8-2012

Fast and Efficient Classification, Tracking, and Simulation in Wireless Sensor Networks

Hao Jiang
Clemson University, jianghao0718@gmail.com

Fast and Efficient Classification, Tracking, and Simulation in Wireless Sensor Networks

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Hao Jiang
August 2012

Accepted by:
Jason O. Hallstrom, Committee Chair
Pradip K. Srimani
Brian A. Malloy
David P. Jacobs

Abstract

Wireless sensor networks are composed of large numbers of resource-lean sensors that collect low-level inputs from the physical world. The applications present challenges for programmers. On the one hand, lightweight algorithms are required given the limited capacity of the constituent devices. On the other, the algorithms must be scalable to accommodate large networks. In this thesis, we focus on the design and implementation of fast and lean (yet scalable) algorithms for classification, simulation, and target tracking in the context of wireless sensor networks. We briefly consider each of these challenges in turn.

The first challenge is to achieve high precision classification of high-level events *in-network* using limited computational and energy resources. We present *in-network* implementations of a Bayesian classifier and a condensed *kd-tree* classifier for identifying events of interest on resource-lean embedded sensors. The first approach uses preprocessed sensor readings to derive a multi-dimensional Bayesian classifier used to classify sensor data in real-time. The second introduces an innovative condensed kd-tree to represent preprocessed sensor data and uses a fast nearest-neighbor search to determine the likelihood of class membership for incoming samples. Both classifiers consume limited resources and provide high precision classification. To evaluate each approach, two case studies are considered, in the contexts of human movement and vehicle navigation, respectively. The classification accuracy is above 85% for both classifiers across the two case studies.

The second challenge is to achieve high performance parallel simulation of sensor network hardware. This is achieved by reducing the synchronization overhead among distributed simulation processes. Traditional parallel simulation strategies introduce significant synchronization overhead, reducing the simulation speed. We present an optimistic simulation algorithm with support for backtracking and re-execution. The algorithm reduces the number of synchronization cycles to the number of transmissions in the network under test. Concretely, we implement *SnapSim*, an extension

to the popular Avrora simulator, based on this algorithm. The experimental results show that our prototype system improves the performance of Avrora by 2 to 10 times for typical network-centric sensor network applications, and up to three orders of magnitude for applications that use the radio infrequently.

The third challenge is to efficiently track a moving target in a network. The difficulty again lies in the conflict between the limited resource capacity of typical sensors and the significant processing requirements of typical tracking algorithms. We introduce an in-network object tracking framework for tracking mobile objects using resource-lean sensors. The framework is based on a distributed, dynamically scoped tracking algorithm which adaptively scopes the event detection region based on object speed. A leader node records the samples across an event region (without the aid of time synchronization) and estimates the object's location in situ. To minimize the number of radio transmissions, the location snapshotting rate is also adjusted based on the object speed.

In this dissertation, focusing on the above challenges, we present the design, implementation, and evaluation of classification, simulation, and tracking contributions.

Dedication

In memory of Su Zhu, a friend and a mentor, who led me to the world of computer sciences.

Acknowledgments

There are many people I would like to thank during my years at Clemson University. First of all, I would like thank my Advisor, Dr. Jason O. Hallstrom, for his support, advice and encouragement, I cannot image how could I reach my destination without him all along the journey. Second, I would like to thank my committee members, Dr. Pradip K. Srimani, Dr. Brian A. Malloy, and Dr. David P. Jacobs, for their help and suggestions on my research and dissertation. Third, I would like to thank my colleagues, Dr. Sally W. Wahba, Biswajit Mazumder, and Jiannan Zhai, for their hard work helping me running experiments and improving my research. Fourth, I would like to thank all the faculty and staff in School of Computing, for their warmhearted support and sagacious instruction. Last but not least, I wish to thank my family. I want to thank my wife, Qian Yang, for her moral support – getting a PhD becomes much easier and enjoyable after she appeared in my world. I want to thank my parents, for their support far from China, even I can hardly accompany with them in these years.

Thank you!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Wireless sensor networks (WSNs) [12] typically comprise a large number of sensor nodes. The constituent nodes, typically limited in terms of computation and energy resources, are equipped with sensors and wireless radios and communicate via specialized network protocols. Sensor networks are popularly used in a variety of applications, including environmental, structural, medical, and traffic monitoring [20, 37, 38, 52, 54, 57, 69, 71]. The numerous application examples reported in the literature suggest a promising future for sensor network technology.

However, sensor network application development is rarely trivial due to the nature of the sensors. The devices are typically lean in computational resources, energy, and bandwidth. Sensors are deployed in a range of environments, usually at large scales. The design challenges are numerous. In this thesis, we focus on three key problems – event classification, network simulation, and target tracking.

Classification. To identify high-level events from low-level sensor signals, the challenge stems from the resource constraints associated with common hardware platforms. Traditional machine learning techniques are not a good fit for resource-lean devices.

Simulation. It is difficult for developers to debug sensor network applications due to their inherently parallel and distributed nature. Network simulation plays an important role in sensor network application development and research. However, high synchronization overhead among simulation processors decreases the efficiency of state-of-the-art parallel simulators.

Tracking. Target tracking is a challenging task for many sensor network applications. The nature of resource-lean sensors requires the algorithms to be lightweight and scalable. Traditional

tracking schemes are inefficient for tracking objects of variable speed.

1.1 Problem Statement

In this section, we more clearly define the challenges to be addressed in the contexts of classification, simulation, and object tracking in sensor networks.

