Summer 2005

Mobility-Pattern Based Localization Update Algorithm for Mobile Wireless Sensor Networks

Mohammad Yacoub Al-laho
Old Dominion University

Mobility-Pattern Based Localization Update Algorithm for Mobile Wireless Sensor Networks

By

Mohammad Yacoub Al-laho

B.S. (Cp.E) December 2003, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
AUGUST 2005

Approved by:

Min Song (Director)

Vijayan K. Asari (Member)

Frederic D. McKenzie (Member)

ABSTRACT

Mobility-Pattern Based Localization Update Algorithm for Mobile Wireless Sensor Networks

Mohammad Yacoub Al-laho
Old Dominion University, August 2005
Director: Dr. Min Song

In mobile wireless sensor networks, sensors move in the monitored area at any direction and speed. Unlike many other networking hosts, sensor nodes do not have global addresses. They are often identified by using a location-based addressing scheme. Therefore, it is important to have the knowledge of the sensor location indicating where the data came from. In this thesis, three localization update algorithms were designed. Specifically, a sensor movement is divided into three states: *Pause*, *Linear*, and *Random*. Each state adopts different localization update algorithm. Since complex movement involves different mobility patterns, a state transition model is developed to accommodate the transition among the three algorithms. This design is called Mobility-pattern Based Localization Update Algorithm.

Simulation results and analysis are provided to study the localization update cost and location accuracy of the proposed mobility-pattern based design. The simulation is developed to accommodate the three different mobility patterns. The analysis to these results indicates that the localization update cost is minimized and the location accuracy is improved.

To my mom, dad and family,

I thank you for your unconditional love, encouragement and support.

ACKNOWLEDGEMENTS

I would like to thank Dr. Min Song for his continuous support and guidance to complete this thesis. I would also like to thank Dr. Vijayan K. Asari and Dr. Frederic D. McKenzie for consenting to be on the thesis advisory committee. I would like to thank my Network Lab colleagues for their help and support and my colleague Jun Wang for his great help and assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Introduction

Wireless sensors have been a hot topic due to its flexibility of gathering information of any kind anywhere. Wireless Sensor Networks (WSNs) consist of densely distributed sensor nodes. Sensor nodes have communication capability, storage and processing resources including sensing capability [5, 16]. Since the quantity of these sensor nodes is generally large and it is hard to change batteries for each node, it is considered that the nodes should have very efficient power consumption. Therefore, the limit of power consumption forces sensor nodes to have short-range transmission and low average bit-rate communication. It is also considered that sensors do not have large computational capabilities due to the power consumption limit and the small size of a node, which is approximately 1-4 cm^3. Example of sensor node's size is shown in figure 1.

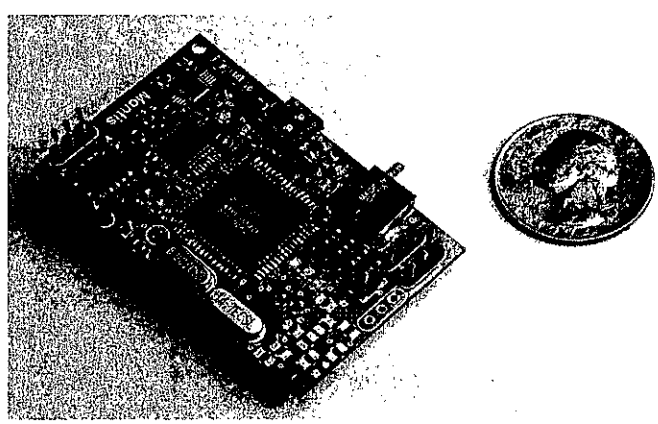


Figure 1: Sensor node size example.

Wireless sensor networks are considered either stationary or mobile. In stationary wireless sensor networks, nodes do not move and hence their locations do not change. On the other hand, in Mobile wireless Sensor Networks (MSNs)*, sensors can move freely in the monitored area at any direction and at any speed. Locating a mobile sensor node in a network is more challenging and needs more aggressive algorithm. MSNs can be used in many useful applications such as military, environmental, health, commercial or home applications [5]. The following only point out a few application examples of MSNs.

Military Applications:

- Monitoring friendly forces, equipment and ammunition.
- Battlefield surveillance.
- Reconnaissance of opposing forces and terrain.
- Battle damage assessment.

Environmental Application:

- Forest fire detection.
- Flood detection.

Health Applications:

- Tracking and monitoring patients and doctors inside a hospital.
- Drug administration in hospitals.

Commercial Applications:

- Managing inventory control.

* MSN is called to specify mobile sensor nodes, while WSN is a more general naming for both mobile and stationary sensor nodes network.

- Vehicle tracking and detection.

The subject matter of this thesis is how and when to locate a mobile sensor node in a network. The localization schemes available in literature are useful for stationary type of wireless sensor networks. The location detection techniques available for the MSNs use techniques with either pure prediction or prediction combined with localization update from other sources in the network. Prediction schemes tend to anticipate the next possible step of a node based on a movement pattern model or movement pattern history of a node.

1.2 Location Importance for Sensor Nodes

Since almost all the MSNs applications require location awareness, tracking sensor nodes' location becomes a large interest. Sensor nodes do not have global addresses. Though, sensor nodes are identified by using a location-based addressing scheme [5]. For example, "temperatures read by the nodes in region A" is an example for location-based naming. The routing process requires location-based naming where the users are more interested in querying a location of a phenomenon, rather than querying an individual node [5]. Therefore, it is important to have the knowledge of the sensor location indicating where the data came from.

1.3 Localization Problem

The method of detecting the current location of an object is called localization scheme. Many researches have been conducted to solve the localization problem of

stationary wireless sensor nodes but few concern localization of mobile nodes. For stationary localization schemes, there are generally two steps involved in the localization process and a third is optional. The first step is to measure the distance between a beacon transmitter and a beacon receiver. This step usually involves using one of the following methods to calculate ranges: Time of Arrival (ToA), Time Difference of Arrival (TDoA), Angle of Arrival (AoA), and Received Signal Strength Indicator (RSSI) [12, 13, 14, 15]. The second step uses the range measurements to estimate the location of an unknown sensor in a network. Three major methods are used in step two, Triangulation, Trilateration, and Multilateration [12, 13, 14]. The third step could be a refining process to assure more accuracy. For the mobile sensor nodes, a localization method is performed frequently when a mobile node moves. Apparently, the more the localization scheme is performed, the more accurate the node's location gets. However, increasing the number of localization updates consumes more power and consequently shortening the effective life of the network. Note that in this thesis, the localization scheme is not the focus of the research. Instead, the research focuses on when to use the localization scheme and how to estimate a node's current location for the period of time between one localization-update and another.

A known localization system is the Global Positioning System (GPS). However, the GPS is not a good choice for MSNs for the following reasons:

- GPS cannot work indoors or in the presence of dense foliage or other obstacles that block the line-of-sight from the GPS satellites.

- The power consumption of GPS receivers reduces the battery life of the sensor nodes and hence reduces the effective lifetime of the entire network.
- It is impossible to equip each sensor node with GPS receiver from economic perspective when the number of sensor nodes in the network is large.
- The size of GPS receivers restricts the sensor nodes from having a small shape form [7].

In another method, the localization of mobile wireless sensors can be performed by using prediction techniques that anticipate the possible next step of a sensor node based on the movement pattern model and the history of the movement pattern [4]. However, the prediction method is rather too complex and needs many calculations and filtering techniques to refine the possible sampling data collected from previous movements. The energy consumption and the memory size restrictions of a sensor node limit the use of the prediction methods. In addition, it has been found that prediction techniques are not always accurate [1]. The following are some reasons that cause inaccurate predictions:

- The developed model can be inaccurate – the sampled points may not be sufficient to discover the mobility pattern.
- We may assume an inappropriate mobility model (e.g., assuming that the node is moving at constant velocity when it has an acceleration component).
- The localization methods may introduce some error in the computed localization points.

- Sensors will typically not follow a predictable model (e.g., there may be unpredictable changes of directions or pauses that will cause the predicted model to go wrong) [1].

Another problem is that mobile localization schemes in literature over-simplify the mobility pattern and they do not work well when nodes move randomly. The more the mobility pattern deviates from the linear model, the more inaccurate the location accuracy gets.

1.4 Objectives and Proposal

The objective of this thesis is to overcome the localization problems mentioned above, and to propose a localization scheme that is confined with the sensor node design aspects like limited power sources. To avoid pure prediction methods, a more reliable method is to use localization update* algorithm combined with prediction techniques. In this case, the localization update will correct any wrong anticipation.

To solve the problem of complex mobility pattern, we feel the need to differentiate among the different mobility patterns in the system. The notion here is to use different localization update schemes for different mobility patterns. In [2] it is found that mobility patterns can be divided into three kinds *Pause*, *Linear* and *Random* as shown in Figure 2.

* Localization update in this context means updating the current location of a sensor node using a localization method.

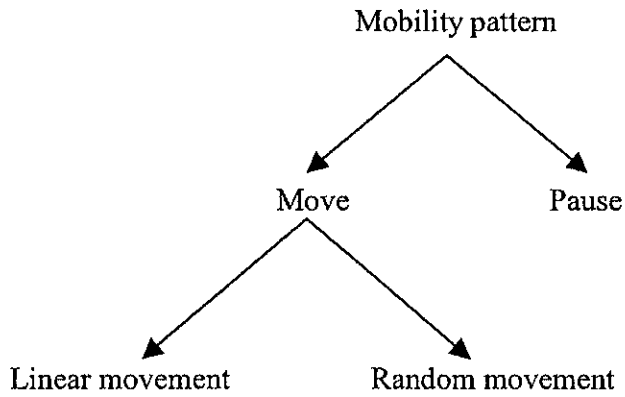


Figure 2: The classification of mobility patterns.

The division of mobility patterns helps dealing with each kind of mobility pattern type alone. So for each type, a different localization-update scheme is used to get the most accurate results.

Moreover, a key design issue in WSNs is simplicity. Keeping the algorithm as simple as possible should be in mind since wireless sensor nodes have limited power consumption and computational capability. The small size of a wireless sensor node limits the size of memory and processor unit.

In this thesis, the topic of location detection of MSNs is investigated. Localization-update algorithms are used in a mobility pattern model where movements are divided into three states. Each state adopts different localization update algorithm. The purpose of the localization update algorithms is to control the period of localization where a sensor node calculates its current location. Also for each state, location prediction methods are used except for the *Pause* state since a node is assumed to be in

the same location. Finally, the goal is to study the tradeoff between localization update cost (the frequency of localization updates performed in a period of time) and the location accuracy errors caused by the prediction techniques. Another primary goal is to reduce the localization updates frequency while keeping the accuracy error low.

CHAPTER II

RELATED WORK

2.1 Introduction

In MSNs, sensors' mobility is considered. As a sensor node moves, it needs to frequently locate itself in the network. The more the sensor node moves, the more frequent it needs to localize itself. Many researches have been conducted to address location detection of stationary sensor nodes, but few works were conducted to enhance location detection of mobile sensor nodes. Some techniques use probabilistic methods to anticipate the mobility pattern of a moving node. In [4], location prediction is performed using two steps, prediction and filtering. In the prediction step, a node draws a set of possible locations that are computed using the previous locations' history and the node's mobility model. The possible location points' set is confined in a circle where the node's previous location is the origin of the circle and the maximum speed is the radius. In the filtering step, a node begins to eliminate certain location points based on beacons* received from neighboring nodes. Lost beacons from certain neighboring nodes indicate that the mobile node is moving away from a certain area, and receiving beacons from other nodes indicate that the mobile node is moving toward another area. Connectivity is very important for this method. Moreover, this method needs a lot of calculations to draw possible locations for each step. Some other prediction methods use linear anticipation calculation, which is known as Dead Reckoning (refer to section

* A beacon in this context is a radio signal transmitted to other nodes with specific data. Usually holds location data.

2.4.1 for more information). Dead reckoning is suitable for linear movement because it does not take into account direction change or acceleration.

