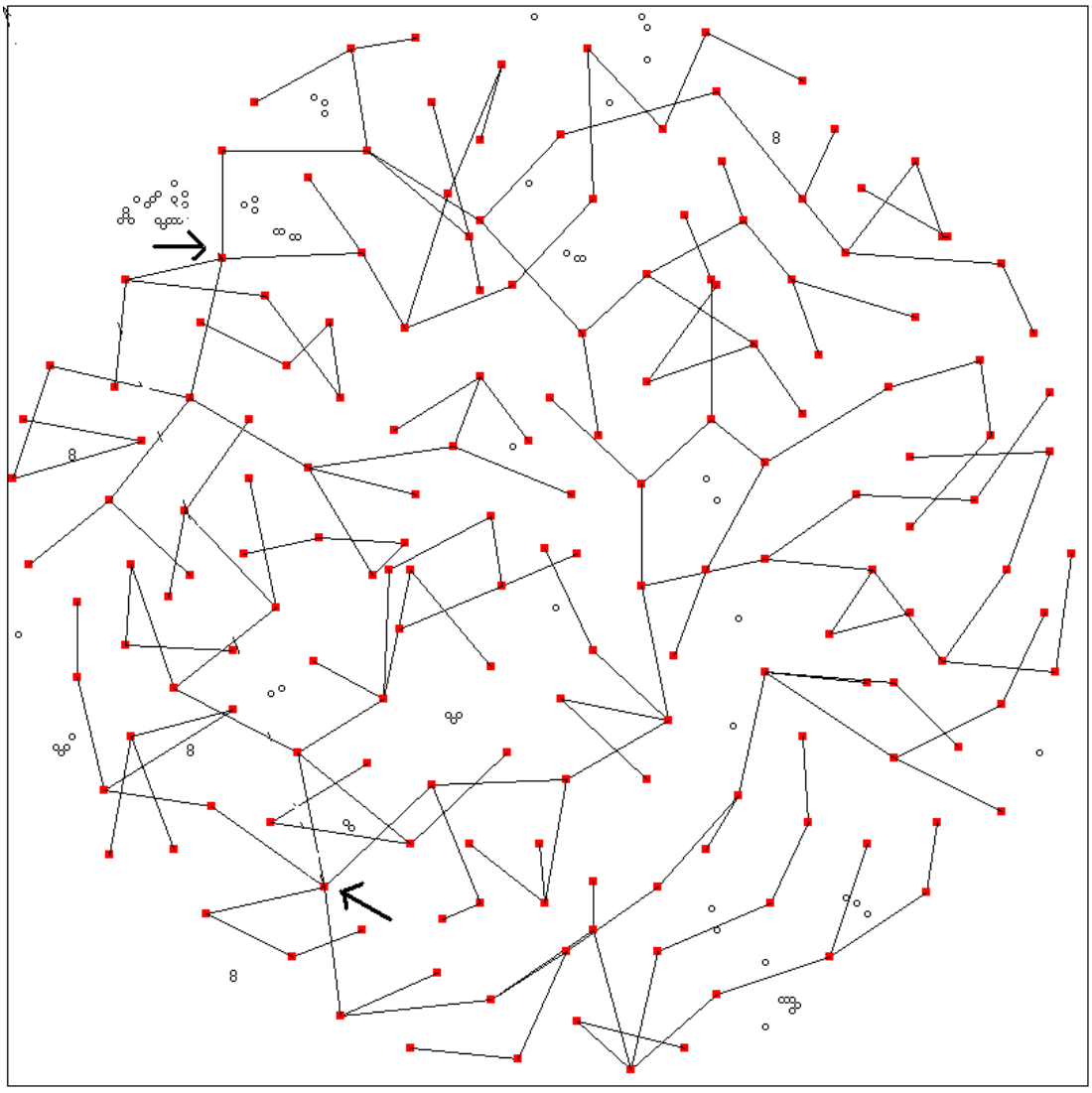


Figure 7.14 – Event based cluster tree formation. Arrows indicate the root node of each cluster tree, P_T = *-20dBm*.

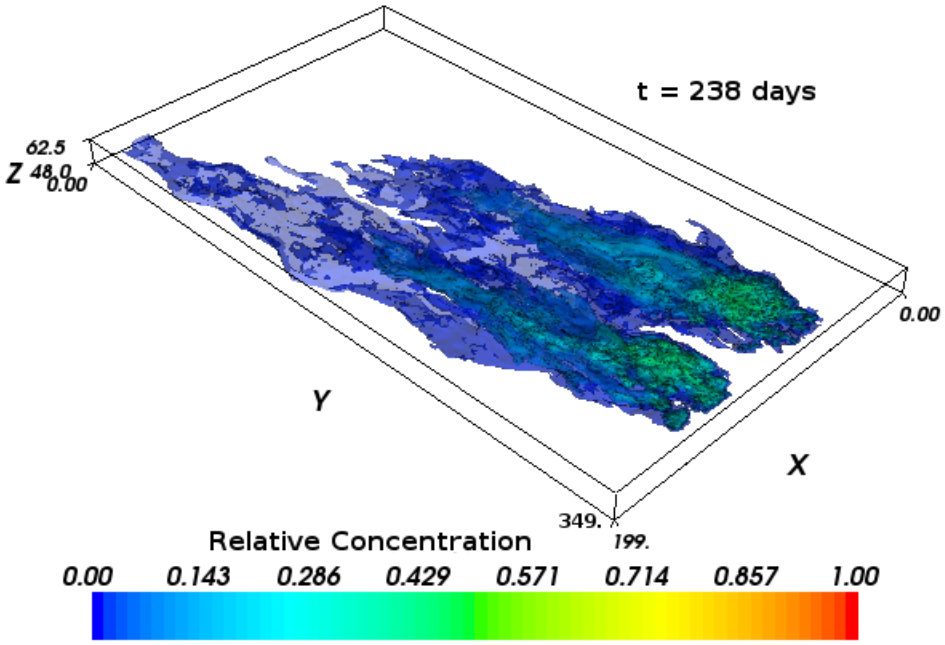


Figure 7.15 – Position of two migrating plumes at day 238.

Figure 7.16 shows the total energy consumption over three years. Both VSN based schemes are able to reduce total power by ~20KJ (20J per node). This is achieved by allowing nodes that are not in a plume to sample slower. Active to sleep power ratio is 3000:1; however, node duty cycle is 2/86400 (or 2/1209600 if samples in every 14 days) for a cluster member and 25/86400 for a CH. Hence, sleep power dominates in such a slow phenomenon that is monitored over several years. This is the reason that we do not see a significant performance improvement in Figure 7.16. With the advancement of technology, nodes are expected to be more energy efficient; therefore, sleep power will not be a significant issue in future. Nevertheless, actual overhead of a VSN based system depends on node's active power, sampling power, and communication cost. Therefore, we analyze the incremental power in Figure 7.17. The ~20KJ saving is clearly visible in Figure 7.17 and this is a 37% improvement over the standard network. VSN and close

loop system consume slightly higher amount of energy than the VSN only case, because of the prediction messages that goes back to the nodes.

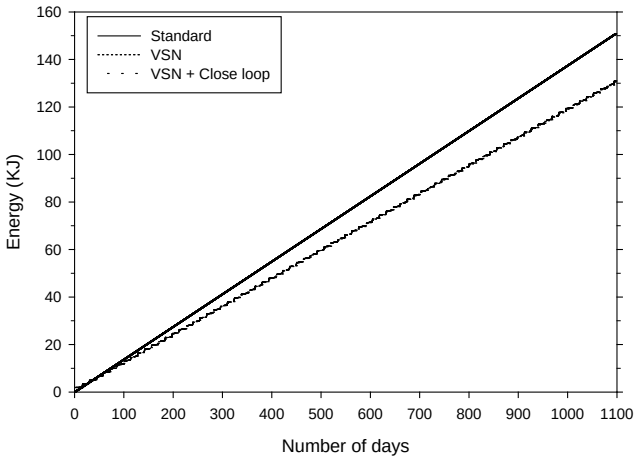


Figure 7.16 – Energy consumed while tracking the plume.

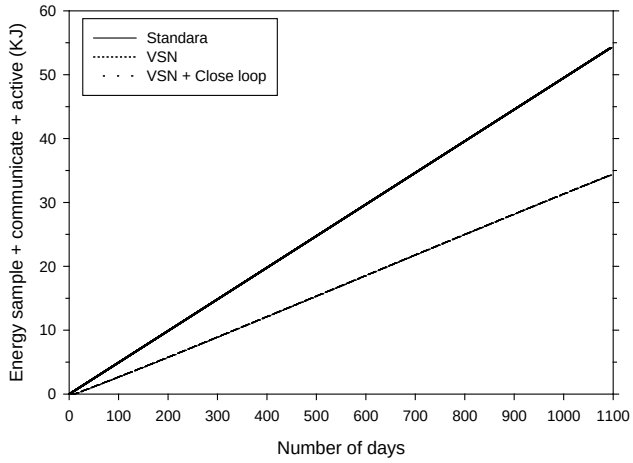


Figure 7.17 – Incremental energy consumed while tracking the plume.

Figure 7.18 shows the results from our energy model for a similar network. Both the simulations and the energy model provide identical results for the standard network. However, the model underestimates the VSN energy consumption. Instead of using time dependent values for number of nodes in the plume ($n_{c\_in\_p}^{\Delta t_i}$), number of CHs in plume (

$n_{CH\_in\_p}^{\Delta t_i}$), and number of CHs in the backbone ($n_{CH\_VSN}^{\Delta t_i}$) we use average values observed from the simulations. This may be the reason that our model under estimates.

To determine the energy saving for a much faster phenomenon, we speedup the plume migration and assume it to be a hazardous gas cloud. We consider each day in the synthetic data to be five-minute interval hence simulated over 91.25 hours. Results were analyzed using both the simulation and energy model. Nodes detecting a hazardous gas sample in every five minutes while other nodes sample in every 30 minutes. Predictions are given in every hour and with a prediction window of one hour. All the other parameters were identical to the plume tracking simulation. Results are shown in Figure 7.19. The VSN based scheme consumes 32.5% less power than the standard network and saves ~17.75KJ within 91.25 hours. Active and sampling power dominates the sleep power because nodes now sample at a much higher rate than the plume tracking simulation (in every five or 30 minutes instead of one or 14 days). 17.75KJ saving over 91.25 hours is much better than 20KJ over the three years. This further strengthens our claim that VSNs can reduce the power consumption by adapting sampling rates.

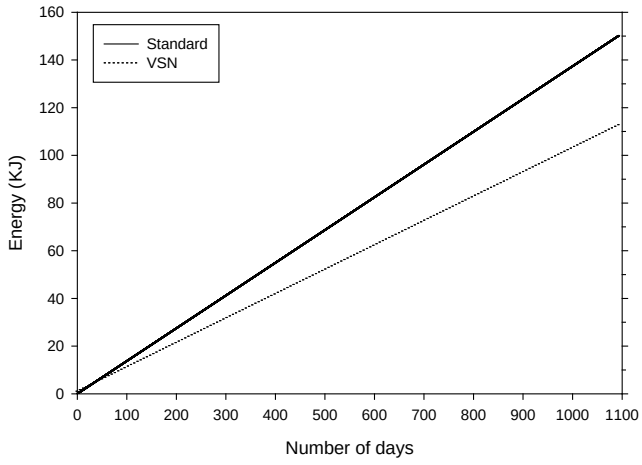
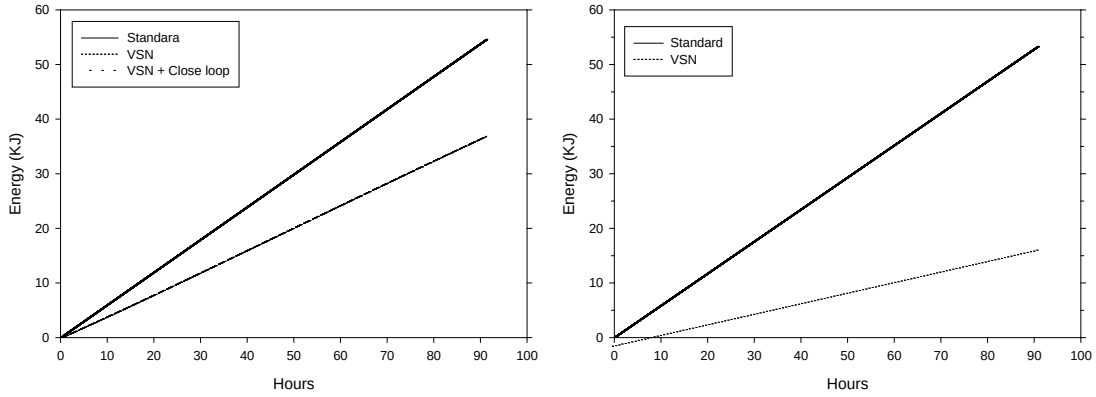


Figure 7.18 – Energy consumed while tracking the plume based on the energy model.

Figure 7.20 show the amount of data transferred between the nodes and the PMP system. Both the standard and VSN based plume tracking system transfer data only to the PMP system where as the close loop system also send messages back to the individual nodes, in the form of predictions. These predictions are useful in reducing the number of missed events. However, such prediction messages somewhat increase both energy consumption and the amount of data transfer between the nodes and close loop system. If predictions are infrequent, as in our plume monitoring case, this overhead is not a significant issue particularly when it reduces the number of missed events. Initially, all the nodes in the network report to the PMP system to indicate their presence in the network. This accounts for the initial ~50KB of data at day 0.



(a) (b)

Figure 7.19 – Energy consumed while tracking hazardous gases. (a) – From simulator, (b) – From energy model.

7.6 Summary

A VSN is formed by connecting nodes observing the same phenomenon. Such a node generates a message that indicates its interest to become a member of a VSN. The message is forwarded towards the root node. As the message travels through the cluster

tree it gets to know about other nodes/CHs that are observing the same phenomenon. All these nodes form a virtual tree that connects one or more VSNs. The logical tree can facilitate both inter-VSN and intra-VSN communication. When the phenomenon is localized cluster tree can deliver more unicast messages. Number of multicast messages significantly reduces with increasing number of nodes that detects the phenomenon. In certain scenarios, cross-links based routing and circular path based routing increase the number of unicast and multicast messages. Therefore, capacity of the network is determined by number of event nodes and how they are distributed within the network. Our simulation-based results further suggest that VSNs can reduce the energy consumption of a network.

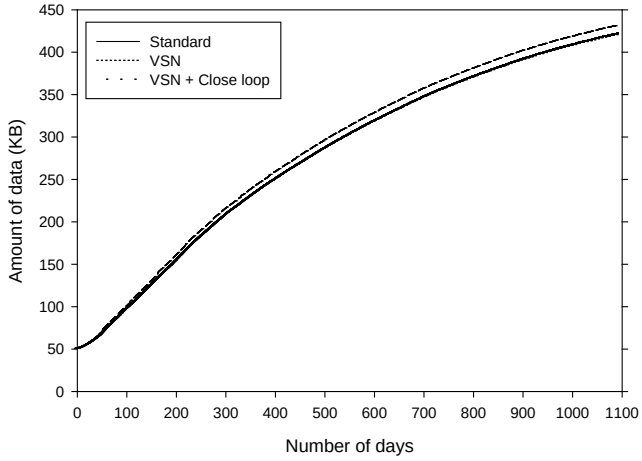


Figure 7.20 – Amount of data transferred between node and plume monitoring and prediction system.

Chapter 8

SECURE BACKBONE DESIGN

Security is a prime concern in large-scale wireless sensor networks used for collaborative and mission critical applications. For example, “future earthquake monitoring systems are expected to be coupled with the electricity grid, gas distribution systems, elevators, traffic lights, etc. that are to be turned off automatically when an earthquake is detected” [30]. If not adequately protected, a malicious attacker can generate false alarms at the sensors and cause massive denial of service. Privacy among different uses is another key requirement in collaborative WSNs.

In order to achieve these application objectives, WSNs at a minimum should provide secure and authenticated communication among sensor nodes. Nodes should be able to communicate securely with each other to relay sensed data and take network wide decisions. In certain applications, nodes should also build some trust relationships with neighbors that generate messages and ensure integrity of the sensed data. Secure and efficient distribution of cryptographic keys is the first step towards achieving these objectives, on top of which many secure protocols can be implemented.

A secure backbone based on the cluster tree can facilitate secure data delivery, dynamic key distribution, and re-keying which are some of the fundamental requirements of secure, large, and collaborative WSNs. Virtual sensor networks can make use of this

secure backbone to distribute keys among dynamically varying subset of sensors. We extend our GTC algorithm to form such a secure backbone using pre-distributed keys.

The secure backbone formation algorithm and how certain desirable characteristics can be achieved are presented in Section 8.1. Section 8.2 presents the simulation results.

8.1 Secure Backbone Formation

8.1.1 Secure GTC Algorithm

The GTC algorithm that forms a secure backbone is shown in Figure 8.1. Extensions to the previous algorithm (Figure 5.1) are underlined. The secure backbone formation is an integral part of the GTC algorithm therefore; it does not introduce any additional overhead other than the cost of sharing cryptographic key identifiers. Only the new changes/additions to the algorithm are described in this section.

As usual, the root node initiates cluster formation by executing the ***Form_Cluster*** function. In addition to cluster formation parameters, the root node also sends its cryptographic key IDs ($keyIDs_{CH}$). All other nodes execute the ***Join_Cluster*** function and waits for a cluster formation broadcast. For simplicity, this broadcast is assumed to be unencrypted. If required, the root node may use a challenge response scheme to authenticate child nodes. A node hearing this broadcast tries to join the cluster if it is not already a member of another cluster and within $hops_{max}$. The node has to share at least one common key with the CH to become a member of the cluster. Common keys are determined using the ***Common_Keys*** function. If no such key exists, the node re-executes the ***Join_Cluster*** function and tries to join a different CH. If a common key exists, the

```
Form_Cluster(NID_CH, CID_CH, delay, n_CCHs, hops_max, TTL_max, depth, keyIDs_CH)
 1  Wait(delay)
 2  TTL ← TTL_max
 3  Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth, keyIDs_CH)
 4  ack_list ← Receive_ACK(NID_child, hops, P_1, P_2,keyIDs_child, timeout_ACK)
 5  IF(ack_list = NULL)
 6      Join_Cluster()
 7  FOR i = 1 TO n_CCHs
 8      CCH_i ← Select_Candidate_CHs(ack_list)
 9      CID_i ← Select_Next_CID(i)
10      delay_i ← Select_Delay(i)
11      depth_i ← depth + 1
12      Request_Form_Cluster(CCH_i, CID_i, delay_i, n_CCHs, hops_max, TTL_max, depth_i)

Join_cluster()
13  Listen_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth, keyIDs_CH)
14  TTL ← TTL – 1
15  hops ← TTL_max - TTL
16  IF(hops ≤ hops_max AND my_CID = 0)
17      IF(Common_Keys(my_keyIDs, keyIDs_CH) ≠ NULL)
18          my_CID ← CID_CH
19          my_CH ← NID_CH
20          my_depth ← depth + 1
21          Send_ACK(my_NID, hops, P_1, P_2, my_keyIDs)
22  IF(TTL > 0)
23      Wait(Random(time_backoff))
24      Forward_Broadcast_Cluster(NID_CH, CID_CH, hops_max, TTL_max, TTL, depth, keyIDs_CH)
25      IF(hops ≤ hops_max)
26          Exit()
27  ELSE
28      IF(Common_Keys(my_keyIDs, keyIDs_CH) ≠ NULL)
29          IF(Wait_Listen_Neighbors(Random(time_backoff)) = FALSE)
30              Send_ACK(my_NID, hops, P_1, P_2)
31              IF(Listen_Form_Cluster(CCH, CID, delay, n_CCHs, hops_max, TTL_max, depth,
    timeout_CCH) = TRUE)
32              Form_Cluster(my_NID, CID, delay, n_CCHs, hops_max, TTL_max, depth,
    my_keyIDs)
33              Exit()
34  Join_cluster()
```

Figure 8.1 – GTC algorithm that forms a secure backbone.

node joins the cluster by setting relevant parameters. Then an acknowledgment (ACK) is sent to the CH to confirm its cluster membership. In addition to the node ID, distance to the CH (*hops*), and properties p_1 and p_2 the ACK also includes list of child's key IDs

(*my_keyIDs*). The CH receiving the ACK (line 4) adds the node to its acknowledged list (*ack_list*). Child's key IDs are used by the CH to determine a common key.

After sending the ACK the child node forwards the broadcast, if *TTL* is not expired. Intermediate nodes just forward the broadcast until *TTL* expires. The nodes that forward the broadcast do not need to share a common key with the CH or with its neighbors, if inter-cluster communication is single-hop. If it is multi-hop, intermediate nodes along the communication path have to share keys with their neighbors. However, it is not necessary to share a key if messages are only encrypted/decrypted at the CHs. Intermediate nodes can just relay the messages without looking into its content. Therefore, we do not check for any common keys in these intermediate nodes.

If *TTL* is expired (line 27), the receiving node is a potential child CH. These nodes have to share a common key with the CH, if they are to be selected as child CHs (line 28). If a common key exist, a node sends an ACK indicating its interest to become a child CH. Before sending the ACK, node waits sometime and listens to the channel to make sure none of its neighbors are interested in becoming a CCH. A node may also send an ACK if it does not share a common key with the neighbor that already sent an ACK, i.e., it cannot join the neighbor's cluster because it does not share a key with the neighbor. Key IDs of CCHs are also sent as part of the ACK. The CH uses these key IDs to determine the common key. Some of the CCHs that receive a cluster formation request, from the parent CH (***Listen_Form_Cluster*** function), form their own clusters. This process continues until the entire sensor field is covered.

8.1.2 Achieving Desirable Characteristics

The solution generated by the algorithm depends on the implementation of different functions and selection of parameters. Only the parameters and functions related to secure backbone formation are described here. Refer Sections 4.2 and 5.1 for other functions and parameters.

Only the nodes that share a common key with the parent CH should be selected as cluster members and CCHs. In the bottom-up approach, clusters are formed independently and later connected together to form a cluster tree. Because CH selection is fully distributed, it is not possible to determine whether two CHs share any common keys during the cluster formation phase. Therefore, these CHs are not guaranteed to connect together and form a fully connected cluster tree. Alternatively, top-down cluster formation allows us to specifically select nodes that share a common key(s) with the parent CH. This ensures that any parent and child CH pair is connected and can securely communicate with each other.

Local connectivity defines the fraction of neighbors that a node shares at least one common key. Higher local connectivity is important to ensure that most of the nodes can join the closest CH. Therefore, underlying key pre-distribution scheme needs have a higher probability of sharing at least one key with each neighbor. A CH has to broadcast all its key IDs during the cluster formation phase so that its neighbors can identify at least one common key. While sending the ACK a node needs to send only the common key ID that was selected to communicate with the CH. If the overhead is not significant, it is desirable to send all the common key IDs to the CH because it is useful in key revocation. The implementation of ***Common_Keys*** function depends on the key pre-distribution

scheme. If a random key pre-distribution scheme is used the function needs to compare the key space of two nodes. If the key pre-distribution scheme is complex, it needs to implement a function that determines one or more common keys. Such a function for combinatorial approach is presented in [42].

A better set of CCHs can be selected if the local connectivity of a node is known in advance. Local connectivity data can be utilized by modifying line 29 of the algorithm:

29 ***IF****(Wait_Listen_Neighbors(d_0/d + Random($time_{backoff}$)) = FALSE)*

where d_0 is the node degree (i.e., number of neighbors) and d is the number of neighbors that share at least one key with the given node. If a node shares a key with most of its neighbors, d_0/d will be closer to one. This reduces the waiting time and allows the node to send the ACK earlier than its neighbors with lower local connectivity. A node with lower connectivity (higher d_0/d) replies only if it does not hear from another CCH. The parent CH can then select some of these CCHs with higher connectivity as its child CHs. This allows us to form much larger and less overlapping clusters.

Neighborhood discovery can be active or passive. In the active approach, nodes get to know about their neighbors either before the cluster formation phase or as part of the clustering algorithm. In the former case, each node may send a broadcast with its list of key IDs so that all the neighbors can determine a common key(s). The later approach uses cluster formation messages to share this information. This increases the complexity of the clustering scheme and its overhead. A passive approach can be used in cluster formation solutions such as [33, 67] that cycle over time. Nodes can keep track of their neighbors' key IDs whenever broadcasts or ACKs occur in different cycles of the cluster formation process. Over time, each node can gain better understanding of its

neighborhood and their keys. Therefore, after several iterations even these solutions can form clusters and a secure cluster tree with better characteristics.

Each parent-child cluster pair identifies their common key(s) by the time the cluster formation phase is completed. Child CH uses the common key to decrypt the messages encrypted by the parent CH or vice versa. The secure backbone is formed by securely connecting these parent-child CHs pairs in the cluster tree. Common key used by each parent-child CH pair may be different. Therefore, a message traveling through secure backbone needs to be decrypted and re-encrypted at each CH. This approach is costly therefore should not be used to deliver messages frequently. Instead, end-to-end encryption can be utilized by assigning a shared key to nodes, CHs, or VSNs that wish to communicate with each other. This is similar to the Virtual Private Network (VPN) concept. The secure backbone can be used to securely share the key between two end points by periodic encryption and decryption at each relay node. It is appropriate to use the secure backbone for dynamic key distribution because such functions are infrequent.

In VSNs, it would be desirable to use a shared key within the entire VSN. This provides privacy while preventing periodic encryption/decryption at all the CHs that facilitates the communication within or across VSNs. However, to ensure security this VSN wide key needs to be periodically changed. VSNs may also include dynamically varying subset of sensors. Whenever a new node joins a VSN, it needs to be given the shared key. When some of the members leave a VSN, shared key may need to change. Therefore, a secure backbone is essential to distribute such dynamic keys securely. Hence the secure GTC algorithm is able to provide a secure communication infrastructure that can be used to dynamically distribute keys within or across VSNs.

8.2 Performance Analysis

Two key pre-distribution schemes are used to evaluate the performance of the secure backbone formation with HHC scheme. The first approach is based on the Deployment Knowledge based Random key pre-distribution (DKR) scheme [26] and the second approach is based on the Random Block Merging in Combinatorial Design (RBMCD) [16]. Both schemes have a much higher local connectivity than most other key pre-distribution schemes. Based on the simulation parameters (Table A.6), RBMCD approach shares 3-4 common keys with its neighbors while DKR shares 4-5 common keys. 5000 nodes are distributed across the network based on a 2-D Gaussian distribution to facilitate the requirement of DKR. Such a node placement scheme did not alter the performance of the original GTC algorithm or RBMCD based cluster formation. The results are based on *100* sample runs (*20 random networks × 5 samples per network*). Refer Appendix A for specific simulation parameters. Following acronyms are used to identify different clustering mechanisms:

- HHC + DKR – HHC clusters and cluster tree formation with deployment knowledge based random key pre-distribution.
- HHC + DKR-Nei – HHC clusters and cluster tree formation with deployment knowledge based random key pre-distribution that make use of neighbor information.
- HHC + RBMCD – HHC clusters and cluster tree formation with random block merging in combinatorial design.
- HHC + RBMCD-Nei – HHC clusters and cluster tree formation with random block merging in combinatorial design that makes use of neighbor information.

