A
Dissertation
On

SIMULATION OF COVERAGE CONFIGURATION PROTOCOL FOR ENERGY EFFICIENT SENSOR NETWORK

Submitted in Partial fulfillment of the requirement
for the award of Degree of

MASTER OF ENGINEERING
(Electronics & Communication Engineering**)**

Submitted By

Puja Krishna
College Roll No: 06/EC/05
University Roll No. 2804

Under the Guidance of:
Mrs. S. Indu
Dept. of Electronics & Communication
Delhi College of Engineering, Delhi

DEPARTMENT OF ELECTRONICS & COMMUNICATION
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY
2005-2007

CERTIFICATE

This is to certify that the work contained in this dissertation entitled **"Simulation of Coverage Configuration Protocol for Energy Efficient Sensor Network"** submitted by **Puja Krishna** in the requirement for the partial fulfillment for the award of the degree of Master of Engineering in Electronics & Communication, Delhi College of Engineering is an account of her work carried out under my guidance and supervision in the academic year 2006-2007.

The work embodies in this dissertation has not been submitted for the award of any other degree to the best of my knowledge.

Approved by*:*	Guided by:
Prof. Asok Bhattacharyya	**Mrs. S. Indu**
H.O.D (E&C)	**Lecturer**
Electronics & Communication	**Electronics & Communication**

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extend my heartiest felt gratitude to everybody who helped me throughout the course of this project.

It is distinct pleasure to express my deep sense of gratitude and indebtedness to my learned supervisor Mrs. S. Indu for her invaluable guidance, encouragement and patient reviews. With her continuous inspiration only, it becomes possible to complete this dissertation. She kept on boosting me with time, to put an extra ounce of effort to realize this work.

I am also thankful to Prof. Asok Bhattacharyya, for his valuable suggestions and constant support.

I would also like to take this opportunity to present my sincere regards to all the faculty members of the Department for their support and encouragement.

I am grateful to my parents for their moral support all the time; they have been always around to cheer me up, in the odd times of this work. I am also thankful to my classmates for their unconditional support and motivation during this work.

Puja Krishna
M.E. (Electronics & Communication)
College Roll No. 06/EC/05
University Roll No. 2804
Department of Electronics & Communication
Delhi College of Engineering, Delhi-110042

ABSTRACT

This thesis gives an insight into the working of CCP (Coverage Configuration Protocol) for energy efficient sensor networks, which provides sensing coverage with the adequate network connectivity. In my minor thesis, I simulated the energy efficient S-MAC protocol for Sensor networks, which provides only network connectivity, but not the sensing coverage.

In sensor networks, there are two critical requirements;

- Sufficient sensing coverage,
- Sufficient network connectivity.

Sensing is one of the responsibilities of a sensor network. To operate successfully, a sensor network must provide satisfactorily sensing coverage and network connectivity. By satisfactorily network connectivity, nodes can communicate for data fusion and reporting to base station. Sensing coverage characterizes the monitoring quality provided by a sensor network in a designated region. The coverage requirement for a sensor network depends on the different applications and also on the number of faults that must be tolerated.

Without sufficient sensing coverage, the network cannot monitor the environment with sufficient accuracy or may even suffer from “sensing voids” locations where no sensing can occur. Without sufficient connectivity, nodes may not be able to coordinate effectively or transmit data back to base station. The combination of coverage and connectivity is a special requirement introduced by sensor networks that integrate multihop wireless communication and sensing capabilities into a single platform.

Hence, this thesis covers the entire connectivity as well as the sensing coverage of sensor network by simulating the efficient energy conservation protocol called CCP (Coverage Configuration Protocol). CCP selects a small number of active nodes to maintain the

sensing coverage and connectivity of a sensor network while scheduling other nodes to sleep. CCP can dynamically configure a sensor network to different degrees of coverage requested by applications. This flexibility allows the network to self configure for a wide range of applications and environments with diverse or changing coverage requirements. Through geometric analysis and simulation results, it can be showed that CCP can maintain robust sensing coverage and network connectivity when communication range is at least twice sensing range.

The problem of coverage configuration can be formulated as follows. Given a convex region A, and a degree of coverage K specified by the application, the number of sleeping nodes must be maximized such that the remaining active nodes must provide K-coverage to region A.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION TO SENSOR NETWORKS & COVERAGE CONFIGURATION PROTOCOL

INTRODUCTION TO SENSOR NETWORKS & COVERAGE CONFIGURATION PROTOCOL

1.1 OVERVIEW OF WIRELESS SENSOR NETWORKS

Efficient design and implementation of wireless sensor networks has become a hot area of research in recent years, due to the vast potential of sensor networks to enable applications that connect the physical world to the virtual world. Wide range of potential applications of wireless sensor networks includes environment monitoring, smart spaces, medical systems and robotic exploration.

Such a network normally consists of a large number of distributed nodes that organize themselves into a multi-hop wireless network. Each node has one or more sensors, embedded processors and low-power radios, and is normally battery operated. Typically, these nodes coordinate to perform a common task.

Wireless sensor network consists of a large number of distributed nodes that organize themselves into a multi-hop wireless network. These nodes are comprised of transducers (sensor or actuator), communication circuitry and behavior logic. These nodes will be embedded in ceiling tiles and will locate things, sense danger and control the environment with minimal human effort. The behavior and characteristics of wireless sensor networks (WSN) are very much different from other wireless networks.

Differences between different types of networks:-

Types	**Number of nodes**	**Range**	**Data rate**	**Mobility**
Cellular	Large	Long	Medium	High
WLAN	Small	Medium	High	Medium
Bluetooth	Small	Short	Medium	Low
WSN	Large	Very short	Low	Low

Types	**Power**	**Cost**	**Size**	**Redundancy**
Cellular	High	High	large	low
WLAN	Medium	Medium	Medium	low
Bluetooth	Low	Low	Small	low
WSN	Very low	Very low	Very small	high

Table 1.1: Differences between different types of networks

Unique Features of Sensor Networks:

It should be noted that sensor networks do share some commonalities with general ad hoc networks. Thus, protocol design for sensor networks must account for the properties of ad hoc networks, including the following:

• Lifetime constraints imposed by the limited energy supplies of the nodes in the network.
• Unreliable communication due to the wireless medium.
• Need for self-configuration, requiring little or no human intervention.

However, several unique features exist in wireless sensor networks that do not exist in general ad hoc networks. These features present new challenges and require modification of designs for traditional ad hoc networks:

• While traditional ad hoc networks consist of network sizes on the order of 10s, sensor networks are expected to scale to sizes of 1000s.
• Since nodes may be deployed in harsh environmental conditions, unexpected node failure may be common.
• Sensor nodes may be much smaller than nodes in traditional ad hoc networks (e.g., PDAs, laptop computers), with smaller batteries leading to shorter lifetimes, less computational power, and less memory.

• Additional services, such as location information, may be required in wireless sensor networks.

• While nodes in traditional ad hoc networks compete for resources such as bandwidth, nodes in a sensor network can be expected to behave more cooperatively, since they are trying to accomplish a similar universal goal, typically related to maintaining an application-level quality of service (QoS), or fidelity.

• Communication is typically data-centric rather than address-centric, meaning that routed data may be aggregated/compressed/prioritized/dropped depending on the description of the data.

• Communication in sensor networks typically takes place in the form of very short packets, meaning that the relative overhead imposed at the different network layers becomes much more important.

• Sensor networks often have a many-to-one traffic pattern, which leads to a "hot spot" problem. Incorporating these unique features of sensor networks into protocol design is important in order to efficiently utilize the limited resources of the network.

An important property of sensor networks is the need of the sensors to reliably disseminate the data to the sink or the base station within a time interval that allows the user or controller application to respond to the information in a timely manner, as out of date information is of no use and may lead to disastrous results.

Another important attribute is the scalability to the change in network size, node density and topology. Sensor networks are very dense as compared to mobile ad hoc and wired networks. This arises from the fact that the sensing range is lesser than the communication range and hence more nodes are needed to achieve sufficient sensing coverage. Sensor nodes are required to be resistant to failures and attacks.

Information routing is a very challenging task in Distributed Sensor Networks due to the inherent characteristics that distinguish these networks from other wireless or adhoc networks. The sensor nodes deployed in an adhoc manner need to be self-organizing as this kind of deployment requires system to form connections and cope with the resultant nodal distribution.

Another important design issue in sensor networks is that sensor networks are application specific. Hence, the application scenario demands the protocol design in a sensor network. Also, the data collected by sensor nodes is often redundant and needs to be exploited by routing protocols to improve energy and bandwidth utilization. The proposed routing protocols for sensor networks should consider all the above issues for it to be very efficient. The algorithms developed need to be very energy efficient, scalable and increase the life of the network in the process.
The multitudes of design challenges imposed on Sensor Networks tend to be quite complex and usually defy the analytical methods that are quite effective for traditional networks. At current stage of technology very few Sensor Networks have come into existence. Although there are many unsolved research problems in this domain, actual deployment and study is infeasible. The only practical alternate to study Sensor Networks is through simulation, which can provide better insight to behavior and performance of various algorithms and protocols.

1.2 THESIS OUTLINE

As wireless sensor networks continue to attract more attention, new ideas for applications are continually being developed, many of which involve consistent coverage with appropriate network connectivity of a given surveillance area. Several other protocols (*e.g.*, ASCENT [2]], SPAN [3], AFECA [4], and GAF [5]) aim to maintain network connectivity, but do not guarantee sensing coverage.

Recently, some protocols and architectures have been proposed to maintain adequate coverage quality with network connectivity [1] while minimizing the drain on the scarce energy resources of the sensor nodes.

An effective approach for energy conservation in wireless sensor networks is scheduling sleep intervals for extraneous nodes, while the remaining nodes stay active to provide continuous service. For the sensor network to operate successfully, the active nodes must maintain both sensing coverage and network connectivity. Furthermore, the network must be able to configure itself to any feasible degrees of coverage and connectivity in order to support different applications and environments with diverse requirements. This Thesis presents the design and analysis of novel protocol that can dynamically configure a network to achieve guaranteed degrees of coverage and connectivity, which differs, from existing connectivity or coverage maintenance protocols in several ways.

Coverage configuration is an important issue in wireless sensor networks (WSNs). Existing coverage configuration methods are generally based on the concept of physical coverage. That is, a point is covered if it is located within the sensing area of at least one sensor but Coverage Configuration depends on the information about the states of neighboring sensors and the intersection points of neighboring sensors within the sensing region of that sensor.

To provide such kinds of network's characteristics, Coverage configuration protocol (CCP) is introduced which preserves the network connectivity with providing the adequate coverage quality that is distinctly different from the previously proposed S-MAC protocol. Coverage Configuration Protocol (CCP) can provide different degrees of coverage requested by applications. This flexibility allows the network to self-configure for a wide range of applications and (possibly dynamic) environments.

This Thesis also provides the geometric analysis of the relationship between coverage and connectivity. This analysis yields key insights for treating coverage and connectivity in a unified framework: this is in sharp contrast to several existing approaches.

CHAPTER 2

COVERAGE CONFIGURATION PROTOCOL DESIGN

COVERAGE CONFIGURATION PROTOCOL DESIGN

An effective approach for energy conservation in wireless sensor networks is scheduling sleep intervals for extraneous nodes [12], while the remaining nodes stay active to provide continuous service. For the sensor network to operate successfully, the active nodes must maintain both sensing coverage [11] and network connectivity. Furthermore, the network must be able to configure itself to any feasible degrees of coverage and connectivity in order to support different applications and environments with diverse requirements.

This chapter presents the design and analysis of novel protocols that can dynamically configure a network to achieve guaranteed degrees of coverage and connectivity, which is different from existing connectivity, or coverage maintenance protocols in several ways:

1) Coverage Configuration Protocol (CCP) is a protocol that can provide different degrees of coverage requested by applications. This flexibility allows the network to self-configure for a wide range of applications and (possibly dynamic) environments.

2) Geometric analysis of the relationship between coverage and connectivity yields key insights for treating coverage and connectivity in a unified framework: this is in sharp contrast to several existing approaches.

2.1 PROBLEM FORMULATION

For design of CCP, a point p is assumed to be ***covered*** (monitored) by a node v if their Euclidian distance is less than the sensing range of v, R_s, i.e., $|P_v| < R_s$. Sensing circle C(v) of node v is defined as the boundary of v's coverage region. Any point p on the sensing circle C (v) (i.e., $|P_v| = R_s$) is not assumed to be covered by v. Based on the above coverage model, a convex region A (that contains at least one sensing circle) is defined as having a coverage degree of K (i.e., being K-covered) if every location inside A is covered by at least K nodes. Practically, a network with a higher degree of coverage can achieve higher sensing accuracy and be more robust against sensing failures. The coverage configuration problem [13] can be formulated as follows.

Given a convex coverage region A, and a coverage degree K specified by the application (either before or after deployment), the number of sleeping nodes must be maximize under the constraint that the remaining nodes must guarantee A is K covered. Despite its simplicity, this coverage model is applicable in a number of applications. For example, it fits well with the decision approach to distributed detection of selfish sensors. Therefore, the statistical nature of sensor network applications and the environments can be incorporated in the definition of sensing range.

In addition, it is assumed that any two nodes u and v can directly communicate with each other if their Euclidian distance is less than a communication range R_c, *i.e.*, $|uv| < R_c$. Given a coverage region A and a sensor coverage degree Ks, the goal of an integrated coverage and connectivity configuration is maximizing the number of nodes that are scheduled to sleep under the constraints that the remaining nodes must guarantee: 1) A is at least Ks–covered, and 2) all active nodes are connected.

2.2 RELATIONSHIP BETWEEN COVERAGE AND CONNECTIVITY

In this section, sufficient condition is first derived when coverage implies connectivity in a network. The relationship between the degree of coverage and connectivity is then quantified. The analysis presented in this section will serve as the foundation for an integrated solution to the problem of integrated coverage and connectivity configuration.

2.2.1 Sufficient Condition for 1-Coverage to Imply Connectivity

In this subsection, the relationship between 1-coverage and connectivity is analyzed in a network. It is noted that connectivity only requires that the location of any active node be within the communication range of one or more active nodes such that all active nodes can form a connected communication backbone, while coverage requires *all* locations in the coverage region be within the sensing range of at least one active node.

Intuitively, the relationship between connectivity and coverage depends on the ratio of the communication range to the sensing range. However, it is easily seen that a connected network may not guarantee its coverage regardless of the ranges. This is because coverage is concerned with whether any location is uncovered while connectivity only requires all locations of active nodes are connected. Hence, focus must be given on analyzing the condition for a covered network to guarantee connectivity in the rest of this section.

Define the graph G(V,E) to be the communication graph of a set of sensors, where each sensor in the set is represented by a node in V, and for any node x and y in V, the edge (x, y) Є E if and only if the Euclidean distance between x and y, |xy| < Rc. Nodes v and u are connected in G(V,E) if and only if a network path consisting of consecutive edges in E exists between node u and v.

Theorem 1: For a set of sensors that at least 1-cover a convex region A, the communication graph is connected if $R_c \geq 2R_s$.

Proof: For any two nodes, u and v in region A, let P_{uv} be the line segment joining them. Since region A is convex, P_{uv} remains entirely within A. Hence, any point on P_{uv} is at least 1-covered. Each point P on P_{uv} has a set of one or more closest sensors equidistant from P. A finite sequence $S_{uv} = s_1 \ldots s_n$ of closest sensor sets can be constructed for contiguous segments 1..n of P_{uv}, where a segment is defined by all points within it having the same set of closest sensors. S_{uv} starts with $s_1 = \{u\}$ and ends with $s_n = \{v\}$, with intervening sets possibly containing other sensors.