For almost all MSNs localization methods, it is important to update the current location to correct any erroneous prediction. The question is when to update the location of a node. The following sections describe three methods for controlling localization periods in MSNs.

2.2 Static Fixed Rate (SFR)

In SFR [1] method, the localization is carried out periodically with a fixed time period t. The energy consumption of the SFR method is independent of mobility. However, SFR accuracy error or performance varies with the mobility of the sensor node. If the sensor node is moving quickly, the error will increase, and if the sensor node is moving slowly, the error will be low.

2.3 Dynamic Velocity Monotonic (DVM)

At this method, a sensor node adapts its localization period as a function of its mobility speed. The simple concept of DVM [1] is that the schedule of when to localize a sensor node depends on the sensor node's velocity. A sensor node measures its speed as it moves by calculating the distance between the current location point and the previous location point and divides it by the elapsed time. Based on the speed value, a node schedules the next time to localize. A parameter named α is set to represent the target maximum error. For each localization measurement, the speed value is compared

with α to estimate how much time needed to reach the target maximum error if the node continues on the same speed. The next localization is scheduled after that time period. Note that a constant velocity is assumed between the two points. This could affect the scheduling time for localization such that a node could reach values of error higher than the target maximum error (threshold error α). In addition, for low speeds, the localization period could be too long. On the other hand, with high speeds, the localization period may be computed to be very short that may drain the power supply fast. To accommodate for these effects, DVM assigns an upper and lower limits for localization periods. This way the performance does not deviate from the desired results.

2.4 Mobility Aware Dead Reckoning Driven (MADRD)

In this method, MADRD [1] tends to predict mobility of nodes in the network using the Dead Reckoning Model (DRM). Based on mobility prediction, the localization periods can be reduced significantly. The idea in MADRD is that as more accurate the mobility prediction gets as less frequent localization updates are performed. To understand MADRD, Dead Reckoning technique has to be introduced.

2.4.1. Dead Reckoning Model (DRM)

DRM [3, 8] predicts mobility based on previous location information. It first calculates the velocity component v_x and v_y along the X and Y axis from two successive

location samples (x_{t_0}, y_{t_0}) and $(x_{t_{-1}}, y_{t_{-1}})$ taken at times t_0 and t_{-1}, where t_0 denotes current time and t_{-1} denotes previous time, thus

$$v_x = \frac{x_{t_{-1}} - x_{t_0}}{t_{-1} - t_0} \tag{1}$$

$$v_y = \frac{y_{t_{-1}} - y_{t_0}}{t_{-1} - t_0} \tag{2}$$

Having the velocity components, we can obtain (x_{t_1}, y_{t_1}), which is the next location coordinates at t_1.

$$x_{t_1} = x_{t_0} + (v_x \times (t_1 - t_0)) \tag{3}$$

$$y_{t_1} = y_{t_0} + (v_y \times (t_1 - t_0)) \tag{4}$$

In MADRD, after $(t_1 - t_0)$ time period (x_{t_1}, y_{t_1}) is compared to the location coordinate obtained from a localization method. The comparison is done by calculating the Euclidean distance between the predicted location and the real location obtained from localization method. The Euclidean distance d is obtained using equation (5).

$$d = \sqrt{(x_{pred} - x_{real})^2 + (y_{pred} - y_{real})^2} \tag{5}$$

Where (x_{pred}, y_{pred}) is the predicted location coordinate and (x_{real}, y_{real}) is the real location coordinate. In MADRD, based on a previously determined threshold error, d is compared to a threshold error value. If the difference is too large that it exceeds a threshold error then the process is moved to a low confidence state where localization is

carried out so often. If the difference is small and below a threshold error then the process moves into a high confidence state where localization is less frequently performed.

2.5 Summary

An overview of selected research works in the field of localization for MSNs was given. All the research works reviewed use localization update to correct any location erroneous predictions. Three localization update control schemes were introduced: Static Fixed Rate (SFR), Dynamic Velocity Monotonic (DVM) and Mobility Aware Dead Reckoning Driven (MADRD). The three protocols are good for linear mobility pattern but they are not suitable for complex mobility patters. A prediction scheme is mentioned that uses sample sets and probabilistic calculations to predict sensor nodes locations. However, this method has too many calculations involved and is rather too complex for MSNs.

CHAPTER III

THE SYSTEM DESIGN

3.1 Introduction

The main objective in designing the algorithm is to manage the localization-update period in optimal rates while keeping the location accuracy error low. There are two parameters to take care of, the cost of update frequency and the node's location accuracy. It is expected that when the location of a node is less updated, the update cost will be low while location accuracy error will be high. However, when the localization update frequency is increased, the update cost will be high and the location accuracy error will be low. The algorithm should keep localization update cost low and location accuracy error low as well. Using one update policy is not enough for complex mobility pattern. [1] uses dead reckoning for prediction, and localization updates are carried out based on how well the prediction is. In [1], it is found that when the movement pattern is not predictive (not linear) the performance decreases significantly. In this thesis, the idea in [2] is adopted where mobility pattern is divided into three types: *Pause*, *Linear*, and *Random* as shown in Figure 2. Different localization update policies are used for different mobility patterns. There are four localization update schemes in literature:

- Time-based: updates are made at fixed time intervals.
- Movement-based: updates are made whenever the number of cell-boundary crossings since the last update exceeds a specified threshold.

- Distance-based: a mobile updates its location whenever its distance from an expected location exceeds a specified value.
- Dead reckoning: a method of predicting a location based on velocity components calculated from previous locations.

Some of the techniques above are not suitable for WSNs since they usually use the knowledge of cells count like in the Movement and Distance based update methods. [9, 10] talk in more details about update schemes.

3.2 Mobility-Pattern Based Localization Update Algorithm (MBLUA)

In the MBLUA, the Time-based scheme is used for the *Pause* state since there is no distance or movement involved in a pause. However, the scheme is changed. In the original Time-based method, the period of localization update is fixed. In our algorithm the localization period is increased as the node's pause time increases. To prevent a long wait time to localize, there is a maximum period of wait time that cannot be exceeded. The motive behind increasing the localization period as a node pause longer is to save more energy. Dead reckoning scheme is most suitable for linear mobility pattern (the *Linear* state) since it works for predictive movement pattern where there is no change of velocity or unpredicted change of direction. The localization period is incremented as long as the distance between the predicted location and the real location (the error) is within a predefined threshold. The localization period is set to the initial localization period if the error exceeds the threshold value. The most suitable policy for the random movement pattern (the *Random* state) is the distance-based scheme. The distance-based scheme used in MBLUA is slightly different than the one in [2, 17]. In

[2, 17] the distance is between the last update in the last cell and the current location in the current cell. Therefore, the distance is actually the number of cells. In wireless sensor networks the concept of cells does not exist. The idea in this design is not to actually measure the distance between last update and current position, but rather to predict when the node will cross the limit of this distance based on the acceleration value measurement of the node. The acceleration is to be measured periodically as the node localizes using equation (6)

$$a = \frac{v_{cur} - v_{prev}}{t_{cur} - t_{prev}} \text{ m/s}^2 \tag{6}$$

where v_{cur} is the current velocity and v_{prev} is the previous velocity, and $t_{cur} - t_{prev}$ is the elapsed time. The localization period is then measured using equation (7)

$$\hat{t} = \sqrt{\frac{d_{thresh}}{|a|}} \tag{7}$$

where *d_thresh* denotes the distance limit to localize; it works like the parameter α in the DVM method in [1]. $\hat{t}$ is an estimate time because we assume a constant acceleration during that time, meanwhile acceleration could change in real movement. After the node localizes immediately, *d_thresh* will be the radius of a circle where the node is on the origin of this circle as shown in Figure 3. A larger *d_thresh* value indicates a larger $\hat{t}$ value and a smaller *d_thresh* value indicates a smaller $\hat{t}$ value regardless of the acceleration value. The value of *d_thresh* will depend on the application. Applications that need more accuracy will have smaller *d_thresh*. Meanwhile, applications that do not require location accuracy will have larger *d_thresh* value.

For a simulation of discrete time intervals, $\hat{t}$ should be converted to discrete time intervals by using equation (8)

$$n = \left\lceil \frac{\hat{t}}{timeslot} \right\rceil \tag{8}$$

where n is the number of time intervals, and *timeslot* is a constant equal the discrete time interval period. During the n time intervals the locations are estimated using equations (9) and (10) that uses dead reckoning plus the a (acceleration) factor. Since a equals to the increment or decrement of velocity per second, a is multiplied by the *timeslot* period to get the increment or decrement in velocity vector for each time interval, $\hat{v} = a \times timeslot$.

$$x_t = x_{t-1} + [(v_x + \hat{v})(timeslot)] \tag{9}$$

$$y_t = y_{t-1} + [(v_y + \hat{v})(timeslot)] \tag{10}$$

Where x_t and y_t are the new estimated position coordinates, x_{t-1} and y_{t-1} are the previous position coordinates, and v_x, v_y are the velocity components.

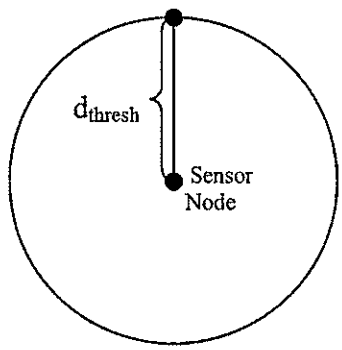


Figure 3: *d_thresh* in the distance-based method.

A limitation of this method is that it does not take into account direction changes as it is in random movement during the localization period. The only consideration here is the acceleration factor. Between each localization update and another, the direction is assumed to be the same. If location information is required during the localization period, before crossing the *d_thresh* limit, current location estimate is obtained from equations (9) and (10). The *d_thresh* value is meant to keep the error tolerable. As a matter of fact, assuming the mobile node to travel in a straight line is the worst-case scenario since the traveled distance will be exactly equal to the *d_thresh* value and may cross it if acceleration is increased during the localization period. Meanwhile, if the mobile node changes its direction, then it will stay within the circle of radius *d_thresh* where error is tolerable. This approach is considered to be conservative and should fit any application by adjusting the *d_thresh* value.

3.3 The System Model

Each mobility pattern presents a state. Conditions to move from one state to another will be tested in each state so the update policy is changed with the state change. Figure 4 shows the initial and the three mobility pattern states.

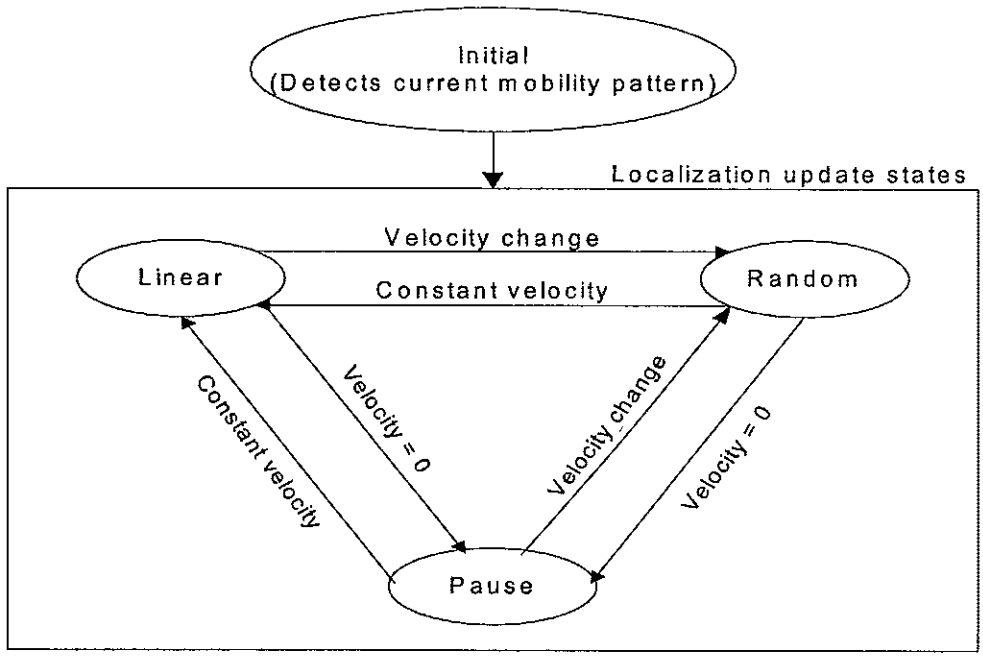


Figure 4: The Mobility Pattern State Transition Diagram.