Figure 8.2 shows the circularity of clusters formed by each solution. HHC clusters without any key pre-distribution has the highest circularity. Both the random key selection approach and the combinatorial approach that make use of neighborhood information form more circular clusters than their standard schemes. Local connectivity of nodes affects circularity of clusters in several ways. If the connectivity is high, most of the neighbors can connect to the CH hence circularity increases. If a node does not share a common key with the CH, it may try to connect to another CH that is within its transmission range (*R*). Such a node may also become a CCH if it shares a common key with another CH that is within 3-hops. It is also possible that a node shares a key with the CH but may not share a key with its neighbors that are already selected as CCHs. Those nodes also try to become CCHs. If two nearby CCHs are selected to form clusters, their cluster will overlap. The RBMCD has a lower local connectivity. These factors reduce the circularity of clusters. Higher local connectivity in DKR helps it to form clusters that are more circular. When data about neighbors is available, nodes with higher local connectivity can be selected as child CHs. Such nodes can form bigger clusters. As a

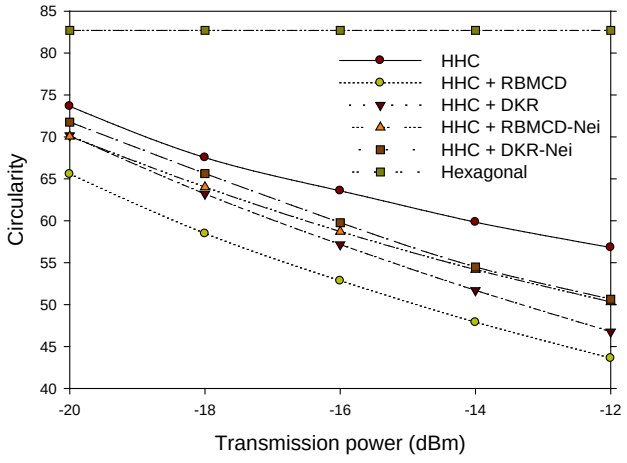


Figure 8.2 – Circularity of clusters.

result, circularity increases. As observed earlier, circularity reduces with the increase in transmission power (P_T).

Figure 8.3 illustrates the number of clusters formed by each approach. Results are not significantly different from the hexagonal packing. HHC clustering without any key pre-distribution produces the lowest number of clusters. Availability of neighborhood information increases the circularity of clusters. When clusters are more circular, number of clusters required to cover a given sensor filed reduces. As a result, both RBMCD-Nei and DKR-Nei produce relatively lower number of clusters than when they do not have neighbor information. Higher local connectivity in DKR can form more circular and bigger clusters; therefore, it forms lower number of clusters than the RBMCD. Clusters become much larger as P_T increases therefore number of clusters produce by each solution reduces with increasing P_T.

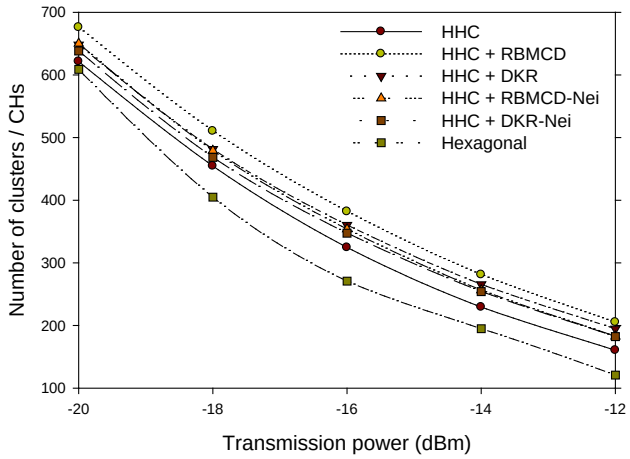


Figure 8.3 – Number of clusters and cluster heads.

Cluster size distribution is shown in Figure 8.4. Use of key pre-distribution somewhat reduces the cluster size. As the transmission power increases area covered by a cluster increases compared to the reduction in circularity. Therefore, clusters become

larger as P_T increases. The RBMCD based clusters has the lowest cluster size as it forms the highest number of clusters (lower circularity). Availability of neighbor information allows the formation of dense cluster therefore both DKR-Nei and RBMCD-Nei have relatively higher cluster size.

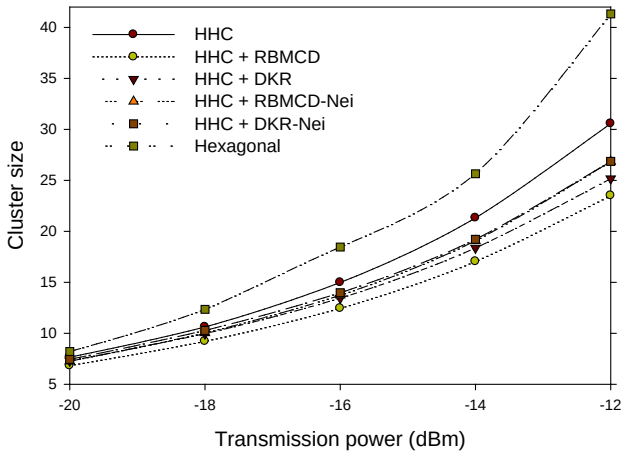


Figure 8.4 – Number of nodes in a cluster.

Figure 8.5 shows the fraction of nodes that are not in a cluster. Nodes can be disconnected from rest of the network due to several reasons. In a randomly deployed network, certain nodes may be isolated from rest of the network due to their location. Some of the isolated nodes can be connected by increasing transmission range of a node. Nodes may not hear cluster formation messages due to collisions. Due to these two reasons around 1-5% of the nodes are anyway disconnected in HHC. In key pre-distribution based networks, nodes can also be disconnected if they do not share common keys with neighbors. Therefore, DKR and RBMCD have higher number of disconnected nodes than HHC without any key requirements. Compared to DKR, 2.5% of additional nodes are disconnected in RBMCD due to lower connectivity. However, availability of neighborhood information significantly improves the performance of RBMCD. This is

because child CHs are always selected from nodes having the highest connectivity in their neighborhoods.

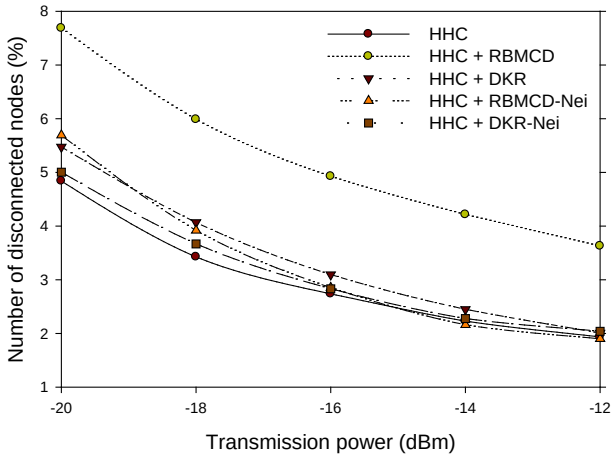


Figure 8.5 – Number of nodes not in a cluster.

From Figures 8.2 to 8.5 it can be seen that these key pre-distribution schemes do not significantly reduce the performance of the HHC scheme. Higher local connectivity is an important property in hierarchical cluster formation because local connectivity of the key pre-distribution scheme directly affects the performance. Availability of neighborhood information can further improve the performance.

Figure 8.6 shows the control message overhead of each solution. The solutions that make use of the neighborhood information have the lowest overhead. It is even lower than the HHC without any key requirements. This was due to the reduction in ACK messages. Nodes with higher connectivity get higher priority in sending their ACKs as CCHs. When a node with higher connectivity sends an ACK, most of its neighbors do not need to send another ACK as they share a key with that node (i.e., if that node is selected as a child CH most of its neighbors can join the new cluster). This reduces the total number of ACKs in the network hence reduces the overhead. In RMBCD approach,

many broadcasts (many nodes are unable to respond to a broadcast) and ACKs (too many CCHs) are wasted due to lack of a common key between CHs and neighbors. Therefore, has a higher overhead. As P_T increases number of neighbors of a node increases. If the local connectivity is lower, most of these nodes may not share a common key with their neighbors. As a result, there will be many CCHs that generate ACKs. Most of these ACKs are wasted because only a subset of CCHs is selected to form a new set of clusters. Therefore, overhead of RMBCD increases with increase in P_T.

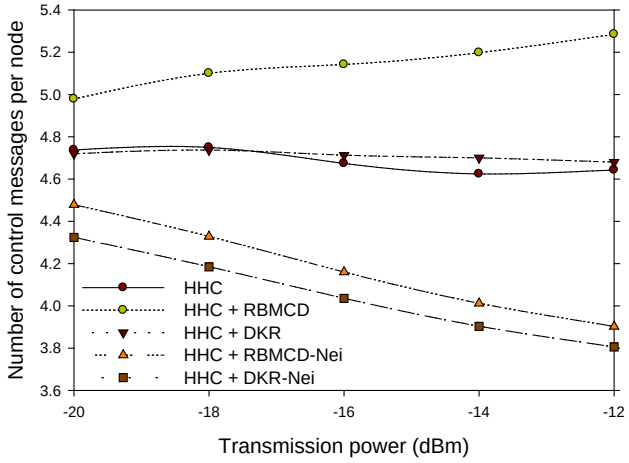


Figure 8.6 – Number of control messages per node.

Figure 8.7 illustrates the control message overhead in terms of message size. Based on our simulation parameters (Table A.6) overhead is calculated as follows. The RBMCD needs 16-bits to indicate a single block ID while DKR needs 24-bits to indicate a key ID. The RBMCD sends four such blocks within a cluster formation broadcast while DKR has to send 120 key IDs. It is assumed that the ACKs contain only the common key ID or group ID. Then the overhead can be calculated from the following equation:

$$\textit{Broadcasts} \times \textit{size of key/group IDs} + \textit{ACKs} \times \textit{size of a key/group ID} \qquad (8.1)$$

For simplicity, we only consider the contribution of key IDs and group IDs to the message payload. Message header size and other parameters (*NID*, *CID*, *depth*, etc.) are independent of the key pre-distribution scheme. Combinatorial approaches use a single group ID to represent a set of keys; therefore, RBMCD has a significantly lower overhead than DKR. In WSNs, even reduction of a single bit is important. Therefore, if the size of a control message is much larger it can significantly affect the performance. Though overhead of DKR is much higher, it produces clusters with better characteristics even without neighborhood information. These characteristics may be more important in long-lived WSNs even through cluster formation overhead is significantly higher.

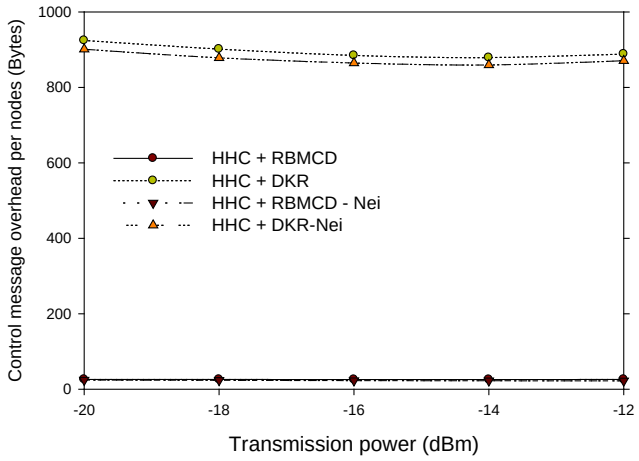


Figure 8.7 – Control message overhead.

Though RBMCD supports up to 5550 nodes (Table A.5) we only deployed 5000 of them in our simulations. When all the 5550 nodes were deployed in the network, RBMCD had much better performance, which was comparable with DKR. In combinatorial design, we need to depend on prime numbers that determines how many times to replicate a given key in multiple nodes [16, 42]. Therefore, designing a network with exact number of sensor nodes is not possible. Local connectivity of RBMCD can be

increased by storing more keys in a node. If nodes have enough flash memory to store 150-200 keys, RBMCD based cluster formation would be a better approach than DKR due to its lower overhead.

Figure 8.8 shows the distribution of CHs in the cluster tree. All the schemes have somewhat similar CH distribution. The ones with neighborhood information have a lower depth and higher breadth. This is due to the fact that they form much bigger and uniform clusters. Nevertheless, these cluster trees include more CHs than the HHC without any key requirements.

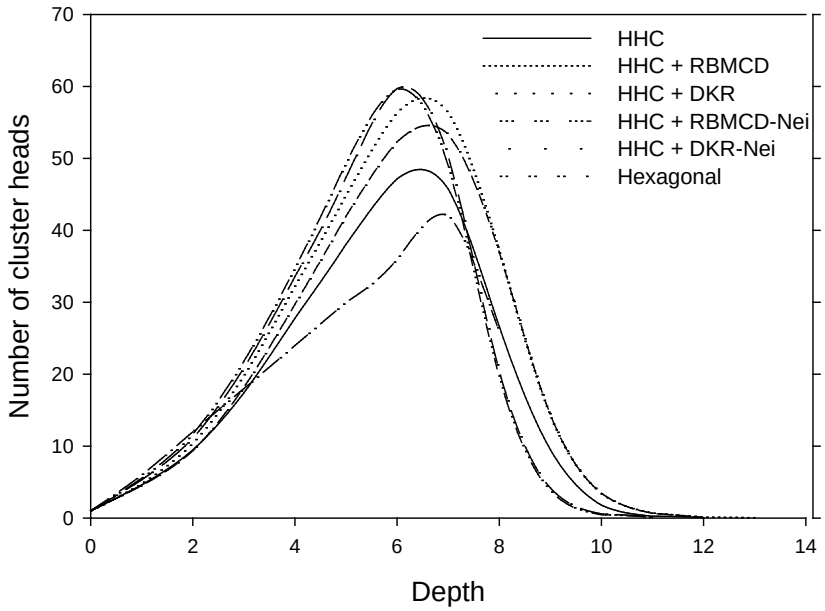


Figure 8.8 – Distribution of CHs in the cluster tree.

When a set of nodes are compromised, attacker gets access to all the keys stored in those nodes. This will directly compromise CHs that uses those keys to secure their links. In hierarchical networks child CHs make use of parent CHs to forward their data. If such a parent CH is directly compromised all the child CHs and their cluster members are indirectly compromised. The impact of random node compromise is shown in Figure 8.9.

As claimed in most key pre-distribution algorithms, impact of direct compromise is lower. However, the indirect compromise is much more significant. When the number of compromised nodes is lower, DKR has a lower number of compromised CHs (direct and indirect). It was observed that these compromised CHs are localized within the region where compromised nodes reside (i.e., in DKR, key pools are overlapped only within certain regions). Number of compromised CHs within the localized region was much higher due to higher local connectivity. Therefore, when nodes from multiple regions are compromised more and more CHs are directly and indirectly affected.

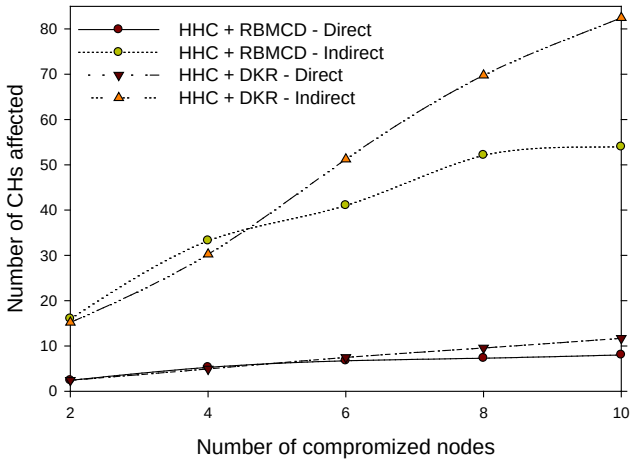


Figure 8.9 – Direct and indirect impact of compromised nodes.

All the nodes in RBMCD have equal likelihood of being directly compromised, regardless of their location. Therefore, the number of nodes that is directly compromised is lower compared to DKR. It was also observed that number of indirectly affected CHs in DKR depends on what nodes are compromised. If the compromised nodes are much closer to the root node, the effect is significant. This is due to the higher connectivity within a given region, which can easily compromise many level 1 CHs. This behavior is not so prominent in RBMCD because of the equal likelihood of direct compromise.

8.3 Summary

The GTC algorithm is extended to form a secure backbone. The algorithm is independent of the pre key-distribution scheme. The algorithm retains most of the desirable cluster and cluster tree characteristics while building a secure cluster tree. Local connectivity of the key pre-distribution scheme directly affects the performance. Therefore, it is important to select a key pre-distribution scheme with higher connectivity. Availability of neighborhood information can further improve the performance. Though DKR retains most of the cluster and cluster tree characteristics, its control message overhead is significant. Combinatorial key pre-distribution schemes are more attractive due to lower overhead. Simulations also suggest that hierarchical networks are more vulnerable to node capture than non-hierarchical networks.

Chapter 9

SUMMARY

Recent technological advances are enabling the deployment of large-scale and collaborative WSNs. Virtual Sensor Networks (VSNs) is an emerging concept that supports collaborative, resource efficient, and multipurpose sensor networks that may involve dynamically varying subset of sensors and users. The goal of the thesis was to design algorithms and protocols that support the formation, usage, and maintenance of VSNs. This chapter provides a concluding summary of work presented, key contributions, and future directions.

9.1 Conclusions

Imposing some structure within the sensor network to effectively achieve application objectives is an attractive option for the self-organization of large-scale and collaborative WSNs. Cluster based organization and arranging clusters in the form of a tree simplify many higher-level functions and distributed application deployments. However, achieving all the properties within a single cluster and tree formation algorithm is not trivial.

Generic Top-down Cluster and cluster tree formation (GTC) algorithm was proposed that achieves most of these properties. The algorithm uses a hybrid top-down

cluster formation approach that combines local and neighborhood information. Use of top-down approach allows the GTC algorithm to control the number of nodes in a cluster, distances between parent and child CHs, breadth and depth of the cluster tree, and provides the ability to select nodes that share common cryptographic keys. The algorithm is configurable, independent of network topology, and does not require a-priory neighborhood information, location awareness, or time synchronization.

Simple Hierarchical Clustering (SHC) is a special case of GTC algorithm that is similar to the IEEE 802.15.4 cluster tree [38]. Another special case, Hop-ahead Hierarchical Clustering (HHC), produces cluster and cluster trees with much better properties. HHC clusters are more circular compared to the clusters formed by FLOC [25] and PHC [9]. Receiver Signal Strength Indicator (RSSI) measurements can be utilized to further improves the properties of the algorithm. The HHC forms more uniform and circular clusters, a cluster tree with a lower depth, and a more ordered network. The properties of HHC are comparable with hexagonal packing particularly for low-density networks and lower transmission power levels. Distance between any parent and child CH is bounded to $3R$, where R is the transmission range of a node. Algorithm has a message complexity of $O(n)$, where n is the number of nodes in the network; therefore, it scales well for large networks. Two-step, cluster and cluster tree optimization phase was designed to further improve the properties. The optimization phase increases the connectivity of the network and enhances the cluster tree; however, it increases the overhead of algorithm. Given all these properties HHC is applicable for many large-scale and collaborative WSNs.

Three routing strategies were proposed that make use of the cluster tree produced by the HHC scheme of the GTG algorithm. A hierarchical addressing scheme that reflects the parent-child relationship among CHs was designed to facilitate node-to-node communication. The addressing scheme simplifies routing and significantly reduces the number of routing entries that needs to be stored in a CH. Variable length hierarchical addresses are used to reduce the overhead. First routing approach is based on the cluster tree and hierarchical routing. However, the root node becomes a single point of failure. Workload on the root node is reduced by forming cross-links within the cluster tree. In cross-links based routing, CHs make use of their neighbors to deliver messages through shorter paths and try to avoid the root node whenever possible. The third routing mechanism, referred to as circular path based routing, makes use of a circular path within the network to relay messages. Depending on the source and destination nodes, messages go either through the cluster tree or through a combination of circular path and cluster tree. Higher number of messages can be delivered by balancing the number of messages relayed by the root node and a node along the circular path. An analytical model is used to determine the best position of the circular path. Both cross-links based routing and circular path based routing at least double the network capacity.

Realization of VSNs requires design and implementation of many algorithms and protocols. As an initial step, a VSN formation mechanism and data delivery platform were presented. Nodes observing similar events send their interest to join a VSN towards the root node. Cluster tree formed with the HHC scheme is used to deliver such messages. Compared to random routing strategies such as Rumor Routing [14] and Ant Routing [35] these messages are guaranteed to meet at the root node. As the message

travels through the network, it discovers other nodes with similar interest and forms a virtual tree that connects members of the VSN. Multiple VSNs may form multiple virtual trees; however, every tree is guaranteed to meet at the root node. This virtual tree can be used to deliver unicast, multicast, and broadcast traffic within and across VSNs. Due to extensive overhead, number of multicast messages delivered by the network significantly reduces. Cross-links based routing scheme was able to at least double the number of unicast messages. However, it was not effective in delivering multicast messages. Circular path based routing scheme was not so effective unless the phenomenon being tracked is distributed. We further demonstrate the efficacy of the approach by simulating a subsurface chemical plume monitoring system. Though this approach does not utilize all the energy in the network, it provides the starting point for formation of VSNs and communication within and across VSNs.

Security is a prime concern in large-scale WSNs used for collaborative and mission critical applications. Secure and efficient distribution of cryptographic keys is the first step towards achieving these security objectives, on top of which many secure protocols can be implemented. Dynamic key distribution is one of the key requirements in secure VSNs that have a time varying set of VSN members. The GTC algorithm is further extended to build a secure backbone on top of the cluster tree, which can be used to distribute network wide or VSN wide keys among different VSNs and users. The extended GTC algorithm is independent of the key pre-distribution scheme. The algorithm retains most of its desirable cluster and cluster tree characteristics while building the secure backbone. Our analysis also shows that hierarchical networks are more susceptible to node capture than non-hierarchical networks.

The goal of the thesis was to design a set of VSN enabling technologies. A cluster and a cluster tree formation scheme, three routing schemes based on the cluster tree, a VSN formation and data delivery scheme, and a secure backbone formation scheme were designed with this objective. A compound solution that combines clustering, cluster tree based routing, and VSN membership discovery addresses the VSN formation problem and data delivery problem within and across VSNs.

9.2 Contributions

The contributions of the thesis include:

1) GTC algorithm
    - A configurable algorithm that is independent of the network topology, and does not require a-priory neighborhood information, location awareness, or time synchronization.
    - The HHC scheme, a special case of the algorithm, forms more uniform and circular clusters and cluster trees with lower depth.
2) Three routing strategies
    - Design of a hierarchical addressing scheme.
    - Cluster tree based routing.
    - Cross-links based routing that at least doubles the network capacity.
    - Circular path based routing further increases the network capacity.
3) Secure backbone design
    - Extended GTC algorithm that can builds a secure cluster tree using pre-distributed keys.
    - Algorithm is independent of the key pre-distribution scheme and retains most of its desirable cluster and cluster tree characteristics.
4) Formation of VSNs

- A cluster tree based VSN formation and data delivery mechanism.
- Achieved by combining contributions one and two.
- Subsurface chemical plume monitoring demonstrates the efficacy of the approach.

9.3 Future Directions

As a starting point towards virtual sensor networks, we designed a set of solutions that facilitate some of the fundamentals requirements of VSNs. However, realization of VSNs requires design and implementation of many other algorithms and protocols. Furthermore, it may be possible to improve proposed algorithms/techniques and adapt them to different applications. Below we discuss some of the possible future research directions.

To test our algorithms for large-scale networks with thousands of sensors, we had to build our own simulation platform. By doing so, we did not implement/simulate the underlying data link layer. It is important to test the performance of our algorithms on a rigorous simulation platform such as TOSSIM [43]. To simulate using TOSSIM, the GTC algorithm and routing schemes needs to be ported to TinyOS [61]. This will also enable us to test the algorithms in an actual testbed such as moteLab [32] before any field implementations.

It was observed that some of the branches in the cluster tree were longer than others and therefore had to handle more workload. This can be a major problem if the network is of an arbitrary shape. After forming the cluster tree, it may need to be rebalanced, e.g., by shifting the position of the root node, so that most of the branches can have similar depths. This further requires the reassignment of hierarchical addresses.

Though such a tree-balancing phase may require some additional overhead, it may extend the network life significantly.

Cross-links based routing and circular path based routing at least double the network capacity. However, these solutions are still not capable of utilizing the energy available in most of the nodes. All these solutions are too dependent on the cluster tree. Though formation of a hierarchy simplifies many functions of collaborative WSNs it is not the best approach when it comes to routing. To our knowledge, not much research has been carried out to overcome the over dependence on the hierarchy or the cluster tree. Applications that require in-network communication do not need to be tied to the cluster tree. Instead, they should maximize the network lifetime by identifying alternative paths to their destinations. Overhead of identifying such paths should be minimum if communication pattern is dynamic, e.g., VSNs. These routing strategies should be energy aware. Allowing CHs to share each other's information about available power levels can enable energy aware path selection. Therefore, it is important to design and develop a routing scheme that maximizes the network lifetime by utilizing most of the energy available in the network.

Heterogeneous sensor nodes may form their own cluster trees and can be considered as individual VSNs. Collaboration among these VSNs is required to effectively achieve each applications objective. For example, consider two different networks that are deployed to monitor rock sliding and animal crossing in a mountainous terrain. Both these networks can make use of each other's cluster tree to deliver their data effectively. The animal crossing network can be used to deliver data across rockslide-monitoring networks placed in two mountains. However, combining or connecting these

two trees is not straightforward. Some of the challenges include; discovering neighbor networks, how to connect multiple trees, where to connect them, how to uniquely identify each CHs address, routing across VSNs without putting extensive burden on each other, etc. Hierarchical addresses that reflect different networks or VSNs can overcome issues such as unique addressing. However, other issues such as detecting, combining, and managing multiple networks are not straightforward and need to be addressed.

VSN membership may change over time due to migrating, merging, splitting, or fading phenomenons. To reduce energy consumption nodes should be allowed to sleep while they are not in the event region. However, when the event moves towards those nodes, they need to be reactivated. The network should be able to predict such changes and inform those sleeping nodes in advance. Though managing these events in resource constrained WSNs is not straightforward, these issues are critical and need to be address to achieve the full potential of VSNs. We demonstrate limited use of such events.

A dynamic key distribution scheme needs to be developed on top of our secure backbone. Such a scheme should be able to assign VSN wide keys to facilitate secure communication within and across VSNs. Depending on the application's security requirements rekeying may be required as the VSN membership changes. It was observed that hierarchical networks are more vulnerable to node capture. This can be a major issue in mission critical larger-scale and collaborative WSNs, hence need to be addressed.

Appendix A

SIMULATOR

A.1 Node Placement

A discrete event simulator is developed using C. 2500, 5000, and 7500 nodes are randomly placed on a $L \times W$ grid. For performance analysis in Chapter 4, nodes were placed on a 101×101 grid with a grid spacing of $10m$ ($1000m \times 1000m$ network). Later, same number of nodes is placed on a circular region with a radius of $500m$. The region is embedded within a 201×201 grid and grid spacing is reduced to $5m$. Circular region is considered to make the comparison with hexagonal packing easier. This network is used in Chapters 5, 6, and 7. In Chapter 8, to facilitate Deployment Knowledge based Random key selection (DKR) scheme [26], 5000 nodes are randomly placed based on a 2-D Gaussian distribution. Grid size is increased to 1001×1001 and grid spacing is reduced to $1m$. Distance between two node deployment points in X and Y-axis is $100m$. Only the deployment points that are within the circular region are considered. This sort of a node placement did not affect the performance of GTC algorithm or Random Block Merging in Combinatorial Design (RBMCD) [16]. In all the networks the root node is placed in the middle of the network. For different node densities and network configurations, 100 networks are generated by randomly placing nodes in the grid. Simulation results are averaged over 100 samples and the pre-generated networks are reused when analyzing

each variable of interest. For each simulation, the random function is initialized with a different seed based on time.

A.2 Cluster And Tree Formation

Single-hop and multi-hop clusters are formed starting from the root node. Breadth-first spanning tree approach described in Section 4.2 is used to form the cluster tree. A subset of CCHs is randomly selected as child CHs. The root node selects six CCHs while all other CHs select three. CHs are assigned hierarchical addresses based on the scheme described in Section 6.1.1.