The distance from each point on the line segment P_{uv} to its closest sensor(s) is always less than R_s, as otherwise the path would go through regions that are not sensor-covered. Furthermore, if there were any two sensors x and y in any consecutive sets s_j and s_{j+1} in S_{uv}, x Є s_j and y Є s_{j+1}, such that $|xy| \geq 2R_s$, then the point p at the intersection of P_{uv} with the sensing circle of x is exactly R_s from x (and not covered by x from the definition of sensing circle) and according to the triangle inequality is at least R_s from y. However, since that point would then have x as one of its closest sensors, it would be at least Rs from *any* sensor and thus would not be sensor-covered. Therefore, the distance between every pair of sensors in consecutive sets in S_{uv} is less than $2R_s$, and is thus less than R_c, so an edge exists between them in the communication graph. Because each set in S_{uv} contains at least one sensor, thus a communication path can be constructed from u to v through each combination of node choices in the sets in S_{uv}. *i.e.*, the communication graph of sensors in region A is connected.

Therefore, Theorem 1 establishes a sufficient condition for a 1-covered network to guarantee 1-connectivity. Under the condition that Rc ≥ 2Rs, a sensor network only needs to be configured to guarantee coverage in order to satisfy both coverage and connectivity.

2.3 RELATIONSHIP BETWEEN THE DEGREE OF COVERAGE AND CONNECTIVITY

In previous section, it is proved that if a region is sensor covered, and then the sensors covering that region are connected as long as their communication range is no less than twice the sensing range. If the condition of $R_c \geq 2R_s$ is maintained, then the relationship between the degree of coverage and connectivity can be maintained. This result is important for applications that require degrees of coverage or connectivity greater than one. Boundary sensor is defined as a sensor whose sensing circle intersects with the boundary of the convex sensor deployment region A. Clearly all boundary sensors are located within R_s distance to the boundary of A. All the other sensors in region A are interior sensors.

Theorem 2: For a Ks-covered convex region A, it is possible to disconnect a boundary node from the rest of the nodes in the communication graph by removing Ks sensors if Rc ≥ 2Rs.

Proof: A sensor u is located at a corner (point q) of the rectangular sensor deployment region A that is Ks-covered as shown in Figure 2.1. Suppose point p is on the sensing circle of sensor u such that pq has a 45^o angle with the horizontal boundary of region A.

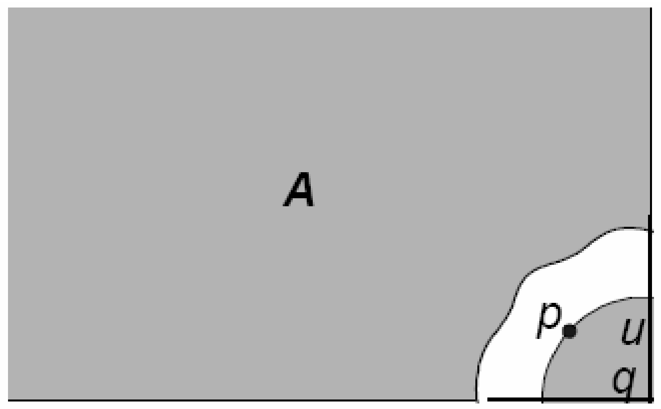


Figure 2.1: Removing Ks nodes disconnects a covered network

Suppose K_s coinciding sensors are located at point p. Clearly, these K_s sensors can K_s-cover the quarter circle of sensor u. In addition, it is assumed that there are no other sensors whose sensing circles intersect with sensing circle of u. Then removing these Ks coinciding sensors will create an uncovered region (*i.e.*, a sensing void) surrounding sensor u. Furthermore, when R_c is equal to $2R_s$, there is no sensor within the communication range of sensor u after the removal of these Ks sensors. *i.e.*, the communication graph is disconnected.

Theorem 3: A set of nodes that Ks-cover a convex region A forms a Ks connected communication graph if $Rc \geq 2Rs$.

Proof: Disconnecting the communication graph G of a set of sensors creates (at least) 3 disjoint sets of nodes, the set of nodes W that is removed, and two sets of nodes V1 and V2, such that there are no edges from any node in V1 to any node in V2 in G. By Theorem 1, if it is possible to draw a continuous path between two nodes so that every point on the path is sensor-covered, then there exists a communication path between those two nodes. Therefore, to disconnect the graph it is necessary to create a sensing void, so that it is impossible to draw a continuous covered path connecting a node in V1 to a node in V2. That is, as illustrated in Figure 2.2, the nodes of V1 may all lie in region S, the nodes in V2 may all lie in region Q, and a set of nodes W must be removed to make a region T that is 0-covered. The nodes that are removed may actually lie in the region labeled S or Q, but their removal leaves the 0-covered region labeled as T.

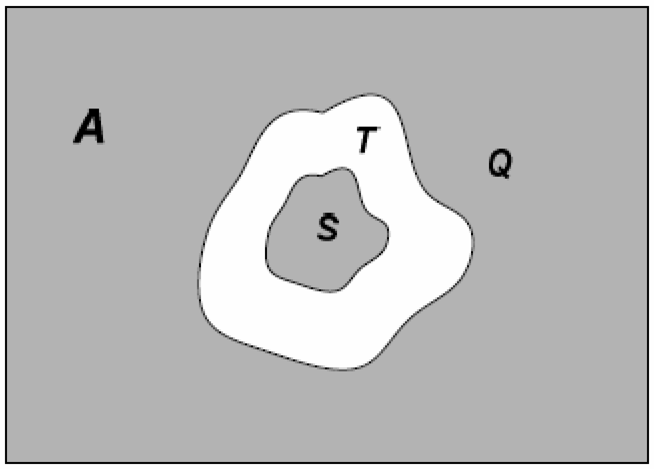


Figure 2.2: A disconnected network

To create a sensing void in an originally Ks-covered region A, it is necessary to remove at least Ks sensors. Thus, the network connectivity is at least Ks. By Theorem 2, removing Ks sensors could disconnect the communication graph. Therefore, the tight lower bound on the connectivity of communication graph is Ks.

Intuitively, the connectivity of the *boundary* sensors dominates the overall connectivity of the communication graph. However, in a large-scale sensor network, the *interior* sensors normally route more traffic and higher connectivity is needed for *interior* sensor to maintain the required throughput. Interior connectivity is defined as the number of sensors (either interior or boundary) that must be removed to disconnect any two *interior* sensors in the communication graph of the sensors.

Theorem 4: For a set of sensors that Ks-cover a convex region A, the *interior connectivity* is $2K_s$ if $R_c \geq 2R_s$.

Proof: Suppose u and v are two interior nodes and the removal of a set of nodes W disconnects node u and node v. In order for nodes v and u to be disconnected, there must be a "void" region that separates node v from node u. There are two cases, either this void is completely contained within the sensor deployment region, or the void merges with the boundary of the region.

Case 1: As illustrated in Figure 2.3, the void does not merge with the boundary. It will prove that one must remove at least $2K_s+1$ sensors in this case to create such a void. It is proved by contradiction.

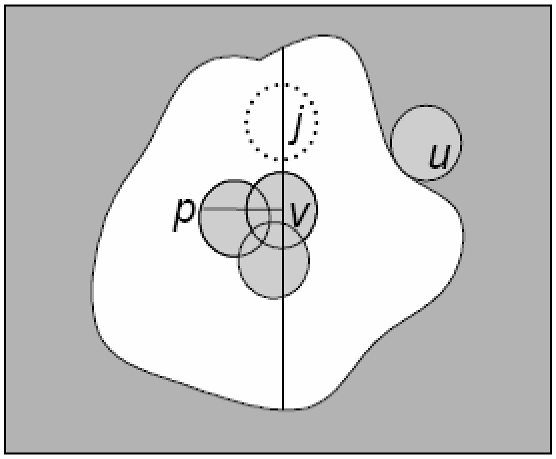


Figure 2.3: Case 1: The void does not merge with boundary

Suppose $|W| < 2K_s+1$. In this case, the void must completely surround a set of nodes including node v. Since node v remains active, the sensing void must be at a distance at least R_s from v. Now a line is drawn from v through a sensor node j in W. Line vj is to be defined as the direction referred to as 'vertical'. Now, there are at most 2Ks-1 remaining sensors (except sensor j) in W which are either on the line vj or to the left or the right of line vj. By the pigeonhole principle, there must be one side that has less than Ks nodes from the set W, which is defined to be the left side. A line is drawn straight left from v until it intersects the void region, and noted this point as P (note that P is covered by zero sensors.) Point P is at least R_s from node v, and is at least R_s from any point on or to the right of the vertical line. However, there are at most $K_s - 1$ nodes in the set W that are to the left of the line. This contradicts the assertion that P was originally K_s covered and the removal of the nodes of W leaves it 0-covered. Thus, |W| is at least $2K_s+1$.

Case 2: The void merges with the boundary of region A, as illustrated in Figure 2.4. In this case, the removal of a set of nodes W creates a void, which separates the nodes v and u, and this void merges with the boundary of the region A that is being sensed.

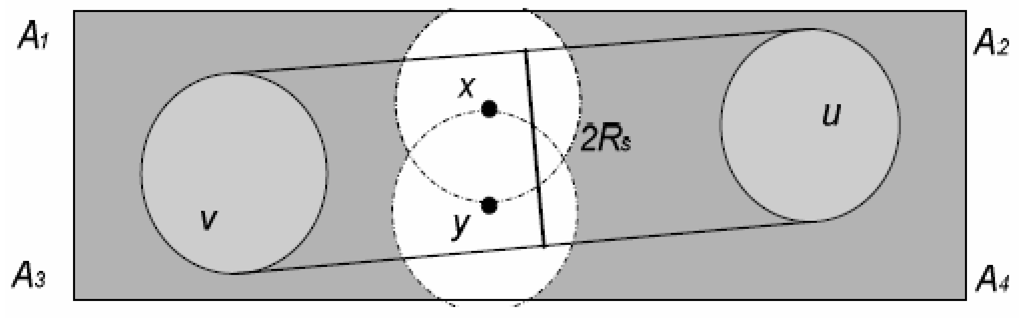


Figure 2.4: Case 2: The void merges with boundary

Since v is an interior node, all the points within a radius R_s from v are inside region A, and the same holds true for u. Furthermore, since the region A is convex, the line connecting any point v' within R_s from v and any point u' within Rs from u are inside the region A and must be intersected by the void, otherwise there will exist a continuous path (vv'u'u) from v to u, which remains entirely within sensor covered region and defines a network path in the communication graph (from Theorem 1). Thus, the minimum width of the void that separates u from v is at least 2 R_s. Now any two points are considered in the void that are a distance of $2R_s$ apart. No sensor can simultaneously cover both points. This implies that at least $2K_s$ sensors were removed in the K_s-covered region A to create the void. This bound is proved tight by the following example. Suppose the K_s-covered region A is a rectangle $A_1A_2A_3A_4$ with width 2 R_s +r ($0 < r < R_s$). Two points x and y are located at perpendicular bisector of A_1A_2 and have distance $(R_s +r)/2 < R_s$ with A_1A_2 and A_3A_4respectively, as shown in Figure 2.4. Suppose there are K_s sensors (shown as dotted circles) located at point x and y respectively. W is composed of these $2K_s$ sensors. It is assumed the sensors (not shown in the figure) whose sensing circles intersect the $2K_s$ sensors in W are far enough from point x and y such that the void created by the removal of W intersects both A_1A_2 and A_3A_4. It is clear that the void disconnects the nodes on left side from the nodes on right side in communication graph.

From the proof of case 1 and case 2, for a set of sensors that K_s cover a convex region, it has shown that the tight lower bound on the *interior connectivity* is $2K_s$.

From the Theorems 3 and 4, one can draw the conclusion that the boundary nodes that are located within R_s distance to the boundary of the coverage region are K_s connected; to the rest of the network, the interior connectivity is $2K_s$.

2.4 COVERAGE AND CONNECTIVITY CONFIGURATION WHEN Rc ≥ 2Rs

Based on Theorems 1, 2 and 3, the integrated coverage and connectivity configuration problem [13] can be handled by a coverage configuration protocol if $R_c \geq 2R_s$. In this section, a new coverage configuration protocol is presented, called CCP that uses this principle. CCP has several key benefits.

1) CCP can configure a network to the specific coverage degree requested by the application.
2) It is a decentralized protocol that only depends on local states of sensing neighbors.
3) Geometric analysis has proven that CCP can provide guaranteed degrees of coverage.

2.4.1 Ks-Coverage Eligibility Algorithm

Each node executes an eligibility algorithm to determine whether it is necessary to become active. Given a requested coverage degree K_s, a node v is ineligible if every location within its coverage range is already Ks-covered by other active nodes in its neighborhood. For example, assume the nodes covering the shaded circles in Figure 5 are active, the node with the bold sensing circle is ineligible for $K_s=1$, but eligible for $K_s > 1$. Before presenting the eligibility algorithm, the following notations are defined.

1) The *sensing region* of node v is the region inside its sensing circle, *i.e.*, a point P is in v's sensing region if and only if $|pv| < R_s$.

2) A point $P \in A$ is called an *intersection point* between nodes u and v, *i.e.*, $P \in u \cap v$, if P is an intersection point of the sensing circles of u and v.

3) A point P on the boundary of the coverage region A is called an *intersection point* between node v and A, i.e., $P \in v \cap A$ if $|pv|= R_s$.

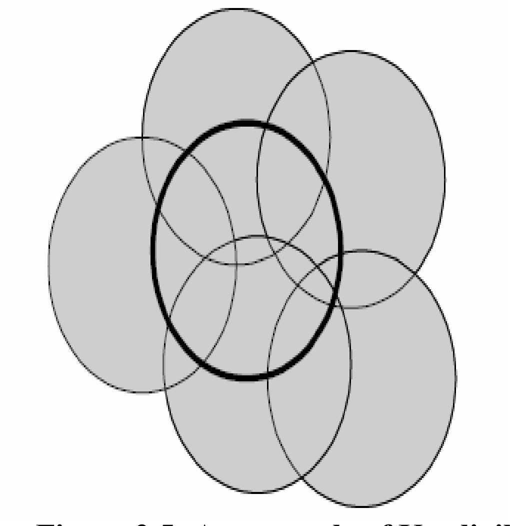


Figure 2.5: An example of Ks-eligibility

Theorem 5: A convex region A is K_s-covered by a set of sensors S if

1) there exist in region A intersection points between sensors or between sensors and A's boundary;

2) all intersection points between any sensors are at least K_s-covered; and

3) all intersections points between any sensor and A's boundary are at least K_s covered.