In the MBLUA, the main test to transit from state to another is based on velocity information. A transition to the *Pause* state occurs when the velocity equals to zero, which is equivalent to detecting a sensor node in the same location for two times in a row. To transit to *Linear* state, velocity should be the same for certain time, which is constant velocity or equivalently acceleration value equals to zero. Finally, transition to the *Random* state occurs when changing in the velocity value is detected, which indicates random mobility pattern. The following section presents the pseudo code for each update policy that will be run in each state including the initial state that detects the current mobility pattern.

3.3.1. The Three States and Initial State Algorithms

The following sections introduce the algorithms in pseudo code to be run in each mobility pattern state including the initial state.

3.3.1.1 Initial State Algorithm:

The Initial state algorithm will be run one time in the beginning of each start. The purpose of this state is to detect the mobility pattern of the sensor node and to initialize the initial update time (*Init_updatetime*), linear threshold (*Lin_thrish*), distance threshold (*d_thresh*), and the maximum waiting time (*max_time*). The *init_updatetime* specifies the initial update time to wait when moving to a new state. Usually the *init_updatetime* equals zero, which means updating each time slot. The *Lin_thrish* is the error threshold for the *Linear* state. *Lin_thrish* equals the tolerable accuracy error (distance error) between the predicted location and the real location. The *d_thresh* is the distance threshold for the *Random* state. *d_thresh* is shown in figure 3. *max_time* is the maximum discrete time periods to wait for the next localization. *max_time* insures that a sensor node does not wait for a long time to localize and hence keeping the accuracy error level low.

Initial state pseudo code:

1. Define *Init_updatetime* // one time slot
2. Define *max_time* //maximum time a node can wait to update again
3. Define *Lin_thrish*
4. Define *d_thresh*
5. Main
6. *loc1* ← loc_update // (x_1, y_1)
7. Wait for *Init_updatetime*
8. *loc2* ← loc_update // (x_2, y_2)

```
9.    calculate v1                    // √((x1 − x2)² + (y1 − y2)²) / (t2 − t1)
10.   If v1 = 0                       // means loc1 = loc2
11.         Then state ← Pause state (t, v, loc)
12.   Else
13.         Wait for Init_updatetime
14.         loc3 ← loc_update         // (x3, y3)
15.         calculate v2              // √((x2 − x3)² + (y2 − y3)²) / (t3 − t2)
16.         If v1 = v2
17.               Then state ← Linear state (t, loc_{t-1})
18.         Else
19.               state ← Random state (t, v_{t-1}, loc_{t-1})
```

3.3.1.2 Pause State Algorithm:

In the Pause state algorithm, a sensor node is assumed to be stationary. The algorithm increases the update time period as long as the sensor node's velocity is zero. The update time period is increased until the *max_time* to avoid infinite waiting time in case of long stationary periods.

Pause state pseudo code:

```
1. Pause state (t, v_{t-1}, loc_{t-1})    // t, v_{t-1} and loc_{t-1} from the previous state
2.     update time ← Init_updatetime
3.  Loop
4.     Wait for update time
5.     loc_to ← loc_update                // Localize
6.     Calculate v_to                     // from loc_{t-1} and loc_to
7.     If v_to != 0 for two consecutive times
8.           Then If v_{t-1} != v_to
9.                 state ← Random state
10.          Else
11.                state ← Linear state
12.    If update_time < max_time
13.          Then update_time ++
14. Repeat
```

3.3.1.3 Linear State Algorithm:

In the Linear state algorithm, a sensor node movement is assumed to be linear which having a constant velocity. At first a node will localize to know its current location. Second, the node will predict its next location using the dead reckoning equations (1), (2), (3), and (4). After the *update_time* period, the node will localize to obtain its real location, and the real location is then compared with the predicted location. If the distance error is within the *Lin_thresh*, the update time is increased. However, if the distance error exceeds the *Lin_thresh*, the update time is set to the *Init_updatetime*.

Linear state pseudo code:

1. Linear state(t, loc_{t-1}) // t and *loc* from the previous state
2. *update_time* ← *Init_updatetime*
3. loc_{to} ← loc_update // (x_{t_o} , y_{t_o}) Localize
4. calculate v_{t-1} // form loc_{t-1} and loc_{to}
5. Loop
6. compute velocity components vx, vy // $v_x=(x_{to}-x_{t-1})/(\Delta t)$, $v_y=(y_{to}-y_{t-1})/(\Delta t)$
7. perform prediction (dead reckoning) // (x_{pred}, y_{pred})
8. Wait for *update_time*
9. Loc_{to} ← loc_update // (x_{t_o} , y_{t_o})
10. calculate v_{to} // v_{t-1} is saved from last calculation
11. If v_{to} != v_{t-1}
12. Then If v_{to} = 0
13. state ← Pause state
14. Else
15. state ← Random state
16. compute d between predicted loc and real loc
17. If d < *Lin_thresh*
18. Then *update_time* ++
19. Else
20. *update_time* ← *Init_updatetime*
21. Repeat loop

3.3.1.4 Random State Algorithm:

In the Random state algorithm, a sensor node movement is considered to have velocity acceleration and the node is expected to change direction. In the algorithm, a sensor node will first localize to obtain its current location then the estimated time $\hat{t}$ is calculated. A node will wait for $\lceil \hat{t}/(timeslot) \rceil$ time periods until it localizes again. During that time a node predicts its current location using equations (9) and (10).

Random state pseudo code:

1. Random state(t, v_{t-1}, loc_{t-1}) // t, v and loc from the previous state
2. $update\_time \leftarrow Init\_updatetime$
3. Loop
4. $loc_{to} \leftarrow$ loc_update // (x_{t_o}, y_{t_o}) localize
5. calculate v_{to} // form loc_{to} and loc_{t-1}
6. calculate a // acceleration form v_{to} and v_{t-1}
7. If $v_{to} = v_{t-1}$
8. Then If $v_{to} = 0$
9. state ← Pause state
10. Else
11. state ← Linear state
12. calculate $\hat{t}$ where d_thresh is to be crossed // $\hat{t} = \sqrt{d_{thresh}/|a|}$
13. calculate $\hat{v}$ // $\hat{v} = a \times timeslot$
14. For($update\_time = \lceil \hat{t}/(timeslot) \rceil$, $update\_time > 0$, $update\_time$ --)
15. { compute velocity components $v_x + \hat{v}$, $v_y + \hat{v}$
16. perform prediction // (x_{pred}, y_{pred}) }
17. Repeat Loop

3.4 State Transition Model

The previous update algorithms are mobility pattern dependent. The State-based Mobility Model (SMM) in [2] is adopted in order to analyze the presented algorithms

over three mobility pattern states. SMM is based on the three mobility patterns *Pause*, *Linear*, and *Random* so its states set is S = {P ≡ Pause, L ≡ Linear, R ≡ Random}. Figure 5 shows an instance of SMM model.

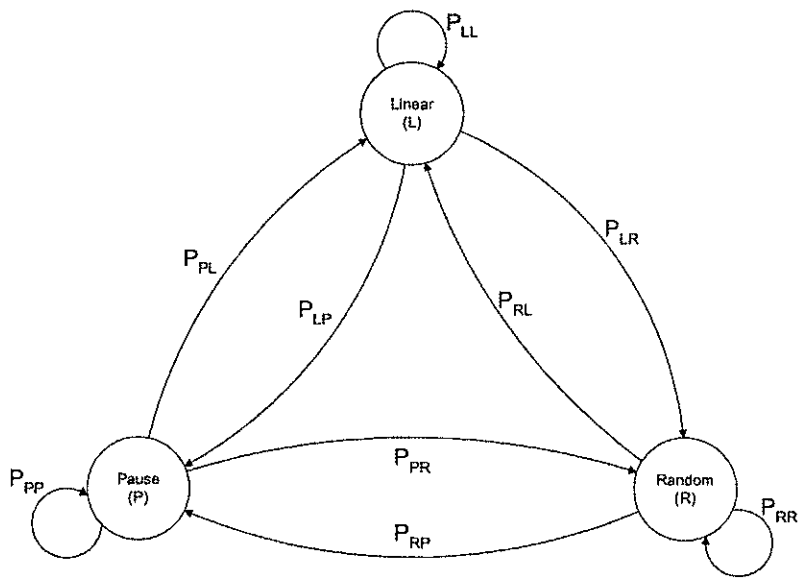


Figure 5: An instance of SMM where P, L, and R ∈ *S*.

From the above model we can derive the steady-state transition probability vector, π, for each state. First, from the balance equations we get (11) and (12):

$$\pi_L(P_{LP} + P_{LR}) = \pi_P P_{PL} + \pi_R P_{RL} \tag{11}$$

$$\pi_R(P_{RP} + P_{RL}) = \pi_P P_{PR} + \pi_L P_{LR} \tag{12}$$

From (11) and (12) we get

$$\pi_L = \frac{\pi_P P_{PL} + \pi_R P_{RL}}{P_{LP} + P_{LR}} \tag{13}$$

$$\pi_R = \frac{\pi_P P_{PR} + \pi_L P_{LR}}{P_{RP} + P_{RL}} \tag{14}$$

Since $\sum_i \pi_i = 1$ then

$$\pi_P + \pi_L + \pi_R = 1 \tag{15}$$

Using equations (13), (14), and (15) we get π for each state:

$$\pi_P = \frac{P_{LP}P_{RP} + P_{LP}P_{RL} + P_{LR}P_{RP}}{(P_{PR} - P_{LR})(P_{LP} + P_{LR} + P_{RL}) + (P_{LP} + P_{LR} + P_{PL})(P_{RP} + P_{RL} + P_{LR})} \tag{16}$$

$$\pi_R = \frac{P_{PR}P_{LP} + P_{PR}P_{LR} + P_{LR}P_{PL}}{(P_{PR} - P_{LR})(P_{LP} + P_{LR} + P_{RL}) + (P_{LP} + P_{LR} + P_{PL})(P_{RP} + P_{RL} + P_{LR})} \tag{17}$$

$$\pi_L = \frac{P_{PR}P_{RL} + P_{RP}P_{PL} + P_{RL}P_{PL}}{(P_{PR} - P_{LR})(P_{LP} + P_{LR} + P_{RL}) + (P_{LP} + P_{LR} + P_{PL})(P_{RP} + P_{RL} + P_{LR})} \tag{18}$$

Equations (16), (17), and (18) are used to calculate the total performance of the SMM model in some scenario as will be shown in section 3.5.

To see how the equations work we will consider the following example. This example should be suited for our mobility model that uses Random Waypoint as an upper level and Gaussian Markovian as a lower level as will be explained in section 3.6. For some system, transitions from state to state are monitored for 105 time slots as following:

PPPPPPRRRRRRRRLLLLLLLLLLLLRRRRRRPPPPPPPPLLLLLLLLLLLLL

LPPPPLLLLLLRRRRRRRRLLLLLRRRRRRRRRPPPPPPRRRRRRRPPPPPP

Counting the number of transitions N_{ij}^2 from state i to state j gives

$$[N_{ij}]_{i,j\in S} = \begin{array}{c} \\ P \\ L \\ R \end{array} \begin{array}{c} \begin{array}{ccc} P & L & R \end{array} \\ \begin{pmatrix} 26 & 2 & 2 \\ 1 & 32 & 3 \\ 3 & 2 & 33 \end{pmatrix} \end{array} \tag{19}$$

The transition probability P_{ij} can be estimated as:

$$\hat{P}_{ij} = \frac{N_{ij}}{\sum_{k\in S} N_{ik}} \tag{20}$$

Calculating each transition probability we get the following matrix:

$$\hat{P} = \begin{pmatrix} \frac{26}{(26+2+2)} & \frac{2}{(26+2+2)} & \frac{2}{(26+2+2)} \\ \frac{1}{(1+32+3)} & \frac{32}{(1+32+3)} & \frac{3}{(1+32+3)} \\ \frac{3}{(3+2+33)} & \frac{2}{(3+2+33)} & \frac{33}{(3+2+33)} \end{pmatrix} = \begin{array}{c} \\ P \\ L \\ R \end{array} \begin{array}{c} \begin{array}{ccc} P & L & R \end{array} \\ \begin{pmatrix} 0.8667 & 0.0667 & 0.0667 \\ 0.0278 & 0.8889 & 0.0833 \\ 0.0789 & 0.0526 & 0.8684 \end{pmatrix} \end{array} \tag{21}$$

From these results of transition probabilities we can calculates π_P, π_L, and π_R by using equations (16), (17), and (18). $\pi_P = 0.288$, $\pi_L = 0.346$, and $\pi_R = 0.366$ were obtained as a steady-state transition probability for each mobility pattern state.