Characteristics of GTC clusters are compared with FLOC [25] and Probabilistic Hierarchical Clustering (PHC) [9]. Two more simulators are developed based on our network model and parameters given in [8] and [21]. For FLOC, the stretch factor m is selected as 2 ($o\text{-}band = 2 \times i\text{-}band$). The effective communication range R was selected as *o-band* ($R = o\text{-}band$). The random wait-time is chosen from the domain $[0 \ldots T]$ and T was appropriately selected based on the node density. Our multi-hop cluster formation is compared with PHC. In PHC, probability of selecting CHs and number of hops with a cluster depends on the area of the sensor field and node density. PHC needs to form 9-hop clusters for the network parameters described in Section A.1. Such large multi-hop clusters are not desirable. Therefore, to obtain realistic and comparable results *2500* nodes were placed on a 50×50 grid.

A.3 Signal Propagation Model

Circular communication model is assumed for signal propagation. Performance analysis in Chapter 4 is based on a fixed transmission range without any fading effects and collisions. Uncertainties in signal strength, fading effect, and collisions are considered in Chapter 5 therefore a more realistic simulation environment was required.

The free space power that receiver's antenna receives can be calculated from the Friis free-space equation [48, 50]. However, the free space mode is an over-idealization because propagation of signal is affected by reflection, diffraction, scattering, and environmental conditions (e.g., indoors, outdoors, rain, etc.) [50]. Based on empirical evidence it is more reasonable to model the receiver power as a log-distance path-loss model [48, 50]:

$$P_R(d) = P_0(d_0) - 10n_p \log(d / d_0) + \mathrm{X}_\sigma \quad \text{(A.1)}$$

where $P_R(d)$ is the receiver power at distance d, $P_0(d_0)$ is the power at a known reference point, d_0 is the distance to the reference point, n_p is the path loss exponent, and X_σ is a zero-mean Gaussian random variable with standard deviation σ. All the power values are in *dBm*.

Integer value of $P_R(d)$ can be considered as the RSSI. The ChipCon CC2420 radio [22] uses the following equation to determine the RSSI value:

$$P_R = RSSI\_VAL + RSSI\_OFFSET \quad \text{(A.2)}$$

where P_R is the receiver power, *RSSI_VAL* is the value of the RSSI register, and *RSSI_OFFSET* is an offset. RSSI values increases linearly with increasing receiver power P_R. However, reliability of RSSI is still debatable due to variation in receiver power. Variability of RSSI is not purely random and tends to increase with the distance [34, 45,

57]. We increase σ with distance to model this behavior. Standard deviation is selected such that:

$$\sigma = \frac{kd}{d_{\max}} \tag{A.3}$$

where k is the variation in RSSI at the receiver sensitivity level, d is the distance, and d_{max} is the transmission range derived from Equation A.1 for a given receiver sensitivity level. Even though RSSI values are noisy, they seem to be time invariant and somewhat stable when antennas are placed above ground.

The Equation A.2 is used to determine the RSSI values. Other relevant parameters are listed in Table A.1. Transmission ranges based on these parameters are listed in Table A.2. Figure A.1 shows the variation in RSSI based on these parameters. It is assumed that a nodes does not accurately hear a broadcast if it is within the collision region of two concurrent broadcasts therefore cannot join a cluster.

Table A.1 – Parameters related to signal propagation model.

Symbol	**Description**	**Value**
n_p	Path loss exponent [30, 28]	*2.2*
k	Variation in RSSI at the receiver sensitivity level [41, 30]	*6dBm*
RSSI_OFFSET	RSSI offset [20]	*-45dBm*
R_T	Receiver sensitivity level [20]	*-90dBm*

Table A.2 – Corresponding transmission ranges for given transmission power levels.

Transmission Power (*dBm*)	**Transmission Range (*m*)**
-10	65.46
-12	53.09
-14	43.07
-16	34.94
-18	28.34
-20	22.96

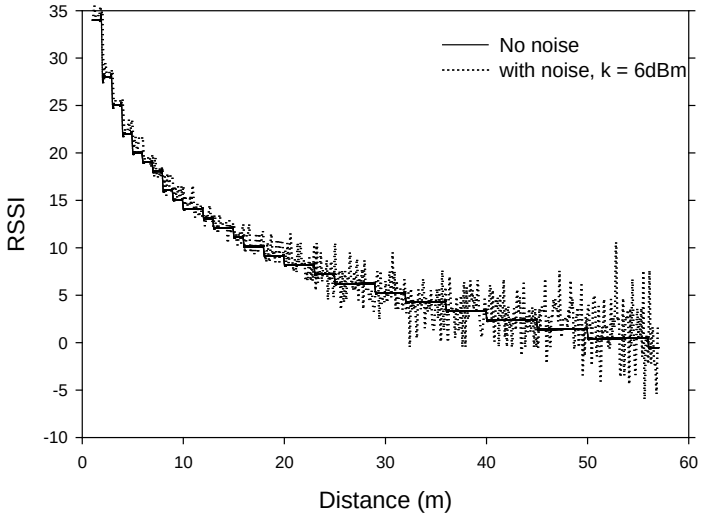


Figure A.1 – Variation in RSSI. P_T = *-12dBm*, k = *6dBm*.

A.4 Energy Model

The energy model is similar to the one used in [5] and [7]. Energy expend by the radio to transmit a *m-bit* message to a node at distance *d* can be written as:

$$E_T = m\left\{E_{circuit} + E_{amp} d^{n_p}\right\} \tag{A.4}$$

where $E_{circuit}$ is the power consumed by the radio (for coding, modulation, filtering, etc.), E_{amp} is the power given to the signal, and n_p is the path loss exponent. Transmission range in our model is fixed therefore $d = R$. Radio also consume power to receive a message. The power expend by the receiver is:

$$E_R = mE_{circuit} \tag{A.5}$$

$E_{circuit}$ tends to dominate the power consumption therefore significantly higher multi-hop communication is not desirable. Both [5] and [7] doubles the path loss exponent n_p, if transmission distance is beyond a certain threshold. Ideally, n_p should gradually increase.

For simplicity, we use the same n_p. Table A.3 show the parameters related to energy consumption. Table A.4 shows the different packets sizes used in the simulation.

Table A.3 – Parameters related to energy model.

Symbol	Description	Value
$E_{circuit}$	Power consumed by the radio [29, 62]	*50.0 nJ/bit*
E_{amp}	power given to the signal [62]	*10 pJ/bit/m^2*
n_p	Path loss exponent [28, 30]	*-2.2*
E	Initial energy of a node	*2.0 J*

Table A.4 – Sizes of different packets.

Description	Size (*bits*)
Data packet	*800*
Cluster formation broadcast	*80*
ACK	*40*
Form new cluster message (during optimization phase)	*120*
Optimize cluster tree message	*40*
Join VSN message	*120*

A.5 Close Loop System

A synthetic data set is generated that simulates the migration pattern of the two plumes (Figure 7.15). Refer [8] for specific details of synthetic data generation. A *200m×350m×15m* sensor field is considered and the plumes migrate from one end of the sensor field to the other, along the longitudinal axis. Plumes migrate over 1095 days (three years). To simplify the analysis, the 3-D data set is converted to a 2-D data set by taking average along the Z-axis. This data set is scaled up to *1000m×1750m* and positioned in the middle of a *2000m×2500m* sensor field. Then 1000 sensor nodes are randomly placed on a *40×50* grid (50% coverage) with a grid spacing of *50m*. Ideal

wireless network is considered, i.e., no packet losses, delayed packets, corrupted packets, collisions, etc.

Nodes read relative chemical concentration values from the time varying synthetic data, based on their location within the sensor field. Initially, each node sends a message to the Plume Modeling and Prediction (PMP) system to indicate its presence in the network. Nodes are configured to sample once a day if they are in a plume and once in every 14 days if they are not in a plume. In each occurrence, five samples are taken and average concentration is reported to the PMP system. Multiple samples are taken to reduce the errors due to noisy reading, out of range reading, and stuck readings that are common to many eco sensors. Plume detection is event drive. If the relative chemical concentration is above 0.005, the sensing node is considered to be in a plume. Then the node sends a message to the PMP system and at the same time joins the VSN. Thereafter, the node sends another message only if the chemical concentration increase/decrease above/below a predefined threshold. If the relative concentration falls below 0.005, the node will unsubscribe from the VSN and switch to a lower sampling rate. We consider Decagon 5TE sensors [24] that are suitable for monitoring subsurface chemical concentrations. Based on these sensors each sampling process takes *150ms* to complete. Node parameters are identical to TELOSB motes [23, 50]. We assume nodes operate at their highest transmission power, i.e., *0dB*, and have a transmission range of *100m*. Nodes take time to walk-up, take multiple samples, communicate, and go back to sleep therefore we assume a cluster member is up for 2 seconds. Clock skew is a major problem in long-lived sensor networks and it can affect the connectivity of the cluster tree. Crystal used on TELOSB and similar motes tends to have a 1.7 seconds clock skew

per day. In our system, some of the CHs only come up in every 14 days hence they are able to synchronize their clock only in every 14 days. To ensure connectivity we assume CHs are up for 25 (1.7×14) seconds. Specific simulation parameters are listed in Table A.5.

Table A.5 – Close loop simulation parameters.

Symbol	Description	
Sensor node related parameters		
P_a	Active power [50]	*3mW*
P_s	Sleep power [50]	*1μW*
P_m	Sampling power [24]	*36mW*
P_d	Transmission power [50]	*45mW*
n_m	Number of samples	*5*
T_m	Time that takes to measure/test a sample [24]	*150ms*
T_c	Time that a child node is active within Δt_i	*2s*
T_{CH}	Time that a cluster head is active within Δt_i	*25s*
B	Bandwidth [23, 50]	*250 Kbps*
Network/VSN related parameters		
n	Number of nodes in the network	*1000*
MAC header	Size of MAC header [38]	*13 bytes*
Network header	Size of network header [38]	*8 bytes*
Network payload	Size of data including hierarchical cluster ID, node ID, VSN ID, and chemical concentration	*22 bytes*
MAC ACK	MAC layer acknowledgement frame size [38]	*9 bytes*
Phenomenon related parameters		
c	Threshold concentration	*0.005*
Δc	Change in concentration that generates a new event	*$\pm$0.05*
Δt	Duration of a time step	*24 hours*
m	Number of time steps before everyone come up again	*14 days*
Prediction rate	Number of days between predictions	*8 weeks*
Prediction window	Number of days predictions are given for	*8 weeks*
α	Changes in concentration level (for energy model)	*0.5*
β	Spatial dynamics of the (for energy model)	*0.1*

Predictions are given in every eight weeks and they are valid for the next eight weeks. A plume prediction model is being developed at Center for Experimental Study of

Subsurface Environmental Processes (CESEP), Colorado School of Mines. Given the size of the plume that we consider this model takes several hours to compute. Therefore, for the time being we derive prefect predictions form the synthetic data. We are already planning to couple the actual plume prediction model to our VSN based close loop system.

A.6 Key Pre-distribution

Two key pre-distribution schemes are used to evaluate the performance of the extended GTC algorithm. The first approach is based on Deployment Knowledge based Random key selection distribution (DKR) [26] while the second approach is based on the Random Block Merging in Combinatorial Design (RBMCD) [16]. Both schemes have a higher local connectivity. Specific parameters of each algorithm are shown in Table A.6. Based on these parameters, RBMCD approach shares 3-4 keys with its neighbors while DKR shares 4-5 keys.

Table A.6 – Parameters of each key pre-distribution scheme.

Parameter	**DKR**	**RBMCD**
Size of key pool	*100000*	*4470*
Number of nodes	*5000*	*5550*
Number of key per node	*120*	*120*
DKR specific parameters		
Number of keys in each group	*1777*	*N/A*
Grid size ($m \times m$)	*100×100*	
Overlaping factor – horizontal/vertical	*0.175*	
Overlaping factor – diagonal	*0.075*	
RBMCD specifc parameters	*N/A*	
Number of key per block		*30*
Number of blocks merged		*4*

A separate program is used to generate key indexes and key IDs. Randomly merged key indexes and key IDs are dumped to a file that enables the use of the same key file against different parameter combinations. From simulation point of view, only key indexes and key IDs are important; therefore, actual keys are not generated. Key indexes/IDs are randomly assigned to nodes before the cluster formation phase. The algorithm given in [42] is used to determine a common key in combinatorial approach. This algorithm requires calculation of modular inverse. Therefore, the algorithm presented in [56] is used to calculate the modular inverse. The inbuilt modulus operator (%) in C does not appropriately calculate modulus of negative numbers; therefore a separate modulus function is implemented based on Euclidean definition [13].

Appendix B

SOURCE CODE

The simulator includes following files:

- types.h - Define data types, simulation parameters, and functions.
- types.c - Implement some of the common functions required by the simulator
- energy.h - Define energy model related parameters and functions
- energy.c - Implement energy model related functions
- simulator.h - Define specific simulation scenario related parameters and cluster and cluster tree formation functions.
- simulator.c - Implement cluster and cluster tree formation related functions. Also include simulation data capture functions.

types.h

```
#define GRIDX             5       //Unit in X direction of grid
#define GRIDY             5       //Unit in Y direction of grid
#define NODESX            201     //No of nodes in X axis
#define NODESY            201     //No of nodes in y axis. Total no of
nodes = NODESX * NODESY
#define MAX_ROUTES        100     // Maximum no of routing entries
#define MAX_VSN_ENTRIES   25      //Maximum number of VSN entries
#define DATA_PACKET_SIZE  800     // Data Packet size in bits
#define CLUSTER_BCAST_SIZE 80     // Cluster formation broadcast size in
bits
#define CLUSTER_ACK_SIZE  40      // ACK size in bits
#define CLUSTER_FORM_SIZE 120      // Cluster formation Size in bits
#define CLUSTER_OPTI_SIZE 40      // Cluster optimization message size
in bits
#define VSN_FORM_SIZE       120   // VSN formation message size in bits
#define NO_COLLISION_NODES 4000   // No of nodes in the collision region
#define MAX_RND_TIME      25      //Maximum random wait time
#define  BIAS_POINT       0       //New time will be (Time – Bias
Point). Should be 0 if no RSSI
#define NODEFILE1       "nodes.txt"
#define NODEFILE2       "nodes_opt.txt"
#define CIRCLEFILE1     "circular.txt"
#define CIRCLEFILE2     "circular_opt.txt"
#define  NODELIST        "input_nodes.txt"
#define  ENERGYFILE      "energy1.txt" //Store energy status after
cluster formation

typedef unsigned char uchar;    //Define uchar
typedef unsigned int uint;      //Define uint

typedef struct {                //Data structure of hierarchical CID
    uint id[4];
} Hie_CID;

typedef struct { //Data structure of routing entry
    uchar valid; //Valid indicate the status of the routing entry.
//Status is indicated by following combination of bits
                // Low in Energy|Learn from neighbor| Route to Parent
//CH |Valid Route
                // bit is 0 is not set. Is 1 if set.
                // Valid = 0 (0000)- Route is not valid
                // Valid = 1 (0001)- Router is valid
                // Valid = 3 (0011)- Route is valid & towards the
//parent CH
                // Valid = 5 (0101)- Router is valid & lean from
//neighbor
                // Valid = 9 (1001)- Router is valid but should be
//avoided whenever possible
                // Valid = 13(1101) -Router is valid but should be
//avoided whenever possible
// Valid = 11 (1011)- Router is valid & to the parent   //CH. But should
be avoided whenever possible
// Valid = 2 (010), 4 (100), 6 (101), 7 (111), 8 //(1000), 10 (1010), 12
(1100), 14 (1110), 15 (1111)- //these status can't exist
    uint NID;          //Neighbors NID
    Hie_CID H_CID;     //Neighbor Hierarchical CID
    uint learn_from;   //Learn from
    uchar hops;        //Number of hops to the destination CH
} router_entry;

typedef struct {       //Data structure for VSN to CH Mapping
    uchar VSN;         //VSN ID
    uint NID;
    uchar node_type ; //Child node (2) or CH (1)
} vsn_entry;

typedef struct {       //Data structure of a node
    uint NID;
    uint CID;
    Hie_CID H_CID;
    Hie_CID Link_H_CID;
    uint CH_NID;
    uint parent_CH_NID; //This will be used only if node is a CH
    uchar tree_depth;
    uchar node_depth;
    uchar link_depth;
    uint no_broadcasts;
    uint no_ACKs;
    uint no_child_nodes;
    uchar no_routing_entries;
    uchar no_vsn_entries;
    uint no_msg_forward;
    uint no_CCHs;
    uint CCHs[200];     //Hold the candidate CHs
    router_entry routing_table[MAX_ROUTES];
    vsn_entry vsn_table[MAX_VSN_ENTRIES];
    float energy;
    uchar node_dead;    //Will be 1 if node is dead
    uint marked_bcast_by_CID;
    uint last_bcast_for_CID;
    uint heard_ACK_for_CID;
    uchar send_ACK_for_CID;
    uchar send_tree_opt_msg;
    uchar in_event;     //Type of event detected
    uchar know_event;   //I know about this event type
    uchar send_routing_info;
} node;

typedef struct {        //Data structure of a data packet
    uint source_NID;
    Hie_CID source_H_CID;
    uint dest_NID;
    Hie_CID dest_H_CID;
} packet;

typedef struct {       //Data structure of neighbor status
```

```
    uchar hops;
    uint nei_NID;
} nei_status;

typedef struct EVENT{   //Data structure of an event
    struct EVENT *next;
    uchar event_type;   //Type of event. 1 - cluster formation, 2 -
                        //timeout, 3 - optimization
    uint time;
    uint NID;
    uint CID;
    Hie_CID H_CID;
    uint CH_NID;
    uchar tree_depth;
    uchar node_depth;
    uchar TTL;
    uint parent_CID;
    uint parent_CH_NID;
    Hie_CID parent_H_CID;
} event;

typedef struct {       //Data structure of a region
    int minx;
    int maxx;
    int miny;
    int maxy;
} region;

Hie_CID generate_CID(Hie_CID parent_ID, uint child_no, uchar
tree_depth);
uchar hop_distance(Hie_CID source_add, Hie_CID dest_add);
region get_node_region(uint x, uint y, float r);
char CID_to_symbol_mapping(int CID);
uint last_event_time(uchar tree_depth, uchar no_CCHs, uint cluster_time,
uint CCH_delay);
char random_wait_time();
void bubble_sort(nei_status *neigh, int n);
```

types.c

```
#include "types.h"
#include <stdio.h>
#include <stdlib.h>

/*-------------------------------------------------------------------*
 * Following function generates the hierarchical cluster id          *
 * based on the parent cluster id & node depth. CIDs are represented *
 * in the following manner                                           *
 *     +-----------------------+----+-----------------+             *
 *     |  ......|child2|child1|parent|  depth (6- bits) |            *
 *     +-----------------------+----+-----------------+             *
 *  depth - 6 bits - can represent up to depth of 38 levels          *
 *  in the tree branching factor (b) - up to 8 branches              *
 *  (3 bits to represent each branch number.                         *
 *  child id = parent id + (branch no << ((depth -1) * b)+ 6) + depth *
 *  with 128-bit CID up to 38 level cluster tree can be represented  *
 *  however there are no 128-bit integer data structures. So an      *
 *  array of four 32-bit integers are used.                          *
 *-------------------------------------------------------------------*/
Hie_CID generate_CID(Hie_CID parent_id, uint child_no, uchar depth)
{
    int tmp_id0, tmp_id1, tmp_id2, tmp_id3, tmp_parent_id[4], tmp_depth;
    Hie_CID new_h_cid;

    tmp_parent_id[0] = parent_id.id[0];
    tmp_parent_id[1] = parent_id.id[1];
    tmp_parent_id[2] = parent_id.id[2];
    tmp_parent_id[3] = parent_id.id[3];

tmp_depth = tmp_parent_id[3] & 63;//extract the depth 64 = 111111 in
                                  // binary
    if ((tmp_depth + 1) != depth)   //if child depth != parent depth + 1
    {
        printf("child depth should be equal to parent depth + 1. Child:
%d Parent: %d\n", depth, tmp_depth);
        exit(0);
    }
    if(depth > 38)
    {
        printf("Hierarchical address overflow.\n");
        exit(1);
    }

    if(depth <= 8)        //up to depth of 8 can be represented
                          //by 32-bits (3 * 8 + 6)
    {
        tmp_id3 = child_no;
        tmp_id3 = tmp_id3 << (((depth -1) * 3) + 6);
        tmp_id3 += tmp_parent_id[3];
        tmp_id3++;           //increment depth by 1
        tmp_id2 = 0;         //the the last (msb) 32-bits zero
        tmp_id1 = 0;         //the the last (msb) 32-bits zero
        tmp_id0 = 0;         //the the last (msb) 32-bits zero
    }
    else if((depth > 8) && (depth < 19))  //If within 9 to 18
    {
        tmp_id3 = tmp_parent_id[3];
        tmp_id3++;                      //Increment depth
        tmp_id2 = child_no;             //Move to the other part of the
                                        //CID and set it
        tmp_id2 = tmp_id2 << ((depth - 9) * 3);//Shift the new no
        tmp_id2 += tmp_parent_id[2];    //Add the MSB9 part of parent
        tmp_id1 = 0;                    //The the last (MSB) 32-bits
                                        //zero
        tmp_id0 = 0;                    //The the last (MSB) 32-bits
                                        //zero
    }
```

```
    else if((depth > 18) && (depth < 29))//If within 19 to 28
    {
        tmp_id3 = tmp_parent_id[3];
        tmp_id3++;                          //increment depth
        tmp_id2 = tmp_parent_id[2];
        tmp_id1 = child_no;                 //move to the other part of the
                                            //cid and set it
        tmp_id1 = tmp_id1 << ((depth - 19) * 3);//shift the new no
        tmp_id1 += tmp_parent_id[1];        //add the MSB part of parent
        tmp_id0 = 0;                        //add the MSB part of parent
    }
    else                                    //If within 29 to 38
    {
        tmp_id3 = tmp_parent_id[3];
        tmp_id3++;                          //increment depth
        tmp_id2 = tmp_parent_id[2];
        tmp_id1 = tmp_parent_id[1];
        tmp_id0 = child_no;                 //move to the other part of the
                                            //cid and set it
        tmp_id0 = tmp_id0 << ((depth - 29) * 3);//shift the new no
        tmp_id0 += tmp_parent_id[0];        //add the MSB part of parent
    }

    new_h_cid.id[0] = tmp_id0;
    new_h_cid.id[1] = tmp_id1;
    new_h_cid.id[2] = tmp_id2;
    new_h_cid.id[3] = tmp_id3;
    return new_h_cid;                       //return new cid
}


/*-------------------------------------------------------------------------*
 * Following function determines the number of hops between two nodes  *
 *-------------------------------------------------------------------------*/
uchar hop_distance(Hie_CID source_add, Hie_CID dest_add)
{
    uchar source_depth, dest_depth, min_depth, tmp_source_add,
tmp_dest_add, i, hops;

    source_depth = source_add.id[3] & 63;       //63 = 111111 in binary
    dest_depth = dest_add.id[3] & 63;

    //If depth > 38 hierarchical address overflow
    if((source_depth > 38) || (dest_depth > 38))
            return 0;

    if(source_depth > dest_depth) //Find minimum depth
        min_depth = dest_depth;
    else
        min_depth = source_depth;
    for(i = 0; i < min_depth ; i++)
    {
        if(i < 8)         //If min-depth is within depth of 8. Check
                //only the LSB (32-bits) of the address
        {
              //Remove depth & hierarchical address of parent
              //CHs starting from root node
            tmp_source_add = (source_add.id[3] >> (6 + i * 3));
            tmp_dest_add = (dest_add.id[3] >> (6 + i * 3));
              // 7 = 111 in binary. Extract only branch number
            tmp_source_add = tmp_source_add & 7;
            tmp_dest_add = tmp_dest_add & 7;
             //If a none matching branch is found
            if(tmp_source_add != tmp_dest_add)
                break;
        }
        else if((i >= 8) && (i < 18)  //Check the next 32-bits
        {
            //Remove the depth info from address
            tmp_source_add = (source_add.id[2] >> ((i - 8) * 3));
            tmp_dest_add = (dest_add.id[2] >> ((i - 8) * 3));
// 7 = 111 in binary. Extract only branch number
            tmp_source_add = tmp_source_add & 7;
            tmp_dest_add = tmp_dest_add & 7;
            //If a none matching branch is found
            if(tmp_source_add != tmp_dest_add)
                break;
        }
        else if((i >= 18) && (i < 28))//Check the next 32-bits
        {
            //Remove the depth info from address
            tmp_source_add = (source_add.id[1] >> ((i - 18) * 3));
            tmp_dest_add = (dest_add.id[1] >> ((i - 18) * 3));
           // 7 = 111 in binary. Extract only branch number
            tmp_source_add = tmp_source_add & 7;
            tmp_dest_add = tmp_dest_add & 7;
          //If a none matching branch is found
            if(tmp_source_add != tmp_dest_add)
                break;
        }
        else
        {
            //Remove the depth info from address
            tmp_source_add = (source_add.id[0] >> ((i - 28) * 3));
            tmp_dest_add = (dest_add.id[0] >> ((i - 28) * 3));
           // 7 = 111 in binary. Extract only branch number
            tmp_source_add = tmp_source_add & 7;
            tmp_dest_add = tmp_dest_add & 7;
           //If a none matching branch is found
            if(tmp_source_add != tmp_dest_add)
                break;
        }
    }

    //Distance to common branching point
    hops = (source_depth - i) + (dest_depth - i);
    return hops;
}
```

```
/*-------------------------------------------------------------------*
 * Following function determines when the next new cluster event     *
 * should start. Starting time is determined based on breadth-first  *
 * tree formation. Each event in the next round should be scheduled  *
 * after the completion of the last event in the current round       *
 * depth         -Depth of the event in the next round               *
 * no_CCHs       -no of candidate CHs                                *
 * cluster_time  -time to form a cluster (timeout)                   *
 * CCH_delay     -Delay for the next child CH for the same parent CH *
 * returns       -time of the last event                             *
*-------------------------------------------------------------------*/
uint last_event_time(uchar depth, uchar no_CCHs, uint cluster_time, uint
CCH_delay)
{
    //dt_CH + (no_CCHs -1)(d -1)t_CCH
    return (depth * cluster_time + (no_CCHs - 1) * (depth -1) *
CCH_delay);

}


/*-------------------------------------------------------------------*
 * Following function return the range of a node                     *
 * given its communication range                                     *
 * x             - x coordinate of node                              *
 * y             - y coordinate of node                              *
 * r             - communication range of node                       *
 * region        - x & y coordinates of the rectangle                *
 *-------------------------------------------------------------------*/
region get_node_region(uint x, uint y, float r)
{
    region tmp_region;
    int minx, maxx, miny, maxy;

    // 1 hop neighbors are in the range of(x - r) to (x + r)
    minx = x - (r/GRIDX) - 1;
    if (minx < 0)          // stay within the grid
        minx = 0;
    maxx = x + (r/GRIDX) + 1;
    if(maxx >= NODESX)
        maxx = NODESX -1;
   // 1 hope neighbors are in the range of(y - r) to (y + r)

    miny = y - (r/GRIDY) - 1;
    if (miny < 0)                 // stay within the grid
        miny = 0;
    maxy = y + (r/GRIDY) + 1;
    if(maxy >= NODESY)
        maxy = NODESY - 1;

    tmp_region.minx = minx;     //set bounds
    tmp_region.maxx = maxx;
    tmp_region.miny = miny;
    tmp_region.maxy = maxy;
    return tmp_region;
}

/*-------------------------------------------------------------------*
 * Following function generates a random time to wait
       *
 * before taking some action      *
 * return        -       +/- random time                             *
*-------------------------------------------------------------------*/
char random_wait_time()
{
    return (rand() % MAX_RND_TIME) - BIAS_POINT; // +/-random(r)
}

/*-------------------------------------------------------------------*
 * Following function maps the Cluster ID to the corresponding symbol*
 * that appears on screen                                            *
*-------------------------------------------------------------------*/
char CID_to_symbol_mapping(int CID)
{
    char symbol;
    switch (CID)
    {
        case 0:
            symbol = 'o' ;
            break;
        case 1:
            symbol = '1' ;
            break;
        case 2:
            symbol = '2' ;
            break;
        case 3:
            symbol = '3' ;
            break;
        case 4:
            symbol = '4' ;
            break;
        case 5:
            symbol = '5' ;
            break;
        case 6:
            symbol = '6' ;
            break;
        case 7:
            symbol = '7' ;
            break;
        case 8:
            symbol = '8' ;
            break;
        case 9:
            symbol = '9' ;
            break;
```

```
case 10:
    symbol = 'A' ;
    break;
case 11:
    symbol = 'B' ;
    break;
case 12:
    symbol = 'C' ;
    break;
case 13:
    symbol = 'D' ;
    break;
case 14:
    symbol = 'E' ;
    break;
case 15:
    symbol = 'F' ;
    break;
case 16:
    symbol = 'G' ;
    break;
case 17:
    symbol = 'H' ;
    break;
case 18:
    symbol = 'I' ;
    break;
case 19:
    symbol = 'J' ;
    break;
case 20:
    symbol = 'K' ;
    break;
case 21:
    symbol = 'L' ;
    break;
case 22:
    symbol = 'M' ;
    break;
case 23:
    symbol = 'N' ;
    break;
case 24:
    symbol = 'O' ;
    break;
case 25:
    symbol = 'P' ;
    break;
case 26:
    symbol = 'Q' ;
    break;
case 27:
    symbol = 'R' ;
    break;
case 28:
    symbol = 'S' ;
    break;
case 29:
    symbol = 'T' ;
    break;
case 30:
    symbol = 'U' ;
    break;
case 31:
    symbol = 'V' ;
    break;
case 32:
    symbol = 'W' ;
    break;
case 33:
    symbol = 'X' ;
    break;
case 34:
    symbol = 'Y' ;
    break;
case 35:
    symbol = 'Z' ;
    break;
case 36:
    symbol = '*' ;
    break;
case 37:
    symbol = '#' ;
    break;
case 38:
    symbol = '$' ;
    break;
case 39:
    symbol = '@' ;
    break;
case 40:
    symbol = '+' ;
    break;
case 41:
    symbol = '-' ;
    break;
case 42:
    symbol = '&' ;
    break;
case 43:
    symbol = '<' ;
    break;
case 44:
    symbol = '!' ;
    break;
case 45:
    symbol = '=' ;
    break;
```

```
case 46:
    symbol = 'a' ;
    break;
case 47:
    symbol = 'b' ;
    break;
case 48:
    symbol = 'c' ;
    break;
case 49:
    symbol = 'd' ;
    break;
case 50:
    symbol = 'e' ;
    break;
case 51:
    symbol = 'f' ;
    break;
case 52:
    symbol = 'g' ;
    break;
case 53:
    symbol = 'h' ;
    break;
case 54:
    symbol = 'i' ;
    break;
case 55:
    symbol = 'j' ;
    break;
case 56:
    symbol = 'k' ;
    break;
case 57:
    symbol = 'l' ;
    break;
case 58:
    symbol = 'm' ;
    break;
case 59:
    symbol = 'n' ;
    break;
case 60:
    symbol = 'p' ;
    break;
case 61:
    symbol = 'q' ;
    break;
case 62:
    symbol = 'r' ;
    break;
case 63:
    symbol = 's' ;
    break;
        case 64:
            symbol = 't' ;
            break;
        case 65:
            symbol = 'u' ;
            break;
        case 66:
            symbol = 'v' ;
            break;
        case 67:
            symbol = 'w' ;
            break;
        case 68:
            symbol = 'x' ;
            break;
        case 69:
            symbol = 'y' ;
            break;
        case 70:
            symbol = 'z' ;
            break;
        default:
            symbol = '?' ;
            break;
    }
    return symbol;
}


/*---------------------------------------------------------------------*
 * Following function sort the neighbor routing entries in the        *
 * increasing order based on hop count.                               *
 * If 2 neighbors have the same hop count they are swapped by tossing *
 * a coin. This prevents the same neighbor being selected again &     *
 * again when they have the same hop count - allow load to be         *
 * distributed                                                        *
 * neigh - pointer to the neighbor routing array                      *
 * n - no of entries in the neighbor array                            *
*---------------------------------------------------------------------*/
void bubble_sort(nei_status *neigh, int n)
{
    char swapped;
    int i;
    nei_status tmp;

    do
    {
        swapped = 0;
        for(i = 0; i < (n-1); i++)
        {
            if(neigh[i].hops > neigh[i+1].hops)
            {
                swapped = 1;
                tmp = neigh[i+1];
```

```
                neigh[i+1] = neigh[i];
                neigh[i] = tmp;
            }
            else if(neigh[i].hops == neigh[i+1].hops)
            {
                //Toss a coin & decide wich entry to use if 0 change if
                //1 keep  previous entry
                if((rand() % 2) == 0 )
                {
                    tmp = neigh[i+1];
                    neigh[i+1] = neigh[i];
                    neigh[i] = tmp;
                }
            }
        }
    }
    while (swapped == 1);   //Repeat until no swapping can be done.
}
```