Proof: It is proved by contradiction. Let P be the point that has the lowest coverage degree k in region A and k < K_s. Furthermore, suppose there is no intersection point in A which is covered to a degree less than K_s. The set of sensing circles partition A into a collection of *coverage patches*, each of them is bounded by arcs of sensing circles and/or the boundary of A, and all points in each coverage patch have the same coverage degree. Suppose point P is located in coverage patch S. First, it is proved that the interior arc of any sensing circle cannot serve as the boundary of S. It is proved by contradiction. Assume there exists an interior arc (of sensing circle C(u)) serving as the boundary of S, crossing this arc (*i.e.* leaving the coverage region of sensor u) would reach an area that is lower covered than point P. This contradicts with the assumption that point P has the lowest coverage degree in region A. The following two cases are now considered:

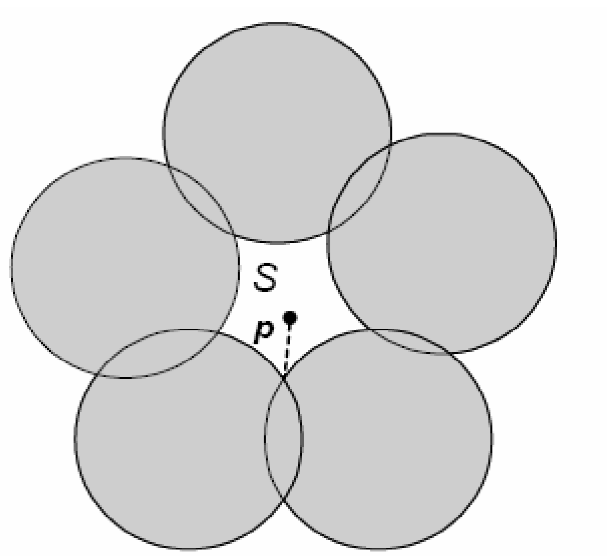


Figure 2.6: A coverage patch bounded by arcs of sensing Circles

1) The point P lies in a coverage region S whose boundary is only composed of exterior arcs of a collection of sensing circles (as Figure 2.6 illustrates). Furthermore, since the sensing circles themselves are outside the sensing range of the nodes that define them, the entire boundary of this coverage patch, including the intersection points of the sensing circles defining the boundary, has the same coverage degree as point P. This contradicts the

assertion that P is covered to a degree less than K_s and all intersection points have coverage degree at least K_s.

2) The point P lies in a coverage region S that is bounded by the exterior arcs of a collection of sensing circles and the boundary of A. As shown in Figure 2.7, point P is in a region bounded by the exterior arcs of sensor u, v, w, x and the boundary of region A. Similarly as case 1), the entire boundary of this coverage patch, *including the intersection points of sensors u, v, w, x and intersection points between sensors w, x and boundary of A,* has the same coverage degree as point P. This contradicts the assertion that P is covered to a degree less than K_s and all intersection points have coverage degree at least K_s.

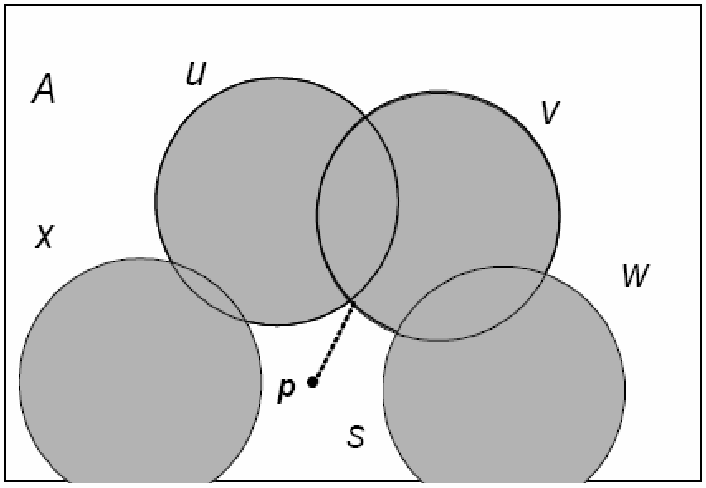


Figure 2.7: A coverage patch bounded by arcs of sensing circles and boundary of a coverage region

Clearly the point P cannot lie in a coverage patch that is bounded solely by the boundary of region A. Otherwise the region A has the same coverage as point P. This contradicts with the assumption that the region A is K_s covered. From the above discussion, the point P with lower coverage degree than K_s does not exist. Thus, the region A is K_s covered.

Theorem 5 allows us to transform the problem of determining the coverage degree of a region to the simpler problem of determining the coverage degrees of all the intersection points in the same region. A sensor is ineligible for turning active if all the intersection points inside its sensing circle are at least Ks-covered. To find all the intersection points inside its sensing circle, a sensor v needs to consider all the sensors in its sensing neighbor set, SN (v).

SN (v) includes all the active nodes that are within a distance of twice of the sensing range to v, i.e., SN (v) = {active node u | |uv|≤ 2Rs and u! =v}. If there is no intersection point inside the sensing circle of sensor v, v is ineligible when there are K_s or more sensors that are located at sensor v's position.

CCP maintains a table of known sensing neighbors based on the beacons (HELLO messages) that it receives from its communication neighbors. When $R_c \geq 2R_s$, the HELLO message from each node only needs to include its own location. When $R_c < 2R_s$, however, a node may not be aware of all sensing neighbors through such HELLO messages. Since some sensing neighbors may be "hidden" from a node, it might activate itself to cover a perceived sensing void that is actually covered by its hidden sensing neighbors. Thus, the number of active nodes would be higher than necessary in this case. To address this limitation, there must be some mechanism for a node to advertise its existence to the neighborhood of 2Rs range.

There are two approaches to make each node aware of its multihop neighbors. One is to broadcast HELLO messages in multiple hops by setting the TTL of each HELLO message. The other is to let each node include the locations of all known multi-hop neighbors in its HELLO messages. Specifically, each node may broadcast the locations and status of all active nodes within $2R_s/R_c$ hops. The second approach reduces the number of broadcasts and is adopted by CCP. Here, it should be noted that, in a network with random topology, such HELLO messages still could not guarantee the discovery of all nodes within a distance of $2R_s$. Since including multi-hop neighbors in the HELLO messages introduce much higher communication overhead compared to a one-hop approach in a dense network, there is a tradeoff between the beacon overhead and the number of active nodes maintained by CCP.

2.5 THE STATE TRANSITION OF CCP

In CCP, each node determines its eligibility using the K_s -coverage eligibility algorithm based on the information about its sensing neighbors, and may switch state dynamically when its eligibility changes. A node can be in one of three states: SLEEP, ACTIVE, and LISTEN, as illustrated in fig 2.8.

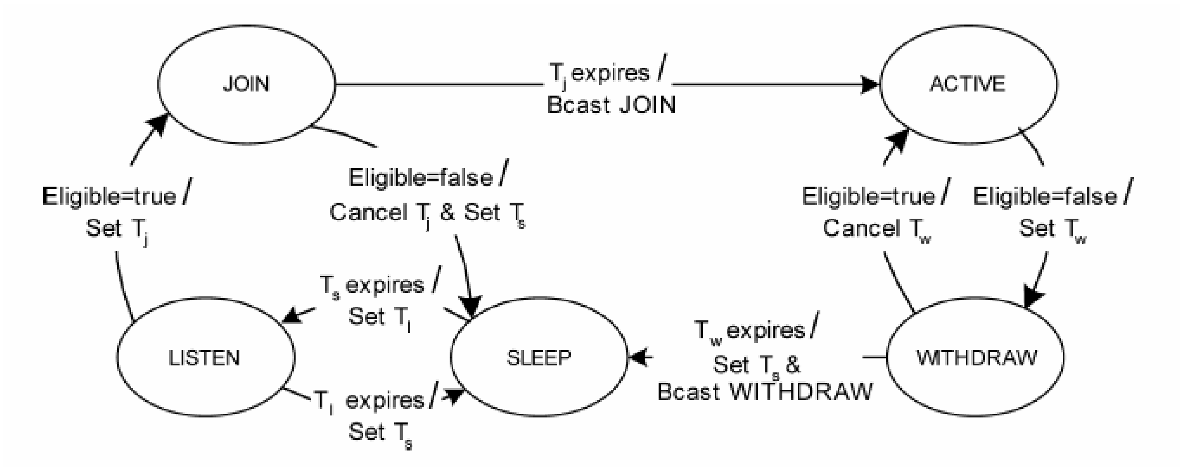

JOIN
ACTIVE
LISTEN
SLEEP
WITHDRAW
T_j expires / Bcast JOIN
Eligible=true / Set T_j
Eligible=false / Cancel T_j & Set T_s
Eligible=true / Cancel T_w
Eligible=false / Set T_w
T_s expires / Set T_l
T_l expires / Set T_s
T_w expires / Set T_s & Bcast WITHDRAW

Figure 2.8: State Diagram of CCP

In the SLEEP state, the node sleeps to conserve energy. In the ACTIVE state, the node actively senses the environment and communicates with other sensors. Each node periodically enters the LISTEN state to collect HELLO messages from its neighbors and reevaluates its eligibility. Two more transient states JOIN and WITHDRAW, are used to reduce the contention among neighbors in the transition from LISTEN to ACTIVE and the transition from ACTIVE to SLEEP, respectively.

1. SLEEP- When the sleep timer T_s expires, a node turns on the radio, starts a listen timer Tl, and enters the LISTEN state.

2. *LISTEN*- When a beacon (HELLO, WITHDRAW, or JOIN message) is received, a node evaluates its eligibility. If it is eligible, it starts a join timer Tj and enters the JOIN state. Otherwise, it sets a sleep timer Ts and returns to the SLEEP state when Tl expires.

3. *JOIN*- If a node becomes ineligible before Tj expires (e.g., due to the reception of a JOIN message), it cancels Tj, starts a sleep timer Ts, and returns to the SLEEP state. If Tj expires, it broadcasts a JOIN message and enters the ACTIVE state.

4. *ACTIVE*- When a node receives a HELLO message, it executes the coverage eligibility algorithm to determine its eligibility to remain active. If it is ineligible, it starts a withdraw timer Tw and enters the WITHDRAW state.

5. *WITHDRAW*- If a node becomes eligible (due to the reception of a WITHDRAW or HELLO message from a neighbor) before the Tw expires, it cancels the Tw and returns to the ACTIVE state. If Tw expires, it broadcasts a WITHDRAW message, starts a sleep timer Ts, and enters the SLEEP mode.

Both the join and withdraw timers are randomized to avoid collisions among multiple nodes that decide to join or withdraw. The values of Tj and Tw affect the responsiveness of CCP. Shorter timers lead to quicker response to the variations in coverage. Both timers should be set appropriately according to the network density. For example, for a denser network where a node has more neighbors, both timers should be increased to give a node enough time to collect the JOIN or WITHDRAW messages from its neighbors.

CHAPTER 3

OVERVIEW OF THE EXISTING NETWORK SIMULATORS

OVERVIEW OF THE EXISTING NETWORK SIMULATORS

Network simulators are very important for analyzing various protocols designed for a network (wired or wireless) and its necessity is very well known in the field of research. Especially, the research challenges in wireless sensor networks brought many open issues to network designers. The techniques used for analyzing the performance of any wireless networks are physical measurement, analytical methods and computer simulation. The constraints imposed on sensor networks, such as energy limitation, fault tolerance make the algorithms for sensor networks to be quite complex and usually defy analytical methods that have been effective for traditional networks. Moreover, physical measurement is not possible because of the unsolved research problems in the field of sensor networks. Hence, computer simulations appear to be the only feasible approach than anything else [6].

ns2, a widely used network simulator in the research community has the extended features to simulate Sensor Networks. It uses object-oriented design for the implementation of various modules of a sensor network [7].

There are modules for energy model, wireless channel, sensor channel which models dynamic inter-action between the physical environment and the sensor nodes. It also has implementations of few protocols that are under development for sensor networks. These include S-MAC, a Sensor MAC protocol at the MAC layer in a Sensor Node protocol stack, Directed Diffusion routing protocol with Geographic Routing. It also has a framework developed for Sensor Networks known as SensorSim that has the detailed implementation of a Sensor Node with a hardware model defining the hardware components of a sensor node and a software model defining the protocol stack of the node. The object-oriented design of ns2 introduces unnecessary interdependence between modules and makes the addition of new protocols very difficult as it can be mastered only by experts in ns2 [6]. This extension might be easy for traditional networks but not for sensor networks where the protocols are not very dominant and it is very unlikely that a single algorithm will be optimal under various circumstances. In addition, various simulation studies show that the memory utilization of ns2 is very high and increases for very large simulations. Since the application, areas in sensor networks require many number of sensor nodes in a sensor field, the simulations in ns2 take lot of memory. In addition, another disadvantage posed by ns2 comes from its open source nature. The documentation is often limited and out of date with the current release of the simulator. The problems can be solved with the help of dynamic news groups and going through the source code. In addition, the consistency of code is lacking as many users develop it. There are no tools describe simulation scenarios and analyze or visualize simulation trace files. The tools for ns2 are written with scripting languages. The lack of generalized analysis tools may lead to different people measuring different values for the same metric names [10].

OPNET modeler is another popular commercial platform for network modeling and simulation, which allows the design and study of communication networks, devices, protocols, and applications with unmatched flexibility and scalability. This is used by many prestigious technology organizations to accelerate the research and development process. OPNET Modeler is based on a series of hierarchical editors that directly parallel the structure of real networks, equipment, and protocols. The wireless model uses a 13- stage pipeline to determine connectivity and propagation among nodes. Modeler's object-oriented modeling and hierarchical editors mirror the structure of actual networks and network components [8]. The difficulty with OPNET Modeler is to build the state machine for each level of the protocol stack. It is difficult to abstract such a state machine starting from a pseudo-coded algorithm. However, state machines are the most practical input for discrete simulators. Hence, it is possible to reuse many existing components (MAC layer, transceivers, links, etc.) improving the deployment process. But on the other hand, any new feature must be described as a finite state machine which can be difficult to debug, extend and validate [10]. In addition, it is commercial and is not available for public, which becomes the biggest disadvantage for working on it.

J-Sim is an open-source, component based network simulation environment developed entirely in Java by Ohio State University (initially and later by University of Illinois). This along with the autonomous component architecture makes it a truly platform-neutral, extensible, and reusable environment. The Sensor Network Framework developed in J-Sim provides an object-oriented definition for target, sensor and sink nodes; sensor and wireless communication channels; and physical media such as seismic channels, mobility model and power model [9]. The simulation analysis described in [10] show that the execution speed of J-Sim is less compared to many other simulators and this happens because of its implementation in JAVA. But the memory consumption of J-Sim is less compared to others and this advantage comes from its garbage collectors.

GloMoSim developed initially at UCLA Computing Laboratory, is a scalable simulation environment for wireless and wired networks systems developed [6]. It is designed using the parallel discrete-event simulation capability provided by a C-based parallel simulation language, Parsec [10]. It currently supports protocols for purely wireless networks and is built using a layered approach. Standard APIs are used between the different layers and allow the rapid integration of models developed at different layers by users. The difficulty with GloMoSim was to describe a simple application that bypasses most OSI layers. The bypass of the protocol stack is not obvious to achieve as most applications usually lie on top of it. The architecture is also not very flexible compared to other simulators. Though many simulators were developed to emulate a Sensor Network, each has its own design complexities to test and verify new protocols.

The current study of coverage configuration protocols is done on OMNeT++ network simulator. OMNeT++ is an object-oriented modular discrete event network simulator. This framework allows the user to debug and test software for distributed sensor networks. OMNeT++ allows developers and researchers in the area of Sensor Networks to investigate topological, phenomenological, networking, robustness and scaling issues, to explore arbitrary algorithms for distributed sensors, and to defeat those algorithms through simulated failure.

CHAPTER 4

INTRODUCTION TO OMNeT++ NETWORK SIMULATOR

INTRODUCTION TO OMNeT++ NETWORK SIMULATOR

OMNeT++ is an object-oriented modular discrete event network simulator. This framework allows the user to debug and test software for distributed sensor networks. OMNeT++ allows developers and researchers in the area of Sensor Networks to investigate topological, phenomenological, networking, robustness and scaling issues, to explore arbitrary algorithms for distributed sensors, and to defeat those algorithms through simulated failure.