3.5 Performance Measurements

The main objective of the MBLUA approach is to reduce the number of localization updates without significantly affect the location accuracy. Since power consumption is very important in WSNs, it is an essential design aspect to reduce the number of localization updates while moving. The lower the number of location updates is, the lower the power consumption. For this reason, a key performance parameter for the algorithm is the cost of localization updates which we will call the update cost C_u.

Another key performance issue is to keep the error of the location accuracy as low as possible. This parameter measurement is called imprecision cost C_e.

The update cost and the imprecision cost are first calculated for each mobility-pattern state as the time progress and then the total update and imprecision costs for the system is calculated as shown later in this section. The update cost for each state, c_u^i where $i \in S$, is measured by calculating how many localization updates occurred in specific discrete time periods as in (26)

$$c_u^i = \frac{\sum_{t=t_0}^{t_{n-1}} \overline{c}_t}{n} \tag{26}$$

where n is the number of time slots. $\overline{c}_t$ is 1 if update occurs at time slot t and 0 if update does not occur. In the algorithm only one update is allowed in a time slot. Therefore, c_u^i is the rate of localization update occurring in n time slots, and $1 \geq c_u^i \geq 0$.

The imprecision cost, c_e^i, for *Linear* and *Random* states are the sum of the Euclidean distances between the actual location and the estimated location over the number of predictions occurred, which is the average accuracy error per prediction as in (27)

$$c_e^i = \frac{\sum_{t=t_0}^{t_{n-1}} d_t}{k}, i \in S = \{L, R\} \tag{27}$$

where k is the number of predictions occurred in n time slots, and d_t is the Euclidean distance between the estimated location and the real location at time slot t. For the Pause state policy the imprecision cost equation is calculated differently since the Pause

state policy does not have predictions. So, in (28) l is the number of updates occurred during the *Pause* state time and d_t is the same as (27).

$$c_e^i = \frac{\sum_{t=t_0}^{t_{n-1}} d_t}{l}, i \in S = \{P\} \tag{28}$$

For the total performance measurements we use the following equations:

$$C_u = \sum_{i \in S} \pi_i c_u^i \tag{29}$$

$$C_e = \sum_{i \in S} \pi_i c_e^i \tag{30}$$

Equation (29) calculates the total localization update cost in the system, where equation (30) calculates the total imprecision cost of the system.

3.6 Mobility Models for Simulation

A mobility model to resemble each state is to be introduced in this section. First the Random Waypoint [6] mobility model is introduced, and second the Gaussian Markovian [6] mobility model is presented. Then a mobility model is introduced in [3] where the mobility model is a hybrid of the Random Waypoint and Gaussian Markovian models as will be explained in section 3.6.3.

3.6.1 Random Waypoint

In Random Waypoint model [6], the Mobile node chooses a random destination in the simulation area and a speed that is uniformly distributed between [*minspeed, maxspeed*]. When it reaches this destination it pauses for a specific period of time

(pause time). The node then chooses a random destination again. All destinations are randomly chosen from within a predefined area. The model is memory-less, which means the speed and direction after the pause are completely independent of the speed and direction before the pause. This model is predictable between the two points but it is completely unpredictable once the node pauses or just before it moves.

3.6.2 Gaussian Markovian

In Gaussian Markovian model [6], initially each node is assigned a current speed and direction. At fixed intervals, the speed and direction are updated. The speed and direction at the n^{th} instance is calculated based upon the value of speed and direction at the $(n\text{-}1)^{th}$ instance and a random variable. The new speed and direction are calculated according to equations (22) and (23).

$$s_n = \alpha s_{n-1} + (1-\alpha)\bar{s} + \sqrt{1-\alpha^2}\, s_{x_{n-1}} \tag{22}$$

$$d_n = \alpha d_{n-1} + (1-\alpha)\bar{d} + \sqrt{1-\alpha^2}\, d_{x_{n-1}} \tag{23}$$

Where s_n and d_n are new speed and direction of a mobile node at time interval n. α is a value from 0 to 1 and it is the tuning parameter used to vary the randomness. $\alpha = 0$ leads to very random motion, while $\alpha = 1$ leads to completely linear motion. $\bar{s}$ and $\bar{d}$ are constants representing the mean value of speed and direction as $n \rightarrow \infty$. Finally $s_{x_{n-1}}$ and $d_{x_{n-1}}$ are random variables from a Gaussian distribution.

At each time interval the next location is calculated based on the current location, speed, and direction of movement. Specifically, at time interval n, a mobile node's position is given by the equations:

$$x_n = x_{n-1} + s_{n-1} \cos d_{n-1} \tag{24}$$

$$y_n = y_{n-1} + s_{n-1} \sin d_{n-1} \tag{25}$$

where (x_n, y_n) and (x_{n-1}, y_{n-1}) are the x and y coordinates of the mobile node's position at the n^{th} and $(n-1)^{th}$ time intervals, respectively, and s_{n-1} and d_{n-1} are the speed and direction of the mobile node, respectively, at the $(n-1)^{th}$ time interval.

3.6.3 Gaussian Markovian Random Waypoint

The mobility pattern in [3] is used with some alterations to suit the SMM model. The Gaussian Markovian Random Waypoint (GMRW) as from the name is a hybrid of the above two models mentioned. The idea is to use Random Waypoint at the macro level and to use the Gaussian-Markovian technique at the micro level. In the GMRW model, the mobile node still chooses a random destination location and speed, and then computes the direction of travel. However, it follows the model presented in equations (22) and (23) to travel in this direction in small time steps. The GMRW model is initialized with s_0 and d_0 being equal to the chosen speed and direction. $\overline{s}$ and $\overline{d}$ are the mean speed and direction computed on line by averaging over all timesteps. The node travels for an amount of time (say, T) equal to what it would take the node to reach the chosen destination using the chosen speed. The mobile node may not reach the destination, but possibly a location close to it depending on the amount of noise. After

the time T, the node pauses for a randomly chosen pause time, chooses a random destination location, and repeats the above process.

To suit the State-based Mobility Model, the GMRW model is modified. The GMRW has the advantage of varying the randomness α parameter to suit the mobility state and still keep choosing a random destination. For the *Linear* state we set α value equal to 1 in equations (22) and (23) so it behaves as pure linear movement. However, the d_{n-1} in (23) should be recalculated to the new location at first. In the random state the α value is set for values between 0.1 and 0.9. The zero value is avoided because a value of zero will make the movement completely random and memoryless that does not depend on the previous speed and direction. Whenever the mobile node goes in pause time it automatically switches to the *Pause* state for that amount of time. The movement of the modified GMRW will be similar to the one in figure 6.

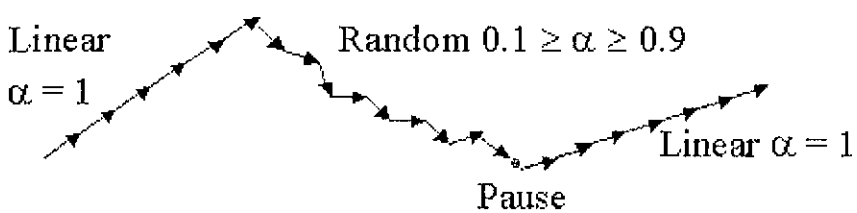


Figure 6: Modified GMRW movement pattern example.

3.7 Summary

This chapter introduces in details the MBLUA algorithm. The MBLUA works on the SMM model where the complex mobility model is divided into three types or states. Two design parameters were introduced, update cost and imprecation cost. The MBLUA is designed to reduce the update cost and keeping the imprecation cost as low as possible. The last section in this chapter presents a mobility model that can vary between the three mobility-pattern states. The modified GMRW is used to simulate the mobile sensor node movement.

CHAPTER IV

EXPERIMENTS, RESULTS AND ANALYSIS

4.1 Introduction

In this section, the experiments' results are presented with the proposed protocol. The mobility model has significant implication on the performance of the localization protocols. With the Mobility-Pattern Based Localization Update Algorithm (MBLUA), a trade-off between update cost and imprecision cost is expected. It is expected that when update cost is low (not updating too much) the imprecision cost will be high, indicating too much error as a node moves. On the other hand, if the update cost is high (updating so often) the imprecision cost will be low, indicating a low error.

In the following sections, simulations are conducted to test the MBLUA design. The test work is divided into two parts. The first part concerns the simulation of the node's actual movement model. The algorithm of this model is the modified GMRW mobility model. A modified BonnMotion tool [11] is used to generate the various scenarios. The second part is the localization update algorithm coding, and it is divided into two parts also. The first part concerns the algorithm of the three states combined using the State-based Mobility Model (SMM), which is the MBLUA design. The second part concerns implementing each State algorithm separately. The purpose of having each State algorithm alone is to compare it with the MBLUA design.

For testing, first the simulation is run to model the real movement of a mobile node. For the MBLUA design, the probability of choosing the next mobility pattern (the next state) is random. However, the probability of having the *Linear* or *Random* mobility patterns is higher than the probability of having the *Pause* state through the entire simulation. The probability of having the *Linear* or *Random* mobility pattern state is 0.4 for each, and the probability of having the *Pause* state is 0.2. Second, the localization update algorithms are run over the output file generated from the mobility model simulation run. Then the localization update algorithms generate results output files and a trace of the estimated and updated locations of a mobile node.

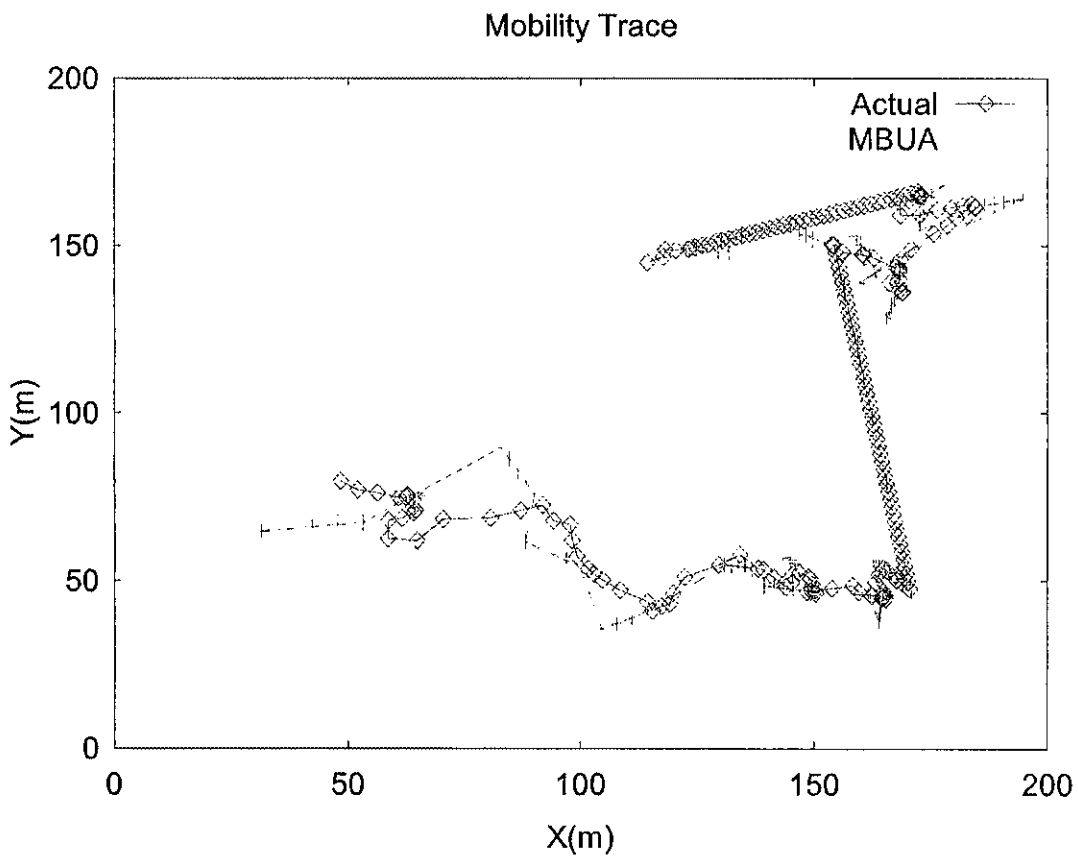
4.2 Network Model

A simulation area of 200 by 200 meters is used, and a two-dimensional area is assumed. The simulation is run over 10 nodes for 900 seconds. The localization scheme (for updating current location) does not affect the proposed algorithm because the algorithm takes care of the localization period only and it is mobility pattern dependent. For this reason, the localization scheme is not included in the simulation, and current location updates are obtained from the mobility model simulation output file.