enrgy.h

```
#define E_CIRCUIT 50.0    //Energy consumed by radio circuit for 1 bit.
//Values are in nano Jule
#define E_AMP_FS 0.01     //Energy for nJ/bit/m^2 - this is for the free
//space model. Values are in nano Jule
#define E_AMP_MP 0.0000013//Energy for nJ/bit/m^4  - this is for the
//multi-path fading model. Values are in nano
//Jule
#define E_NODE   2000000000//Initial energy of a node - in nano Jule
//#define T_RANGE 87.7    //Threshold distance to use the multipath
//fading model
#define  RF_POWER_LP      -20.0    //RF power of the transmeter when at
//Low Power state
#define  RF_POWER_HP      -20.0    //RF power of the transmeter when at
//High Power state
#define MY_N              2.2      //Path loss exponent
#define  REC_SENSITIVITY  -90.0    //Receiver sensitivity in dBm
#define USE_NOISE       0         //If 1 - use of noise
#define MY_K            0.0000989 //K if free space equation
#define RSSI_VAR        0.0       //Maximum variation in RSSI

float energy_to_receive(int no_bits);
float energy_to_transmit(int no_bits, float distance);
char RSSI(float distance, unsigned char lp);
float transmission_range(unsigned char lp);
void gaussian_rnd_init();
void gaussian_rnd_remove();
```

energy.c

```
#include "energy.h"
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

gsl_rng *r;
const gsl_rng_type *T;


/*--------------------------------------------------------------------*
 * Following function calculate the amount of energy required to      *
 * receive a data packet of a geven length                            *
 * no_bits - no of bits                                               *
 *--------------------------------------------------------------------*/
float energy_to_receive(int no_bits)
{
    return (float)(no_bits * E_CIRCUIT);
}


/*--------------------------------------------------------------------*
 * Following function calculate the amount of energy required to      *
 * transmit a data packet of given size and to a given distance       *

 * If the distance is less than R * CH_CH_R_FACT then free space mode *
 * is used. Otherwise multi=path fading model is used                 *
 * no_bits - no of bits to transmit                                   *
 * distance - distance between communicating nodes or maximum         *
 * transmission range                                                 *
 * fs_or_mp - Free space model or multi-path model to use            *
 * if fs_or_mp = 0 free space                                         *
 * if fs_or_mp = 1 multi-path                                         *
 *--------------------------------------------------------------------*/
float energy_to_transmit(int no_bits, float distance)
{
    float circuit_energy, amp_energy;

    circuit_energy = (float)(no_bits * E_CIRCUIT);
    //Enable commented line if different multi-path fading factors are
    //to be used
/*
    if(distance < T_RANGE)        //If below threshold use free space
model
        amp_energy = no_bits * E_AMP_FS * pow(distance, 2);
    else                  //Else use multipath fading model
        amp_energy = no_bits * E_AMP_MP * pow(distance, 4);
*/
    amp_energy = no_bits * E_AMP_FS * pow(distance, MY_N);
    return (amp_energy + circuit_energy);
}


/*--------------------------------------------------------------------*
 * Function calculates Received Signal Strength                       *
```

```
 * Indicator (RSSI) value based on the equation                          *
 * P_r(d) = 10log(kP_t) - 10nlog(d)                                      *
 * where P_r is the receiver power in dBm, D_0 is                        *
 * the close-in distance                                                 *
 * n is the path loss exponent                                           *
 * distance - distance between transmitter & receiver                    *
 * lp - whether to use LP or high power                                  *
*-----------------------------------------------------------------------*/
char RSSI(float distance, unsigned char lp)
{
    double trans_power, rec_power, diff, sigma, noise;

    if(lp == 0)
        trans_power = pow(10, (RF_POWER_LP/10)); //convert power to mW
    else
        trans_power = pow(10, (RF_POWER_HP/10)); //convert power to mW

    if(USE_NOISE == 0)      // No noise is added
        rec_power = 10 * log10(MY_K * trans_power) - 10 * MY_N *
log10(distance);
    else                    //To add noise
    {
        if(lp == 0)
            sigma = (RSSI_VAR * distance)/(2 * transmission_range(0));
        else
            sigma = (RSSI_VAR * distance)/(2 * transmission_range(1));
        noise = gsl_ran_gaussian(r,sigma);
        rec_power = 10 * log10(MY_K * trans_power) - 10 * MY_N *
log10(distance) + noise;
    }
    diff = fabs(REC_SENSITIVITY) - fabs(rec_power);
    if(diff < 0)
        diff = diff - 1;
    return (char)diff;
}


/*-----------------------------------------------------------------------*
 * Following function determine the transmission range                   *
 * lp - If 0 use low power else use High power                           *
 *-----------------------------------------------------------------------*/
float transmission_range(unsigned char lp)
{
    double trans_power, distance;

    if(lp == 0)   //Use Low Power
    {
        trans_power = pow(10, (RF_POWER_LP/10)); //convert power to mW
        distance = pow(10, (10*log10(MY_K * trans_power) -
REC_SENSITIVITY)/(10 * MY_N));
    }
    else                        //Use High Power
    {
        trans_power = pow(10, (RF_POWER_HP/10)); //convert power to mW
        distance = pow(10, (10*log10(MY_K * trans_power) -
REC_SENSITIVITY)/(10 * MY_N));
    }
    return (float)distance;
}


/*-----------------------------------------------------------------------*
 * Initialize the Gaussian random number generator                       *
 *-----------------------------------------------------------------------*/
void gaussian_rnd_init()
{
    unsigned long int seed;

    seed = time(NULL);
    gsl_rng_env_setup();

    T=gsl_rng_default;
    r = gsl_rng_alloc(T);
    gsl_rng_set(r,seed);
}


/*-----------------------------------------------------------------------*
 * Remove the Gaussian random number generator                           *
 *-----------------------------------------------------------------------*/
void gaussian_rnd_remove()
{
    gsl_rng_free (r);
}
```

Simulator.h

```
#include "types.h"

#define STARTX            100   //x value of the starting node
#define STARTY            100   //y value of the starting node
#define MAX_HOPS 1              //Maximum no of hops within cluster
#define MAX_TTL             3   //Max no of hops to propagate the
                                //cluster formation bcast
#define NO_CCHS             6   //No of candidate CHs for level 1
#define CH_CH_R_FACT        3   //Maximum distance between two CHs

#define EVENT_OFFSET_X     10   //used to define event region
#define EVENT_OFFSET_Y     25
#define EVENT_DISTANCE_X   30
#define EVENT_DISTANCE_Y   60
#define NO_EVENT_NODES     50   //Number of nodes reading the same event
#define NO_OF_PACKETS  100000   //Number of packets to send

#define NONODES          5000   //Number of nodes in the network
#define NO_PACKETS          1   //Number of VSN data packets to send
```

```
//Following 2 parameters need to be set depending on whether
//RSSI is used or not
#define DELAY_CCH     200 //Delay to wait before forming a new cluster
#define TIMEOUT       100 //Time to wait before selecting CCHs
#define RANDOM_WAIT   100 //random time to wait before broadcasting
                          //itself as CH
#define SHOW_NODE_DATA  0 //If 1 show node data on terminal
#define USE_NODE_FILE   1 //If 1 use pre-generated node file
#define USE_COLLISIONS  1 //If 1 consider node collisions
#define USE_HP          0 //If 1 use High Power for
                          //CH-to-CH communication

void init(uchar use_file);
void add_event(uchar event_type, int start_time, uint nid, uint cid,
uint ch_nid, Hie_CID h_cid, uchar tree_depth, uchar node_depth, uchar
ttl, uint parent_cid, uint parent_ch_nid, Hie_CID parent_h_cid);
void remove_event(uchar event_type, uint start_time, uint nid);
uchar process_event_list();
void add_nodes_to_cluster(uint start_time, uint nid, uint cid, uint
ch_nid, uchar tree_depth, uchar node_depth, uchar ttl);
void forward_bcast_cluster(uint start_time, uint nid, uint cid, uint
ch_nid, uchar ttl, uchar node_depth);
void send_ACK_as_CCH(uint nid, uint cid, uint ch_nid);
void select_child_CHs(uchar event_time, uchar no_cchs, uint parent_cid,
uint parent_ch_nid, Hie_CID parent_h_cid, uchar tree_depth, uchar
node_depth);
void calculate_circularity(uchar file);
void print_nodes(uchar pnt_console, uchar file);
void print_cluster_energy();
void mark_collision_region(uint nid1, uint nid2);
char is_in_collision(uint nid);
void opti_cluster_tree(uint nid, uint CH_nid, uchar tree_depth, uchar
node_depth, uchar ttl);
void opti_none_cluster_nodes();
void update_child_nodes();
void form_my_own_cluster();
double total_energy();

//Following functions are related to routing
int next_hop(Hie_CID dest_add, int current_nid, int sender_nid);
void send_data();
unsigned char send_data_packet(packet data_packet);
void form_vsn();
uchar send_form_vsn_msg(packet data_packet);
void send_vsn_unicast_data();
void send_vsn_multicast_data();
uchar send_vsn_multicast_packet(packet data_packet);
void inform_neighbors();

void form_second_cluster_tree(uchar tree_depth);
void add_ch_to_tree(uint nid, uint parent_nid, uchar tree_depth, uchar
link_depth, Hie_CID h_cid);

void discover_neighbors_of_link(uchar tree_depth);
void send_link_info(uint nid, uchar tree_depth);
void who_died(uchar d);
```