An OMNeT++ model consists of hierarchically nested modules. The depth of module nesting is not limited, which allows the user to reflect the logical structure of the actual system in the model structure. Modules communicate through message passing. Messages can contain arbitrarily complex data structures. Modules can send messages either directly to their destination or along a predefined path, through gates and connections.

Modules can have their own parameters. Parameters can be used to customize module behavior and to parameterize the model's topology. Modules at the lowest level of the module hierarchy encapsulate behavior. These modules are termed simple modules, and they are programmed in C++ using the simulation library.

OMNeT++ simulations can feature varying user interfaces for different purposes: debugging, demonstration and batch execution. Advanced user interfaces make the inside of the model visible to the user, allow control over simulation execution and to intervene by changing variables/objects inside the model. This is very useful in the development/debugging phase of the simulation project. User interfaces also facilitate demonstration of how a model works.

The simulator as well as user interfaces and tools are portable. They are known to work on Windows and on several UNIX flavors, using various C++ compilers.

OMNeT++ also supports parallel-distributed simulation. OMNeT++ can use several mechanisms for communication between partitions of a parallel-distributed simulation, for example MPI or named pipes. The parallel simulation algorithm can easily be extended or new ones plugged in. Models do not need any special instrumentation to be run in parallel – it is just a matter of configuration. OMNeT++ can even be used for classroom presentation of parallel simulation algorithms, because simulations can be run in parallel even under the GUI, which provides detailed feedback on what is going on.

4.1 ARCHITECTURE OF OMNeT++

OMNeT++ has a modular architecture. The high-level architecture of OMNeT++ simulations is shown in fig.4.1.

The rectangles in the picture represent components:

• **Sim:** It is the simulation kernel and class library. Sim exists as a library you link your simulation program with.

• **Envir:** It is another library, which contains all code that is common to all user interfaces. main () is also in Envir. Envir provides services like ini file handling for specific user interface implementations. Envir presents itself towards Sim and the executing model via the ev facade object, hiding all other user interface internals. Some aspects of Envir can be customized via plug-in interfaces. Embedding OMNeT++ into applications can be achieved implementing a new user interface in addition to Cmdenv and Tkev, or by replacing Envir with another implementation of ev.

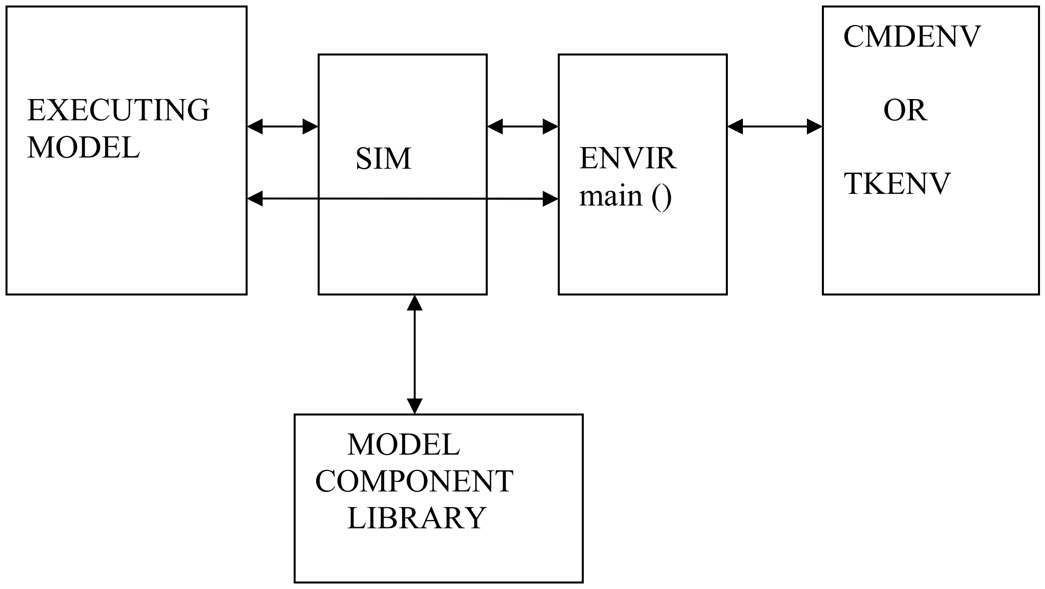


Figure 4.1: Architecture of OMNeT++ simulation programs

• **Cmdenv and Tkenv:** These are specific user interface implementations. A simulation is linked with either Cmdenv or Tkenv.

• **Model Component Library:** It consists of simple module definitions and their C++ implementations, compound module types, channels, networks, message types and in general everything that belongs to models and has been linked into the simulation program. A simulation program is able to run any model that has all necessary components linked in.

• **Executing Model:** It is the model that has been set up for simulation. It contains objects (modules, channels, etc.) that are all instances of components in the model component library. The arrows in the figure show how components interact with each other:

4.2 COMPONENTS OF OMNeT++:

- Simulation kernel library
- Compiler for the NED topology description language

- Graphical network editor for NED files (GNED)
- GUI for simulation execution, links into simulation executable (Tkenv)
- Command-line user interface for simulation execution (Cmdenv)
- Graphical output vector plotting tool (Plove)
- Utilities (random number seed generation tool, makefile creation tool, etc.)
- Documentation, sample simulations, contributed material, etc.

4.3 PLATFORMS OF OMNeT++:

OMNeT++ works well on multiple platforms. It was first developed on Linux. OMNet++ runs on most UNIX systems and Windows platforms (works best on NT4.0, W2K or XP). The best platforms used are:

- Solaris, Linux (or other Unix-like systems) with GNU tools.
- Win32 and Cygwin32 (Win32 port of gcc)
- Win32 and Microsoft Visual C++

4.4 LICENSING FOR OMNeT++:

OMNeT++ is free for any non-profit use. The author must be contacted if it is used in a commercial project. The GNU General Public License can be chosen on OMNeT++.

4.5 MODELING CONCEPTS:

OMNeT++ provides efficient tools for the user to describe the structure of the actual system. Some of the main features are:

- hierarchically nested modules
- modules are instances of module types
- modules communicate with messages through channels
- flexible module parameters
- topology description language

Hierarchical modules

An OMNeT++ model consists of hierarchically nested modules, which communicate by passing messages to each another. OMNeT++ models are often referred to as networks. The top level module is the system module. The system module contains sub modules, which can also contain sub modules themselves (Fig 4.2). The depth of module nesting is not limited; this allows the user to reflect the logical structure of actual system in the model structure.

Model structure is described in OMNeT++'s NED language.

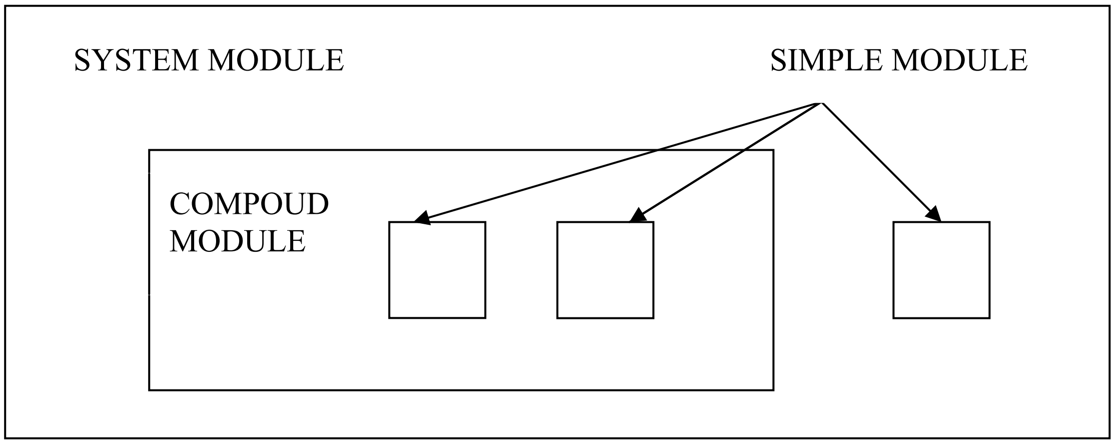


Figure 4.2: Simple and compound modules

Modules that contain submodules are termed compound modules, as opposed simple modules, which are at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model. The user implements the simple modules in C++, using the OMNeT++ simulation class library.

Module types

Both simple and compound modules are instances of module types. While describing the model, the user defines module types; instances of these module types serve as components for more complex module types. Finally, the user creates the system module as an instance of a previously defined module type; all modules of the network are instantiated as submodules and sub-submodules of the system module.

When a module type is used as a building block, there is no distinction whether it is a simple or a compound module. This allows the user to split a simple module into several simple modules embedded into a compound module, or vice versa, aggregate the functionality of a compound module into a single simple module, without affecting existing users of the module type.

Module types can be stored in files separately from the place of their actual usage. This means that the user can group existing module types and create component libraries.

Messages, gates, links

Modules communicate by exchanging messages. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queuing network or other types of mobile entities. Messages can contain arbitrarily complex data structures. Simple modules can send messages either directly to their destination or along a predefined path, through gates and connections.

The "local simulation time" of a module advances when the module receives a message. The message can arrive from another module or from the same module (self-messages are used to implement timers).

Gates are the input and output interfaces of modules; messages are sent out through output gates and arrive through input gates. Each connection (also called link) is created within a single level of the module hierarchy: within a compound module, one can connect the corresponding gates of two submodules, or a gate of one sub module and a gate of the compound module (Fig 4.3 &Fig 4.4).

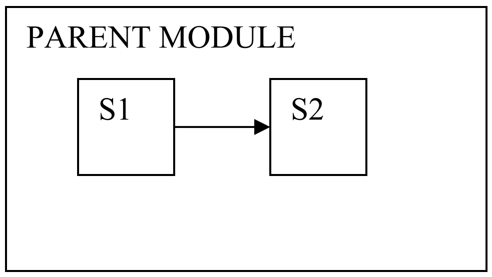


Figure 4.3: Submodules connected to each other

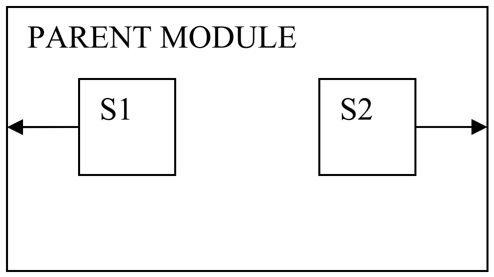


Figure4.4: Each submodule connected to parent module

Due to the hierarchical structure of the model, messages typically travel through a series of connections, to start and arrive in simple modules. Such series of connections that go from simple module to simple module are called routes. Compound modules act as 'cardboard boxes' in the model, transparently relaying messages between their inside and the outside world.

4.6 SIMULATION MODELING IN OMNeT++

The following are types of modeling that can be used:

- Communication protocols
- Computer networks and traffic modeling
- Multi-processor and distributed systems
- Administrative systems
- In addition, any other system where the discrete event approaches is suitable.

Library Modules

Object libraries can be made using simple modules. The best simple modules to be used for library modules are the ones that implement:

- Physical/Data-link protocols: Ethernet, Token Ring, FDDI, LAPB etc.
- Higher layer protocols: IP, TCP, X.25 L2/L3, etc.
- Network application types: E-mail, NFS, X, audio etc.
- Basic elements: message generator, sink, concentrator/simple hub, queue etc.
- Modules that implement routing algorithms in a multiprocessor or network

Network Modeling

A model network consists of "nodes" connected by "links. The nodes representing blocks, entities, modules, etc, while the link representing channels, connections, etc. The structure of how fixed elements (i.e. nodes) in a network are interconnected together is called topology.

Omnet++ uses NED language, thus allowing for a more user friendly and accessible environment for creation and editing. It can be created with any text-processing tool (Perl, awk, etc). It has a human-readable textual topology. It also uses the same format as that of a graphical editor. It also supports submodule testing. Omnet++ allows for the creation of a driver entity to build a network at run-time by program.

Organization of Network Simulation:

Omnet++ follows a hierarchical module structure allowing for different levels of organization.

- ***Physical Layer:***

    1. Top-level network
    2. Sub network (site)
    3. LAN
    4. node

- ***Topology within a node:***

    1. OSI layers. The Data-Link, Network, Transport, Application layers are of greater importance.
    2. Applications/protocols within a layer.

4.7 NED LANGUAGE:

The topology of a model is specified using the NED language. The NED language facilitates the modular description of a network. This means that a network description may consist of a number of component descriptions (channels, simple/compound module types). The channels, simple modules and compound modules of one network description can be reused in another network description.

Files containing network descriptions generally have a .ned suffix. NED files can be loaded dynamically into simulation programs, or translated into C++ by the NED compiler and linked into the simulation executable.

Components of a NED description

A NED description can contain the following components, in arbitrary number or order:

- import directives
- channel definitions
- simple and compound module definitions
- network definitions

4.8 PROGRAMMING THE ALGORITHMS:

The simple modules of a model contain algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported by the OMNeT++ simulation class library. The simulation programmer can choose between event-driven and process-style description, and can freely use object-oriented concepts (inheritance, polymorphism etc) and design patterns to extend the functionality of the simulator.

Simulation objects (messages, modules, queues etc.) are represented by C++ classes. They have been designed to work together efficiently, creating a powerful simulation-programming framework. The following classes are part of the simulation class library:

- modules, gates, connections etc.
- parameters
- messages
- container classes (e.g. queue, array)
- data collection classes
- statistic and distribution estimation classes (histograms, P2 algorithm for calculating quintiles etc.)
- transient detection and result accuracy detection classes

The classes are also specially instrumented, allowing one to traverse objects of a running simulation and display information about them such as name, class name, state variables or contents. This feature has made it possible to create a simulation GUI where all internals of the simulation are visible.

4.9 USER INTERFACES

OMNeT++ user interface is used with the simulation execution. OMNeT++'s design allows the inside of model to be seen by the user. It also allows the user to initiate and terminate simulations, as well as change variable inside simulation models. These features are handy during the development and debugging phase of modules in a project. Graphical interface is a user-friendly option in Omnet++ allows access to the internal workings of the model. The interaction of the user interface and the simulation kernel is through a well defined interface. Without changing the simulation kernel, it is possible to implement several types

of user interfaces. Also without changing the model file, the simulation model can run under different interfaces. The user would test and debug the simulation with a powerful graphical user interface, and finally run it with a simple and fast user interface that supports batch execution.

The user interfaces are a form of interchangeable libraries. When linking into a created simulation executable, the user can choose the interface libraries they would like to use.

Currently, two user interfaces are supported

- **Tkenv**: Tk-based graphical, windowing user interface (X-Window, Win95, WinNT etc...)
- **Cmdenv**: command-line user interface for batch execution

Simulation is tested and debugged under Tkenv, while the Cmdenv is used for actual simulation experiments since it supports batch execution.