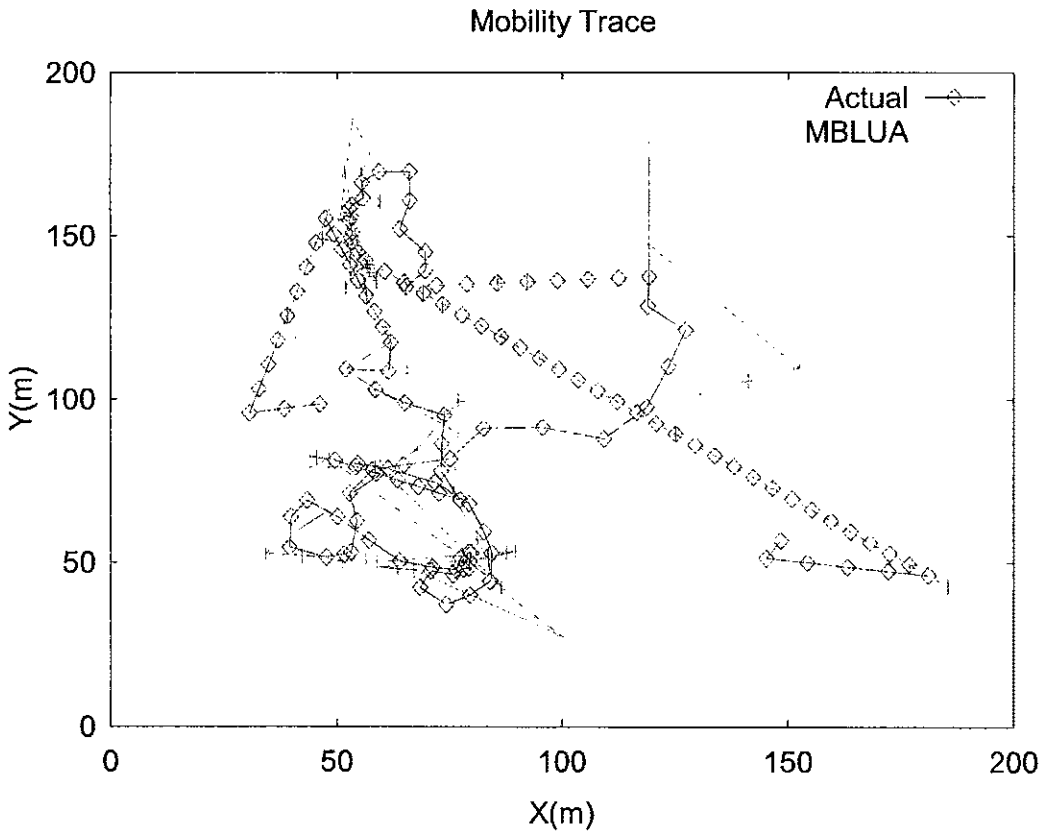
4.3 Results and Analysis

In figure 7, the trace results for one slow node and one fast node is shown. The figure shows the actual movement trace and the estimated trace using the MBLUA design for two different speed ranges. For the slow node, the speed range is from

0.5m/s to 1m/s, and for the fast node, the speed range is from 2m/s to 4m/s. The maximum pause time is 60 seconds. In both (a) and (b) we can see that the MBLUA design shows a good result when the movement is linear, and it tends to have some error when having abrupt turns in the movements. In Figure 7 (b) the error increases with higher speed range.



(a) Slow speed (0.5-1 m/s) (Max pause time 60 sec.)
(*max_time* = 5 time lots, *Lin_thresh* = 4m, and *d_thresh* = 5m)



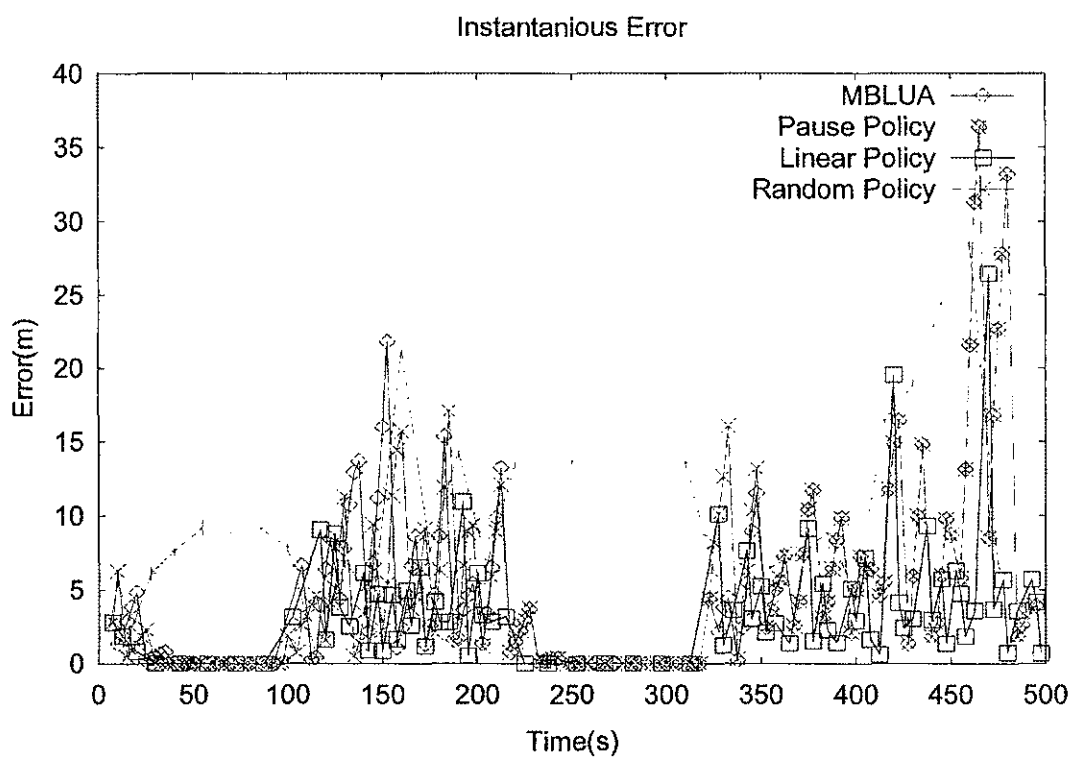
(b) Fast speed (2-4 m/s) (Max pause time 60 sec.)
(*max_time* = 5 time lots, *Lin_thresh* = 4m, and *d_thresh* = 5m)

Figure 7: Mobility trace for slow and fast speeds.

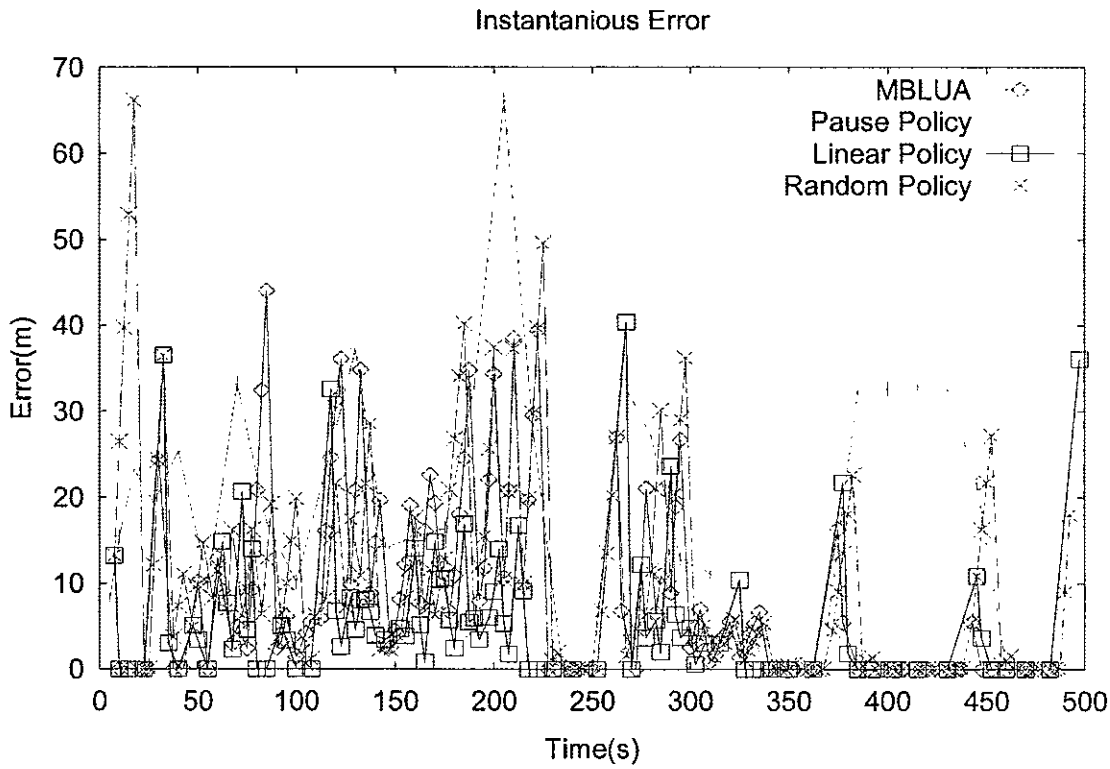
The reason of having mismatch between the actual and the MBLUA is because the mobile node's real movement went off the location estimation during the wait period. Localization is performed after this wait period and the estimation is corrected. When the actual location trace is off by itself (in figure 7), it means that the mobile node was in a *Pause* state and the MBLUA assumes that the mobile node is still in the same position during the wait time. On the other hand, if the MBLUA location trace is off by itself (in figure 7) it means that the mobile node was in motion and suddenly stopped

(*Pause* state) and the MBLUA assumes that the mobile node is still in motion during the wait time.

Figure 8 shows the instantaneous error as a function of time using MBLUA, Pause, Linear, and Random policies for slow and fast movement patterns. The results are obtained from one mobile node (say mobile node 1). For MBLUA the parameters are: *max_time* = 5 time slots as the maximum wait time, *Lin_thresh* = 4m as the Linear state error threshold and *d_thresh* = 5m as the Random state error threshold. The Linear and the Random states policies use the same value for *Lin_thresh* and *d_thresh* respectively.



(a) Slow speed (0.5-1 m/s)



(b) Fast speed (2-4 m/s)

Figure 8: Instantaneous error for slow and fast movement.

It was found that the Pause state policy has the worst error value since localization update occurs at fixed time periods regardless of the error value. The lowest two error values are the MBLUA and the Linear state policy. At random motion periods, the MBLUA performs better than the Linear state policy.