simulator.c

```
/*-------------------------------------------------------------------*
 * Version - 8.2                                                     *
 * By - Dilum Bandara                                                *
 * e-mail - dilumb@engr.colostate.edu                                *
 * This implementation supports the following functions              *
 * Basic HHC algorithm                                               *
 * Breadth First tree formation                                      *
 * Hierarchical naming                                               *
 * Consider node collisions                                          *
 * Noise and RSSI                                                    *
 * CCHs send ACK only if they don't hear an ACK from                 *
 * a neighboring node                                                *
 * Node energy consumption - send/receive                            *
 * Cluster and Tree optimization phase                               *
 * Circularity calculation                                           *
 * Routing                                                           *
*-------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "simulator.h"
#include "energy.h"

node nodes[NODESX][NODESY];     //Hold the node information
event *root;                    //Root of the CH event list
char msg[100];                  //Hold temporary messages
                                //before dumping data
uint next_cid = 1;              //Next CID
 //List of nodes in the collision region for current time
uint collision_nodes[NO_COLLISION_NODES];
 //Number of nodes in the collision region
uint no_collision_nodes = 0;
 //Last 2 nodes that caused the collision
uint last_collision_set[2] = {0, 0};
uint last_type3_event_time = 0; //Time related to forming my own cluster
 //Hold the list of nodes related to the event
int event_nodes[NO_EVENT_NODES];
uchar optimized = 0;            // Node optimization is not done
 //Transmission range based on power & fading factor
float R;


int main()
{
     //initialize nodes & set relevant parameters.
     //Set the 1st event (from root node)
```

```
    init(USE_NODE_FILE);
     //Process the event list until no events are found
    while(process_event_list());

    //enable/disable following functions based on the properties that
    //needs to be measured
    if(optimized == 0)        //If not optimization phase is already run
    {
        calculate_circularity(1); //Dump node circularity info to file
        print_nodes(SHOW_NODE_DATA, 1);//Dump node information to file
        //print_cluster_energy();      //Dump node energy info to file
        optimized = 1;                 //Set as optimized

        opti_none_cluster_nodes(); //Optimize none cluster nodes
        form_my_own_cluster();     //If can't join a cluster form my own
        //Process the event list until no events are found
        while(process_event_list());
        //Optimze cluster tree
        opti_cluster_tree(nodes[STARTX][STARTY].NID,
nodes[STARTX][STARTY].NID, 0, 0, MAX_TTL);
        //Process the event list until no events are found
        while(process_event_list());
        update_child_nodes();      //Update child nodes depth info

    }

    calculate_circularity(2);     //Dump node circularity info to file
    print_nodes(SHOW_NODE_DATA, 2); //Dump node information to file
    //inform_neighbors();
    //discover_neighbors_of_link(4);
    //send_data();                  //Send data until packet drops
    //form_vsn();
    //send_vsn_unicast_data();
    //send_vsn_multicast_data();
    //print_nodes(SHOW_NODE_DATA, 1); //Dump node information to file
    //print_cluster_energy();       //Dump node energy info to file
    //form_second_cluster_tree(3);  //Form a 2nd cluster tree
    //who_died(4);
    //Free memory allocated for random no generator
    gaussian_rnd_remove();
    exit(0);
}


/*---------------------------------------------------------------------*
 * Following function initialise the nodes & the event list           *
 * Nodes can be either randomly genrated or assigned                  *
 * based on a pregenerated file.                                      *
 * use_file - if 1 use the pregenerated node id file, else            *
 * place nodes randomly                                               *
*---------------------------------------------------------------------*/
void init(uchar use_file)
{
    int count, x, y, i, j, start_NID, NID;
    FILE *nodes_fd;
    char str[10];
    Hie_CID root_H_CID;

    count = 0;                      //Nunber of nodes
    srand(time(NULL));              //Set the seed for rand()
    gaussian_rnd_init();   //Initialize Gaussian random number generator
    R = transmission_range(0);      //Determine communication range
based on transmission power

    //Set all the node parameters. Some of the node location may not be
    //used. So they need to be set to 0
    for (i = 0; i < NODESX ; i++)    //All nodes in X direction
    {
        for(j = 0 ; j < NODESY ; j++)//All nodes in Y direction
        {
            nodes[i][j].CID = 0;     // Initially set all nodes to 0
            nodes[i][j].NID = 0;     // Set NIDs from left to right
            nodes[i][j].no_broadcasts = 0;
            nodes[i][j].no_ACKs = 0;
            nodes[i][j].CH_NID = 0;
            nodes[i][j].tree_depth = 0;
            nodes[i][j].node_depth = 0;
            nodes[i][j].link_depth = 255;
            nodes[i][j].parent_CH_NID = 0;
            nodes[i][j].no_child_nodes = 0;
            nodes[i][j].no_routing_entries = 0;
            nodes[i][j].no_vsn_entries = 0;
            nodes[i][j].no_msg_forward = 0;
            nodes[i][j].node_dead = 0;
            nodes[i][j].energy = 0;
            nodes[i][j].marked_bcast_by_CID = 0;
            nodes[i][j].last_bcast_for_CID = 0;
            nodes[i][j].heard_ACK_for_CID = 0;
            nodes[i][j].send_ACK_for_CID = 0;
            nodes[i][j].send_tree_opt_msg = 255;
            nodes[i][j].no_CCHs = 0;
            nodes[i][j].in_event = 0;
            nodes[i][j].know_event = 0;
            nodes[i][j].send_routing_info = 0;
        }
    }

    //if nodes are to be generated randomly
    if(use_file == 0)
    {
        while (1)                   //Loop forever
        {
            x = rand() % NODESX;    //Select random X
            y = rand() % NODESY;    //Select random Y
            //If node has not already being assigned
            if(nodes[x][y].NID == 0)
            {
         //Set NIDs from left to right. Start numbering from 1
```

```
nodes[x][y].NID = y * NODESX + x + 1;
                //Set the node's initial energy
                nodes[x][y].energy = E_NODE;
                count++;     //Increment no of nodes generated
            }
             //Break if require no of nodes are generated
            if(count == NONODES)
                break;
        }
    }
    else    //if previously generated node file to be used
    {
        nodes_fd = fopen(NODELIST, "r");   //Open file in read-only mode
        if(nodes_fd == NULL)   //If file is not open print error message
        {
            perror("ERROR:");
            exit(1);
        }

        for(i = 0; i <    NONODES; i++)   //Read each node ID from file
        {
            fgets(str, 10, nodes_fd);
            NID = atoi(str);             //Convert to integer
            x = (NID - 1) % NODESX;      //Given NID determine X & Y
            y = (NID - 1) / NODESX;
            // Set NIDs from left to right on the grid. Set nos from 1
            nodes[x][y].NID = NID;
            nodes[x][y].energy = E_NODE; //Set the nodes initial energy

        }

        fclose(nodes_fd);                //Close the file
    }

     //See whether the starting node exist
    if(nodes[STARTX][STARTY].NID == 0)
    {
        printf("Can't continue. Initial node doesn't exist\n");
        exit(1);
    }

    //Add the first event to the event list
    //Determine the NID of the root node
    start_NID = STARTY * NODESX + STARTX + 1;
    root_H_CID.id[0] = 0; //Hierarchical cluster ID of the root node
    root_H_CID.id[1] = 0;
    root_H_CID.id[2] = 0;
    root_H_CID.id[3] = 0;

    //set the 1st cluster formation event. Event type 1.
    //Set start time as 1, CID as 1, no CH_NID since this node is the
    //CH, hierarchical ID is 0 for 1st cluster, depth 0,
    //no parent CH_NID or CID
        add_event(1, 1, start_NID, 1, start_NID, root_H_CID, 0, 0, MAX_TTL,
0, 0, root_H_CID);
        //Add my own routing entry
        nodes[STARTX][STARTY].routing_table[0].valid = 1;
        nodes[STARTX][STARTY].routing_table[0].H_CID = root_H_CID;
        nodes[STARTX][STARTY].routing_table[0].NID = start_NID;
        nodes[STARTX][STARTY].routing_table[0].learn_from = start_NID;
        nodes[STARTX][STARTY].routing_table[0].hops = 0;
        nodes[STARTX][STARTY].no_routing_entries++;
    }


    /*-------------------------------------------------------------------*
     * Following function add events to the Events List                  *
     * Implementation of this function will slightly vary                *
     * depending on the tree                                             *
     * formation approach (breadth-first, depth first, etc.)            *
     * This implementation is for the Breadth-first                      *
     * event_type     - Inter-cluster or Intra-cluster event             *
     * start time     - starting time of the event. Events are sorted    *
     * nid            - node ID of the event related node                *
     * cid            - CID of the event related node                    *
     * ch_nid         - CH NID. This will be 0 if the event              *
     *                  is related to a new                              *
     * cluser formation. Will be > 0 if event is within cluster          *
     * h_cid          - Hierarchical cluster ID of the cluster           *
     * tree_depth     - depth of the node broadcasting the message       *
     *                  based on cluster tree                            *
     * node_depth     - depth of the node broadcasting the message       *
     * ttl            - current TTL value of the cluster                 *
     *                  formation broadcast                              *
     * parent_cid     - CID of the parent CH                             *
     * parent_ch_nid - NID of the parent CH                              *
     * parent_h_cid   - Hierarchical CID of the parent                   *
    *-------------------------------------------------------------------*/
    void add_event(uchar event_type, int start_time, uint nid, uint cid,
    uint ch_nid, Hie_CID h_cid,
            uchar tree_depth, uchar node_depth, uchar ttl, uint parent_cid,
    uint parent_ch_nid, Hie_CID parent_h_cid)
    {
        event *new, *current, *previous;

        if(root == NULL)      //If root of the event list is not defined
        {
            root = (event *)malloc(sizeof(event));
            if(root == NULL)  //If unable to allocate memory
            {
                perror("Error while allocating memory - in add_event
    function\n");
                exit(1);
            }
            root->next = NULL;
        }
        new = (event *)malloc(sizeof(event));  //New event
```

```
    if(new == NULL)
    {
        perror("Error while allocating memory - in add_event
function\n");
        exit(1);
    }

    //Set the parameters for the new event
    new->event_type = event_type;
    new->time = start_time;
    new->NID = nid;
    new->CID = cid;
    new->CH_NID = ch_nid;
    new->H_CID = h_cid;
    new->tree_depth = tree_depth;
    new->node_depth = node_depth;
    new->TTL = ttl;
    new->parent_CID = parent_cid;
    new->parent_CH_NID = parent_ch_nid;
    if(parent_ch_nid != 0)
        new->parent_H_CID = parent_h_cid;

    if(root->next == NULL)    //if event list is blank
    {
        root->next = new;
        new->next = NULL;
    }
    else                      //if event list contains at least one item
    {
        previous = root;
        current = root->next;

        while(1)               //Find the proper location
        {
            if(current->time > start_time)
            {
                new->next = current;    //Set the new event
                previous->next = new;
                break;
            }
            else if(current->next == NULL)
            {
                current->next = new;
                new->next = NULL;
                break;
            }
            else
            {
                previous = current;
                current = current->next;
            }
        }
    }
}
```

```
/*-------------------------------------------------------------------*
 * Following function remove the given event from the event list     *
 * start_time    - remove the event related to the given start time  *
 * NID           - NID of the node related to the event              *
 *-------------------------------------------------------------------*/
void remove_event(uchar event_type, uint start_time, uint nid)
{
    event  *current, *previous;

    if(root->next == NULL) //If event list is already empty
    {
        printf("Event list is already empty\n");
        exit(1);
    }

    //Locate the event with given time & NID
    previous = root;
    current = root->next;

    while (1)
    {
        //If no more events is found break the loop
        if(current->next == NULL)
            break;

        //if exact event is found break the loop
        if((current->event_type == event_type) && (current->time ==
start_time) && (current->NID == nid))
            break;

        previous = current;
        current = current->next;
    }

    if(current->next != NULL) //If event is the last event
    {
        previous->next = current->next;
        free(current);
    }
    else                  //if event is the 1st event or in the middle
    {
        previous->next = NULL;
        free(current);
    }
}


/*-------------------------------------------------------------------*
 * Process the event list starting from
 * the first event, one event at a time                              *
 * If no more events are available, print data about nodes & exit.   *
*-------------------------------------------------------------------*/
```

```
uchar process_event_list()
{
    event *next_event;
    uint x, y, parentx, parenty, parent_nid, next_route_no,
current_time, new_timeout;
    packet data_packet;
    Hie_CID tmp_h_cid;
    uchar result;

    tmp_h_cid.id[0] = 0;     //Initialize hierarchical ID
    tmp_h_cid.id[1] = 0;
    tmp_h_cid.id[2] = 0;
    tmp_h_cid.id[3] = 0;

    //If no more events to handle print data & exit
    if(root->next == NULL)
        return 0;
    else
    {
        //Get event at the top of the event list
        next_event = root->next;
        //Set current time to the time of the event
        current_time = next_event->time;
         //X coordinates of node related to event
        x = (next_event->NID - 1) % NODESX;
        //Y coordinates of node related to event
        y = (next_event->NID - 1) / NODESX;

        //event related to a node broadcasting/forwarding
        //a cluster formation broadcast.
        if(next_event->event_type == 1)  //Intra-cluster event
        {
            //Check for collisions
            if(next_event->next != NULL)   //If there are at least 2
                                           //events
            {
                //if two nodes are broadcasting at
                // the same time & if collisions to be considered
                if((next_event->time == next_event->next->time) &&
(USE_COLLISIONS == 1))
                    mark_collision_region(next_event->NID, next_event-
>next->NID);    //Mark the collision region
                //If the node was not in a collision region
                else if(last_collision_set[1] != next_event->NID)
                    no_collision_nodes = 0;
            }

             //If node is the CH. Has the highest TTL
            if(MAX_TTL == next_event->TTL)
            {
                //if not assigned to another cluster
                if(nodes[x][y].CID == 0)
                {
                   //if not a node trying to form a cluster by it self
                    if(next_event->tree_depth != 254)
                        add_nodes_to_cluster(current_time, next_event-
>NID, next_event->CID, next_event->NID,
                                             next_event->tree_depth, next_event-
>node_depth, next_event->TTL);
                    else
                        add_nodes_to_cluster(current_time, next_event-
>NID, next_event->CID, next_event->NID,
                                             next_event->tree_depth, next_event-
>node_depth,  1);

                    //Mark my last broadcast
                    nodes[x][y].last_bcast_for_CID = next_event->CID;
                    //I send another broadcasts
                    nodes[x][y].no_broadcasts++;
                    nodes[x][y].energy -=
energy_to_transmit(CLUSTER_BCAST_SIZE, R); //Consume energy
                    if((optimized == 0) && (nodes[x][y].no_child_nodes >
0)) //If able to attract child node(s)
                    {
                        nodes[x][y].CID = next_event->CID;
//set the CID of the starting node
                        nodes[x][y].H_CID = next_event->H_CID;
//set the Hierarchical CID of starting node
                        nodes[x][y].CH_NID = next_event->NID;
//set me as my own CH
                        nodes[x][y].tree_depth = next_event->tree_depth;
//Set depth based on logical tree
                        nodes[x][y].node_depth = next_event->node_depth;
//set depth based on physical tree
                        nodes[x][y].parent_CH_NID = next_event-
>parent_CH_NID; //Keep track of parent

                //Add routing entries. Not valid if its root node
                //if not the root node
                        if(next_event->parent_CH_NID != 0)
                        {
                            //Add my own routing entry
                            nodes[x][y].routing_table[0].valid = 1;
                            nodes[x][y].routing_table[0].H_CID =
next_event->H_CID;
                            nodes[x][y].routing_table[0].NID =
next_event->NID;
                            nodes[x][y].routing_table[0].learn_from =
next_event->NID;
                            nodes[x][y].routing_table[0].hops = 0;
                            nodes[x][y].no_routing_entries++;
                            //Add parent routing entry
                            nodes[x][y].routing_table[1].valid = 3;
                            nodes[x][y].routing_table[1].H_CID =
next_event->parent_H_CID;
                            nodes[x][y].routing_table[1].NID =
next_event->parent_CH_NID;
```

```
                            nodes[x][y].routing_table[1].learn_from =
next_event->parent_CH_NID;
                            nodes[x][y].routing_table[1].hops = 1;
                            nodes[x][y].no_routing_entries++;

                            //add my entry to parent routing table
                            parent_nid = nodes[x][y].parent_CH_NID;
                            parentx = (parent_nid - 1) % NODESX;
                            parenty = (parent_nid - 1) / NODESX;
                            next_route_no =
nodes[parentx][parenty].no_routing_entries;

nodes[parentx][parenty].routing_table[next_route_no].valid = 1;

nodes[parentx][parenty].routing_table[next_route_no].H_CID = next_event-
>H_CID;

nodes[parentx][parenty].routing_table[next_route_no].NID = next_event-
>NID;

nodes[parentx][parenty].routing_table[next_route_no].learn_from =
next_event->NID;

nodes[parentx][parenty].routing_table[next_route_no].hops = 1;

nodes[parentx][parenty].no_routing_entries++;
                        }

                        //Add a timeout event for this CH
                        new_timeout = current_time + TIMEOUT;
                        add_event(2, new_timeout, nodes[x][y].NID,
nodes[x][y].CID, nodes[x][y].CH_NID,
                                  nodes[x][y].H_CID,
nodes[x][y].tree_depth, nodes[x][y].node_depth, 0, 0, 0, tmp_h_cid);
                    }
                    else if(optimized == 1)
                //Allow single node clusters in optimization phase
                    {
                        //set the cid of the starting node
                        nodes[x][y].CID = next_event->CID;
                        nodes[x][y].H_CID = next_event->H_CID;
                        //set me as my own CH
                        nodes[x][y].CH_NID = next_event->NID;
                        nodes[x][y].tree_depth = next_event->tree_depth;
                        nodes[x][y].node_depth = next_event->node_depth;
                        nodes[x][y].parent_CH_NID = next_event-
>parent_CH_NID;
                    }
                }
                remove_event(1, current_time, next_event->NID);
        //remove event
            }
            //If bcast message is within MAX_HOPS, add
            //receiving nodes to cluster
                else if((MAX_TTL - next_event->TTL) < MAX_HOPS)
                {

                     //if assigned to same cluster allow to add more nodes
                    if(nodes[x][y].CID == next_event->CID)
                    {
                      //Haven't send the same broadcast earlier
                       if(nodes[x][y].last_bcast_for_CID != next_event->CID)
                       {
                            add_nodes_to_cluster(current_time, next_event-
>NID, next_event->CID,
                                        next_event->CH_NID, next_event-
>tree_depth, next_event->node_depth, next_event->TTL);
                            nodes[x][y].energy -=
energy_to_transmit(CLUSTER_BCAST_SIZE, R);
                            nodes[x][y].last_bcast_for_CID = next_event-
>CID;
                            nodes[x][y].no_broadcasts++;
                        }
                    }
                    remove_event(1, current_time, next_event->NID);
            //remove event
                }
                //if out side cluster
                else if((MAX_TTL - next_event->TTL) < (MAX_TTL))
                {
                    //if assigned to be bcast by for the same cluster
                    if(nodes[x][y].marked_bcast_by_CID == next_event->CID)
                    {
                        if(nodes[x][y].last_bcast_for_CID != next_event-
>CID)
                        {
                            forward_bcast_cluster(current_time, next_event-
>NID, next_event->CID,
                                        next_event->CH_NID, next_event->TTL,
next_event->node_depth);
                            nodes[x][y].last_bcast_for_CID = next_event-
>CID;
                            nodes[x][y].no_broadcasts++;
                            nodes[x][y].energy -=
energy_to_transmit(CLUSTER_BCAST_SIZE, R);
                        }
                    }
                    remove_event(1, current_time, next_event->NID);
            //remove event
                }
                else //I'm suppose to send an ACK as a CCH
                {
                    //if assigned to be bcast by for the same cluster &
                    //not already heard an ACK from a neighbor
                    if(nodes[x][y].marked_bcast_by_CID == next_event->CID)
                        send_ACK_as_CCH(next_event->NID, next_event->CID,
next_event->CH_NID);
```

```
            remove_event(1, current_time, next_event->NID);
        //remove event
            }
        }
        //event related to a CH timeout. Then select CCHs
        else if (next_event->event_type == 2)
        {
            if(nodes[x][y].no_CCHs > 255)
                printf("Error: No of CCHs > 255. Overflow\n");

            select_child_CHs(current_time, nodes[x][y].no_CCHs,
next_event->CID, next_event->CH_NID,
                    next_event->H_CID, next_event->tree_depth,
next_event->node_depth);
           //remove event
            remove_event(2, current_time, next_event->NID);
        }
        //Handle cluster optimization events
        else if (next_event->event_type == 3)
        {
            nodes[x][y].energy -= energy_to_transmit(CLUSTER_OPTI_SIZE,
R);
            opti_cluster_tree(next_event->NID, next_event->CH_NID,
next_event->tree_depth,
                    next_event->node_depth, next_event->TTL);
            //remove event
            remove_event(3, current_time, next_event->NID);
        }
        //Event related to a VSN multicast
        else if (next_event->event_type == 4)
        {
            data_packet.source_NID = next_event->NID;
            data_packet.dest_NID = next_event->CH_NID;
            data_packet.source_H_CID.id[0] = next_event->H_CID.id[0];
            data_packet.source_H_CID.id[1] = next_event->H_CID.id[1];
            data_packet.source_H_CID.id[2] = next_event->H_CID.id[2];
            data_packet.source_H_CID.id[3] = next_event->H_CID.id[3];
            data_packet.dest_H_CID.id[0] = next_event-
>parent_H_CID.id[0];
            data_packet.dest_H_CID.id[1] = next_event-
>parent_H_CID.id[1];
            data_packet.dest_H_CID.id[2] = next_event-
>parent_H_CID.id[2];
            data_packet.dest_H_CID.id[3] = next_event-
>parent_H_CID.id[3];
            result = send_vsn_multicast_packet(data_packet);
            //Go back to caller & say failed.
            //Caller assume 1 as success so don't give 1
            //remove event
            remove_event(4, current_time, next_event->NID);
            if(result != 0)
                return result + 1;
        }
        else if(next_event->event_type == 5)
        {
            add_ch_to_tree(next_event->NID, next_event->parent_CH_NID,
next_event->tree_depth,
                    next_event->node_depth, next_event->H_CID);
            //remove event
            remove_event(5, current_time, next_event->NID);
        }
        else if(next_event->event_type == 6)
        {
            send_link_info(next_event->NID, next_event->tree_depth);
            //remove event
            remove_event(6, current_time, next_event->NID);
        }
    }
    return 1;   //More event(s) in list
}


/*--------------------------------------------------------------------*
 * Following function add nodes to a cluster                          *
 * Mark all the nodes in the communication range of the broadcasting  *
 * node & without a cluster. In 1-hop cluster nid == ch_nid           *
 * but this will be different in multi-hop clusters.                  *
 * start_time    - start time of the current event                    *
 * nid           - NID of the node sending the bcast                  *
 * cid           - CID of the new clusters                            *
 * ch_nid        - NID of the CH                                      *
 * depth         - depth of the node sending the bcast                *
 * ttl           - TTL of the bcast                                   *
*--------------------------------------------------------------------*/
void add_nodes_to_cluster(uint start_time, uint nid, uint cid, uint
ch_nid, uchar tree_depth,
        uchar node_depth, uchar ttl)
{
    int x, y, minx, miny, maxx, maxy, ch_x, ch_y, l, k;
    float distance;//distance between receiving & transmitting node
    char rec_rssi; //Received signal strength
    uchar new_tree_depth, new_node_depth, new_ttl;
    uint new_start_time;
    Hie_CID tmp_h_cid;
    region my_region;

    x = (nid - 1) % NODESX;       //my X, Y coordinates
    y = (nid - 1) / NODESX;
    ch_x = (ch_nid - 1) % NODESX; //X, Y coordinates of CH
    ch_y = (ch_nid - 1) / NODESX;

    my_region = get_node_region(x, y, R);  //determine my neighborhood
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    tmp_h_cid.id[0] = 0;     //Temporary Hierarchical CID
```

```
    tmp_h_cid.id[1] = 0;
    tmp_h_cid.id[2] = 0;
    tmp_h_cid.id[3] = 0;

    //As the message travels depth increases & TTL reduces
    new_tree_depth = tree_depth + 1;//Depth in logical tree
    new_node_depth = node_depth + 1;//Depth in physical tree
    new_ttl = ttl - 1;              //New TTL

    //set CID for only neighboring nodes
    for(l = miny; l <= maxy; l++)
    {
        for (k = minx; k <= maxx; k++)
        {
            //Cartesian distance. Then determine RSSI for given distance
            distance = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l - y)*(l-
y)*(GRIDY * GRIDY));
            rec_rssi = RSSI(distance, 0);

            // if within communication range, if
            //the node exist & not the same node (distance != 0)
            if ((rec_rssi >= 0) && (nodes[k][l].NID != 0) && (distance
!= 0))
            {
                //if not a member of a cluster or not in
                //the collision region assigned to the current cluster
if((nodes[k][l].CID == 0) && (is_in_collision(nodes[k][l].NID) == 0))
                {
                    nodes[k][l].CID = cid;
                    nodes[k][l].CH_NID = ch_nid;
                    nodes[k][l].tree_depth = new_tree_depth;
                    nodes[k][l].node_depth = new_node_depth;
                    nodes[ch_x][ch_y].no_child_nodes++;
//Add as a child node of CH
                    nodes[k][l].no_ACKs++;
//I'm sending an ACK
                    nodes[k][l].energy -=
//energy consumed to receive a message
energy_to_receive(CLUSTER_BCAST_SIZE);
                    nodes[k][l].energy -=
energy_to_transmit(CLUSTER_ACK_SIZE, R); //energy consumed to send ACK
                    nodes[ch_x][ch_y].energy -=
energy_to_receive(CLUSTER_ACK_SIZE); //Energy to receive the ACK

                    //Wait random time before forwarding. Chose
                    //one  of the Following lines if RSSI is used
                    new_start_time = start_time + random_wait_time();
                    //new_start_time = start_time + rec_rssi *
                       MAX_RND_TIME + random_wait_time();
                    if(new_start_time <= start_time)
                        new_start_time = start_time + 1;

            //Add to event list based on RSSI value. Add as type 1 event
                    if((new_ttl > 0) && (nodes[k][l].marked_bcast_by_CID
!= cid))    //if TTL has not expired
                    {
                        add_event(1, new_start_time, nodes[k][l].NID,
cid, ch_nid, tmp_h_cid,
                            new_tree_depth, new_node_depth, new_ttl, 0,
0, tmp_h_cid);
                        nodes[k][l].marked_bcast_by_CID = cid;
                    }
                }
            }
        }
    }
}


/*--------------------------------------------------------------------*
 * Following function forward the cluster formation broadcast         *
 * start_time   - start time of the current event                     *
 * nid          - NID of the node sending the bcast                   *
 * cid          - CID of the new clusters                             *
 * ch_nid       - NID of the CH                                       *
 * ttl          - TTL of the bcast                                    *
*--------------------------------------------------------------------*/
void forward_bcast_cluster(uint start_time, uint nid, uint cid, uint
ch_nid, uchar ttl,
        uchar node_depth)
{
    int x, y, minx, miny, maxx, maxy, ch_x, ch_y, l, k;
    float distance;//distance between receiving & transmitting node
    uchar new_ttl;
    char rec_rssi;       //Received RSSI
    uint new_start_time;
    Hie_CID tmp_h_cid;
    region my_region;

    ch_x = (ch_nid - 1) % NODESX; //X, Y coordinates of CH
    ch_y = (ch_nid - 1) / NODESX;
    x = (nid - 1) % NODESX;       //My X, Y coordinates
    y = (nid - 1) / NODESX;

    my_region = get_node_region(x, y, R);  //Get my region
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    tmp_h_cid.id[0] = 0;
    tmp_h_cid.id[1] = 0;
    tmp_h_cid.id[2] = 0;
    tmp_h_cid.id[3] = 0;

    new_ttl = ttl - 1; //As the message forwards, TTL reduces
```

```
    //set CID for only neighboring nodes
    for(l = miny; l <= maxy; l++)
    {
        for (k = minx; k <= maxx; k++)
        {
            //Distance & corresponding RSSI value
            distance = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l - y)*(l-
y)*(GRIDY * GRIDY));
           //determine the RSSI for the signal for the given distance
            rec_rssi = RSSI(distance, 0);

            // if within communication range & node
            // exist & not the same node (distance != 0)
            if ((rec_rssi >= 0) && (nodes[k][l].NID != 0) && (distance
!= 0))
            {
                //if not a member of a cluster & not used to send
                // the same bcast use it to bcast the message
                if((nodes[k][l].CID == 0) &&
(nodes[k][l].marked_bcast_by_CID != cid)
                        && (is_in_collision(nodes[k][l].NID) == 0))
                {
                    nodes[k][l].energy -=
energy_to_receive(CLUSTER_BCAST_SIZE); //energy to receive a message

                    //Select one of the following lines depending
                    // on RSSI based forwarding or not
                    new_start_time = start_time + random_wait_time();
/*
                    if(new_ttl > 0)     //If not at the edge
                        new_start_time = start_time + rec_rssi *
MAX_RND_TIME + random_wait_time();
                    else
                        new_start_time = start_time + (90 - rec_rssi) *
MAX_RND_TIME + random_wait_time();
*/
                    if(new_start_time <= start_time)
                        new_start_time = start_time + 1;

                    add_event(1, new_start_time, nodes[k][l].NID, cid,
ch_nid, tmp_h_cid, 0,
                            (node_depth + 1), new_ttl, 0, 0, tmp_h_cid);
                    nodes[k][l].marked_bcast_by_CID = cid;
                }
            }
        }
    }
}


/*--------------------------------------------------------------------*
 * Send an acknowledgement to the CH indicating that                  *
 * node is a candidate to be a a new CH. Consider all the nodes in    *
 * the communication range of broadcasting node & without a cluster.  *
 * Nodes that has already heard a neighbor ACK will                   *
 * not respond to the CH                                              *
 * start_time  - start time of the current event                      *
 * nid          - NID of the node sending the bcast                   *
 * cid          - CID of the new clusters                             *
 * ch_nid       - NID of the CH                                       *
 * ttl          - TTL of the bcast                                    *
*--------------------------------------------------------------------*/
void send_ACK_as_CCH(uint nid, uint cid, uint ch_nid)
{
    int x, y, minx, miny, maxx, maxy, ch_x, ch_y, l, k, i;
    float distance;
    uint tmp_no_CCHs;
    region my_region;
    char rec_rssi;

    x = (nid - 1) % NODESX;        //X, Y coordinates of me
    y = (nid - 1) / NODESX;
    ch_x = (ch_nid - 1) % NODESX; //X, Y coordinates of CH
    ch_y = (ch_nid - 1) / NODESX;

    //Make sure that node has not already send a ACK
    //for the same cluster or heard an ACK from a neighbor
    if((nodes[x][y].send_ACK_for_CID != cid) &&
(nodes[x][y].heard_ACK_for_CID != cid))
    {
    //Makesure I have not already ACK
        for(i = 0; i < nodes[ch_x][ch_y].no_CCHs; i++)
        {
            if(nid == nodes[ch_x][ch_y].CCHs[i])
                break;
        }
    //If no match add me as a CCH
        if(i ==  nodes[ch_x][ch_y].no_CCHs)
        {
            tmp_no_CCHs = nodes[ch_x][ch_y].no_CCHs;
            nodes[ch_x][ch_y].CCHs[tmp_no_CCHs] = nid;
            if(nodes[ch_x][ch_y].no_CCHs > 200)
            {
                printf("No of candidate CHs overflow.
Terminating.....\n");
                exit(1);
            }
            nodes[ch_x][ch_y].no_CCHs = tmp_no_CCHs + 1;
            nodes[x][y].send_ACK_for_CID = cid;
            nodes[x][y].heard_ACK_for_CID = cid;
            //ACK forwarded 3- hops
            nodes[x][y].no_ACKs = nodes[x][y].no_ACKs + 3;
            //Energy to send & receive ACK
            nodes[x][y].energy -= energy_to_transmit(CLUSTER_ACK_SIZE,
R);
            nodes[ch_x][ch_y].energy -=
energy_to_receive(CLUSTER_ACK_SIZE);
        }
```

```
        else
            return;       //if so discard
    }
    else                  //if so discard
        return;

    my_region = get_node_region(x, y, R);  //get my region
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    //Mark the neighbors indicating that they heard my ACK to the CH
    //set CID for only neighboring nodes
    for(l = miny; l <= maxy; l++)
    {
        for (k = minx; k <= maxx; k++)
        {
            distance = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l - y)*(l-
y)*(GRIDY * GRIDY));
            rec_rssi = RSSI(distance, 0);

            // if within communication range & if the node
            //exist & not the same node (distance != 0)
            if ((rec_rssi >= 0) && (nodes[k][l].NID != 0) && (distance
!= 0))
            {
                //if not a member of a cluster & marked
                //to forward the bcast
                if((nodes[k][l].CID == 0) &&
(nodes[k][l].marked_bcast_by_CID == cid)
                        && (is_in_collision(nodes[k][l].NID) == 0))
                {
                    nodes[k][l].heard_ACK_for_CID = cid;
                    nodes[k][l].energy -=
                 //Cost of neighbors receiving the ACK
energy_to_receive(CLUSTER_ACK_SIZE);
                }
            }
        }
    }
}


/*-------------------------------------------------------------------------*
 * Following function selects new CHs from list of available CCHs     *
 * no_cchs       - no of CCHs                                          *
 * depth          - depth of the parent CH                             *
 * parent_cid    - CID of the parent CH                                *
 * parent_h_cid  - Hierarchical CID of the parent CH                   *
 * parent_ch_nid  - NID of the parent CH                               *
*-------------------------------------------------------------------------*/
void select_child_CHs(uchar event_time, uchar no_cchs, uint parent_cid,
uint parent_ch_nid,
        Hie_CID parent_h_cid, uchar tree_depth, uchar node_depth)
{
    uint no_new_chs, next_ch, j, l, new_time, x, y, cch_x, cch_y;
    uint selected_ch_list[10];
    uint no_selected_chs = 0;
    Hie_CID new_h_cid;

   //if no CCHs are there to be elected, just return back to caller
    if(no_cchs == 0)
        return;

    x = (parent_ch_nid - 1) % NODESX;  //my X, Y coordinates
    y = (parent_ch_nid - 1) / NODESX;

    if(tree_depth == 0)        //If depth 0 use maximum branching factor
        no_new_chs= NO_CCHS;
    else                       //else use only half of it
        no_new_chs= NO_CCHS/2;

    if(no_new_chs > no_cchs)//if no of possible CCHs are less than
                            //what needs to be created
        no_new_chs = no_cchs;//generate the maximum possible no of CCHs

    //select given no of candidate neighbors as CHs
    for(j = 0; j < no_new_chs; j++)
    {
        while(1)
        {
            //randomly select one of the nodes to be the CH
            next_ch = rand() % no_cchs;
            //check whether it has been already selected
            for(l = 0 ; l < no_selected_chs ; l++)
            {
                if(selected_ch_list[l] == nodes[x][y].CCHs[next_ch])
                    break;
            }

            //if already selected diacard & select another
            if(l != no_selected_chs)
                continue;

            //else select it as a new CH
            selected_ch_list[no_selected_chs] =
nodes[x][y].CCHs[next_ch];
            no_selected_chs++;  //Increment no of new CHs
            next_cid++;         //generate the next cid
            new_h_cid = generate_CID(parent_h_cid, j, (tree_depth + 1));

            //set the timing such that tree is formed using the breadth-
first tree formation. This needs to be changed if depth-first tree
formation is used
            if (j == 0)
                new_time = last_event_time((tree_depth + 1), NO_CCHS,
TIMEOUT, DELAY_CCH) + 1 ;
```

```
            else
                new_time = last_event_time((tree_depth + 1), NO_CCHS,
TIMEOUT, DELAY_CCH) + j * DELAY_CCH ;
            if(new_time <= event_time) //Make sure new time > old time
                new_time = event_time + 1;

            //add as a new CH event
            add_event(1, new_time, nodes[x][y].CCHs[next_ch], next_cid,
0, new_h_cid,
                    (tree_depth + 1), (node_depth + MAX_TTL), MAX_TTL,
parent_cid, parent_ch_nid, parent_h_cid);
            nodes[x][y].no_broadcasts = nodes[x][y].no_broadcasts +3
            //Bcast is send 3-hops
            //Cost of sending the form_cluster function
            nodes[x][y].energy -= energy_to_transmit(CLUSTER_FORM_SIZE,