Tkenv

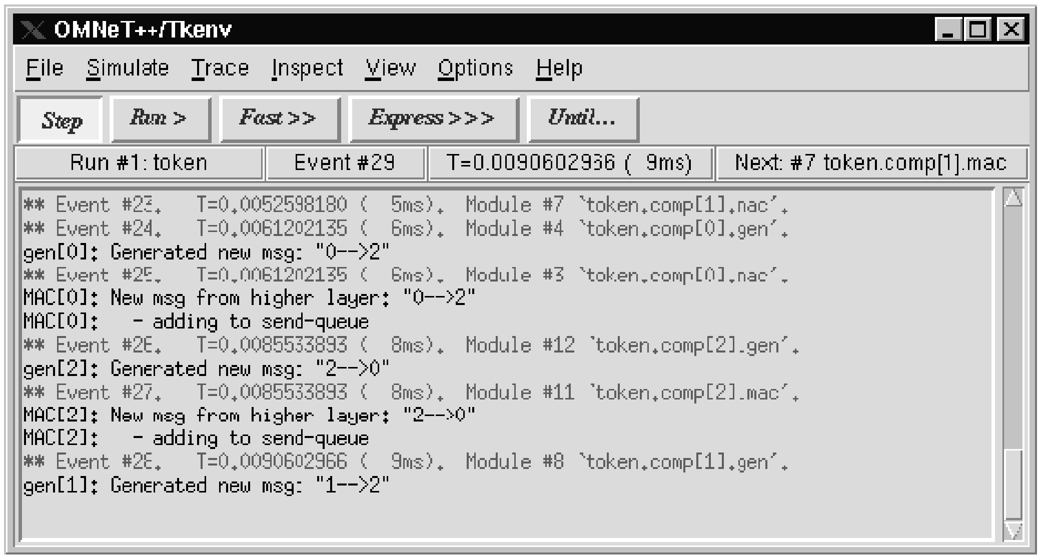
Tkenv is a portable graphical windowing user interface. Tracing, debugging, and simulation execution is supported by Tkenv. It has the ability to provide a detailed picture of the state of the simulation at any point during the execution. This feature makes Tkenv a good candidate in the development stage of a simulation or for presentations. A snapshot of a Tkenv interface is shown in figure 4.5.

Important features in Tkenv

- separate window for each module's text output
- scheduled messages can be watched in a window as simulation progresses
- event-by-event execution
- execution animation
- labeled breakpoints
- inspector windows to examine and alter objects and variables in the model
- Graphical display of simulation results during execution. Results can be displayed as histograms or time-series diagrams.

- simulation can be restarted
- Snapshots (detailed report about the model: objects, variables etc.)

It is recommended for testing and debugging when used with gdb or xxgdb. Tkenv provides a good environment for experimenting with the model during executions and verification of the correct operation during the simulation program. Since this is possible to display simulation results during execution.



```
Run #1: token    Event #29    T=0.0090602936 (  9ms)    Next: #7 token.comp[1].mac
```

```
** Event #23.   T=0.0052598180 (  5ms).  Module #7  `token.comp[1].mac'.
** Event #24.   T=0.0061202135 (  6ms).  Module #4  `token.comp[0].gen'.
gen[0]: Generated new msg: "0-->2"
** Event #25.   T=0.0061202135 (  6ms).  Module #3  `token.comp[0].mac'.
MAC[0]: New msg from higher layer: "0-->2"
MAC[0]:   - adding to send-queue
** Event #26.   T=0.0085533893 (  8ms).  Module #12 `token.comp[2].gen'.
gen[2]: Generated new msg: "2-->0"
** Event #27.   T=0.0085533893 (  8ms).  Module #11 `token.comp[2].mac'.
MAC[2]: New msg from higher layer: "2-->0"
MAC[2]:   - adding to send-queue
** Event #28.   T=0.0090602966 (  9ms).  Module #8  `token.comp[1].gen'.
gen[1]: Generated new msg: "1-->2"
```

Figure 4.5: Example of a Tkenv User Interface in OMNeT++

Cmdenv

Cmdenv is designed primarily for batch execution. It is a portable and small command line interface that is fast. It compiles and runs on all platforms. Cmdenv simply executes all simulation runs that are described in the configuration file.

4.10 BUILDING AND RUNNING SIMULATION

An OMNeT++ model consists of the following parts:

- NED language topology description(s) (.ned files) which describe the module structure with parameters, gates etc. NED files can be written using any text editor or the GNED graphical editor.
- Message definitions (.msg files). You can define various message types and add data fields to them. OMNeT++ translates message definitions into full-fledged C++ classes.
- Simple modules sources. They are C++ files, with .h/.cc/.cpp suffix.

Simulation programs are built from the above components. First, .msg files are translated into C++ code using the opp_msgc program. Then all C++ sources are compiled, and linked with the simulation kernel and a user interface library to form a simulation executable.

Running the simulation and analyzing the results

The simulation executable is a standalone program, thus it can be run on other machines without OMNeT++ or the model files being present. When the program is started, it reads a configuration file (usually called omnetpp.ini). This file contains settings that control how the simulation is executed, values for model parameters, etc.

CHAPTER 5

IMPLEMENTATION

IMPLEMENTATION

5.1 OMNeT++ SIMULATOR REPRESENTING DIFFERENT STATES OF NODES

In CCP, each node determines its eligibility using the K_s -coverage eligibility algorithm based on the information about its sensing neighbors, and may switch state dynamically when its eligibility changes. A node can be in one of three states: SLEEP, ACTIVE, and LISTEN, as shown in Fig 5.1. Two more transient states JOIN and WITHDRAW, are used to reduce the contention among neighbors in the transition from LISTEN to ACTIVE and the transition from ACTIVE to SLEEP, respectively.

5.1.1 Different States :

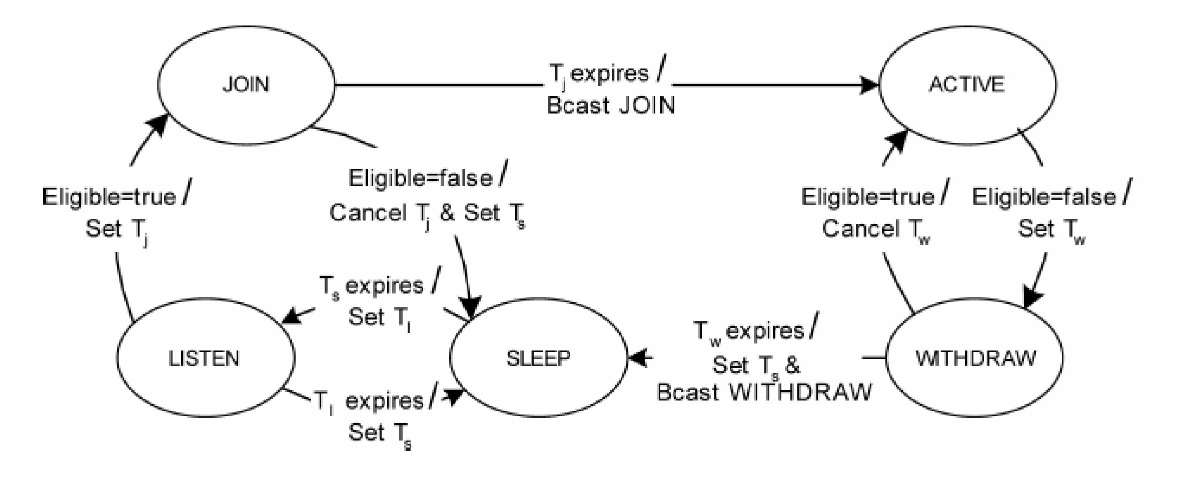

JOIN
ACTIVE
T_j expires / Bcast JOIN
Eligible=true / Set T_j
Eligible=false / Cancel T_j & Set T_s
Eligible=true / Cancel T_w
Eligible=false / Set T_w
T_s expires / Set T_l
LISTEN
SLEEP
WITHDRAW
T_w expires / Set T_s & Bcast WITHDRAW
T_l expires / Set T_s

Figure 5.1: State Diagram of CCP

1. SLEEP- When the sleep timer T_s expires, a node turns on the radio, starts a listen timer Tl, and enters the LISTEN state.

2. ***LISTEN***- When a beacon (HELLO, WITHDRAW, or JOIN message) is received, a node evaluates its eligibility. If it is eligible, it starts a join timer Tj and enters the JOIN state. Otherwise, it sets a sleep timer Ts and returns to the SLEEP state when Tl expires.

3**.** ***JOIN***- If a node becomes ineligible before Tj expires (e.g., due to the reception of a JOIN message), it cancels Tj, starts a sleep timer Ts, and returns to the SLEEP state. If Tj expires, it broadcasts a JOIN message and enters the ACTIVE state.

4**.** ***ACTIVE***- When a node receives a HELLO message, it executes the coverage eligibility algorithm to determine its eligibility to remain active. If it is ineligible, it starts a withdraw timer Tw and enters the WITHDRAW state.

5. ***WITHDRAW***- If a node becomes eligible (due to the reception of a WITHDRAW or HELLO message from a neighbor) before the Tw expires, it cancels the Tw and returns to the ACTIVE state. If Tw expires, it broadcasts a WITHDRAW message, starts a sleep timer Ts, and enters the SLEEP mode.

5.1.2 Different Files for Simulation:

1. NED File:

```
simple txc13
   parameters:
      sleepTimeMean: numeric,
      burstTimeMean: numeric,
      sendJTime: numeric,
      sendIATime: numeric,
      msgLength: numeric;
   gates:
      in: in[];
      out: out[];
endsimple

module States
   submodules:
      tic: txc13[6];
```

```
            display: "i=block/process";
    connections:
        tic[0].out++ --> delay 100ms --> tic[1].in++;
        tic[0].in++ <-- delay 100ms <-- tic[1].out++;

        tic[1].out++ --> delay 100ms --> tic[2].in++;
        tic[1].in++ <-- delay 100ms <-- tic[2].out++;

        tic[1].out++ --> delay 100ms --> tic[4].in++;
        tic[1].in++ <-- delay 100ms <-- tic[4].out++;

        tic[3].out++ --> delay 100ms --> tic[4].in++;
        tic[3].in++ <-- delay 100ms <-- tic[4].out++;

        tic[4].out++ --> delay 100ms --> tic[5].in++;
        tic[4].in++ <-- delay 100ms <-- tic[5].out++;
endmodule

network states : States
endnetwork
```

2. CPP File:

```
#define FSM_DEBUG
#include <omnetpp.h>

class txc13 : public cSimpleModule
{
  protected:
// parameters
                                double sleepTimeMean;
                                double burstTimeMean;
                                double sendJTime;
                                double sendIATime;
                                cPar *msgLength;

// FSM and its states
                                cFSM fsm;
                                enum
                                    {
                                 WITHDRAW       = 0,
                                  SLEEP         = FSM_Steady(1),
                                  LISTEN        = FSM_Steady(2),
                                  JOIN          = FSM_Steady(3),
                                  ACTIVE        = FSM_Transient(1),
                                 };

// variables used
                                int i;
                                cMessage *startStopBurst;
                                cMessage *sendJMessage;
                                cMessage *sendMessage;

// the virtual functions
                                virtual void initialize();
                                virtual void handleMessage(cMessage *msg);
};

Define_Module(txc13);

void txc13::initialize()
 {
                                fsm.setName("fsm");
```

```
                    sleepTimeMean     = par("sleepTimeMean");
                    burstTimeMean     = par("burstTimeMean");
                    sendJTime         = par("sendJTime");
                    sendIATime        = par("sendIATime");
                    msgLength         = &par("msgLength");
                    i                 = 0;

                    WATCH(i); // always put watches in initialize()
                    startStopBurst    = new cMessage("startStopBurst");
                    sendJMessage      = new cMessage("sendJMessage");
                    sendMessage       = new cMessage("sendMessage");
                    scheduleAt(0.0,startStopBurst);

}

void txc13::handleMessage(cMessage *msg)
{
                    FSM_Switch(fsm)
                     {
                       case FSM_Exit(WITHDRAW):
// transition to SLEEP state
                         FSM_Goto(fsm,SLEEP);
                         break;

                       case FSM_Enter(SLEEP):
                       cancelEvent(startStopBurst);
                        bubble("enter SLEEP");
// schedule end of sleep period (start of next burst)

                    scheduleAt(simTime()+exponential(sleepTimeMean),startStopB
                    urst);
                    break;

                      case FSM_Exit(SLEEP):
                      cancelEvent(startStopBurst); bubble("exit SLEEP");
// schedule end of this burst

                    scheduleAt(simTime()+exponential(burstTimeMean),startStopB
                    urst);
// transition to LISTEN state:
                       if (msg!=startStopBurst)
                         {
                              FSM_Goto(fsm,WITHDRAW);

                         }
                      FSM_Goto(fsm,LISTEN);
                       cancelEvent(startStopBurst);
                       break;

                      case FSM_Enter(LISTEN):
                      cancelEvent(sendJMessage); bubble(" enter LISTEN");

// schedule next sending
                       scheduleAt(simTime()+exponential(sendJTime),
                       sendJMessage);
                       break;

                      case FSM_Exit(LISTEN): bubble(" exit LISTEN");

// transition to either JOIN or SLEEP
                       if (msg==sendJMessage)
                         {
```

```
                                   FSM_Goto(fsm,JOIN);
                              }
                              else
                              {
                                   cancelEvent(sendJMessage);
                                   FSM_Goto(fsm,SLEEP);
                              }

                              break;

                              case FSM_Enter(JOIN):
                              cancelEvent(sendJMessage); bubble("enter JOIN");

//schedule next sending
                              scheduleAt(simTime()+exponential(sendIATime),
                              sendMessage);
                              break;

                              case FSM_Exit(JOIN): bubble("exit JOIN");


//transition to either ACTIVE or SLEEP or WITHDRAW
                              if (msg==sendMessage)
                              {
                                      bubble(" enter ACTIVE");
                                      FSM_Goto(fsm,ACTIVE);
                              }
                              else if (msg==startStopBurst)
                              {
                                        cancelEvent(sendMessage);
                                      FSM_Goto(fsm,SLEEP);
```

```
                                    }

                                  else
                                    {
                                            bubble("enter WITHDRAW");
                                            cancelEvent(sendMessage);
                                           FSM_Goto(fsm,WITHDRAW);
                                    }
                                  break;

                                  case FSM_Exit(ACTIVE):
                                    {
// generate and send out job
                                            char msgname[32]; bubble("ACTIVE");
                                            sprintf( msgname, "job-%d", ++i);
                                            ev   <<   "Generating   HELLO   MESSAGE"   <<
msgname                                     << endl;
                                            cMessage *job = new cMessage(msgname);
                                            job->setLength( (long) *msgLength );
                                            job->setTimestamp();
                                            send( job, "out" );
// return to LISTEN
                                            bubble("enter SLEEP");
                                            FSM_Goto(fsm,SLEEP);
                                  break;
                                  }
                              }
}
```

3. Configuration (ini) File :

```
# This file is shared by states network simulation.
# Lines beginning with `#' are comments

[General]
preload-ned-files=*.ned
network= states  # this line is for Cmdenv, Tkenv will still let you choose from a dialog
sim-time-limit=500000s
output-vector-file=states.vec
[Parameters]
```

5.2 OMNeT++ SIMULATOR REPRESENTING THE SENSING OF PARAMETERS

Sense is a Network, which contains twelve nodes from which six nodes are Temperature sensing nodes and other six are Voltage sensing nodes. Each node is a simple module, which goes in one of the five different states according to the broadcasting of hello messages from other nodes.

When temperature is sensed, then all six nodes, sensing temperature, come in coverage of each other and go in one of the five states to transmit the message. At this time, voltage-sensing nodes remain in SLEEP or LISTEN state, but not in ACTIVE state.