To demonstrate the comparison better, Table 1 shows the update and the imprecision costs for the three update policies and the MBLUA design for the fast and the slow movements. Again the algorithm parameters are: *max_time* = 5 time lots, *Lin_thresh* = 4m, and *d_thresh* = 5m. The difficulty faced on gathering the results of

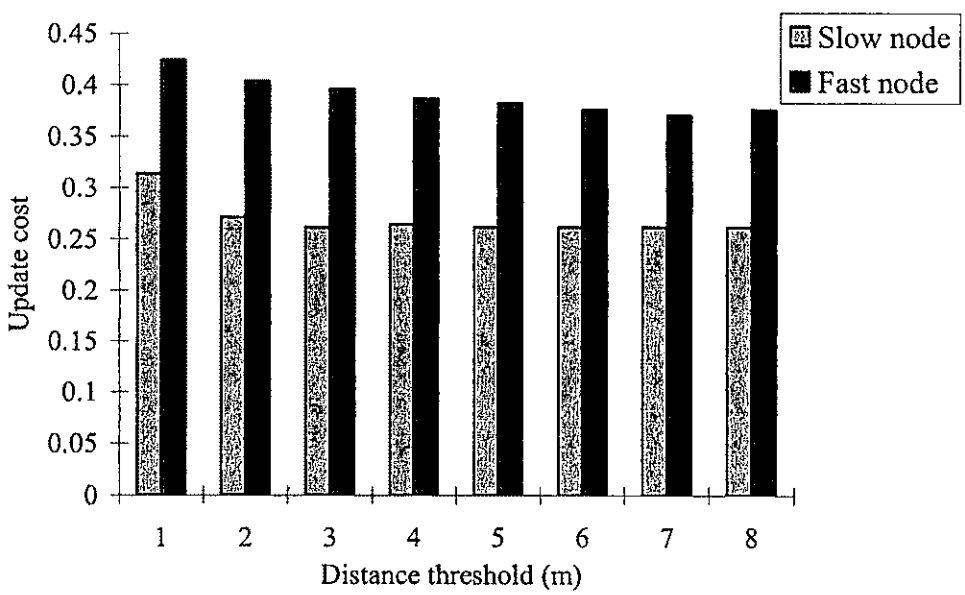
Table 1 is that each node in the network has different mobility scenario. So it is hard to have an average of ten distinct motions or mobility patterns. The results presented in Table 1 are obtained from only one mobile node (say mobile node 1). A prompt question is: what is the purpose of having many mobile nodes in the network model? Well the answer to this question is simply to have mobility movement model exactly the same when having many sensor nodes in the network. Moving alone in an area is definitely different than moving with other objects around.

Table 1: Policies update and imprecision costs comparison.

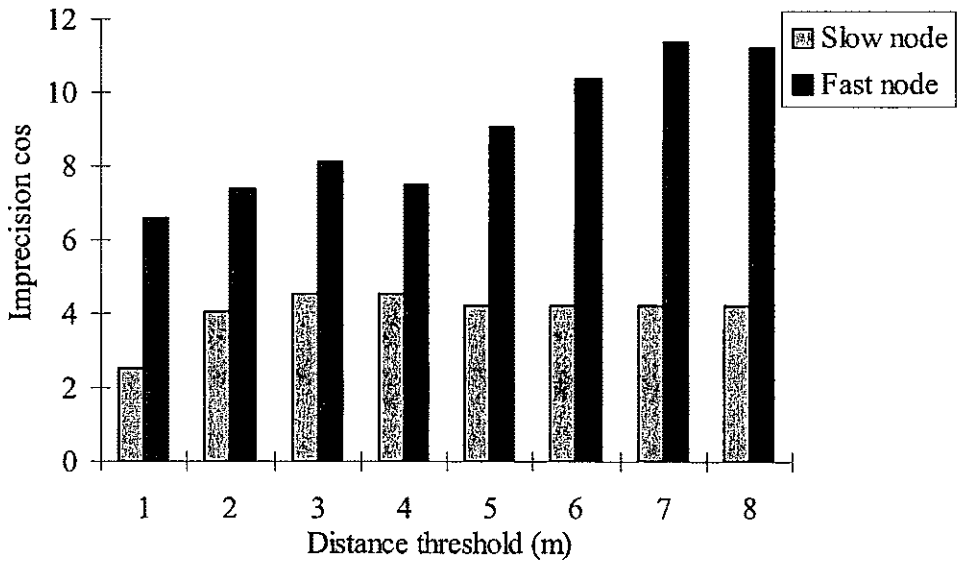
Policy	Fast node (2-4 m/s)		Slow node (0.5-1 m/s)	
	Update cost	Imprecision cost	Update cost	Imprecision cost
MBLUA	0.373776	10.19647	0.241907	5.388964
Pause Policy	0.176768	18.63553	0.176768	10.07666
Linear Policy	0.464646	6.349763	0.40404	3.734811
RandomPolicy	0.208122	16.60858	0.177665	8.137531

From table 1, the Pause state policy has the least update cost, but it has the most imprecision cost (the larger accuracy error). The Linear state policy has the lowest imprecision cost, but it has the highest update cost. The Random state policy stands in the middle between the Pause and the Linear state policies, but it has considerably high imprecision cost. Comparing the MBLUA results to the other policies results, it is found that MBLUA has the second lowest imprecision cost after the Linear state policy. Also, the MBLUA update cost is less than the Linear policy update cost. Therefore, the MBLUA has considerably low update cost with an acceptable imprecision cost.

Figure 9 shows the update and the imprecision costs for MBLUA design when varying the *d_thresh* parameter. Figure 9 (a) shows the relationship between *d_thresh* and the update cost, while figure 9 (b) shows the relationship between *d_thresh* and the imprecision cost. It is observed that when *d_thresh* is low, the update cost is high indicating more frequent updates, and the imprecision cost is low since location knowledge is updated more frequently. As *d_thresh* increases, the update cost decreases and the imprecision cost increases, since a larger *d_thresh* value results in larger wait time to localize.



(a) Update cost vs. *d_thresh* (*max_time* = 5 time lots and *Lin_thresh* = 4m)



(b) Imprecision cost vs. *d_thresh* (*max_time* = 5 time lots and *Lin_thresh* = 4m)

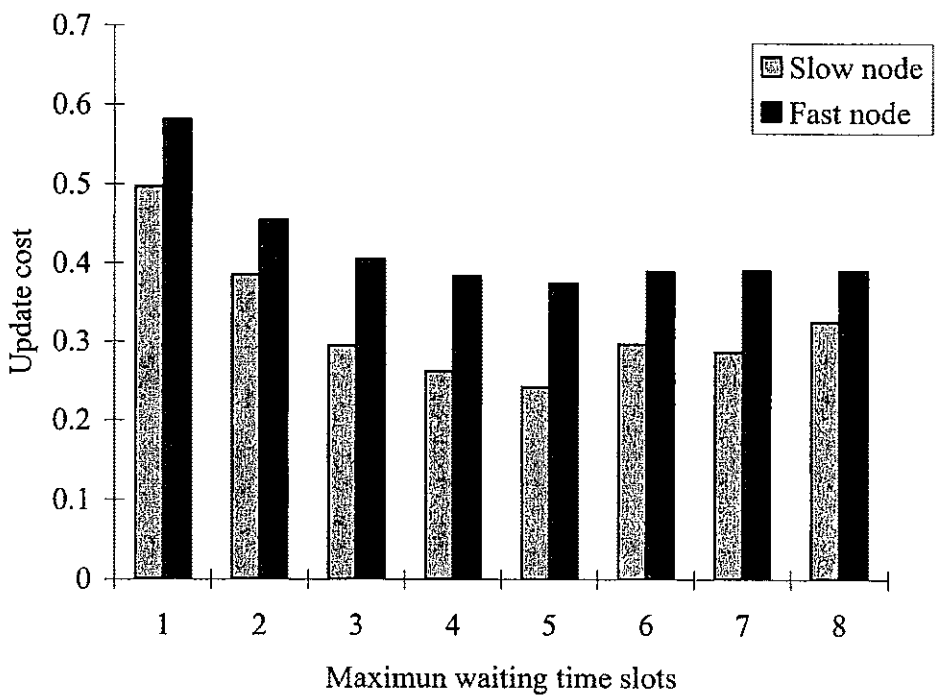
Figure 9: Update and imprecision costs as a function of *d_thresh*.

From the results of figure 9, it is observed that any *d_thresh* value greater than 5 does not have effect on the update cost meanwhile imprecision cost does increase. The reason for this is that the maximum time (*max_time*) to localize is fixed in the simulations. As a result, after *d_thresh* value 5, the sensor node will localize after *max_time* period no matter how large the *d_thresh* value is which stabilizes the update cost. Consequently, the main factor of changing the imprecision cost when the update cost stabilizes would be the acceleration value.

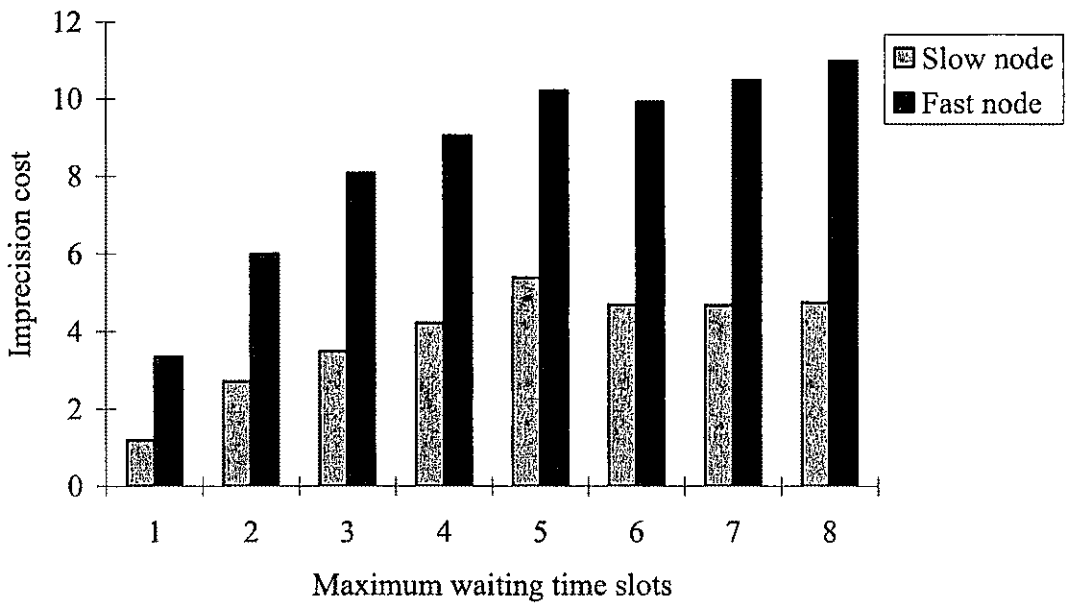
It is found that changing *Lin_thresh* has a little effect on the results because the Linear state policy tends to increase the wait time to the maximum time period (*max_time*) as long as the prediction error is less than *Lin_thresh*. This is usually the case when having linear movement. Therefore, the localization period is equal to the

max_time most of the time period of the linear movement, which causes the update and imprecision costs to stabilize.

Figure 10 shows the update and the imprecision costs for MBLUA design when varying the maximum wait time (*max_time*). Figure 10 (a) shows the relationship between *max_time* and the update cost, and figure 10 (b) shows the relationship between *max_time* and the imprecision cost.



(a) Update cost vs. Max. waiting time slots (*max_time*) (*Lin_thresh* = 4m, and *d_thresh* = 5m)



(b) Imprecision cost vs. Max. waiting time slots (*max_time*) (*Lin_thresh* = 4m, and *d_thresh* = 5m)

Figure 10: Update and imprecision costs as a function of maximum waiting time slots (*max_time*).