R);
            //Cost of receiving the form_cluster function
            cch_x = (nodes[x][y].CCHs[next_ch] - 1) % NODESX;
            cch_y = (nodes[x][y].CCHs[next_ch] - 1) / NODESX;
            nodes[cch_x][cch_y].energy -=
energy_to_receive(CLUSTER_FORM_SIZE);

            break;
        }
    }
}


/*--------------------------------------------------------------------*
 * Following function optimize the cluster tree                       *
 * by broadcasting cluster                                            *
 * optimization message                                               *
 * nid           - NID of the node                                    *
 * CH_nid        - NID of the CH sending the bcast                    *
 * tree_depth    - depth of the CH broadcasting the message           *
 * node_depth    - depth of the node broadcasting or forwarding       *
 *                 the bcast                                          *
 * ttl           - TTL of the message                                 *
*--------------------------------------------------------------------*/
void opti_cluster_tree(uint nid, uint CH_nid, uchar tree_depth, uchar
node_depth, uchar ttl)
{
    int x, y, minx, miny, maxx, maxy, l, k;
    float distance, r;
    Hie_CID tmp_H_CID;
    region my_region;
    char rec_rssi;

    x = (nid - 1) % NODESX;
    y = (nid - 1) / NODESX;

    if(USE_HP == 0) //If cluster optimization phase is high power
        r = transmission_range(0);
    else
        r = transmission_range(1);

    my_region = get_node_region(x, y, r);  //get my region
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    tmp_H_CID.id[0] = 0;    //Hierarchical CID of the root node
    tmp_H_CID.id[1] = 0;
    tmp_H_CID.id[2] = 0;
    tmp_H_CID.id[3] = 0;

    if((ttl -1 ) < 0)       //if message expired
        return;

    for(l = miny; l <= maxy; l++)
    {
        for (k = minx; k <= maxx; k++)
        {
            distance = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l - y)*(l-
y)*(GRIDY * GRIDY));
            if(USE_HP == 0)
                rec_rssi = RSSI(distance, 0);
            else
                rec_rssi = RSSI(distance, 1);

            //if in region, node exist and not the same node
            if((rec_rssi >= 0) && (nodes[k][l].NID != 0) && (distance !=
0))
            {
                if(nodes[k][l].NID != nodes[k][l].CH_NID)  //if not a CH
                {
                    if(((ttl - 1) > 0) && (nodes[k][l].send_tree_opt_msg
> (node_depth + 1)))
                    {
                        last_type3_event_time++;
                        add_event(3, last_type3_event_time,
nodes[k][l].NID, 0, CH_nid, tmp_H_CID,
                                tree_depth, (node_depth + 1), (ttl - 1),
0, 0, tmp_H_CID);
                       //Broadcast cluster changes to child nodes
                        nodes[k][l].no_broadcasts++;
                        nodes[k][l].energy -=
energy_to_receive(CLUSTER_OPTI_SIZE); //energy to receive message
                        nodes[k][l].send_tree_opt_msg = node_depth + 1;
                    }
                }
                else
                {
                    //if a CH and current depth is higher
                    if(nodes[k][l].node_depth > (node_depth + 1))
                    {
                        nodes[k][l].tree_depth = tree_depth + 1;
```

```
                    //Set new depth & parent CH
                        nodes[k][l].node_depth = node_depth + 1;
                        nodes[k][l].parent_CH_NID = CH_nid;
                        nodes[k][l].no_ACKs = nodes[k][l].no_ACKs +
(MAX_TTL - ttl);
                        nodes[k][l].energy -=
energy_to_receive(CLUSTER_OPTI_SIZE);  //energy to receive message
                        nodes[x][y].energy -=
energy_to_transmit(CLUSTER_ACK_SIZE, r);  //energy to send ACK
                    }
                    if(nodes[k][l].send_tree_opt_msg >
(nodes[k][l].node_depth)) //If new message is better
                    {
                        last_type3_event_time++;
                        if(USE_HP == 0)
                            add_event(3, last_type3_event_time,
nodes[k][l].NID, 0, nodes[k][l].NID, tmp_H_CID,
                                nodes[k][l].tree_depth,
nodes[k][l].node_depth, MAX_TTL, 0, 0, tmp_H_CID);
                        else
                            add_event(3, last_type3_event_time,
nodes[k][l].NID, 0, nodes[k][l].NID, tmp_H_CID,
                                nodes[k][l].tree_depth,
nodes[k][l].node_depth, MAX_HOPS, 0, 0, tmp_H_CID);

          //Broadcast cluster changes to child nodes
                     nodes[k][l].no_broadcasts++;
nodes[k][l].send_tree_opt_msg = nodes[k][l].node_depth;
                    }
                }
            }
        }
    }
}


/*--------------------------------------------------------------------*
 * Allow nodes that are not in a cluster to join a neighboring cluster*
 *--------------------------------------------------------------------*/
void opti_none_cluster_nodes()
{
    int i, j, k, l, minx, maxx, miny, maxy;
    region my_region;
    float distance;
    uchar my_exit = 0;
    char rec_rssi;

    for (j = 0; j < NODESY ; j++) //check for all the nodes
    {
        for(i = 0 ; i < NODESX ; i++)
        {
            //if the node exist but not in a cluster
            if((nodes[i][j].NID != 0) && (nodes[i][j].CID == 0))
            {
                    my_exit = 0;

                    my_region = get_node_region(i, j, R);//get my region
                    minx = my_region.minx;
                    miny = my_region.miny;
                    maxx = my_region.maxx;
                    maxy = my_region.maxy;

                    for(l = miny; l <= maxy; l++) //in my region
                    {
                        for (k = minx; k <= maxx; k++)
                        {
                            distance = sqrt((k - i)*(k - i)*(GRIDX * GRIDX)
+ (l - j)*(l - j)*(GRIDY * GRIDY));
                            rec_rssi = RSSI(distance, 0);

                      // if within communication range & if the node exist &
                      //a CH, join that cluster
                            if((rec_rssi >= 0) && (nodes[k][l].NID != 0) &&
(nodes[k][l].NID == nodes[k][l].CH_NID))
                            {
                                nodes[i][j].CID = nodes[k][l].CID;
                                nodes[i][j].CH_NID = nodes[k][l].CH_NID;
                                nodes[i][j].tree_depth =
nodes[k][l].tree_depth + 1;
                                nodes[i][j].node_depth =
nodes[k][l].node_depth + 1;
                                nodes[i][j].no_ACKs++;
                                nodes[k][l].no_child_nodes++;
                                nodes[i][j].energy -=
energy_to_transmit(CLUSTER_ACK_SIZE, R);  //energy to send ACK
                                nodes[k][l].energy -=
energy_to_receive(CLUSTER_ACK_SIZE);  //energy to receive ACK
                                my_exit = 1;//Exit both loops
                                break;
                            }
                        }
                        if(my_exit == 1)   //exit 1st outer loop
                            break;
                    }
                }
            }
        }
    }
}


/*--------------------------------------------------------------------*
 * Following function update the child nodes if parent nodes          *
 * changes its location in the cluster tree                           *
*--------------------------------------------------------------------*/
void update_child_nodes()
{
    int k, l, CH_x, CH_y;
    uchar tree_depth_CH, node_depth_CH;
```

```
    for(l = 0; l < NODESY; l++) //check for all nodes
    {
        for (k = 0; k < NODESX; k++)
        {
            //If I'm in a cluster and if I'm not a CH
            if((nodes[k][l].CID != 0) && (nodes[k][l].NID !=
nodes[k][l].CH_NID))
            {
                CH_x = (nodes[k][l].CH_NID - 1) % NODESX;
                CH_y = (nodes[k][l].CH_NID - 1) / NODESX;
                tree_depth_CH = nodes[CH_x][CH_y].tree_depth;
                node_depth_CH = nodes[CH_x][CH_y].node_depth;

                //My depth & my CH depth don't agree
                if(nodes[k][l].tree_depth != (tree_depth_CH + 1))
                {
                    nodes[k][l].tree_depth = tree_depth_CH  + 1;
                    nodes[k][l].node_depth = node_depth_CH  + 1;
                }
                if(nodes[k][l].node_depth != (node_depth_CH + 1))
                {
                    nodes[k][l].tree_depth = tree_depth_CH  + 1;
                    nodes[k][l].node_depth = node_depth_CH  + 1;
                }
            }
        }
    }
}


/*-------------------------------------------------------------------------*
* Form my own cluster if unable to join a neighboring cluster         *
*-------------------------------------------------------------------------*/
void form_my_own_cluster()
{
    int k, l, new_time;
    Hie_CID tmp_h_cid;

    tmp_h_cid.id[0] = 0;
    tmp_h_cid.id[1] = 0;
    tmp_h_cid.id[2] = 0;
    tmp_h_cid.id[3] = 0;

    for(l = 0; l < NODESY; l++)
    {
        for (k = 0; k < NODESX; k++)
        {
            //if node exist & not in a cluster
            if((nodes[k][l].NID != 0) && (nodes[k][l].CID == 0))
            {
                new_time = rand() % RANDOM_WAIT;
                add_event(1, new_time, nodes[k][l].NID, next_cid, 0,
tmp_h_cid, 254, 254,
                          MAX_TTL, 0, 0, tmp_h_cid);
                next_cid++;
            }
        }
    }
}


/*-------------------------------------------------------------------------*
 * Following functions randomly generate a socue and a destination     *
 * node & then sends a message. It count the no of messages            *
 * successfully delivered & terminates when the 1st message get        *
 * dropped. It indicates the reason why the packet get dropped.        *
*-------------------------------------------------------------------------*/
void send_data()
{
    int s_x, s_y, d_x, d_y, source_CH_x, source_CH_y, dest_CH_x,
dest_CH_y, CH_NID, i;
    uchar result;
    uint no_msg_delivered = 0;   //No of successfully delivered messages
    uint no_msg_dropped = 0;     //No of messages dropped
    uint no_route = 0;           //No of routes not found
    packet data_packet;          //Data packet to be transmitted
    double energy_before;

    energy_before = total_energy();
    //for(i = 0 ; i < NO_OF_PACKETS; i++)
    while (1)    //Loop until packet get dropped
    {
        while (1)//Generate source node
        {
            s_x = rand() % NODESX;
            s_y = rand() % NODESY;
            //Make sure node is available & in a cluster
            if((nodes[s_x][s_y].NID == 0) || (nodes[s_x][s_y].CID == 0))

                continue;
                 //Source node is dead find another
            else if(nodes[s_x][s_y].node_dead == 1)
                continue;
            else //Form the source info of the data packet
            {
                data_packet.source_NID = nodes[s_x][s_y].NID;
                CH_NID = nodes[s_x][s_y].CH_NID;
                source_CH_x = (CH_NID - 1) % NODESX;
                source_CH_y = (CH_NID - 1) / NODESX;
                data_packet.source_H_CID =
nodes[source_CH_x][source_CH_y].H_CID;
                break;
            }
        }
        while (1)//Generate destination node
        {
            d_x = rand() % NODESX;
```

```
            d_y = rand() % NODESY;
            //Make sure node is available & a member of a cluster
            if((nodes[d_x][d_y].NID == 0) || (nodes[d_x][d_y].CID == 0))
                continue;
            //if both source & destination is equal find another
            else if((s_x == d_x) && (s_y == d_y))
                continue;
            else     //Form the destination info of the data packet
            {
                data_packet.dest_NID = nodes[d_x][d_y].NID;
                CH_NID = nodes[d_x][d_y].CH_NID;
                dest_CH_x = (CH_NID - 1) % NODESX;
                dest_CH_y = (CH_NID - 1) / NODESX;
                data_packet.dest_H_CID =
nodes[dest_CH_x][dest_CH_y].H_CID;
                break;
            }
        }

        //Send data from random source to a random destination
        result = send_data_packet(data_packet);
        if(result == 1)//If the message is dropped due to low energy
        {
            no_msg_dropped++;
            break;
        }
        else if (result == 2) //If message drop due to the wrong route
        {
            no_route++;
            break;
        }
        else
            no_msg_delivered++;
     }
    //printf("%d\n", no_msg_delivered);
    //printf("%d\t%d\t%d\n", no_msg_delivered, no_msg_dropped,
    //no_route);
    printf("%d\t%f\t%f\t", no_msg_delivered, energy_before,
total_energy());
}


/*-------------------------------------------------------------------------*
 * Following function try to deliver a message between                     *
 * a given source & a destination. Function returns:                       *
 * 0 - On sucess                                                           *
 * 1 - when not enough energy to deliver the packet                        *
 * 2 - When route to destination is not found                              *
 * data_packet - header of the data packet to be delivered                 *
*-------------------------------------------------------------------------*/
unsigned char send_data_packet(packet data_packet)
{
    int s_x, s_y, d_x, d_y, source_CH_x, source_CH_y, dest_CH_x,
dest_CH_y, receiver_x,
            receiver_y, current_x, current_y;
    int neighbor_to_forward, current_NID;
    int msg_send_by;      //Node that send the message

    receiver_x = -1;
    receiver_y = -1;
    msg_send_by = 0;

    s_x = (data_packet.source_NID - 1) % NODESX;
    s_y = (data_packet.source_NID - 1) / NODESX;
    d_x = (data_packet.dest_NID - 1) % NODESX;
    d_y = (data_packet.dest_NID - 1) / NODESX;
    source_CH_x = (nodes[s_x][s_y].CH_NID - 1) % NODESX;
    source_CH_y = (nodes[s_x][s_y].CH_NID - 1) / NODESX;
    dest_CH_x = (nodes[d_x][d_y].CH_NID - 1) % NODESX;
    dest_CH_y = (nodes[d_x][d_y].CH_NID - 1) / NODESX;

    //If souce is not a CH. Then forward the message to the CH
    if(data_packet.source_NID != nodes[s_x][s_y].CH_NID)
    {
        nodes[s_x][s_y].energy -= energy_to_transmit(DATA_PACKET_SIZE ,
R);
        if(nodes[s_x][s_y].energy < 0)     //If not enough energy
        {
            nodes[s_x][s_y].node_dead = 1; //Mark node as dead
            return 1;
        }
        //If the rceiving CH is dead, drop message
        if(nodes[source_CH_x][source_CH_y].node_dead == 1)
            return 1;

        nodes[source_CH_x][source_CH_y].energy -=
energy_to_receive(DATA_PACKET_SIZE);
        //If not enough energy to receive
        if(nodes[source_CH_x][source_CH_y].energy < 0)
        {
            //Mark node as dead
            nodes[source_CH_x][source_CH_y].node_dead = 1;
            return 1;
        }
        msg_send_by = nodes[s_x][s_y].NID;
        //Another message is forwarded
        nodes[s_x][s_y].no_msg_forward++;
    }

    //if source and destination has the same CH
    if(nodes[s_x][s_y].CH_NID == nodes[d_x][d_y].CH_NID)
    {
        nodes[source_CH_x][source_CH_y].energy -=
energy_to_transmit(DATA_PACKET_SIZE , R);
        //If not enough energy
        if(nodes[source_CH_x][source_CH_y].energy < 0)
        {
            //Mark node as dead
```

```
            nodes[source_CH_x][source_CH_y].node_dead = 1;
            return 1;
        }
        //Another message
        nodes[source_CH_x][source_CH_y].no_msg_forward++;

        if(nodes[d_x][d_y].node_dead == 1)
            return 1;
        nodes[d_x][d_y].energy -= energy_to_receive(DATA_PACKET_SIZE);
        if(nodes[d_x][d_y].energy < 0)     //If not enough energy
        {
            nodes[d_x][d_y].node_dead = 1; //Mark node as dead
            return 1;
        }
        return 0;
    }

    current_NID = nodes[s_x][s_y].CH_NID;// Start forwarding from the CH

    //Loop until destination CH is found
    while(1)
    {
        //Find next hop
        neighbor_to_forward = next_hop(data_packet.dest_H_CID,
current_NID, msg_send_by);
        if (neighbor_to_forward == 0)     //No route to destination
            return 2;
        else if(neighbor_to_forward == current_NID)//Same as destination
            break;
        else
        {
            current_x = (current_NID - 1) % NODESX;
            current_y = (current_NID - 1) / NODESX;

            //If is dead can't forward messages
            if(nodes[current_x][current_y].node_dead == 1)
                return 1;

            //Reduce energy to transmit. CH to CH messages are
            //high power with within R*TTL_max
            nodes[current_x][current_y].energy -=
energy_to_transmit(DATA_PACKET_SIZE, (CH_CH_R_FACT * R));
            //If no energy to tranmit
            if(nodes[current_x][current_y].energy < 0)
            {
                //Mark node as dead
                nodes[current_x][current_y].node_dead = 1;
                return 1;              //Not enough energy
            }

            receiver_x = (neighbor_to_forward - 1) % NODESX;
            receiver_y = (neighbor_to_forward - 1) / NODESX;
            //Node is dead can't receive messages
            if(nodes[receiver_x][receiver_y].node_dead == 1)
                return 1;

            nodes[receiver_x][receiver_y].energy -=
energy_to_receive(DATA_PACKET_SIZE);
            //Not enough energy
            if(nodes[receiver_x][receiver_y].energy < 0)
            {
                //Mark node as dead
                nodes[receiver_x][receiver_y].node_dead = 1;
                return 1;
            }
            msg_send_by = nodes[current_x][current_y].NID;
            //Another message is forwarded
            nodes[current_x][current_y].no_msg_forward++;
            //Forward packet to neighbor. Neighbor becomes current node
            current_NID = neighbor_to_forward;
        }
    }
    if(current_NID != data_packet.dest_NID)
    {
        //current node x, y values are cacluated in the previous loop
        //Node is dead can't forward message
        if(nodes[receiver_x][receiver_y].node_dead == 1)
            return 1;

        nodes[receiver_x][receiver_y].energy -=
energy_to_transmit(DATA_PACKET_SIZE, R);
        if(nodes[receiver_x][receiver_y].energy < 0)//Not enough energy
        {
            //Mark node as dead
            nodes[receiver_x][receiver_y].node_dead = 1;
            return 1;
        }

        nodes[d_x][d_y].energy -= energy_to_receive(DATA_PACKET_SIZE);
        if(nodes[dest_CH_x][dest_CH_y].energy < 0)  //Not enough energy
        {
            nodes[d_x][d_y].node_dead = 1;   //Mark node as dead
            return 1;
        }
        //Another message is forwarded
        nodes[receiver_x][receiver_y].no_msg_forward++;
    }
    return 0;    //Packet is sucessfully delivered
}


/*---------------------------------------------------------------------*
 * Following function forms a VSN                                      *
 * Nodes in a given region form a VSN by send a message                *
*---------------------------------------------------------------------*/
void form_vsn()
{
    int s_x, s_y, source_CH_x, source_CH_y, CH_NID, i, j;
```

```
    uchar ret_value;
    packet data_packet;

    for(i = 0 ; i < NO_EVENT_NODES; i++)
    {
        while (1)//Generate source node within event region
        {
            s_x = rand() % (EVENT_DISTANCE_X + 1);
            s_y = rand() % (EVENT_DISTANCE_Y + 1);

            s_x += EVENT_OFFSET_X;
            s_y += EVENT_OFFSET_Y;

            //Make sure node is available & in a cluster
            if((nodes[s_x][s_y].NID == 0) || (nodes[s_x][s_y].CID == 0))

                continue;
            //Source node is dead find another node
            else if(nodes[s_x][s_y].node_dead == 1)
                continue;
            else //Form the source info of the data packet
            {
                //shik if same event node is found
                for(j = 0 ; j < i; j++)
                {
                    if(event_nodes[j] == nodes[s_x][s_y].NID)
                        continue;
                }
                nodes[s_x][s_y].in_event = 1;   //In event region
                //Add to list of nodes in event
                event_nodes[i] = nodes[s_x][s_y].NID;

                data_packet.source_NID = nodes[s_x][s_y].NID;
                CH_NID = nodes[s_x][s_y].CH_NID;
                source_CH_x = (CH_NID - 1) % NODESX;
                source_CH_y = (CH_NID - 1) / NODESX;
                data_packet.source_H_CID =
nodes[source_CH_x][source_CH_y].H_CID;

                data_packet.dest_NID = nodes[STARTX][STARTY].NID;
               //Send to root node
                data_packet.dest_H_CID.id[0] = 0;
                data_packet.dest_H_CID.id[1] = 0;
                data_packet.dest_H_CID.id[2] = 0;
                data_packet.dest_H_CID.id[3] = 0;
                break;
            }
        }

        ret_value= send_form_vsn_msg(data_packet);
        if(ret_value == 1)
            printf("Unable to send VSN formation message. Not enugh
energy.\n");
        else if(ret_value == 2)
            printf("Unable to send VSN formation message. No route
towards root node.\n");
        }
    }


/*--------------------------------------------------------------------*
* Following function send a form a VSN message                        *
* Each node that detects an event sends a message                     *
* towards the root node                                               *
* If a CH has already send a message it will not send another         *
* If two events meet message will stop there & sending                *
* node will get info on                                               *
* where they meet (Hierarchical address is send).                     *
* data_packet - header of the VSN formation message                   *
* return 0 on sucess, 1 if no energy & 2 is no route                  *
*--------------------------------------------------------------------*/
uchar send_form_vsn_msg(packet data_packet)
{
    int s_x, s_y, source_CH_x, source_CH_y, receiver_x, receiver_y,
current_x, current_y;
    int neighbor_to_forward, current_NID, i;
    int msg_send_by;      //Node that send the message
    uchar tmp_vsn_entries;

    receiver_x = -1;
    receiver_y = -1;
    msg_send_by = 0;

    s_x = (data_packet.source_NID - 1) % NODESX;
    s_y = (data_packet.source_NID - 1) / NODESX;
    if(nodes[s_x][s_y].know_event == 1)   //If already know event type 1
        return 0;
    else
        nodes[s_x][s_y].know_event = 1;

    source_CH_x = (nodes[s_x][s_y].CH_NID - 1) % NODESX;
    source_CH_y = (nodes[s_x][s_y].CH_NID - 1) / NODESX;

    //If souce is not a CH. Then forward the message to the CH
    if(data_packet.source_NID != nodes[s_x][s_y].CH_NID)
    {
        nodes[s_x][s_y].energy -= energy_to_transmit(VSN_FORM_SIZE , R);

        if(nodes[s_x][s_y].energy < 0)     //If not enough energy
        {
            nodes[s_x][s_y].node_dead = 1;  //Mark node as dead
            return 1;
        }
        nodes[s_x][s_y].know_event = 1;     //Know about event

        //If the rceiving CH is dead, drop message
        if(nodes[source_CH_x][source_CH_y].node_dead == 1)
            return 1;
```

```
        nodes[source_CH_x][source_CH_y].energy -=
energy_to_receive(VSN_FORM_SIZE);
        //If not enough energy to receive
        if(nodes[source_CH_x][source_CH_y].energy < 0)
        {
            //Mark node as dead
            nodes[source_CH_x][source_CH_y].node_dead = 1;
            return 1;
        }

        //Add child node to VSN table
        tmp_vsn_entries =
nodes[source_CH_x][source_CH_y].no_vsn_entries;
        //Makesure there is no duplicate entries
        for(i = 0 ; i < tmp_vsn_entries; i++)
        {
            //check for event type and NID
            if((nodes[source_CH_x][source_CH_y].vsn_table[i].NID ==
data_packet.source_NID)
                    && (nodes[source_CH_x][source_CH_y].vsn_table[i].VSN
== 1))
                break;
        }
        if(i == tmp_vsn_entries)
        {
            if(nodes[source_CH_x][source_CH_y].no_vsn_entries >
MAX_VSN_ENTRIES)
            {
                printf("No of VSN entries overflow. Terminating..\n");
                exit(1);
            }

nodes[source_CH_x][source_CH_y].vsn_table[tmp_vsn_entries].NID =
data_packet.source_NID;

nodes[source_CH_x][source_CH_y].vsn_table[tmp_vsn_entries].VSN = 1;

nodes[source_CH_x][source_CH_y].vsn_table[tmp_vsn_entries].node_type =
2; //Child node
            nodes[source_CH_x][source_CH_y].no_vsn_entries++;
        }

        //Another message is forwarded
        nodes[s_x][s_y].no_msg_forward++;

        //If CH already know event type 1
        if(nodes[source_CH_x][source_CH_y].know_event == 1)
            return 0;
        else
            nodes[source_CH_x][source_CH_y].know_event = 1;
        msg_send_by = nodes[s_x][s_y].NID;
    }
```

```
    current_NID = nodes[s_x][s_y].CH_NID;// Start forwarding from the CH

    //Loop until destination CH is found
    while(1)
    {
        //Find next hop
        neighbor_to_forward = next_hop(data_packet.dest_H_CID,
current_NID, msg_send_by);
        if(neighbor_to_forward == current_NID)  //Same as destination
            break;
        else if (neighbor_to_forward == 0)     //No route to destination
            return 2;
        else
        {
            current_x = (current_NID - 1) % NODESX;
            current_y = (current_NID - 1) / NODESX;

            //If is dead can't forward messages
            if(nodes[current_x][current_y].node_dead == 1)
                return 1;

            //Reduce energy to transmit. CH to CH messages
            //are high power with within R*TTL_max
            nodes[current_x][current_y].energy -=
energy_to_transmit(VSN_FORM_SIZE, (CH_CH_R_FACT * R));
            //If no energy to tranmit
            if(nodes[current_x][current_y].energy < 0)
            {
                //Mark node as dead
                nodes[current_x][current_y].node_dead = 1;
                return 1;       //Not enough energy
            }

            //Add parent CH to VSN table
            tmp_vsn_entries =
nodes[current_x][current_y].no_vsn_entries;
            //Makesure there are no duplicate entries
            for(i = 0 ; i < tmp_vsn_entries; i++)
            {
                //check for event type and NID
                if((nodes[current_x][current_y].vsn_table[i].NID ==
current_NID)
                        && (nodes[current_x][current_y].vsn_table[i].VSN
== 1))
                    break;
            }
            if(i == tmp_vsn_entries)
            {

nodes[current_x][current_y].vsn_table[tmp_vsn_entries].NID =
neighbor_to_forward;

nodes[current_x][current_y].vsn_table[tmp_vsn_entries].VSN = 1;
```

```
nodes[current_x][current_y].vsn_table[tmp_vsn_entries].node_type = 1;
//CH
                nodes[current_x][current_y].no_vsn_entries++;
            }

            receiver_x = (neighbor_to_forward - 1) % NODESX;
            receiver_y = (neighbor_to_forward - 1) / NODESX;

            //Node is dead can't receive messages
            if(nodes[receiver_x][receiver_y].node_dead == 1)
                return 1;

            nodes[receiver_x][receiver_y].energy -=
energy_to_receive(VSN_FORM_SIZE);
            //Not enough energy
            if(nodes[receiver_x][receiver_y].energy < 0)
            {
                //Mark node as dead
                nodes[receiver_x][receiver_y].node_dead = 1;
                return 1;
            }

            //Add child CH to VSN table
            tmp_vsn_entries =
nodes[receiver_x][receiver_y].no_vsn_entries;
            //Makesure there are no duplicate entries
            for(i = 0 ; i < tmp_vsn_entries; i++)
            {
                //check for event type and NID
                if((nodes[receiver_x][receiver_y].vsn_table[i].NID ==
current_NID)
                        &&
(nodes[receiver_x][receiver_y].vsn_table[i].VSN == 1))
                    break;
            }
            if(i == tmp_vsn_entries)
            {

nodes[receiver_x][receiver_y].vsn_table[tmp_vsn_entries].NID =
current_NID;

nodes[receiver_x][receiver_y].vsn_table[tmp_vsn_entries].VSN = 1;

nodes[receiver_x][receiver_y].vsn_table[tmp_vsn_entries].node_type = 1;
//CH
                nodes[receiver_x][receiver_y].no_vsn_entries++;
            }
               //Another message is forwarded
nodes[current_x][current_y].no_msg_forward++;
            //If already know event
            if(nodes[receiver_x][receiver_y].know_event == 1)
                return 0;
            else
                nodes[receiver_x][receiver_y].know_event = 1;

            msg_send_by = nodes[current_x][current_y].NID;
           //Forward packet to neighbor. Neighbor becomes current node
            current_NID = neighbor_to_forward ;
        }
    }

    return 0;    //Packet is sucessfully delivered

}


/*--------------------------------------------------------------------*
* Following functions sends VSN data packets from                     *
* randomly selected node to                                           *
* another randomly selected node.                                      *
* It count the no of messages sucessfully                             *
* delivered & terminates either when                                  *
* the 1st message get dropped or given number of messages             *
* are transmitted                                                     *
*--------------------------------------------------------------------*/
void send_vsn_unicast_data()
{
    int s_x, s_y, d_x, d_y, source_CH_x, source_CH_y, dest_CH_x,
dest_CH_y, i;
    uint s_nid, d_nid, rnd, ch_nid;
    uchar result;
    uint no_msg_delivered = 0;    //No of sucessfully delivered messages
    uint no_msg_dropped = 0;      //Nof of messages dropped
    uint no_route = 0;            //No fo routes not found
    packet data_packet;           //Data packet to be transmitted
    double energy_before, energy_after;

    energy_before = total_energy();
    for(i = 0 ; i < NO_PACKETS; i++)  //Send NO_PACKETS
    {
        while(1)
        {
            //Generate source node
            rnd = rand() % NO_EVENT_NODES;
            s_nid = event_nodes[rnd];
            rnd = rand() % NO_EVENT_NODES;
            d_nid = event_nodes[rnd];

            if(s_nid == d_nid)  //If source & destination is same
                continue;

            s_x = (s_nid - 1) % NODESX;
            s_y = (s_nid - 1) / NODESX;
            ch_nid = nodes[s_x][s_y].CH_NID;
            source_CH_x = (ch_nid - 1) % NODESX;
            source_CH_y = (ch_nid - 1) / NODESX;
```

```
            d_x = (d_nid - 1) % NODESX;
            d_y = (d_nid - 1) / NODESX;
            ch_nid = nodes[d_x][d_y].CH_NID;
            dest_CH_x = (ch_nid - 1) % NODESX;
            dest_CH_y = (ch_nid - 1) / NODESX;

            data_packet.source_NID = s_nid;
            data_packet.dest_NID = d_nid;

            data_packet.source_H_CID =
nodes[source_CH_x][source_CH_y].H_CID;
            data_packet.dest_H_CID = nodes[dest_CH_x][dest_CH_y].H_CID;
            break;
        }

     //Send data from source to destination. Send as normal data packets
        result = send_data_packet(data_packet);
        if(result == 1) //If the message is dropped due to low energy
        {
            no_msg_dropped++;
            break;
        }
        else if(result == 2)    //If message drop due to the wrong route
        {
            no_route++;
            break;
        }
        else
            no_msg_delivered++;
     }
    energy_after = total_energy();
    //printf("%d\n", no_msg_delivered);
    //printf("%d\t%d\t%d\n", no_msg_delivered, no_msg_dropped,
no_route);
    printf("%d\t%f\t%f\n", no_msg_delivered, energy_before,
energy_after);
}


/*-------------------------------------------------------------------------*
 * Following function generates a VSN multicast packet and                 *
 * add it for routing                                                      *
 * to the event list.                                                      *
 * After calling this function process_event() list must                   *
 * be caled to deliver                                                     *
 * multicast messages. Delivery will be handled by                         *
 * the send_vsn_multicast_data()                                           *
*-------------------------------------------------------------------------*/
void send_vsn_multicast_data()
{
    int s_x, s_y, d_x, d_y, source_CH_x, source_CH_y, i, j, nid, ch_nid,
des_nid, rnd;
    Hie_CID source_h_cid, dest_h_cid;
    uchar result;

    uint no_msg_delivered = 0;    //No of sucessfully delivered messages
    uint no_msg_dropped = 0;      //Nof of messages dropped
    uint no_route = 0;            //No fo routes not found

    for(i = 0 ; i < NO_PACKETS; i++)
    {
        //Generate source node
        rnd = rand() % NO_EVENT_NODES;
        nid = event_nodes[rnd];
        s_x = (nid - 1) % NODESX;
        s_y = (nid - 1) / NODESX;

        ch_nid = nodes[s_x][s_y].CH_NID;
        source_CH_x = (ch_nid - 1) % NODESX;
        source_CH_y = (ch_nid - 1) / NODESX;

        if(nodes[s_x][s_y].NID != nodes[s_x][s_y].CH_NID) //If not a CH
        {
            source_h_cid.id[0] = 0;
            source_h_cid.id[1] = 0;
            source_h_cid.id[2] = 0;
            source_h_cid.id[3] = 0;
            dest_h_cid = nodes[source_CH_x][source_CH_y].H_CID;

            //Parameters in the event list has following meanings
            //nid - Nid of source node
            //ch_nid - NID of destination node
            //h_cid - Hierarchical CID of source
            //parent_h_cid - - Hierarchical CID of destination
            add_event(4, last_type3_event_time, nid, 0, ch_nid,
source_h_cid, 0, 0, 0, 0, 0, dest_h_cid);
            last_type3_event_time++;
        }
        else  //if CH send a seperate packet for each entry in VSN table
        {
            for(j = 0 ; j < nodes[s_x][s_y].no_vsn_entries; j++)
            {
                des_nid = nodes[s_x][s_y].vsn_table[j].NID;
                d_x = (des_nid - 1) % NODESX;
                d_y = (des_nid - 1) / NODESX;

                source_h_cid = nodes[source_CH_x][source_CH_y].H_CID;

                //If destination is a CH
                if(des_nid == nodes[d_x][d_y].CH_NID)
                    dest_h_cid = nodes[d_x][d_y].H_CID;
                else    //if it's a child node
                {
                    dest_h_cid.id[0] = 0;
                    dest_h_cid.id[1] = 0;
                    dest_h_cid.id[2] = 0;
                    dest_h_cid.id[3] = 0;
                }
```

```
                //Parameters in the event list has following meanings
                //nid - Nid of source node
                //ch_nid - NID of destination node
                //h_cid - Hierarchical CID of source
                //parent_h_cid - - Hierarchical CID of destination
                add_event(4, last_type3_event_time, nid, 0, des_nid,
source_h_cid, 0, 0, 0, 0, 0, dest_h_cid);
                last_type3_event_time++;
            }
        }
        while(1)    //Process the event list until no events are found
        {
            result = process_event_list();
           //Actual result is incremented by 1, by the sender
            if(result != 1)
                break;
        }
        if(result == 2)  //If the message is dropped due to low energy
        {
            no_msg_dropped++;
            break;
        }
        else if (result == 3) //If message drop due to the wrong route
        {
            no_route++;
            break;
        }
        else
            no_msg_delivered++;
    }
    //printf("%d\n", no_msg_delivered);
    printf("%d\t%d\t%d\n", no_msg_delivered, no_msg_dropped, no_route);
}


/*--------------------------------------------------------------------*
 * Following function forwards a VSN multicats message               *
 * between a given source                                            *
 * & a destination. If the receiver has VSN entries                  *
 * in it's VSN table new events                                      *
 * are added to the event list.                                      *
 * data_packet - header of the VSN formation message                 *
 * return 0 on sucess, 1 if no energy & 2 is no route                *
*--------------------------------------------------------------------*/
uchar send_vsn_multicast_packet(packet data_packet)
{
    int s_x, s_y, d_x, d_y, i, des_nid;
    Hie_CID source_h_cid, dest_h_cid;

    s_x = (data_packet.source_NID - 1) % NODESX;
    s_y = (data_packet.source_NID - 1) / NODESX;
    d_x = (data_packet.dest_NID - 1) % NODESX;
    d_y = (data_packet.dest_NID - 1) / NODESX;

    //receive the message
    //if source is a child node
    if(nodes[s_x][s_y].NID != nodes[s_x][s_y].CH_NID)
    {
    //If is dead can't forward messages
        if(nodes[s_x][s_y].node_dead == 1)
            return 1;

        //Reduce energy to transmit. CH to CH messages are high power
with within R*TTL_max
        nodes[s_x][s_y].energy -=  energy_to_transmit(DATA_PACKET_SIZE,
R);
        if(nodes[s_x][s_y].energy < 0)  //If no energy to tranmit
        {
            nodes[s_x][s_y].node_dead = 1;  //Mark node as dead
            return 1;                       //Not enough energy
        }

        //Node is dead can't receive messages
        if(nodes[d_x][d_y].node_dead == 1)
            return 1;

        nodes[d_x][d_y].energy -= energy_to_receive(DATA_PACKET_SIZE);
        if(nodes[d_x][d_y].energy < 0)     //Not enough energy
        {
            nodes[d_x][d_y].node_dead = 1;  //Mark node as dead
            return 1;
        }
    }
    else    //if source is a another CH
    {
       //If destination is a child node
        if(nodes[d_x][d_y].NID != nodes[d_x][d_y].CH_NID)
        {
          //If is dead can't forward messages
            if(nodes[s_x][s_y].node_dead == 1)
                return 1;

            //Reduce energy to transmit. CH to CH messages are high
power with within R*TTL_max
            nodes[s_x][s_y].energy -=
energy_to_transmit(DATA_PACKET_SIZE, R);
            if(nodes[s_x][s_y].energy < 0)  //If no energy to tranmit
            {
                nodes[s_x][s_y].node_dead = 1;  //Mark node as dead
                return 1;                       //Not enough energy
            }

                  //Node is dead can't receive messages
            if(nodes[d_x][d_y].node_dead == 1)
                return 1;

            nodes[d_x][d_y].energy -=
energy_to_receive(DATA_PACKET_SIZE);
```

```
            if(nodes[d_x][d_y].energy < 0) //Not enough energy
            {
                nodes[d_x][d_y].node_dead = 1;  //Mark node as dead
                return 1;
            }
            return 0;   //Sucessfully delivered to destination
        }
        else    //if destination is a CH
        {
            //If is dead can't forward messages
            if(nodes[s_x][s_y].node_dead == 1)
                return 1;

            //Reduce energy to transmit. CH to CH messages are high
power with within R*TTL_max
            nodes[s_x][s_y].energy -=
energy_to_transmit(DATA_PACKET_SIZE, (CH_CH_R_FACT * R));
            if(nodes[s_x][s_y].energy < 0)  //If no energy to tranmit
            {
                nodes[s_x][s_y].node_dead = 1;  //Mark node as dead
                return 1;                       //Not enough energy
            }

            //Node is dead can't receive messages
            if(nodes[d_x][d_y].node_dead == 1)
                return 1;

            nodes[d_x][d_y].energy -=
energy_to_receive(DATA_PACKET_SIZE);
            if(nodes[d_x][d_y].energy < 0) //Not enough energy
            {
                nodes[d_x][d_y].node_dead = 1;  //Mark node as dead
                return 1;
            }
        }
    }

    //Now source and destination changes. New destination is
    //what is in my VSN table
    //Code will come this point only if it's a CH.
    s_x = d_x;
    s_y = d_y;

    for(i = 0 ; i < nodes[s_x][s_y].no_vsn_entries; i++)
    {
        des_nid = nodes[s_x][s_y].vsn_table[i].NID;
        //Skip if the entry is for sender. Prevents loops
        if(des_nid == data_packet.source_NID)
            continue;

        d_x = (des_nid - 1) % NODESX;
        d_y = (des_nid - 1) / NODESX;

        source_h_cid = nodes[s_x][s_y].H_CID;

        if(des_nid == nodes[d_x][d_y].CH_NID)   //If destination is a CH
            dest_h_cid = nodes[d_x][d_y].H_CID;
        else    //if it's a child node
        {
            dest_h_cid.id[0] = 0;
            dest_h_cid.id[1] = 0;
            dest_h_cid.id[2] = 0;
            dest_h_cid.id[3] = 0;
        }