Similarly, when voltage is sensed, all voltage sensing nodes come in coverage of each other and go in one of the five different states to transmit message, but temperature-sensing nodes remain in SLEEP or LISTEN state.

5.2.1 Different Files for Simulation:

1. NED File:

```
simple txc13
   parameters:
      sleepTimeMean: numeric,
      burstTimeMean: numeric,
      sendJTime: numeric,
      sendIATime: numeric,
      msgLength: numeric;
   gates:
      in: in[];
      out: out[];
endsimple

module Sense
```

```
submodules:
    temp: txc13[6];
        display: "i=block/process,cyan";
    volt: txc13[6];
        display: "i=block/app2,gold";
connections:
    temp[0].out++ --> delay 100ms --> temp[1].in++;
    temp[0].in++ <-- delay 100ms <-- temp[1].out++;

    temp[1].out++ --> delay 100ms --> temp[2].in++;
    temp[1].in++ <-- delay 100ms <-- temp[2].out++;

    temp[1].out++ --> delay 100ms --> temp[4].in++;
    temp[1].in++ <-- delay 100ms <-- temp[4].out++;

    temp[3].out++ --> delay 100ms --> temp[4].in++;
    temp[3].in++ <-- delay 100ms <-- temp[4].out++;

    temp[4].out++ --> delay 100ms --> temp[5].in++;
    temp[4].in++ <-- delay 100ms <-- temp[5].out++;


    volt[0].out++ --> delay 100ms --> volt[4].in++;
    volt[0].in++ <-- delay 100ms <-- volt[4].out++;

    volt[2].out++ --> delay 100ms --> volt[5].in++;
    volt[2].in++ <-- delay 100ms <-- volt[5].out++;

    volt[0].out++ --> delay 100ms --> volt[2].in++;

    volt[0].in++ <-- delay 100ms <-- volt[2].out++;
```

```
        volt[1].out++ --> delay 100ms --> volt[0].in++;
        volt[1].in++ <-- delay 100ms <-- volt[0].out++;

        volt[2].out++ --> delay 100ms --> volt[3].in++;
        volt[2].in++ <-- delay 100ms <-- volt[3].out++;

endmodule

network sense : Sense
endnetwork
```

2.CPP File

```
#define FSM_DEBUG
#include <omnetpp.h>

class txc13 : public cSimpleModule
{
  protected:
// parameters
                    double sleepTimeMean;
                    double burstTimeMean;
                    double sendJTime;
                    double sendIATime;
                    cPar *msgLength;

// FSM and its states
                    cFSM fsm;
                    enum
                      {

                     WITHDRAW     = 0,
                              SLEEP          = FSM_Steady(1),
                              LISTEN         = FSM_Steady(2),
                              JOIN           = FSM_Steady(3),
                              ACTIVE         = FSM_Transient(1),
                             };

// variables used
                             int i;
                             cMessage *startStopBurst;
                             cMessage *sendJMessage;
                             cMessage *sendMessage;

// the virtual functions
                             virtual void initialize();
                             virtual void handleMessage(cMessage *msg);
};

Define_Module(txc13);

void txc13::initialize()
 {
                             fsm.setName("fsm");
                             sleepTimeMean    = par("sleepTimeMean");
                             burstTimeMean    = par("burstTimeMean");
                             sendJTime        = par("sendJTime");
                             sendIATime       = par("sendIATime");
                             msgLength        = &par("msgLength");
                             i                = 0;

                             WATCH(i); // always put watches in initialize()
                             startStopBurst   = new cMessage("startStopBurst");
```

```
                              sendJMessage        = new cMessage("sendJMessage");
                              sendMessage         = new cMessage("sendMessage");
                              scheduleAt(0.0,startStopBurst);

}

void txc13::handleMessage(cMessage *msg)
{
                              FSM_Switch(fsm)
                                {
                                  case FSM_Exit(WITHDRAW):
// transition to SLEEP state
                                    FSM_Goto(fsm,SLEEP);
                                    break;

                                  case FSM_Enter(SLEEP):
                                  cancelEvent(startStopBurst);
                                   bubble("enter SLEEP");
// schedule end of sleep period (start of next burst)

                              scheduleAt(simTime()+exponential(sleepTimeMean),startStopB
                              urst);
                              break;

                                  case FSM_Exit(SLEEP):
                                  cancelEvent(startStopBurst); bubble("exit SLEEP");
// schedule end of this burst

                              scheduleAt(simTime()+exponential(burstTimeMean),startStopB
                              urst);
// transition to LISTEN state:
```

```
                        if (msg!=startStopBurst)
                          {
                                FSM_Goto(fsm,WITHDRAW);

                          }
                        FSM_Goto(fsm,LISTEN);
                        cancelEvent(startStopBurst);
                        break;

                       case FSM_Enter(LISTEN):
                       cancelEvent(sendJMessage); bubble(" enter LISTEN");

// schedule next sending
                        scheduleAt(simTime()+exponential(sendJTime),
                        sendJMessage);
                        break;

                       case FSM_Exit(LISTEN): bubble(" exit LISTEN");

// transition to either JOIN or SLEEP
                        if (msg==sendJMessage)
                          {
                              FSM_Goto(fsm,JOIN);
                          }
                        else
                          {
                              cancelEvent(sendJMessage);
                              FSM_Goto(fsm,SLEEP);
                          }

                        break;
```

```
                                case FSM_Enter(JOIN):
                                cancelEvent(sendJMessage); bubble("enter JOIN");

//schedule next sending
                                scheduleAt(simTime()+exponential(sendIATime),
                                sendMessage);
                                break;

                                case FSM_Exit(JOIN): bubble("exit JOIN");


//transition to either ACTIVE or SLEEP or WITHDRAW
                                if (msg==sendMessage)
                                 {
                                     bubble(" enter ACTIVE");
                                     FSM_Goto(fsm,ACTIVE);
                                 }
                                else if (msg==startStopBurst)
                                 {
                                       cancelEvent(sendMessage);
                                     FSM_Goto(fsm,SLEEP);
                                 }

                                else
                                 {
                                     bubble("enter WITHDRAW");
                                      cancelEvent(sendMessage);
                                      FSM_Goto(fsm,WITHDRAW);
                                 }
                                break;
```

```
                        case FSM_Exit(ACTIVE):
                         {
// generate and send out job
                                char msgname[32]; bubble("ACTIVE");
                               sprintf( msgname, "job-%d", ++i);
                               ev   <<   "Generating   HELLO   MESSAGE"   <<
msgname                        << endl;
                               cMessage *job = new cMessage(msgname);
                               job->setLength( (long) *msgLength );
                               job->setTimestamp();
                               send( job, "out" );
// return to LISTEN
                               bubble("enter SLEEP");
                               FSM_Goto(fsm,SLEEP);
                        break;
                        }
                      }
}
```

3. Configuration (ini) File:

```
# This file is shared by sense network simulation.
# Lines beginning with `#' are comments

[General]
preload-ned-files=*.ned
network= sense  # this line is for Cmdenv, Tkenv will still let you choose from a dialog
sim-time-limit=500000s
output-vector-file= sense.vec
[Parameters]
```

5.3 APPLICATION OF CCP IN DETECTING THE SELFISH NODE

The simulator is based on OMNeT++ and implements the CCP algorithm for providing sensing coverage and connectivity in a Wireless Sensor Network. The simulator depicts a Sensor Network with a parameterizable number of hosts that are distributed in a field free of obstacles. Each host has a defined transmission power that affect the range within a communication is feasible. The signal power degradation is modeled by the Free Space Propagation Model, *which* states that the received signal strength is inversely proportional to the node distance square.

Each host is a compound module, which encapsulates the following simple modules:

- A physical layer
- A MAC layer
- A route layer
- An application layer;

The communication between the modules is made via messages exchange. Each module (simple or compound) can be replaced by other newly implemented one simply modifying the omnetpp.ini file. Then there is no need to bother about writing any new instruction in the other simulator models.

5.3.1 Physical Model

It implements the physical layer of each host. In particular, it cares about the on-fly creation of gates that allow the exchange of messages among the hosts. This dynamic capability represents an important contribute to the existing wireless module for OMNeT++ available in Internet. Every time an inter-distance check on each node is performed. If a host gets close enough (depending on the transmission power of the moving node) to a new neighbor, these operations take place:

1. A new gate is created for both the compound modules (the two hosts modules).

2. A new gate is created on each of the physic simple module contained in the host module.
3. A link is created between the newly created simple module gate and the compound module new gate.
4. A link is created between the two hosts modules. This last link uses the "etere" channel property. "Etere" is a channel type that I defined and that gives to the link a delay, throughput and error probability characteristics.

When the two nodes get too far that is the first node has no intersection point within its coverage region with second node, it means these two are out of coverage, these gates are deleted, so the link between these two hosts are not created. Each node has its own transmission power so it can happen that a node has a link toward another host but there is not a reverse link.

The physic module can receive messages from other hosts. When this happen, if the message comes from outside and does not contains errors, it is sent directly to the higher levels. When a higher-level module needs to send a message, it sends it to the physical level that will care about the correct delivery. This module has a list of the current neighbors so scanning this list entries, it sends a new copy of the original message through the gates that connect the host to the other nodes. The simulator kernel, accordingly to the gate settings and the message length, will care about the correct delivery time of the message to the neighbor.

5.3.2 Mac Layer

This module depicts the ISO/OSI MAC layer. Here it is possible to insert different channel contention protocols such as CSMA/CA, MACA, MACAW and any other existent algorithm. All of these protocols are very complex and their implementation is a new project worth.

The layer implemented is much a simpler one. The outgoing messages are let pass through. The incoming one instead is delivered to the higher levels with a MM1 queue policy. When a in-coming message arrives the module check a flag that advise if the higher level is busy. If it is the message, is put in buffer or, if the buffer is full, it is dropped. When the higher level is no busier, the MAC module picks the first message in the buffer, send it upward and schedule to itself an end of service message that will trigger a new pick from the buffer or set the busy-flag as free.

This level check all the incoming messages and watching their *mac address*. It let pass only those who are addressed to this module or are broadcast one.

The node can even work in promiscuous mode meaning that all the messages, even the other module's one, are allowed to be elaborated by the higher levels. This is a dangerous thing for the security of the network communication but it is as well a very important resource for all the on-demand routing protocols.

5.3.3 Routing model

The routing model is the simulator heart. This model depicts the routing protocol and it is set between the MAC module and the application module. It receives *DATA* messages from the higher layers and tries to find a route to the chosen destination looking in its neighbor table or sending control messages **(a rreq)** to get a new route.

This CCP simulator implements those options suitable for a wireless sensor network and in particular:

- HELLO message exchange between neighbor nodes**.** As stated by the standard, when the underlying layer does not provide any information about the link status, HELLO messages are used to check the neighbor status. Each node enters all the information it receives through broadcasting the HELLO message & updates into a neighbor table. Consequently, this neighbor table contains a list of neighbors and for each neighbor, a list of its neighbors. The fields (bits) of HELLO message (broadcast message) of a node is shown below in table5.1.

Fields of HELLO message
Source ID Destination ID Neighbors list

Table 5.1: Neighbor Table

- The expanding ring search optimization is used to regulate RREQ broadcast. This means that initially a route request is sent using a small ttl hoping that some neighbor knew a route toward the chosen destination. If the request fails, a new one with a bigger ttl is sent. A fixed number of retrials is allowed after which the transmission trial is aborted.

- Due to the asymmetrical nature of the wireless link, ACK messages are used to confirm the correct delivery of a RREP message. Data message acknowledgment is referred to the transport layer.

- A black list is used to avoid unreliable neighbor nodes. A node, call as X, inserts a neighbor Y, in the black list when X, after trying a number of times to send a RREP message to Y, it does not receive any acknowledgment. When this happen it means that the link between X and Y is unidirectional, X "hears" the messages sent by Y but Y does not do the same with X's messages. CCP standard says that when a node X put a neighbor Y in the black list, X will not consider any new RREQ messages coming from Y for a fixed amount of time. Hello message coming from Y are still processed by X. This may bring some problems to the node that might wish to communicate directly with to Y and because Y is in X's neighbor table, no RREQ will be sent and the straight route will be used.

5.3.4 Traffic model

This module generates the data traffic that triggers all the routing operations. Each host has its own traffic generator that can be switched on/off setting the active parameter in the omnetppp.ini file. This module schedules a self-message to trigger the data sending operation.

The traffic is modeled by generating a packet burst of sixty four messages sent to a randomly chosen destination that stays the same for all the burst length. The rate of each burst sending messages is defined by the rate parameter.

As previously mentioned, a host is identified by its ID number that the OMNeT++ kernel assigns at the simulation beginning.

These IDs are not in sequence and may vary depending on the total number of modules that work in a simulation. To generate a correct destination number, avoiding the burden of scanning all the module vector of pointers kept by the simulator kernel, the module uses a pointer to the physic layer that already has a list of all available destinations.

5.3.5 Modular architecture of Network:

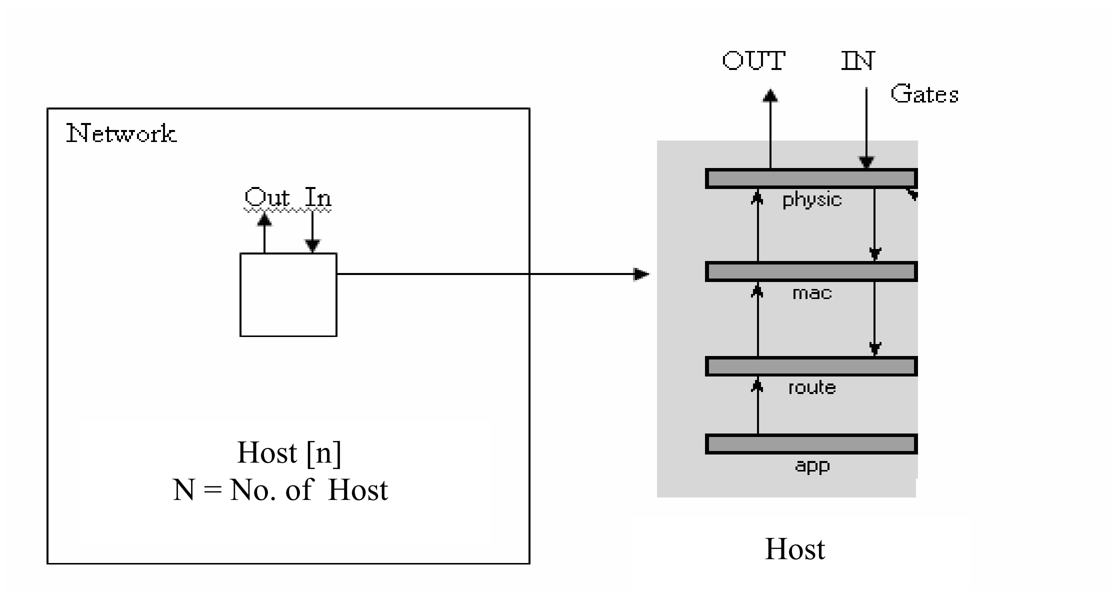


Fig 5.2: Modular Architecture of Network

Network is a simple module, which contains n Host submodules. Each Host submodule contains four submodules to represent physical, mac, network, and application layers. Each Host submodule has two gates In and Out for external communication.