When having low *max_time* values, the update cost gets high (more updates) and the imprecision cost gets low (less error) because the mobile node is forced to localize more frequently. In contrast, as the *max_time* increases the update cost decreases (less updates) and the imprecision cost increases (more error) because the mobile node has larger waiting time limit so it localizes less frequently. In figure 10 (a), the *max_time* value 5 seems to be the optimal value for best update cost result and any larger value does not have large effect on the update cost because the *d_thresh* and *Lin_thresh* would limit the localization period then.

The previous results presented show that fast movement has more update and imprecision costs. This is because when a mobile node moves faster, the estimation

deviates from the real location in higher rates, which causes larger estimation errors. Thus, as long as the error value exceeds the predefined error thresholds (*d_thresh* or *Lin_thresh*), a node will localize more frequently.

4.4 Summary

In this chapter, the simulation and the MBLUA design results were introduced and analyzed. First, a comparison movement trace between the actual location and the estimated location is presented. Second, a performance comparison between the MBLUA design and each state update policy is introduced by showing the instantaneous error as a function of time. To emphasize the comparison, the update and imprecision costs for MBLUA and each update policy are compared in Table 1. It is found that the MBLUA model has considerably low update cost and moderate imprecision cost compared to the other mobility state policies apart. Being able to change the update policy based on the mobility pattern gives a great compromise between the update cost and the imprecision cost. We can see that there are no extreme results for either the update cost or the imprecision cost in the MBLUA. Finally, a trade-off between the update cost and the imprecision cost were studied when changing the *d_thresh* and the *max_time* values.

CHAPTER V

CONCLUSION AND FUTURE WORK

5.1 Conclusion

A Mobility-Pattern Based Localization Update Algorithms was designed for mobile wireless sensor networks. The main idea is to divide sensor movements into three states *Pause*, *Linear*, and *Random*. Based on the nature of each state, a different localization update algorithm is designed. The State-based Mobility Model is used to transit from one mobility-state to another. Simulations were performed and analyzed to verify the design.

The results demonstrated how the localization cost is minimized and location accuracy is improved in the proposed algorithm. Also the MBLUA design compromises between the update cost and the imprecision cost values as shown in table 1. This research is significant to conserve the power consumption in sensor nodes and to track the locations of mobile sensors in a real-time manner.

5.2 Future Work

For further improvements, a suggestion is to equip a motion sensor in each mobile sensor node. The motion sensor notifies the mobile node if it is in motion or in pause state. This method could improve the accuracy a lot since no erroneous

estimations will be made when a mobile node goes from pause state to motion or vise versa.

A future work field is to study group mobility models. In group mobility models, a number of mobile sensor nodes move together. For military application, we can imagine a group of soldiers moving together to achieve certain goal. In group mobility models, a different localization update algorithm may be considered to anticipate a group mobility pattern.

APPENDIX

SOURCE CODE

```
/*************************************************************************
//
// Author: Mohammad Y. Al-laho
//
// Date  : June 2005
//
// Description:
//
//                Implementing the Mobility-pattern Based Update Algorithm (MBUA)
//
//************************************************************************/

#include <string>
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

//initial update time, in time slots (time periods)
const int Init_updatetime = 0;

//the time interval period, in seconds
const double time_slot = 2.5;

//Define global variables to hold position and time values everywhere
//& update and imprecision values
double xg, vg, yg, tg;
double P_update = 0;
double L_update = 0;
double R_update = 0;
double P_imp = 0;
double L_imp = 0;
double R_imp = 0;
double P_time_intervals = 0;
double L_time_intervals = 0;
double R_time_intervals = 0;
double P_pred_num = 0;
double L_pred_num = 0;
double R_pred_num = 0;

//Mobility model parameters variable
```

```
int nodes_num, sim_time_slots;

//simulation number of time slots counter
int num_of_slots = 0;

//functions declerations
int Pause_state(double t, double v, double x, double y, FILE *read, FILE *derror, int
max_time);
int Linear_state(double t, double x, double y, FILE *read, FILE *derror, int max_time,
int Lin_thresh);
int Random_state(double t, double v, double x, double y, FILE *read, FILE *derror, int
max_time, int d_thresh);

// The three states implementation
// Pause state
int Pause_state(double t, double v, double x, double y, FILE *read, FILE *derror, int
max_time)
{
        double x1, x2, y1, y2, t1, t2, v1, v2, d1;
        double t_testend = 0;
        int k = 0;
        int update_time, n;
        update_time = Init_updatetime;

        printf("PAUSE STATE \n");
        printf("t = %lf, x = %lf, y = %lf\n", t, x, y);
        printf("v = %lf\n", v);

        x2 = x;
        y2 = y;
        t2 = t;

        while((!feof(read)) && (num_of_slots < sim_time_slots))
        {
                //wait for update_time period
                for(n = update_time; n > 0; n--)
                {
                        fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
                        P_time_intervals++;
                        num_of_slots++;
                        //if end of file or end of node date break
                        if((t1 == t_testend) || (num_of_slots == sim_time_slots))
                        {
                                break;
                        }
```

```
            t_testend = t1;
    }
//save last time
t2 = t1;

//if end of node data break
if(num_of_slots == sim_time_slots)
 {
                    break;
 }

//Localize
fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
P_time_intervals++;
P_update++;
num_of_slots++;
P_pred_num++;

//if end of node data break
if(num_of_slots == sim_time_slots)
 {
                    break;
 }

//calculate distance and velocity
d1 = sqrt((pow ((x1 - x2), 2)) + (pow ((y1 - y2), 2)));
v1 = d1/(t1 - t2);

printf("P t1 = %lf, x1 = %lf, y1 = %lf\n", t1, x1, y1);
printf("v1 after calc = %lf\n", v1);

printf("error value d = %lf\n", d1);
//calculate imprecision cost
P_imp = P_imp + d1;
printf("imprecision cost = %lf\n", P_imp);
fprintf(derror,"%lf  %lf  %lf  %lf\n", t1, d1, x1, y1);

//if second time with velocity not equal zero
if(v1 != 0)
 {
                    k++;

                    //reset update_time
                    update_time = Init_updatetime;
 }
```

```
        //test mobility state
        if(k == 2)
          {
              if(v1 == v2)
                {
                 // Linear_state(t1, x1, y1, read);
                 xg = x1;
                 yg = y1;
                 tg = t1;
                 printf("P_time_intervals = %lf\n", P_time_intervals);
                 printf("P_update = %lf\n", P_update);
                 return 2;
                }
              else
                {
                 // Random_state(t1, v1, x1, y1, read);
                 xg = x1;
                 yg = y1;
                 tg = t1;
                 vg = v1;
                 printf("P_time_intervals = %lf\n", P_time_intervals);
                 printf("P_update = %lf\n", P_update);
                 return 3;
                }
          }
        if(v1 == 0)
          {
                    if(update_time < max_time)
                    {
                              update_time++;
                    }
                    k = 0; //reset k if v1 = 0 again
          }

        //save previous value into x2, y2, t2, and v2
        x2 = x1;
        y2 = y1;
        t2 = t1;
        v2 = v1;

      }// end while loop

}
// Linear state
```

```
int Linear_state(double t, double x, double y, FILE *read, FILE *derror, int max_time,
int Lin_thresh)
{
        double x1, x2, y1, y2, t1, t2, v1, v2, d1, d, vx, vy, xp, yp, tdr;
        double t_testend = 0;
        int update_time, n;
        int pred_num = 0;
        update_time = Init_updatetime;

        printf("LINEAR STATE \n");
        printf("t = %lf, x = %lf, y = %lf\n", t, x, y);

        x2 = x;
        y2 = y;
        t2 = t;

        //Localize
        fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
        L_update++;
        L_time_intervals++;
        num_of_slots++;

        tdr = t1;

        //calculate distance and velocity
        d1 = sqrt((pow ((x1 - x2), 2)) + (pow ((y1 - y2), 2)));
        v2 = d1/(t1 - t2);

        while((!feof(read)) && (num_of_slots < sim_time_slots))
        {
          //compute velocity components
          vx = (x1 - x2)/(t1 - t2);
          vy = (y1 - y2)/(t1 - t2);
          //Dead Reckoning
          xp = x1 + (vx * (tdr - t2));
          yp = y1 + (vy * (tdr - t2));
          printf("xp = %lf\n", xp);
          printf("yp = %lf\n", yp);
          L_pred_num++;
          x2 = x1;
          y2 = y1;
          t2 = t1;

          //wait for update_time period
                for(n = update_time; n > 0; n--)
```

```
    {
    fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
    L_time_intervals++;
    num_of_slots++;
    //if end of file or end of node date break
    if((t1 == t_testend) || (num_of_slots == sim_time_slots))
      {
       printf("inside break \n");
       break;
      }
    t_testend = t1;
    }

//if end of node date break
if(num_of_slots == sim_time_slots)
 {
            break;
 }
//Localize
fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
num_of_slots++;

if(t1 != t_testend)
 {
    L_time_intervals++;
    L_update++;

    //calculate distance and velocity
    d1 = sqrt((pow ((x1 - x2), 2)) + (pow ((y1 - y2), 2)));
    v1 = d1/(t1 - t2);

    printf("L t1 = %lf, x1 = %lf, y1 = %lf\n", t1, x1, y1);

    //compute d between predicted loc and real loc
    d = sqrt((pow ((x1 - xp), 2)) + (pow ((y1 - yp), 2)));
    printf("error value d = %lf\n", d);
    //calculate imprecision cost
    L_imp = L_imp + d;
    printf("imprecision cost = %lf\n", L_imp);
    fprintf(derror,"%lf  %lf  %lf  %lf\n", t1, d, xp , yp);

    printf("v1 after calc = %lf\n", v1);
    printf("v2 after calc = %lf\n", v2);
 }
```

```
//if end of node data break
if(num_of_slots == sim_time_slots)
 {
                    break;
 }

//update update_time
if((d < Lin_thresh) && (v1 != 0))
 {
            if(d == 0)
            {
                    if(update_time < max_time)
                    {
                            update_time++;
                            //advance t1 to wait time for dead reckoning
                            tdr = t1 + time_slot;
                            printf("update tdr C= %lf\n", tdr);
                    }
                    else
                    {
                            tdr = t1;
                            printf("update tdr C+= %lf\n", tdr);
                     }
            }
            else //if small error within L_thresh
            {
                    //compute tdr when reset update_time
                    tdr = t1 - (update_time * time_slot);
                    //reset update_time
                    update_time = Init_updatetime;
                    printf("tdr = %lf\n", tdr);
            }

 }
else
 {
    //test mobility state
    printf("mobility test\n");
    if(v1 == 0)
     {
      // Pause_state(t1, v1, x1, y1, read);
      xg = x1;
      yg = y1;
      tg = t1;
      vg = v1;
```

```
                return 1;
              }
            else
              {
               // Random_state(t1, v1, x1, y1, read);
               xg = x1;
               yg = y1;
               tg = t1;
               vg = v1;
               printf("L_update = %lf, L_time_intervals = %lf\n", L_update,
               L_time_intervals);
               return 3;
              }
          }

        //save previous velocity in v2
        v2 = v1;

      }// end of while

}
// Random state
int Random_state(double t, double v, double x, double y, FILE *read, FILE *derror, int
max_time, int d_thresh)
{
        double x1, x2, y1, y2, t1, t2, v1, v2, d1, d, vx, vy, a, t_cap;
        double t_testend = 0;
        double xp = 0;
        double yp = 0;
        double update_time, n;

        update_time = Init_updatetime;

        printf("RANDOM STATE \n");
        printf("t = %lf, x = %lf, y = %lf\n", t, x, y);
        printf("v = %lf\n", v);

        x2 = x;
        y2 = y;
        t2 = t;
        v2 = v;

        while((!feof(read)) && (num_of_slots < sim_time_slots))
        {
           //Localize
```

```
fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
R_time_intervals++;
R_update++;
num_of_slots++;
//if end of node date break
if(num_of_slots == sim_time_slots)
 {
                    break;
 }

//calculate distance, velocity & acceleration
d1 = sqrt((pow ((x1 - x2), 2)) + (pow ((y1 - y2), 2)));
v1 = d1/(t1 - t2);
a = (v1 - v2) / (t1 - t2);
printf("R t1 = %lf, x1 = %lf, y1 = %lf\n", t1, x1, y1);
printf("v1 after calc = %lf\n", v1);
printf("v2 after calc = %lf\n", v2);
printf("a after calc = %lf\n", a);
//state mobility test
if((v1 == v2) || (v1 == 0))
 {
    if(v1 == 0) //Pause state
     {
      xg = x1;
      yg = y1;
      tg = t1;
      vg = v1;
      return 1;
     }
    else //Linear state
     {
      xg = x1;
      yg = y1;
      tg = t1;
      return 2;
     }
 }
//calculate t_cap where the d_thresh is to be crossed
// abs_a = fabs(a);
t_cap = sqrt(d_thresh / fabs(a));
printf("t_cap = %lf\n", t_cap);
//compute discrete time intervals
update_time = ceil(t_cap / time_slot);
if(update_time > max_time)
 {
```

```
                    update_time = max_time;
 }

//compute velocity components
vx = (x1 - x2)/(t1 - t2);
vy = (y1 - y2)/(t1 - t2);

xp = x1;
yp = y1;

//save previous update values into x2, y2, t2, and v2
x2 = x1;
y2 = y1;
t2 = t1;
v2 = v1;

//wait for update_time period
for(n = update_time; n > 0; n--)
 {
        printf("update_time = %lf\n", n);

        //Prediction during update_time wait period
        xp = xp + ((vx + (a*time_slot)) * (time_slot));
        yp = yp + ((vy + (a*time_slot)) * (time_slot));

        printf("xp = %lf\n", xp);
        printf("yp = %lf\n", yp);

        fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);
        num_of_slots++;
        //if end of file or end of node date break
        if((t1 == t_testend) || (num_of_slots == sim_time_slots))
        {
                break;
        }
        t_testend = t1;
        R_time_intervals++;
        printf("inside for: t1 = %lf, x1 = %lf, y1 = %lf\n", t1, x1, y1);

        //compute d between predicted loc and real loc
        d = sqrt((pow ((x1 - xp), 2)) + (pow ((y1 - yp), 2)));
        fprintf(derror,"%lf  %lf  %lf  %lf\n", t1, d, xp, yp);
 }

R_pred_num++;
```

```
            R_imp = R_imp + d;

            xp = 0;
            yp = 0;


          }
}

//*********************************************************************
// Initial state
int main(int argc, char* argv[])
{
        double x1, x2, y1, y2, t1, t2, v1, v2, d1, d2;
        double P_updatecost, L_updatecost, R_updatecost, P_impcost, L_impcost,
        R_impcost;
        double PYp, PYl, PYr, total_updatecost, total_impcost;
        int i = 0;
        int n = 0;
        int nodes_count = 0;

        int max_time = atoi(argv[1]);    //for all states, max wait time, in time periods
        int Lin_thresh = atoi(argv[2]);          //for the linear state, in meters
        int d_thresh = atoi(argv[3]);    //for the random state, in meters


        //create file (read) for reading
        FILE *read;
        read = fopen("scenario13.movements", "r");
        if (read == (FILE *)0)
        {
         fprintf(stderr, "Error opening file (printed to standard error)\n");
         exit (1);
        }
        //create file (pyfile) for reading
        FILE *pyfile;
        pyfile = fopen("scenario13.tt", "r");
        if (pyfile == (FILE *)0)
        {
         fprintf(stderr, "Error opening file (printed to standard error)\n");
         exit (1);
        }
        //create file (results) for writing
        FILE *results;
        results = fopen("final_results.txt","a");
```

```
//create file (derror) for writing, for instantanious error
FILE *derror;
derror = fopen("d_error.txt","w");

// read model parameters
fscanf(pyfile, "%d%d", &nodes_num, &sim_time_slots);
printf("Nodes num = %d, sim_time_slots = %d\n", nodes_num, sim_time_slots);

for(n = nodes_num; n > 0; n--)
 {
  // Localize loc1
  fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);

  // Localize loc2
  fscanf(read, "%lf%lf%lf", &t2, &x2, &y2);

  //calculate distance and velocity
  d1 = sqrt((pow ((x2 - x1), 2)) + (pow ((y2 - y1), 2)));
  v1 = d1/(t2 - t1);

  printf("t1 = %lf, x1 = %lf, y1 = %lf\n", t1, x1, y1);
  printf("t2 = %lf, x2 = %lf, y2 = %lf\n", t2, x2, y2);
  printf("v1 = %lf\n", v1);

  //Localize loc3
  fscanf(read, "%lf%lf%lf", &t1, &x1, &y1);

  //calculate distance and velocity
  d2 = sqrt((pow ((x1 - x2), 2)) + (pow ((y1 - y2), 2)));
  v2 = d2/(t1 - t2);

  printf("t3 = %lf, x3 = %lf, y3 = %lf\n", t1, x1, y1);
  printf("v2 = %lf\n", v2);

  /*Comparing velocity values and assign i to 1, 2, or 3
    representing Pause, Linear, or Random respectively*/
  if(v2 == 0)
    {
      //Pause state
      i = 1;
      xg = x1;
      yg = y1;
      tg = t1;
      vg = v2;
```

```
 }
else
 {
     if(v1 == v2)
      {
       //Linear state
       i = 2;
       xg = x1;
       yg = y1;
       tg = t1;
      }
     else
      {
       //Random_state
       i = 3;
       xg = x1;
       yg = y1;
       tg = t1;
       vg = v2;
      }
 }

num_of_slots = 3;

//node counter
nodes_count = nodes_num - n;

// loop until end of file (until end of simulation)
while((!feof(read)) && (num_of_slots < sim_time_slots))
 {
              if(i == 1) //Pause state
              { i = Pause_state(tg, vg, xg, yg, read, derror, max_time); }

              if(i == 2) //Linear state
              { i = Linear_state(tg, xg, yg, read, derror, max_time,
              Lin_thresh); }

              if(i == 3) //Random state
              { i = Random_state(tg, vg, xg, yg, read, derror, max_time,
              d_thresh); }
 }

printf("tg = %lf, xg = %lf, yg = %lf\n", tg, xg, yg);
printf("vg = %lf\n", vg);
```

```
fscanf(pyfile, "%lf%lf%lf", &PYp, &PYl, &PYr);

//calculate update & imprecision costs for all states
P_updatecost = P_update / P_time_intervals;
P_impcost = P_imp / P_pred_num;

L_updatecost = L_update / L_time_intervals;
L_impcost = L_imp / L_pred_num;

R_updatecost = R_update / R_time_intervals;
R_impcost = R_imp / R_pred_num;

if((P_update == 0) || (P_imp == 0))
 {
        P_updatecost = 0;
        P_impcost = 0;
 }
if((L_update == 0) || (L_imp == 0))
 {
        L_updatecost = 0;
        L_impcost = 0;
 }
if((R_update == 0) || (R_imp == 0))
 {
        R_updatecost = 0;
        R_impcost = 0;
 }

total_updatecost = (PYp * P_updatecost)+(PYl * L_updatecost)+(PYr *
R_updatecost);
total_impcost = (PYp * P_impcost)+(PYl * L_impcost)+(PYr * R_impcost);


printf("Results of Node # %d\n", nodes_count);
printf("P_updatecost = %lf, P_impcost = %lf\n", P_updatecost, P_impcost);
printf("L_updatecost = %lf, L_impcost = %lf\n", L_updatecost, L_impcost);
printf("R_updatecost = %lf, R_impcost = %lf\n", R_updatecost, R_impcost);

printf("Total updatecost = %lf, Total impcost = %lf\n\n", total_updatecost,
total_impcost);

//output results to a file
fprintf(results,"%d %d %d %d ", nodes_count, max_time, Lin_thresh,
d_thresh);
fprintf(results,"%lf  %lf\n", P_updatecost, P_impcost);
```

```
            fprintf(results,"%lf  %lf\n", L_updatecost, L_impcost);
            fprintf(results,"%lf  %lf\n", R_updatecost, R_impcost);

            fprintf(results,"%lf  %lf\n", total_updatecost, total_impcost);
            fprintf(derror,"\n");
            //reset all update and imp variables for new node count
            P_update = 0;
            L_update = 0;
            R_update = 0;
            P_imp = 0;
            L_imp = 0;
            R_imp = 0;
            P_time_intervals = 0;
            L_time_intervals = 0;
            R_time_intervals = 0;
            num_of_slots = 0;

          }//end of for loop for # of nodes

        printf("End of simulation \n");
        fclose(derror);
        fclose(results);
        fclose(pyfile);
        fclose(read);
        return 0;
}// end of main
```