        //Parameters in the event list has following meanings
        //nid - Nid of source node
        //ch_nid - NID of destination node
        //h_cid - Hierarchical CID of source
        //parent_h_cid - - Hierarchical CID of destination
        add_event(4, last_type3_event_time, nodes[s_x][s_y].NID, 0,
des_nid, source_h_cid, 0, 0, 0, 0, 0, dest_h_cid);
        last_type3_event_time++;
    }
    return 0;
}


/*--------------------------------------------------------------------*
* Following function is used to inform neighboring CHs about         *
* a particular CH's                                                  *
* Hierarchical address. This help to build corss                     *
* links along cluster tree                                           *
*--------------------------------------------------------------------*/
void inform_neighbors()
{
    int x, y, minx, miny, maxx, maxy, l, k, i;
    region my_region;
    float length, range;

    range = CH_CH_R_FACT * R;

    for(x = 0 ; x < NODESX; x++)    //Check for CHs
    {
        for(y = 0; y < NODESY; y++)
        {
              //If node exists and is a CH
            if((nodes[x][y].NID == nodes[x][y].CH_NID) &&
(nodes[x][y].NID != 0))
            {
                nodes[x][y].energy -=
energy_to_transmit(DATA_PACKET_SIZE, (CH_CH_R_FACT * R));
     //determine my neighborhood

                my_region = get_node_region(x, y, range);
                minx = my_region.minx;
                miny = my_region.miny;
                maxx = my_region.maxx;
```

```
                maxy = my_region.maxy;

                for(l = miny; l <= maxy; l++)
                {
                    for (k = minx; k <= maxx; k++)
                    {
                        length = sqrt((k - x)*(k - x)*(GRIDX * GRIDX) +
(l - y)*(l - y)*(GRIDY * GRIDY));

                        // If within communication range, if
                        //node exists, if node is a CH
                        if ((length <= range) && (nodes[k][l].NID != 0)
&& (nodes[k][l].NID ==
                                nodes[k][l].CH_NID) && (length != 0))
                        {
                            //Check whther the routing entry
                            //already exit (parent/child will exist)
                            for(i = 0; i <
nodes[k][l].no_routing_entries; i++)
                            {
                                if(nodes[k][l].routing_table[i].NID ==
nodes[x][y].NID)
                                    break;
                            }
                            if(i != nodes[k][l].no_routing_entries)
//Skip if CH is already in
                                continue;
                            else    //Else add to the routing table
                            {
                                if(nodes[k][l].no_routing_entries >
MAX_ROUTES)
                                    printf("Number of routing entries
overflow.\n");

                                nodes[k][l].routing_table[i].NID =
nodes[x][y].NID;
                                nodes[k][l].routing_table[i].H_CID =
nodes[x][y].H_CID;
                                nodes[k][l].routing_table[i].learn_from
= nodes[x][y].NID;
                                nodes[k][l].routing_table[i].valid = 5;
        //Entry learn from neighbor, 5 = 101
                                nodes[k][l].routing_table[i].hops = 1;
                                nodes[k][l].no_routing_entries++;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
/*-------------------------------------------------------------------*
* Following function forms a nother cluster tree based on            *
* already existing CHs                                               *
* Multiple such trees can be build by modifiying this                *
* function & other related                                           *
* functions                                                          *
* Such trees will form a connected graph in the network & can        *
* faciliate better                                                   *
* node-to-node routing                                               *
* tree_depth - if tree to be formed by combining nodes               *
* at a particular depth in                                           *
* the orginal cluster tree. Consider if                              *
*   depth <= CH depth <= depth + 1                                   *
* Function needs to be modified if depth is not restricted.          *
*-------------------------------------------------------------------*/
void form_second_cluster_tree(uchar tree_depth)
{
    int x, y, node_list[100], nid;
    uchar no_nodes = 0;
    Hie_CID h_cid;

    for(x = 0; x < NODESX; x++) //Find list of nodes in given depth
    {
        for(y = 0; y < NODESY; y++)
        {   //if its a CH & in given depth
            if((nodes[x][y].NID == nodes[x][y].CH_NID) &&
(nodes[x][y].NID != 0)
                    && (nodes[x][y].tree_depth == tree_depth))
            {
                printf("%d\n", nodes[x][y].NID);
                if(no_nodes < 100)
                {
                    node_list[no_nodes] = nodes[x][y].NID;
                    no_nodes++;
                }
                else
                    continue;
            }
        }
    }

    nid = node_list[(rand() % no_nodes)];    //Pick a random node
    x = (nid - 1) % NODESX;
    y = (nid - 1) / NODESX;
    h_cid.id[0] = 0;
    h_cid.id[1] = 0;
    h_cid.id[2] = 0;
    h_cid.id[3] = 0;
    nodes[x][y].Link_H_CID = h_cid;
    nodes[x][y].link_depth = 0;
    //Parameters in the event list has following meanings
    //This is a type 5 event
    //nid - Nid of source CH
    //h_cid - Hierarchical CID of source CH
```

```
    //tree_depth - tree depth in the cluster tree
    //node_depth - tree depth in the new tree
    //parent_nid - nid of the parent CH
    add_event(5, last_type3_event_time, nid, 0, 0, h_cid, tree_depth, 0,
0, 0, 0, h_cid);
    last_type3_event_time++;
    while(process_event_list());
}


/*-------------------------------------------------------------------------*
 * Following function add already existing CHs to a new clustr tree   *
 * This function needs to rerun again & again until                  *
 * tree finish spanning                                              *
 * Will be initially called by the form_second_cluster_tree function. *
 * For simplicity this process is sequential (no parallel events      *
 * nid - NID of the CH sending the message                           *
 * parent_nid - NID of the parent CH                                 *
 * tree_depth - Depth in the original tree                           *
 * link_depth - depth in the new tree/link                           *
 * h_cid - Hierarchical CID in the original cluster tree             *
*-------------------------------------------------------------------------*/
void add_ch_to_tree(uint nid, uint parent_nid, uchar tree_depth, uchar
link_depth, Hie_CID h_cid)
{
    int x, y, minx, miny, maxx, maxy, l, k;
    region my_region;
    float length;
    uchar child_no, new_link_depth;
    Hie_CID tmp_h_cid;

    child_no = 0;
    new_link_depth = link_depth + 1;
    tmp_h_cid.id[0] = 0;
    tmp_h_cid.id[1] = 0;
    tmp_h_cid.id[2] = 0;
    tmp_h_cid.id[3] = 0;

    x = (nid - 1) % NODESX;
    y = (nid - 1) / NODESX;
    my_region = get_node_region(x, y, (CH_CH_R_FACT * R));//My region
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    nodes[x][y].energy -=  energy_to_transmit(CLUSTER_BCAST_SIZE,
(CH_CH_R_FACT * R));
    if(nodes[x][y].energy < 0)  //If no energy to tranmit
    {
        nodes[x][y].node_dead = 1;  //Mark node as dead
        return ;                    //Not enough energy
    }

    for(l = miny; l <= maxy; l++) //Check within my region
    {
        for (k = minx; k <= maxx; k++)
        {
         //Concept of RSSI is not applicable here as far as R is known
            length = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l - y)*(l-
y)*(GRIDY * GRIDY));
            // if within communication range
            if((length <= (CH_CH_R_FACT * R)) && (nodes[k][l].NID != 0)
&& (nodes[k][l].NID == nodes[k][l].CH_NID))
            {
                if((nodes[k][l].tree_depth == tree_depth) ||
(nodes[k][l].tree_depth == (tree_depth + 1)))
                {
                    if (nodes[k][l].NID != parent_nid)  //skip parent
                    {
                  //Node is dead can't receive messages
                        if(nodes[k][l].node_dead == 1)
                            continue;
                        nodes[k][l].energy -=
energy_to_receive(CLUSTER_BCAST_SIZE);
                        if(nodes[k][l].energy < 0)  //Not enough energy
                        {
                            //Mark node as dead
                            nodes[k][l].node_dead = 1;
                            continue;
                        }

                        if(nodes[k][l].link_depth > new_link_depth)
                        {
                            //can't add more than 8 child nodes
                            if(child_no < 7)
                            {
                                nodes[k][l].Link_H_CID =
generate_CID(h_cid, (int)child_no, new_link_depth);
                                nodes[k][l].link_depth = new_link_depth;
                                child_no++;
                    //Parameters in the event list has following meanings
                    //This is a type 5 event
                    //nid - Nid of source CH
                    //h_cid - Hierarchical CID of source CH
                    //tree_depth - tree depth in the cluster tree
                    //node_depth - tree depth in the new tree
                    //parent_nid - nid of the parent CH
                                add_event(5, last_type3_event_time,
nodes[k][l].NID, 0, 0,
                                          nodes[k][l].Link_H_CID,
tree_depth, new_link_depth, 0, 0, nid, tmp_h_cid);
                                last_type3_event_time++;
                    //send the ACK & its received by the parent
                                nodes[k][l].energy -=
energy_to_transmit(CLUSTER_ACK_SIZE, (CH_CH_R_FACT * R));
                                nodes[x][y].energy -=
energy_to_receive(CLUSTER_ACK_SIZE);
```

```
                        }
                        else
                            continue;
                    }
                    else
                        continue;
                }
                else
                    continue;
            }
        }
    }
}


/*--------------------------------------------------------------------*
* Following function discover CHs at a given depth & depth + 1       *
* in the cluster                                                     *
* tree. It then initiates sharing of roting information between them. *
* tree_depth - Depth in the original tree                            *
*--------------------------------------------------------------------*/
void discover_neighbors_of_link(uchar tree_depth)
{
    int x, y;
    Hie_CID h_cid;

    h_cid.id[0] = 0;
    h_cid.id[1] = 0;
    h_cid.id[2] = 0;
    h_cid.id[3] = 0;

    for(x = 0; x < NODESX; x++) //Find list of nodes in given depth
    {
        for(y = 0; y < NODESY; y++)
        {   //if its a CH & in given depth
            if((nodes[x][y].NID == nodes[x][y].CH_NID) &&
(nodes[x][y].NID != 0) &&
                ((nodes[x][y].tree_depth >= tree_depth) &&
nodes[x][y].tree_depth <= (tree_depth + 1)))
            {
                //Parameters in the event list has following meanings
                //This is a type 6 event
                //nid - Nid of CH sending routing table
                //tree_depth - tree depth in the cluster tree
                add_event(6, last_type3_event_time, nodes[x][y].NID, 0,
0, h_cid, tree_depth, 0, 0, 0, 0, h_cid);
                last_type3_event_time++;
                nodes[x][y].send_routing_info = 0;
            }
        }
    }
    //process untill all events are completed
    while(process_event_list());
}


/*--------------------------------------------------------------------*
* Following function share routing table info amonge                 *
* CHs at a given depth &                                             *
* depth + 1 in the cluster tree. Info is shared only                 *
* if they are neighbors,                                             *
* within the given depth range and if the new info                   *
* is better than going through                                       *
* the cluster tree (will add entry if same as distance               *
* through routing table)                                             *
* nid - node broadcasting its routing table                          *
* tree_depth - Depth in the original tree                            *
*--------------------------------------------------------------------*/
void send_link_info(uint nid, uchar tree_depth)
{
    int x, y, minx, maxx, miny, maxy, i, j, k, l, data_size;
    region my_region;
    float length;
    uchar hops, depth;
    Hie_CID h_cid;

    h_cid.id[0] = 0;
    h_cid.id[1] = 0;
    h_cid.id[2] = 0;
    h_cid.id[3] = 0;

    x = (nid - 1) % NODESX;
    y = (nid - 1) / NODESX;
    data_size = sizeof(router_entry) * nodes[x][y].no_routing_entries;

    //Discard if no update happened
    if(nodes[x][y].send_routing_info == 1)
        return;

    if(nodes[x][y].node_dead == 1) //Node is dead can't receive messages
        return;
    nodes[x][y].energy -=  energy_to_transmit(data_size, (CH_CH_R_FACT *
R));
    if(nodes[x][y].energy < 0)    //Not enough energy
    {
        nodes[x][y].node_dead = 1;  //Mark node as dead
        return;
    }

    //Get my region
    my_region = get_node_region(x, y, (CH_CH_R_FACT * R));
    minx = my_region.minx;
    miny = my_region.miny;
    maxx = my_region.maxx;
    maxy = my_region.maxy;

    for(l = miny; l <= maxy; l++) //Check within my region
```

```
    {
        for (k = minx; k <= maxx; k++)
        {
       //Node should exist, should be a CH, should be within given depth
            if((nodes[k][l].NID != 0) && (nodes[k][l].NID ==
nodes[k][l].CH_NID) &&
                    ((nodes[k][l].tree_depth >= tree_depth) &&
(nodes[k][l].tree_depth <= (tree_depth + 1))))
            {
                length = sqrt((k - x)*(k-x)*(GRIDX * GRIDX) + (l -
y)*(l-y)*(GRIDY * GRIDY));
                // if within communication range
                if((length <= (CH_CH_R_FACT * R)) && (length != 0))
                {
                    //Energy to receive
                    //Node is dead can't receive messages
                    if(nodes[k][l].node_dead == 1)
                        return;
                    nodes[k][l].energy -=  energy_to_receive(data_size);
                    if(nodes[k][l].energy < 0)      //Not enough energy
                    {
                        nodes[k][l].node_dead = 1;  //Mark node as dead
                        continue;
                    }

                    for(i = 0 ; i < nodes[x][y].no_routing_entries; i++)
//for each routing entry
                    {
                        depth = nodes[x][y].routing_table[i].H_CID.id[3]
& 63;
                        //skip the entries beyound the given depth range
                        if((depth < tree_depth) || (depth > (tree_depth
+ 1)))
                            continue;
                        else if(nodes[x][y].routing_table[i].valid == 0)
//continue if invalid
                            continue;
                        else //if valid
                        {
                            //compare with each entry in the receiver
                            for(j = 0; j <
nodes[k][l].no_routing_entries; j++)
                            {
                                depth =
nodes[k][l].routing_table[j].H_CID.id[3] & 63;
                        //skip the entries beyound the given depth range
                                if((depth < tree_depth) || (depth >
(tree_depth + 1)))
                                    continue;
                                else
if(nodes[k][l].routing_table[j].valid == 0) //skip if invalid
                                    continue;
                                else
if(nodes[k][l].routing_table[j].valid == 3) //skip my parent entry
                                    continue;
                                else if(nodes[x][y].routing_table[i].NID
== nodes[k][l].routing_table[j].NID)
                                    {
                                //if same entry see whether new info is better
                                //if new imformation is not useful

if((nodes[x][y].routing_table[i].hops + 1) >=
nodes[k][l].routing_table[j].hops)
                        //break inner loop if useless, can't be two matching entries
                                        break;
                                    else    //if new info is useful
                                    {

nodes[k][l].routing_table[j].valid = 5; //valid & from neighbor
                                        nodes[k][l].routing_table[j].NID
= nodes[x][y].routing_table[i].NID;

nodes[k][l].routing_table[j].H_CID = nodes[x][y].routing_table[i].H_CID;

nodes[k][l].routing_table[j].learn_from = nid;

nodes[k][l].routing_table[j].hops = nodes[x][y].routing_table[i].hops +
1;
                                //Parameters in the event list has following meanings
                                //This is a type 6 event
                                //nid - Nid of CH sending routing table
                                //tree_depth - tree depth in the cluster tree
                                        add_event(6,
last_type3_event_time, nodes[k][l].NID, 0, 0, h_cid, tree_depth, 0, 0,
0, 0, h_cid);
                                        last_type3_event_time++;
                                        nodes[k][l].send_routing_info =
0;
                                        break;
                                    }
                                }
                            }

                            //No matching entry
                            if(j == nodes[k][l].no_routing_entries)
                            {
                                if(nodes[k][l].no_routing_entries <
MAX_ROUTES)
                                {
                               //determine distance through cluster tree
                                    hops =
hop_distance(nodes[k][l].H_CID, nodes[x][y].H_CID);
                        //if cluster tree distance is >= to what I learn from neighbor
                                    if(hops >=
(nodes[x][y].routing_table[i].hops + 1))
                                    {

nodes[k][l].routing_table[j].valid = 5; //valid & from neighbor
```

```
                                        nodes[k][l].routing_table[j].NID
= nodes[x][y].routing_table[i].NID;

nodes[k][l].routing_table[j].H_CID = nodes[x][y].routing_table[i].H_CID;

nodes[k][l].routing_table[j].learn_from = nid;

nodes[k][l].routing_table[j].hops = nodes[x][y].routing_table[i].hops +
1;

nodes[k][l].no_routing_entries++;
                    //Parameters in the event list has following meanings
                        //This is a type 6 event
                        //nid - Nid of CH sending routing table
                        //tree_depth - tree depth in the cluster tree
                                        add_event(6,
last_type3_event_time, nodes[k][l].NID, 0, 0, h_cid, tree_depth, 0, 0,
0, 0, h_cid);
                                        last_type3_event_time++;
                                        nodes[k][l].send_routing_info =
0;
                                    }
                                }
                                else
                                    printf("Number of routing entries
overflow.\n");
                            }
                        }
                    }
                }
            }
        }
    }
    nodes[x][y].send_routing_info = 1;
}


/*-------------------------------------------------------------------------*
* Follwing function calculates the maximum achievable                      *
* circularity (MAC) for                                                    *
* each cluster & sump circularity to the circularity.txt file              *
* this functions works only up to 3-hop clusters.                          *
* Can be extened to other cases                                            *
* file - if 1 write to 1st file else write to 2nd file                     *
* (optimization phase)                                                     *
*-------------------------------------------------------------------------*/
void calculate_circularity(uchar file)
{
    int i, j, x, y, minx, miny, maxx, maxy, l, k, retvalue;
    float length, in_cluster, outof_cluster, circularity;
    int nb_list[3][500];
    int level0, level1, level2, n, m, p, q;
    region my_region;

    FILE *circlefd;
    if(file == 1) //which file
        circlefd = fopen(CIRCLEFILE1, "w");
    else
        circlefd = fopen(CIRCLEFILE2, "w");

    if(circlefd == NULL)
        perror("ERROR: No circularity data will be written.....");

    for (i = 0; i < NODESX ; i++) //Check for a CH from all nodes
    {
        for(j = 0 ; j < NODESY ; j++)
        {
            //make sure that the node is a CH
            if((nodes[i][j].NID == nodes[i][j].CH_NID) &&
(nodes[i][j].NID != 0))
            {
                in_cluster = 0;    //no of nodes inside cluster
                outof_cluster = 0;//no of nodes outside cluster
               //check no of hops for single & multi-hop clustering
                level0 = level1 = level2 = 0;

               //Get my X, Y coordinates
                x = (nodes[i][j].NID - 1) % NODESX;
                y = (nodes[i][j].NID - 1) / NODESX;
                my_region = get_node_region(x, y, R);//Get my region
                minx = my_region.minx;
                miny = my_region.miny;
                maxx = my_region.maxx;
                maxy = my_region.maxy;

                for(l = miny; l <= maxy; l++) //Check within my region
                {
                    for (k = minx; k <= maxx; k++)
                    {
           //Concept of RSSI is not applicable here as far as R is known
                        length = sqrt((k - x)*(k - x)*(GRIDX * GRIDX) +
(l - y)*(l - y)*(GRIDY * GRIDY));
                        if((length <=  R) && (nodes[k][l].NID != 0))
// if within communication range
                        {
                            if(nodes[k][l].CID == nodes[i][j].CID)
//If with the same CID, inside
                                in_cluster++;
                            else
//Else outside
                                outof_cluster++;
                            nb_list[0][level0] = nodes[k][l].NID;
                            level0++;
                        }
                    }
                }
                //If multi-hop clusters
                if( (MAX_HOPS > 1) && (level0 != 0))
```

```
                {
                    for(n=0; n < level0 ; n++)
                    {
                        x = (nb_list[0][n] - 1) % NODESX;
                        y = (nb_list[0][n] - 1) / NODESX;
                        //get my region
                        my_region = get_node_region(x, y, R);
                        minx = my_region.minx;
                        miny = my_region.miny;
                        maxx = my_region.maxx;
                        maxy = my_region.maxy;

                        for(l = miny; l <= maxy; l++) //within my region
                        {
                            for (k = minx; k <= maxx; k++)
                            {
                                length = sqrt((k - x)*(k - x)*(GRIDX *
GRIDX) + (l-y)*(l-y)*(GRIDY * GRIDY));
                                if((length <=  R) && (nodes[k][l].NID !=
0))
                                {
                                    for(m = 0; m < level0; m++)

//neighbors at level 1
                                    {
                                        if(nb_list[0][m] ==
nodes[k][l].NID)
                                            break;
                                    }
                                    for( p = 0; p < level1; p++)

//neighbors at level 2
                                    {
                                        if(nb_list[1][p] ==
nodes[k][l].NID)
                                            break;
                                    }
                                    if((m == level0) && (p == level1))
                                    {
                                        if(nodes[k][l].CID ==
nodes[i][j].CID)
                                         //If with same CID in cluster
                                            in_cluster++;
                                        else     //not in cluster
                                            outof_cluster++;
                                        nb_list[1][level1] =
nodes[k][l].NID;
                                        level1++;
                                    }
                                }
                            }
                        }
                    }
                }
                if( (MAX_HOPS > 2) && (level1 != 0)) //if 3 hops or more
                {
                    for(n=0; n < level1 ; n++)
                    {
                        x = (nb_list[1][n] - 1) % NODESX;
                        y = (nb_list[1][n] - 1) / NODESX;
                        my_region = get_node_region(x, y, R);
            //get my region
                        minx = my_region.minx;
                        miny = my_region.miny;
                        maxx = my_region.maxx;
                        maxy = my_region.maxy;

                        for(l = miny; l <= maxy; l++)
                        {
                            for (k = minx; k <= maxx; k++)
                            {
                                length = sqrt((k - x)*(k - x)*(GRIDX *
GRIDX) + (l-y)*(l-y)*(GRIDY * GRIDY));
                                if((length <=  R) && (nodes[k][l].NID !=
0))
                                {
                                    for(m = 0; m < level0; m++)//level 0
                                    {
                                        if(nb_list[0][m] ==
nodes[k][l].NID)
                                            break;
                                    }
                                    for( p = 0; p < level1; p++)//level 1
                                    {
                                        if(nb_list[1][p] ==
nodes[k][l].NID)
                                            break;
                                    }
                                    for(q= 0; q < level2; q++) //level 2
                                    {
                                        if(nb_list[2][q] ==
nodes[k][l].NID)
                                            break;
                                    }

                                    if((m == level0) && (p == level1) &&
(q == level2))
                                    {
                                        if(nodes[k][l].CID ==
nodes[i][j].CID)
                                            in_cluster++;
                                        else
                                            outof_cluster++;
                                        nb_list[2][level2] =
nodes[k][l].NID;
                                        level2++;
                                    }
                                }
                            }
                        }
                    }
                }
```

```
                }
            }
        //circularity = total inside / total nodes in range
        //for multihop clusters a node is considerd to be in range only
        //if there is a path from node to CH
            circularity = (in_cluster/(in_cluster + outof_cluster))
* 100;
            retvalue = sprintf(msg, "%f\n", circularity);
            //retvalue = sprintf(msg, "%d\t%f\n", nodes[i][j].CID,
circularity);
            fputs(msg, circlefd);
        }
    }
  }
  fclose(circlefd);     //close file
}


/*---------------------------------------------------------------------*
* Following function dump remaining energy of each node to a text file*
*---------------------------------------------------------------------*/
void print_cluster_energy()
{
    int i, j, retvalue;
    FILE *energyfd;

    energyfd = fopen(ENERGYFILE, "w");  //open file
    if(energyfd == NULL)
        perror("ERROR: No energy data will be written.....");

    //Following code writes node energy to the text file
    for (i = 0; i < NODESY ; i++)
    {
        for(j = 0 ; j < NODESX ; j++)
        {
            if(nodes[j][i].NID != 0) //if node exist
            {
                if(nodes[j][i].energy > 0.0)
                    retvalue = sprintf(msg, "%f\n", nodes[j][i].energy);
                else
                    retvalue = sprintf(msg, "%f\n", 0.00);
                fputs(msg, energyfd);
            }
        }
    }
    fclose(energyfd);   //close file
}


/*---------------------------------------------------------------------*
* This function either printer the node status on                      *
* the terminal or print node                                           *
* data to a text file named nodes.txt                                  *
* symbols:                                                             *
* '.' - Indicate grid points with nodes                                *
* 'o' - Indicate nodes with a CH                                       *
* '?' - Indicare nodes clusters that don't have a represnetable symbol*
* Cluster symbol followed by a . indicate CHs                          *
* pnt_console  - print data to console                                 *
* file - which file to use. 1 - 1st file, 2 - 2nd file (optimized)     *
*---------------------------------------------------------------------*/
void print_nodes(uchar pnt_console, uchar file)
{
    int i, j, retvalue;
    FILE *nodefd;

    if(file == 1)   //open file
        nodefd = fopen(NODEFILE1, "w");
    else
        nodefd = fopen(NODEFILE2, "w");

    if(nodefd == NULL)
        perror("ERROR: No node data will be written.....");

    //Following code writes node data to the text file
    for (i = 0; i < NODESY ; i++)
    {
        for(j = 0 ; j < NODESX ; j++)
        {
            if(nodes[j][i].NID != 0)
            {
                //form the text string
                retvalue = sprintf(msg,
"%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", nodes[j][i].NID,
nodes[j][i].CID,
                        nodes[j][i].CH_NID, nodes[j][i].parent_CH_NID,
nodes[j][i].tree_depth,
                        nodes[j][i].no_child_nodes,
nodes[j][i].no_broadcasts, nodes[j][i].no_ACKs, nodes[j][i].node_depth);

/*                retvalue = sprintf(msg,
"%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", nodes[j][i].NID,
nodes[j][i].CID,
                        nodes[j][i].CH_NID, nodes[j][i].parent_CH_NID,
nodes[j][i].tree_depth,
                        nodes[j][i].no_child_nodes,
nodes[j][i].no_broadcasts, nodes[j][i].no_ACKs, nodes[j][i].node_depth,
                        nodes[j][i].in_event, nodes[j][i].know_event);
*/
                fputs(msg, nodefd); //write to file
            }
        }
    }
    fclose(nodefd); //close file

    if(pnt_console == 1)    //if needs to print to console
    {
        for (i = 0; i < NODESY ; i++)
```

```
        {
            for(j = 0 ; j < NODESX ; j++)
            {
                if(nodes[j][i].NID == 0)
                    printf(". ");   //No node in this location
                else {
                    printf("%c",
CID_to_symbol_mapping(nodes[j][i].CID));

                    if(nodes[j][i].NID == nodes[j][i].CH_NID)
                        printf(".");
                    else
                        printf(" ");
                }
            }
            printf("\n");
        }
    }
}


/*-------------------------------------------------------------------------*
* Following function add a node to the collision list if its is in the*
* communication range of two broadcasting nodes    the same time.     *
* nid1 - NID of the first node                                           *
* nid2 - NID of the second node                                          *
*-------------------------------------------------------------------------*/
void mark_collision_region(uint nid1, uint nid2)
{
    float ch_distance, distance1, distance2;
    int x1, y1, x2, y2, minx, maxx, miny, maxy, l, k, i;
    region my_region;

    //Reset the list if last collision is not related
    if(last_collision_set[1] != nid1)
        no_collision_nodes = 0;

   //Set the last nodes related to the collision
    last_collision_set[0] = nid1;
    last_collision_set[1] = nid2;

    x1 = (nid1 - 1) % NODESX;//X, Y coordinates of node related to event
    y1 = (nid1 - 1) / NODESX;
    x2 = (nid2 - 1) % NODESX;//X, Y coordinates of node related to event
    y2 = (nid2 - 1) / NODESX;

    ch_distance = sqrt((x1 - x2)*(x1 - x2)*(GRIDX * GRIDX) + (y1 -
y2)*(y1 - y2)*(GRIDY)*(GRIDY));
    if(ch_distance <= (2 * R))             //if within each others range
    {
        my_region = get_node_region(x1, y1, R);     //get my region
        minx = my_region.minx;
        miny = my_region.miny;
        maxx = my_region.maxx;
        maxy = my_region.maxy;

        for(l = miny; l <= maxy; l++)
        {
            for (k = minx; k <= maxx; k++)
            {
                distance1 = sqrt((k - x1)*(k - x1)*(GRIDX * GRIDX) + (l
- y1)*(l - y1)*(GRIDY * GRIDY));
                if((distance1 <= R) && (nodes[k][l].NID != 0))
                {
                    distance2 = sqrt((k - x2)*(k - x2)*(GRIDX * GRIDX) +
(l - y2)*(l - y2)*(GRIDY * GRIDY));
                    if((distance2 <= R) && (nodes[k][l].NID != 0))
                    {
                        //Check for overflows. If needed to change
                        //set the value in header file
                        if(no_collision_nodes < NO_COLLISION_NODES)
                        {
                            for(i = 0; i < no_collision_nodes; i++)
//Don't put the same node again & again
                            {
                                if(collision_nodes[i] ==
nodes[k][l].NID)
                                    break;
                            }
                            //if no match found add
                            if(i == no_collision_nodes)
                            {
                                //Add node to the collision region
                                collision_nodes[no_collision_nodes] =
nodes[k][l].NID;
                            //Increment no of nodes in collision region
                                no_collision_nodes++;
                            }
                        }
                        else
                        {
                            printf("No of collision nodes
overflow...\n");
                            exit(0);
                        }
                    }
                }
            }
        }
    }
}


/*-------------------------------------------------------------------------*
* Following function checks whether a given node is in                    *
* the collision range                                                     *
* nid - NID of the node                                                   *
* Return - 1 if in the collision region & 0 if not                        *
```

```
*-------------------------------------------------------------------------*/
char is_in_collision(uint nid)
{
    int i;

    //Check to see whether node is in collsion list
    for(i = 0; i < no_collision_nodes; i++)
    {
        if(collision_nodes[i] == nid)       //if so break
            break;
    }
    if(i != no_collision_nodes)
        return 1;                  //if in collision region
    else
        return 0;
}


/*-------------------------------------------------------------------------*
* Follwoing function returns the NID of the next hop                       *
* to forward the message                                                   *
* If it is for the current cluster same NID is returned                    *
* If its for a neighboring cluster its NID is returned                     *
* Otherwice the NID of the next hop is returned based                      *
* on the entries in the                                                    *
* routing table.                                                           *
* If no suitable next hop can't be found 0 is returned                     *
* dest_add  - Destination hierarchical address                             *
* current_NID - NID of the node trying to determine next hop               *
* sendder_NID - NID of the node that forwarded the message                 *
*-------------------------------------------------------------------------*/
int next_hop(Hie_CID dest_add, int current_nid, int sender_nid)
{
    int x, y, i, j;
    //Hold hop count for each router table entry
    nei_status neighbors[MAX_ROUTES];
    int no_routing_entries;
    uchar result;

    x = (current_nid - 1) % NODESX;
    y = (current_nid - 1) / NODESX;

    no_routing_entries = nodes[x][y].no_routing_entries;
    //For each entry in the routing table
    for(i = 0; i < no_routing_entries; i++)
    {
        //If route is invalid discard
        if((nodes[x][y].routing_table[i].valid & 1) == 0)
            neighbors[i].hops = 255; //Set as unreachable
       //Check whether the destination is my address or neighbor address
        else if((nodes[x][y].routing_table[i].H_CID.id[0] ==
dest_add.id[0]) //neighbor is destination
            && (nodes[x][y].routing_table[i].H_CID.id[1] ==
dest_add.id[1])
            && (nodes[x][y].routing_table[i].H_CID.id[2] ==
dest_add.id[2])
            && (nodes[x][y].routing_table[i].H_CID.id[3] ==
dest_add.id[3]))
            return nodes[x][y].routing_table[i].learn_from;
        else if(nodes[x][y].routing_table[i].hops == 0) //Skip my entry
            neighbors[i].hops = 255;       //Set as unreachable
        else
        {
            result = hop_distance(nodes[x][y].routing_table[i].H_CID,
dest_add);
            if(result == 0)
                neighbors[i].hops = 255;   //Set as unreachable
            else
            {
                neighbors[i].hops = result +
nodes[x][y].routing_table[i].hops;
                neighbors[i].nei_NID =
nodes[x][y].routing_table[i].learn_from;
            }
        }
    }

    bubble_sort(neighbors, no_routing_entries);//Sort based on hop count

    if(neighbors[0].hops == 255)    //No matching next hop found
        return 0;
    else
    {
        //If 2 or best entries skip parent entry
        if((neighbors[0].hops == neighbors[1].hops) &&
(neighbors[0].nei_NID != neighbors[1].nei_NID)
            && (neighbors[0].nei_NID == nodes[x][y].parent_CH_NID)
&& (neighbors[1].nei_NID != sender_nid))
        {
            return neighbors[1].nei_NID;
        }
        //If next hop is not same as the sender
        else if (neighbors[0].nei_NID != sender_nid)
            return neighbors[0].nei_NID;
        else    //If so pick the next best node
        {
           //Find a node which is not energy constrained
            for(j = 1 ; j < no_routing_entries; j++)
            {
                if((neighbors[j].hops < 255) && (neighbors[j].nei_NID !=
sender_nid)
                   && (neighbors[j].nei_NID != current_nid)) //If found stop
                     return neighbors[j].nei_NID;
            }
            if(j == no_routing_entries)    //No possible next node
                 return 0;
        }
    }
```

```
    return 0;
}


/*---------------------------------------------------------------------*
 * Following function add us the total energy remaining in the network*
 * return total energy                                                 *
 *---------------------------------------------------------------------*/
double total_energy()
{
    int i, j;
    double energy = 0.00;

    for(i = 0 ; i < NODESX; i++)
    {
        for(j = 0 ; j < NODESY; j++)
        {
            if(nodes[i][j].NID != 0)
                energy += (double)nodes[i][j].energy;
        }
    }
    return energy;
}


/*---------------------------------------------------------------------*
 * Which node died first, root node or a node along the circular link *
 *---------------------------------------------------------------------*/
void who_died(uchar d)
{
    int x, y;

    for(x = 0 ; x < NODESX; x++)
    {
        for(y = 0 ; y < NODESY; y++)
        {
            if(nodes[x][y].node_dead == 1)
            {
                if((nodes[x][y].tree_depth == d) ||
(nodes[x][y].tree_depth == (d + 1)))
                    printf("0\n");
                else
                    printf("1\n");
            }
        }
    }
}
```