5.3.6 Different Files for simulation:

1. NED Files:

a) For World Network:

```
module World
   parameters:
      dim: numeric,
      width: numeric,
      height: numeric;
   submodules:
        Host: Host[dim];
            parameters:
                numHost = dim,
                Xbound = width,
                Ybound = height,

                //x = width /2,
                //y = height /2;
                x = intuniform (5, width -5),
                y = intuniform (5, height -5);
                                              //x = 60 + (index % 5 ) * 120,
                                              //y = 30 + (index - index %  5 ) * 30 ;

                                              //display: "p=95, 40; b=20, 20";
            display: "p=10, 10; b=$width, $height";
    connections:
    display: "b=0, 12";
endmodule
```

b) <u>For simple modules:</u>

```
   simple Physic
    parameters:
        txPower: numeric,
        rxThreshold: numeric,
        channelDelay: numeric,
        channelDatarate: numeric,
        channelError: numeric;
    gates:
        in: fromMobility;
        in: fromMac;
        out: toMac;
```

```
endsimple

simple Mac
   parameters:
      inBufferSize: numeric,
      promisqueMode: bool;
   gates:
      in: fromPh;
      in: fromRoute;
      out: toRoute;
      out: toPh;
endsimple

simple Application
   parameters:
      rate: numeric, //paket per secod
      pktSize: numeric,
      hostNum: numeric,
      active: numeric,
      burstInterval: numeric; // time(s) between two data bursts
   gates:
      out: out;
endsimple

simple Routing
   gates:
      in: fromMac;
      in: fromApp;
      out: toMac;
endsimple
```

2. Configuration (ini) File:

```
#omnetpp.ini
[General]
preload-ned-files=*.ned
network = world
sim-time-limit = 60s
total-stack-kb = 32768
num-rngs=5

[Parameters]

#world module
;world.height = 500
;world.width = 500
;world.dim = 50

#include randWP.ini

        #sensor host module
#world.Host [*].x = intuniform (5, 55)
#world.Host [*].y = intuniform (5, 55)

world.Host[*].routeAlgorithm = "CCP"
world.Host [*].macAlgorithm = "SimpleMac"

        #pyisic module
world.Host [*].physic.txPower = uniform (9000, 9900)
world.Host [*].physic.rxThreshold = 1
world.Host [*].physic.channelDelay = 0.0001
world.Host [*].physic.channelDatarate = 11.04858e+6
world.Host [*].physic.channelError = 0.000001
```

```
        #mac module
world.Host [*].mac.promisqueMode = true;
world.Host [*].mac.inBufferSize = 8.38864e6


        #application module
; pakets per secod
world.Host [*].app.rate = 3
; pakets of 512 byte = 4096 bit
world.Host [*].app.pktSize = 4096;
// time elapsed between two data burst
world.Host [*].app.burstInterval = truncnormal (2, 1.0)
world.Host [*].app.active = 1
```

CHAPTER 6

RESULTS

RESULTS

6.1 OUTPUT OF NETWORK FOR SHOWING DIFFERENT STATES OF NODES:

When simulation is started, then the numbers of States of nodes of a network (as specified by the parameters) is displayed on the screen and nodes of the network are appeared to move in different states as shown in Figure 6.1. These states are ACTIVE, SLEEP, LISTEN, JOIN & WITHDRAW.

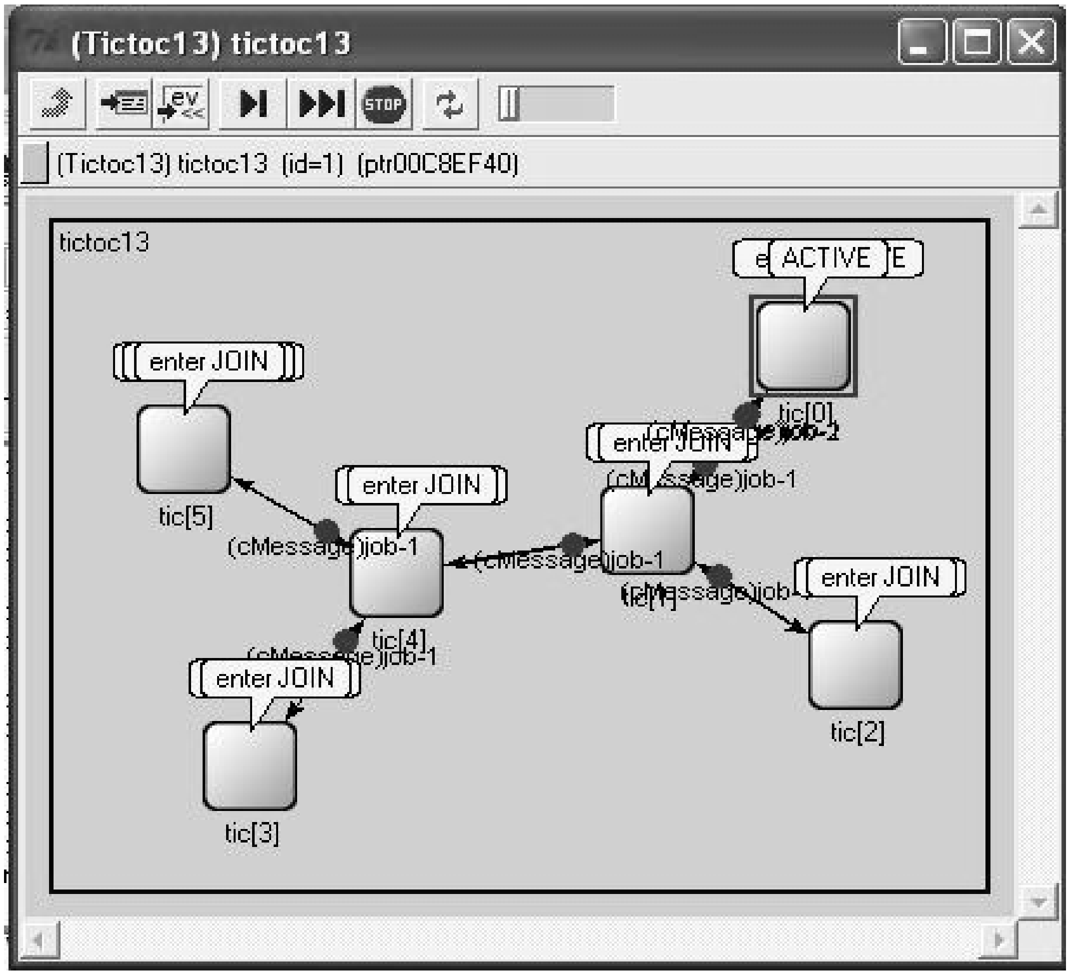
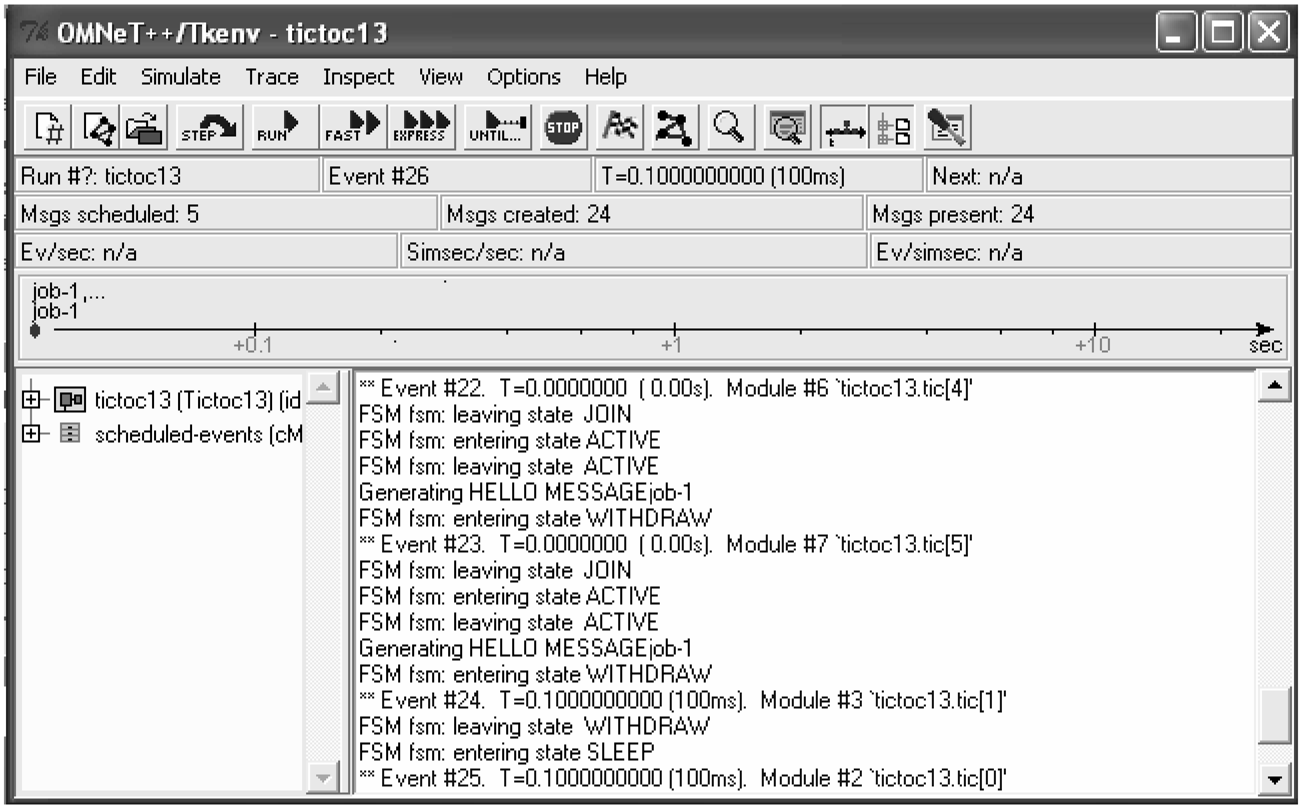


Figure 6.1: Simulation snapshot of different states of wireless sensor network with CCP using OMNET++



```
Run #?: tictoc13    Event #26    T=0.1000000000 (100ms)    Next: n/a
Msgs scheduled: 5    Msgs created: 24    Msgs present: 24
Ev/sec: n/a    Simsec/sec: n/a    Ev/simsec: n/a

** Event #22.  T=0.0000000  ( 0.00s).  Module #6 `tictoc13.tic[4]'
FSM fsm: leaving state  JOIN
FSM fsm: entering state ACTIVE
FSM fsm: leaving state  ACTIVE
Generating HELLO MESSAGEjob-1
FSM fsm: entering state WITHDRAW
** Event #23.  T=0.0000000  ( 0.00s).  Module #7 `tictoc13.tic[5]'
FSM fsm: leaving state  JOIN
FSM fsm: entering state ACTIVE
FSM fsm: leaving state  ACTIVE
Generating HELLO MESSAGEjob-1
FSM fsm: entering state WITHDRAW
** Event #24.  T=0.1000000000 (100ms).  Module #3 `tictoc13.tic[1]'
FSM fsm: leaving state  WITHDRAW
FSM fsm: entering state SLEEP
** Event #25.  T=0.1000000000 (100ms).  Module #2 `tictoc13.tic[0]'
```

Figure 6.2: Output screen with messages of wireless sensor network with CCP protocol using OMNeT++

6.2 OUTPUT OF NETWORK FOR SENSING TEMPERATURE AND VOLTAGE :

When simulation is started, then the numbers of States of nodes of a network (as specified by the parameters) are displayed on the screen and appear to move in different states as shown in Figure 6.3. Here in this simulation, nodes are parameter sensitive that is some node sense the temperature and some voltage. When Temperature is sensed, then temperature-sensing nodes come in active state and exchange the messages.

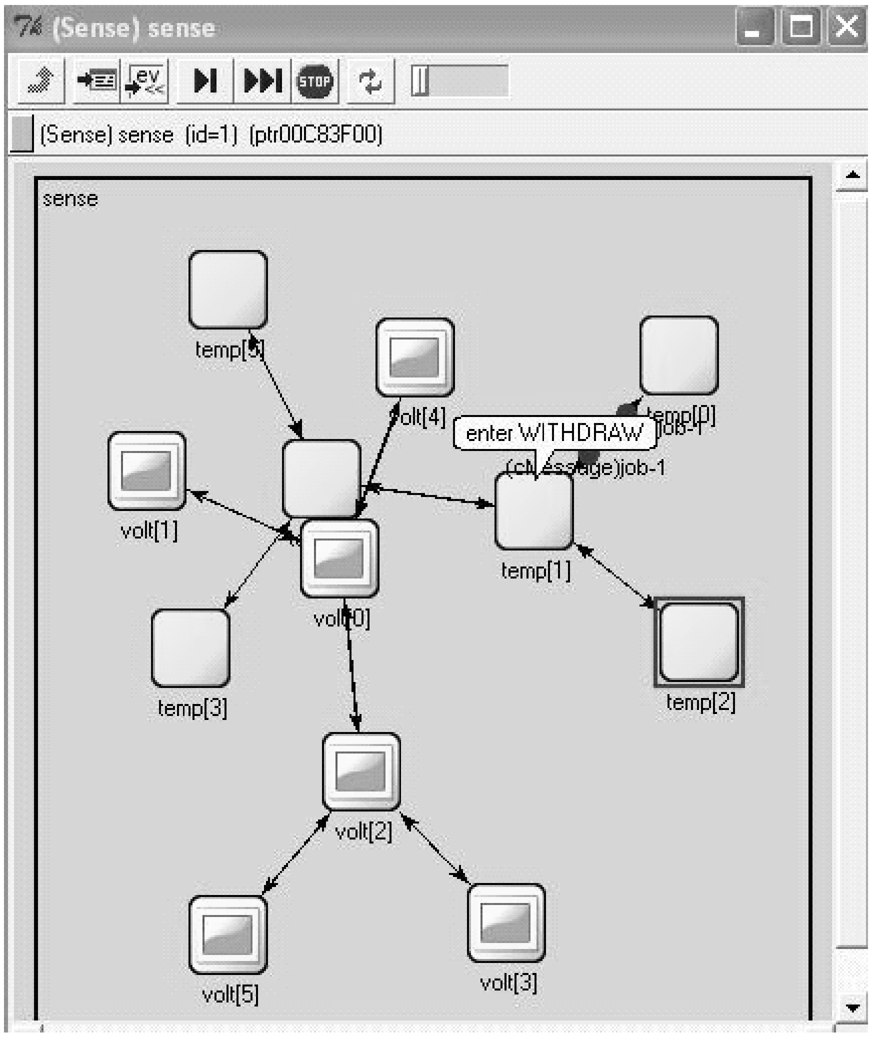
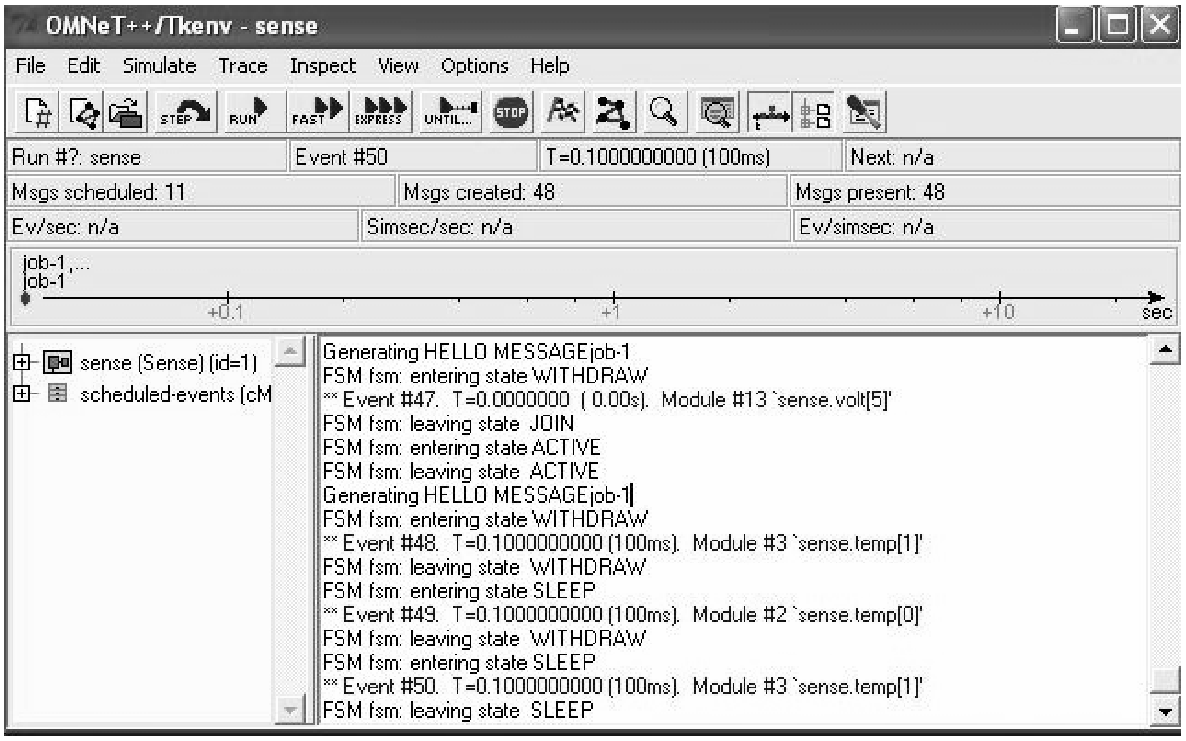