REFERENCES

[1] S. Tilak, V. Kolar, N. B. Abu-Ghazaleh, and K. D. Kang, "Dynamic Localization Control for Mobile Sensor Networks," *Proc. of IEEE International Workshop on Strategies for Energy Efficiency in Ad Hoc and Sensor Networks*, pp. 587–592, Apr. 2005.

[2] M. Song, K. Park, J. Ryu, and C. Hwang, "Modeling and Tracking Complexly Moving Objects in Location-Based Services," *Journal of Information Science and Engineering*, vol. 20, no. 3, pp. 517–534, May 2004.

[3] V. Kumar, and S. R. Das, "Performance of Dead Reckoning-Based Location Service for Mobile Ad Hoc Networks," *Wireless Communications and Mobile Computing Journal*, vol. 4, no. 2, pp. 189–202, Mar 2004.

[4] L. Hu, and D. Evans, "Localization for Mobile Sensor Networks," *Proc. of Tenth Annual Internationa Conference on Mobile Computing and Networking*, pp. 45–57, 2004.

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

[6] T. Camp, J. Boleng, and V. Davies, "A Survey of Mobility Models for Ad Hoc Network Research," *Wireless Communication & Mobile Computing, Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, vol. 2, no. 5, pp. 483-502, 2002.

[7] A. Savvides, C. C. Han, and M. B. Srivastava, "Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors," *Proc. of the 7th ACM/IEEE International Conference on Mobile Computing and Networking* (MobiCom), pp 166–179, Jul. 2001.

[8] A. Agarwal, and S. R. Das, "Dead Reckoning in Mobile Ad Hoc Networks," *Proc. of the IEEE Wireless Communications and Networking Conference*, vol.3, pp. 1838–1843, Mar. 2003.

[9] A. Bar-Noy, I. Kessler, and M. Sidi, "Mobile users: To update or not to update?" *Wireless Networks*, vol. 1, no. 2, pp. 175–186, July 1995.

[10] V. W. S. Wong, and V. C. M. Leung, "An Adaptive Distance-Based Location Update Algorithm for Next-Generation PCS Networks," *IEEE J. Select. Areas on Communications*, vol. 19, no. 10, pp. 1942–1952, Oct. 2001.

[11] BonnMotion. http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/, May 15, 2005.

[12] X. Cheng, A. Thaeler, G. Xue, and D. Chen, "TPS: A Time-Based Positioning Scheme for Outdoor Sensor Networks," *IEEE INFOCOM, Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies,* vol. 4, pp. 2685–2696, March 2004.

[13] A. Nasipuri, and K. Li, "A Directionality Based Location Discovery Scheme for Wireless Sensor Networks," *Proc. ACM WSNA, Atlanta, GA*, pp. 105–111, Sept. 2002.

[14] X. Cheng, X. Huang and D.-Z. Du, "Location Discovery in Ad-hoc Wireless Sensor Networks," *Kluwer Academic Publishers, Norwell, MA,* 2003.

[15] L. Doherty, K. S. J. Pister, and L. El Ghaoui, "Convex Position Estimation in Wireless Sensor Networks," *IEEE INFOCOM*, volume 3, pp. 1655–1663, April 2001.

[16] D. Culler, D. Estrin, and M. Srivastava, "Guest Editors' Introduction: Overview of Sensor Networks," *Computer*, volume 37, issue 8, pp. 41–49, Aug. 2004.

[17] B. Liang, and Z. J. Haas, "Predictive Distance-Based Mobility Management for PCS Networks," *INFOCOM '99, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pp. 1377–1384, March 1999.