REFERENCES

[1] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," *Journal of Ad Hoc Networks*, vol.3, 2005, pp. 325-349.

[2] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Journal of Ad Hoc Networks (Elsevier)*, vol. 2, no. 4, Oct. 2004, pp. 351-367.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications Magazine*, vol. 40, no. 8, Aug. 2002, pp. 102-114.

[4] A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh, "Max-Min d-cluster formation in wireless ad-hoc networks," *In Proc. Conference on Computer Communications (IEEE INFOCOM 2000)*, Tel Aviv, Mar. 2000. pp. 32-41.

[5] H. M. N. D. Bandara and A. P. Jayasumana, "An enhanced top-down cluster and cluster tree formation algorithm for wireless sensor networks," *In Proc. 2nd International Conference on Industrial and Information Systems (ICIIS 2007)*, Sri Lanka, Aug. 2007, pp. 37-42.

[6] H. M. N. D. Bandara, A. P. Jayasumana, T. H. Illangasekare, and Qi Han, "A wireless sensor network based system for underground chemical plume tracking," *ISTec Student Research Poster Contest - 2008*, Colorado State University, Fort Collins, CO, Feb. 2008, Available: http://hdl.handle.net/10217/1550

[7] H. M. N. D. Bandara, A. P. Jayasumana, and I. Ray, "Key pre-distribution based secure backbone design for wireless sensor networks", *In proc. 3rd International Workshop on Practical Issues in Building Sensor Network Applications*, Montreal, Canada, Oct. 2008, *To be published*.

[8] H. M. N. D. Bandara, A. P. Jayasumana, and T. H. Illangasekare, "Cluster tree based self organization of virtual sensor networks," *In Proc. Wireless Mesh and Sensor Networks: Paving the Way to the Future or yet Another...?*, New Orlens, Nov. 2008, *To be published*.

[9] S. Bandyopadhyay and E. J. Coyle, "An energy efficient hierarchical clustering algorithm for wireless sensor networks," *In Proc. 22nd Conference on Computer Communications (IEEE INFOCOM 2003)*, vol. 3, San Francisco, Mar.-Apr. 2003.

[10] T. Banka, G. Tandon, and A. P. Jayasumana, "Zonal rumor routing for wireless sensor networks," *In Proc. International Conference on Information Technology: Coding and Computing*, vol. 02, 2005, pp. 562-567.

[11] K. Barnhart, "Creation of synthetic data," Center for Experimental Study of Subsurface Environmental Processes (CESEP), Colorado School of Mines, Golden, CO 80401, *Unpublished*.

[12] S. Basagni, "Distributed clustering for ad-hoc networks," *In Proc. International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'99)*, Australia, Jun. 1999, pp. 310-315.

[13] R. T. Boute, "The Euclidean definition of the functions div and mod," *ACM Transactions on Programming Languages and Systems*, vol. 14, no. 2, Apr. 1992, pp. 127-144.

[14] D. Braginsky and D. Estrin, "Rumor routing algorithm for sensor networks," *In Proc. 1st ACM International Workshop on Wireless Sensor Networks and Applications*, Atlanta, Sep. 2002, pp. 22-31.

[15] S. A. Camtepe and B. Yener, "Key distribution mechanisms for wireless sensor networks: a survey," *Technical Report TR-05-07*, Dept. CS, Rensselaer Polytechnic Institute, Mar. 2003.

[16] D. Chakrabarti, S. Maitra, and B. Roy, "A key pre-distribution scheme for wireless sensor networks: merging blocks in combinatorial design," *International Journal of Information Security*, vol. 5, no. 2, Apr. 2006, pp. 105-114.

[17] H. Chan and A. Perrig, “ACE: An emergent algorithm for highly uniform cluster formation,” *In Proc. 1st European Workshop on Wireless Sensor Networks*, Germany, Jan. 2004, pp. 154-171.

[18] H. Chan, A. Perrig, and D. Song, “Key distribution techniques for sensor networks,” *Wireless sensor networks*, Kluwer Academic Publishers, Norwell, MA, 2004, pp. 277-303.

[19] H. Chan, A. Perrig, and D. Song, “Random key predistribution schemes for sensor networks,” Symposium on Security and Privacy 2003, May 2003, pp. 197-213.

[20] K. S. Chan, H. Pishro-Nik, and F. Fekri, “Analysis of hierarchical algorithms for wireless sensor network routing protocols,” *In Proc. IEEE Wireless Communications and Networking Conference*, vol. 3, New Orleans, Mar. 2005, pp. 1830-1835.

[21] G. Chen and I. Stojmenovic, “Clustering and routing in mobile wireless networks,” *Technical report, SITE*, University of Ottawa, 1999.

[22] Crossbow Technology, “Crossbow MPR/MIB user’s manual,” Rev. A, June 2007.

[23] Crossbow Technology, “TELOSB mote platform,” 6020-0094-02 Rev A.

[24] Decagon Devices, Inc., “5TE water content, EC and temperature sensors,” ver. 1.

[25] M. Demirbas, A. Arora, V. Mittal, and V. Kulathumani, “A fault-local self-stabilizing clustering service for wireless ad-hoc networks,” *IEEE Trans. Parallel and Distributes Systems*, vol. 17, no. 9, Sep. 2006.

[26] W. Du, J. Deng, Y. S. Han, S. Chen, and P. K. Varshney, “A key management scheme for wireless sensor networks using deployment knowledge,” *IEEE INFOCOM 2004*, vol. 1, Mar. 2004, pp. 586-597.

[27] X. Du, Y. Xiao, S. Ci, M. Cuizani, and H. Chen, “A routing driven key management scheme for heterogeneous sensor networks,” *In Proc. IEEE International Conference on Communications*, June 2007, pp. 3407-3412.

[28] A. Durresi and V. Paruchuri, "Adaptive clustering protocol for sensor networks," *IEEE Aerospace Conference 2005*, Mar. 2005, pp. 1-8.

[29] e-Sense project, Available: www.ist-e-sense.org

[30] J. Elson and D. Estrin, "Sensor networks: a bridge to the physical world," in *Wireless Sensor Networks*, C. S. Raghavendra, K. M. Sivalingam, and T. Znati, eds., Kluwer Academic Publishers, Norwell, MA, 2004, pp. 3–20.

[31] L. Eschenauer and V.D. Gligor, "A key-management scheme for distributed sensor networks," *In Proc. 9th ACM Conference on Computer and Communications Security*, Nov. 2002, pp. 41-47.

[32] Harvard Sensor Network Testbed, Available: http://motelab.eecs.harvard.edu

[33] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan, "An application-specific protocol architecture for wireless microsensor networks," *IEEE Trans. Wireless Communications*, vol. 1, no. 4, Oct. 2002.

[34] M. M. Holland, R. G. Aures, and W. B. Heinzelman, "Experimental investigation of radio performance in wireless sensor networks," *IEEE Workshop on Wireless Mesh Networks*, Sep. 2006, pp. 140-150.

[35] O. Hussein and T. Saadawi, "Ant routing algorithm for mobile ad-hoc networks (ARAMA)," *In Proc. IEEE International Conference on Performance, Computing, and Communications*, Apr. 2003, pp 281-290.

[36] T. T. Huynh and C. S. Hong, "A novel multi-layer architecture for wireless sensor networks," *In Proc. 7th International Conference on Advanced Communication Technology*, vol. 2, 2005, pp. 1143-1146.

[37] J. Ibriq and I. Mahgoub, "A hierarchical key establishment scheme for wireless sensor networks," *In Proc. 21st International Conference on Advanced Networking and Applications*, 2007, pp. 210–219.

[38] IEEE Computer Society, "IEEE.802.15.4: Wireless medium access control and physical layer specifications for low-rate wireless personal area networks," Sep. 2006.

[39] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," *In Proc. 6th Annual International Conference on Mobile Computing and Networking*, Aug. 2000, Boston, pp. 56–67.

[40] A. P Jayasumana, Q. Han, and T. Illangasekare, "Virtual sensor networks – a resource efficient approach for concurrent applications," *In Proc. 4th International Conference on Information Technology*, Las Vegas, Apr. 2007.

[41] J. Kulik, W. Rabiner, and Hari Balakrishnan, "Adaptive protocols for information dissemination in wireless sensor networks," *In Proc. 5th ACM/IEEE International Conference on Mobile Computing and Networking*, Seattle, Aug. 1999.

[42] J. Lee and D. R. Stinson, "A combinatorial approach to key predistribution for distributed sensor networks," *In Proc. Wireless Communications and Networking*, vol. 2, Mar. 2005, pp. 1200-1205.

[43] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," *In Proc. 1st international conference on Embedded Networked Sensor Systems*, Los Angeles, Nov. 2003, pp. 126-137.

[44] C. R. Lin and M. Gerla, "Adaptive clustering for mobile wireless networks," *IEEE Journal of Selected Areas in Communications*, vol. 15, no. 7, Sep. 1997, pp. 1265-1275.

[45] D. Lymberopoulos, Q. Lindsey, and A. Savvides, "An empirical analysis of radio signal strength variability in IEEE 802.15.4 networks using monopole antennas," *ENALAB Technical Report* 050501.

[46] M. Maeda and Ed Callaway, "Cluster tree protocol (ver. 0.6)", Apr. 2001, Available: http://www.ieee802.org/15/pub/2001/May01/01189r0P802-15_TG4-Cluster-Tree-Network.pdf

[47] A, Manjeshwar and D. P. Agrawal, "TEEN: a routing protocol for enhanced efficiency in wireless sensor networks," *In proc. 15th International Parallel and Distributed Processing Symposium*, Apr. 2001, San Francisco, pp. 2009-2015.

[48] G. Mao, B. Fidan, and B. D. O. Anderson, "Wireless sensor network localization techniques," *International Joirnal of Computer and Telecommunications Networking*, vol. 51 , no. 10 , July 2007, pp. 2529-2553.

[49] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, June 2004, pp. 53–57.

[50] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," *In Proc. 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, Apr. 2005, pp. 364-369.

[51] S. Rao, "Estimating the ZigBee transmission-range ISM band," *EDN*, May 2007, pp. 67-72.

[52] C. Schurgers and M. B. Srivastava, "Energy efficient routing in wireless sensor networks," *In Proc. Military Communications Conference 2001*, vol. 1, Virginia, Oct. 2001, pp. 357-361.

[53] M. Shehab, E. Bertino, and A. Ghafoor, "Efficient hierarchical key generation and key diffusion for sensor networks," *In Proc. Sensor and Ad Hoc Communications and Networks*, Sep. 2005, pp. 76-84.

[54] K. Simonova, A. C. H. Ling, and X. S. Wang, "Location-aware key predistribution scheme for wide area wireless sensor networks," *Security of ad hoc and Sensor Networks*, Virginia, Oct. 2003, pp. 157-168.

[55] Smart dust, Available: http://robotics.eecs.berkeley.edu/~pister/SmartDust

[56] H. J. Smith, Modular inverse, Nov. 2007, Available: http://www.geocities.com/hjsmithh/Numbers/InvMod.html

[57] K. Srinivasan and P. Levis, "RSSI is under appreciated," *In Proc. 3rd Workshop on Embedded Networked Sensors*, May 2006.

[58] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Communications of the ACM*, vol. 47 , no. 6, June 2004, pp. 34-40.

[59] Texas Instruments, "2.4GHz IEEE 802.15.4/ZigBee-Ready RF transceiver," Rev. B," Mar. 2007.

[60] H. Tian, H. Shen, and T. Matsuzawa, "Random walk routing for wireless sensor networks," *In Proc. 6th International Conference on Parallel and Distributed Computing Applications and Technologies*, 2005, pp. 196-200.

[61] TinyOS, Availiable: http://www.tinyos.net/

[62] U-City project, Available: http://ucta.or.kr/en/ucity/concept.php

[63] N. Vlajic and D. Xia, "Wireless sensor networks: to cluster or not to cluster?," *In Proc. International Symposium on World of Wireless, Mobile and Multimedia Networks*, June 2006.

[64] X. Wang and T. Berger, "Self-organizing redundancy-cellular architecture for wireless sensor networks," *In Proc. Wireless Communications and Networking Conference*, New Orleans, Mar. 2005, pp. 1945-1951.

[65] J. E. Wieselthier, G. D. Nguyen, and A. Ephremides, "Energy-efficient broadcast and multicast trees in wireless networks," *Mobile Networks and Applications*, vol. 7, no. 6, Dec. 2002, pp. 481-492.

[66] L. Ying and Y. Haibin, "Energy adaptive cluster-head selection for wireless sensor Networks," *In Proc. 6th International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT 05)*, China, Dec. 2005, pp. 634-638.

[67] O. Younis and S. Fahmy, "HEED: a hybrid, energy-efficient, distributed clustering approach for ad-hoc sensor networks," *IEEE Trans. Mobile Computing*, vol. 3, no. 4, Oct.-Dec. 2004, pp. 366-379.

[68] O. Younis, M Krunz, and S. Ramasubramanian, “Node clustering in wireless sensor networks: recent developments and deployment challenges,” *IEEE Network*, vol. 20, no. 3, May-June 2006, pp. 20-25.

[69] W. Zhang and G. Cao, “Optimizing tree reconfiguration for mobile target tracking in sensor networks,” *In Proc. Conference on Computer Communications (IEEE INFOCOM 2004)*, vol. 4, Mar. 2004, pp. 2434–2445.

ABBREVIATIONS

ACE	Algorithm for Cluster Establishment
ACK	Acknowledgment
ACP	Adaptive Clustering Protocol
CCH	Candidate Cluster Head
CH	Cluster Head
CID	Cluster IDentifier
CNRL	Computer Networking Research Laboratory
DCA	Distributed Clustering Algorithm
DKR	Deployment Knowledge based Random key selection
DNS	Domain Name System
ECC	Elliptic Curve Cryptography
FLOC	Fast, LOcal Clustering service
GPS	Global Positioning System
GTC	Generic Top-down Cluster and cluster tree formation
HEED	Hybrid Energy-Efficient Distributed clustering
HHC	Hop-ahead Hierarchical Clustering
i-band	inner-band
ID	IDentifier
IEEE	Institute of Electrical and Electronics Engineers
LEACH	Low Energy Adaptive Clustering Hierarchy

LEACH-C	Low Energy Adaptive Clustering Hierarchy – Centralized
LSD	Least Significant Digit
MAC	Media Access Control
NID	Node Identifier
o-band	outer-band
PAN	Personal Area Network
PAN ID	Personal Area Network IDentifier
PHC	Probabilistic Hierarchical Clustering
PMP	Plume Modeling and Prediction system
RBMCD	Random Block Merging in Combinatorial Design
RIP	Routing Information Protocol
RSSI	Receiver Signal Strength Indicator
RHHC	Receiver signal strength indicator based Hop-ahead Hierarchical Clustering
RSHC	Receiver signal strength indicator based Simple Hierarchical Clustering
SHC	Simple Hierarchical Clustering
SPIN	Sensor Protocols for Information via Negotiation
STD	STandard Deviation
TEEN	Threshold sensitive Energy Efficient sensor Network
TTL	Time To Live
VPN	Virtual Private Network
VSN	Virtual Sensor Network
WSN	Wireless Sensor Network
ZRR	Zonal Rumor Routing