Figure 6.3: Simulation snapshot of different states of wireless sensor network sensing temperature or voltage once at a time with CCP using OMNET++



```
Run #?: sense        Event #50        T=0.1000000000 (100ms)        Next: n/a
Msgs scheduled: 11        Msgs created: 48        Msgs present: 48
Ev/sec: n/a        Simsec/sec: n/a        Ev/simsec: n/a

Generating HELLO MESSAGEjob-1
FSM fsm: entering state WITHDRAW
** Event #47.  T=0.0000000  ( 0.00s).  Module #13 `sense.volt[5]'
FSM fsm: leaving state  JOIN
FSM fsm: entering state ACTIVE
FSM fsm: leaving state  ACTIVE
Generating HELLO MESSAGEjob-1
FSM fsm: entering state WITHDRAW
** Event #48.  T=0.1000000000 (100ms).  Module #3 `sense.temp[1]'
FSM fsm: leaving state  WITHDRAW
FSM fsm: entering state SLEEP
** Event #49.  T=0.1000000000 (100ms).  Module #2 `sense.temp[0]'
FSM fsm: leaving state  WITHDRAW
FSM fsm: entering state SLEEP
** Event #50.  T=0.1000000000 (100ms).  Module #3 `sense.temp[1]'
FSM fsm: leaving state  SLEEP
```

Figure 6.4: Output screen with messages of wireless sensor network sensing Temperature or voltage once at a time with CCP Using OMNeT++

6.3 Output of Network for Selfish Node Detection:

When the simulation stared, then a number of hosts (as specified by the parameters) are displayed on the screen and appear to remain in different directions. When a host comes within the sensing range of another host, they start communicating with each other. As shown in fig.6.3.The coverage of this network follows the rule of CCP protocol.

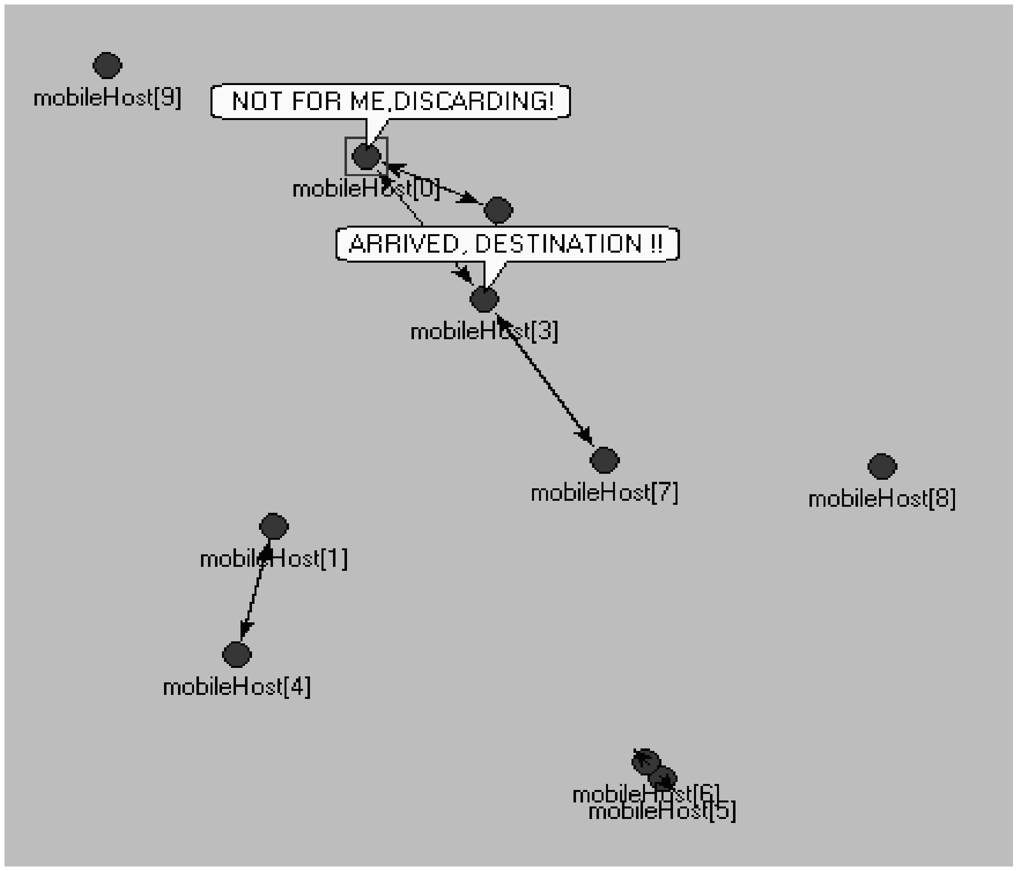
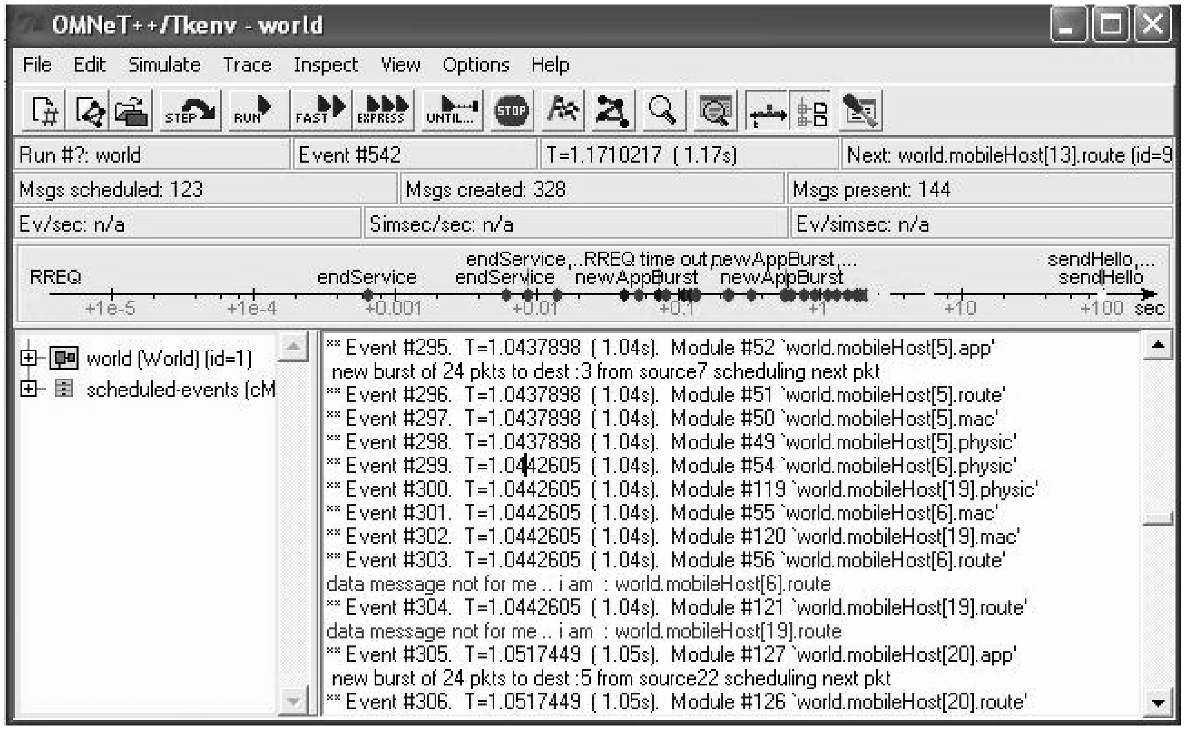


Figure 6.5: Simulation snapshot of selfish node detection in network by following the CCP using OMNET++


OMNeT++/Tkenv - world
File Edit Simulate Trace Inspect View Options Help
STEP RUN FAST EXPRESS UNTIL... STOP
RREQ
endService
endService,..RREQ time out
endService
newAppBurst,...
newAppBurst
newAppBurst
sendHello,...
sendHello
+1e-5 +1e-4 +0.001 +0.01 +0.1 +1 +10 +100 sec
world (World) (id=1)
scheduled-events (cM

```
Run #?: world        Event #542        T=1.1710217 ( 1.17s)        Next: world.mobileHost[13].route (id=9
Msgs scheduled: 123        Msgs created: 328        Msgs present: 144
Ev/sec: n/a        Simsec/sec: n/a        Ev/simsec: n/a
```

```
** Event #295.  T=1.0437898  ( 1.04s).  Module #52 `world.mobileHost[5].app'
 new burst of 24 pkts to dest :3 from source7 scheduling next pkt
** Event #296.  T=1.0437898  ( 1.04s).  Module #51 `world.mobileHost[5].route'
** Event #297.  T=1.0437898  ( 1.04s).  Module #50 `world.mobileHost[5].mac'
** Event #298.  T=1.0437898  ( 1.04s).  Module #49 `world.mobileHost[5].physic'
** Event #299.  T=1.0442605  ( 1.04s).  Module #54 `world.mobileHost[6].physic'
** Event #300.  T=1.0442605  ( 1.04s).  Module #119 `world.mobileHost[19].physic'
** Event #301.  T=1.0442605  ( 1.04s).  Module #55 `world.mobileHost[6].mac'
** Event #302.  T=1.0442605  ( 1.04s).  Module #120 `world.mobileHost[19].mac'
** Event #303.  T=1.0442605  ( 1.04s).  Module #56 `world.mobileHost[6].route'
data message not for me .. i am  : world.mobileHost[6].route
** Event #304.  T=1.0442605  ( 1.04s).  Module #121 `world.mobileHost[19].route'
data message not for me .. i am  : world.mobileHost[19].route
** Event #305.  T=1.0517449  ( 1.05s).  Module #127 `world.mobileHost[20].app'
 new burst of 24 pkts to dest :5 from source22 scheduling next pkt
** Event #306.  T=1.0517449  ( 1.05s).  Module #126 `world.mobileHost[20].route'
```

Figure 6.6 Output screen with messages of wireless sensor network detecting the selfish node

CHAPTER 7

CONCLUSION

CONCLUSION

This Thesis presents a new protocol, Coverage Configuration Protocol, for wireless sensor networks to provide both desired coverage and connectivity.

This Thesis also provides the geometric analysis that

1) proves sensing coverage implies network connectivity when the sensing range is no more than half of the communication range; and

2) quantify the relationship between the degree of coverage and connectivity.

It develops the Coverage Configuration Protocol (CCP) that can achieve different degrees of coverage requested by applications, and also reduce the energy consumption by allowing the nodes of network to go in SLEEP and LISTEN modes most of the time when they do not sense anything. This flexibility allows the network to self-configure for a wide range of applications and (possibly dynamic) environments.

CHAPTER 8

APPLICATION & FUTURE WORK

APPLICATION & FUTURE WORK

Wireless sensor networks have a wide range of applications. They offers their networked wireless systems across a broad range of applications, including industrial automation, building automation, security, home automation, consumer, medical and transportation. Sensing coverage and communication coverage are two fundamental qualities of service for wireless sensor networks. In this Thesis, work on energy efficient sensing coverage and communication are presented. Several schemes for sensing coverage subject to different requirements and constraints are designed respectively. It also propose a broadcasting communication protocol with high energy efficiency and low latency for large scale sensor networks based on the Small World network theory.

In future, Study of CCP will extend the solution to handle more sophisticated coverage models and connectivity configuration and develop adaptive coverage reconfiguration for energy-efficient distributed detection and tracking techniques.

REFERENCES

WEBSITES:

http://www.cse.wustl.edu/
http://www.csc.lsu.edu/~iyengar/index.html
http://www.csc.lsu.edu/~iyengar/publications.html#papers
http://www.cse.wustl.edu/~lu/papers/sensys03_ccp.pdf
http://www.monarch.cs.cmu.edu

PAPERS:

[1] Gouliang Xing, Xiaorui Wang, Yuanfang Zhang, Chenyang Lu, Robert Pless and Christopher Gill, “Integrated coverage and connectivity configuration in wireless sensor networks”, Washington University in St. Louis.

[2] A. Cerpa and D. Estrin, "ASCENT: Adaptive Self- Configuring Sensor Networks Topologies," INFOCOM, June 2002.

[3] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks," ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001), Rome, Italy, July 16-21, 2001

[4] Y. Xu, J. Heidemann, and D. Estrin, "Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks," Research Report 527, USC/Information Sciences Institute, October 2000.

[5] Y. Xu, J. Heidemann, and D. Estrin, "Geography-informed Energy Conservation for Ad Hoc Routing," ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2001), Rome, Italy, July 16-21, 2001.

[6] Chen., J. Branch, M. J. Pflug, L. Zhu and B. Szymanski SENSE: A Sensor Network Simulator. Advances in Pervasive Computing and Networking. B. Szymanksi and B. Yener, Springer: 249-267 (2004)..

[7] Kevin Fall, Kannan Varadhan, Editors, The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, the ns Manual.

[8] OPNET Technolgies, Inc. OPNET Modeler.

[9] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang, J-Sim: A Simulation and Emulation Environment for Wireless Sensor Networks.

[10] David Cavin, Yoav Sasson, Andre Schiper, Distributed Systems Laboratory, On the accuracy of MANET simulators.

[11] P. Hall, Introduction to the Theory of Coverage Processes. John Wiley & Sons Inc., New York.

[12] S. Meguerdichian and M. Potkonjak. "Low Power 01 Coverage and scheduling Techniques in Sensor Networks." UCLA Technical Reports 030001. June 2003.

[13] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. Srivastava, "Coverage Problems in Wireless Ad-Hoc Sensor Networks." INFOCOM'01, Vol 3, April 2001.