1.1.1 Classification

Wireless sensor networks offer the potential to monitor high-level events using simple sensor signals. Representative applications include bridge health monitoring using accelerometers [37], patient symptom detection using wearable accelerometers [25, 29], location context identification using microphones [53], and vehicle detection using acoustic and seismic sensors [40]. Event detection involves extracting information from raw sensor readings and reporting the occurrence of interesting events in the physical world. Signal thresholding techniques are most frequently used for event detection, but are difficult to apply across detection scenarios due to their ad hoc design. In contrast, machine learning techniques are more general, accurate, and transferable. However, the challenge lies in the limited resources associated with typical sensor devices. The algorithms are usually too complicated to be realized on resource-lean sensors. **Designing a general purpose, lightweight classification approach for event detection is key to applying in situ machine learning on resource-lean sensors.**

1.1.2 Network Simulation

Network simulation is an important tool for programmers and researchers to understand and debug sensor network applications. However, due to the large scale of many sensor networks, traditional single-threaded simulators [45] introduce significant simulation delays, especially in the context of cycle-accurate simulation. Multi-threaded, parallel simulators [65, 68] provide a more scalable solution. State-of-the-art parallel simulators are designed based on an interval synchronization strategy which introduces significant synchronization overhead, especially when the simulation processors themselves are distributed. **Reducing the number of synchronization cycles is the key to improving the performance of parallel simulators.**

1.1.3 Object Tracking

Object tracking is a challenging and interesting application area for sensor network systems. Environmental scientists may be interested in tracking animal movements in an environmental surveillance network; military defense agencies may be interested in tracking vehicle movements in the battlefield [13]. However, tracking fast mobile objects in-network using resource lean sensors is difficult. It usually requires a high sampling rate, as well as a high degree of communication, both constrained by the limited resources available on typical sensors; moreover, the algorithms need to be decentralized, lightweight, and accurate. **Designing a distributed, efficient, and lightweight object tracking algorithm is the key to tracking fast moving objects in-network.**

1.2 Research Approach and Contributions

1.2.1 Classification

The main challenge addressed by our work centers on the mismatch between computationally intensive classification techniques and resource-constrained sensor nodes. We present a generic, *node-level* classification framework for resource-constrained sensors, such as the popular Tmote platform [10], with an MSP430 microprocessor. We present two different classification techniques and a preprocessing technique for accelerometer data. In the first design, we use a multi-dimensional Bayesian Classifier, which is relatively lightweight and suitable for resource-constrained devices. In the second design, we use an innovative classifier, a condensed kd-tree, which can reduce the number of leaves in a regular kd-tree by 90.0%. Using k-nearest neighbor search in the condensed kd-tree, we classify incoming events in $O(n^{\frac{1}{2}})$ time for the 2-dimensional case, where n denotes the tree size. The preprocessing technique uses a sliding window to smooth the accelerometer readings in $O(1)$ time; this improves the performance of the classifiers significantly. A post-classification voting method further improves accuracy. Both classifiers yield high classification accuracy in our case studies and reduce communication overhead and energy consumption when compared to the raw data collection approach. Moreover, since each classifier is generated in-network, re-training is energy-efficient.

1.2.2 Network Simulation

In this component of the dissertation, the main challenge addressed by our work centers on reducing the synchronization overhead in parallel simulation systems. Without using traditional interval synchronization, which introduces significant synchronization overhead, we present an optimistic distributed synchronization algorithm that uses probing execution and backtracking. Jefferson [33] has argued that general rollback is too complex and inefficient to implement. We show, however, that our design is capable of reducing the number of synchronization cycles significantly and improves performance by 2 to 10 times for representative WSN applications, compared to the interval synchronization algorithm.

1.2.3 Object Tracking

Finally, in the third focus area of the dissertation, the main challenge addressed by our work centers on designing a fast and efficient in-network tracking framework which detects and tracks moving objects. The framework is based on a distributed algorithm which tracks a dynamically scoped event region in the network. (An *event region* comprises a set of neighboring nodes which detect the target and report their sensor readings.) A leader is elected within each event region and adaptively scopes the region according to the speed of the target. The leader receives sample reports from nodes in the event region and estimates the target's position across the region. The tracking framework is adaptive; the sampling rate is adjusted based on the speed of the tracked object. It is also hierarchy-free, scalable, and does not require time synchronization. The object tracking task is executed on-node and is capable of capturing fast moving objects in-network.

1.2.4 Contributions

1.2.4.1 Contribution 1

(1) We develop a multi-dimensional Bayesian classifier on resource-constrained sensor nodes without transmitting raw data back to a host; the training and classification phases are both implemented in-network. (2) We design a new condensed kd-tree data structure and use k-nearest neighbor search as a classification function in-network. (3) We describe a general preprocessing technique for accelerometer data to improve the performance of both classifiers. The classification accuracy is above 85% for both classifiers in the two case studies considered.

1.2.4.2 Contribution 2

(1) We present an optimistic synchronization algorithm that abandons interval synchronization in favor of probing execution and backtracking. The number of synchronization cycles is reduced to an optimal number. (2) We implement *SnapSim*, an extension to the popular Avrora simulator, based on this algorithm. (3) We evaluate the performance of SnapSim using a series of standard sensor network applications from the TinyOS [46] distribution. The results show that for typical sensor network applications, SnapSim is significantly faster than the state-of-the-art simulator.

1.2.4.3 Contribution 3

(1) We present a novel distributed tracking algorithm which uses a dynamically-scoped event region tuned to the object's speed. (2) We present an adaptive snapshotting technique to estimate the target's location according to the speed of the object without the use of time synchronization. (3) We implement the tracking framework and evaluate the system on a large-scale testbed, capable of providing ground truth for fast moving light objects with high accuracy. The framework can be applied in other sensor networks using different proximity sensors with little adaptation.

1.3 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents background material related to classification, network simulation, and object tracking. Chapter 3 presents elements of related work in the relevant areas. Chapter 4 presents the in-situ classification contributions. Chapter 5 presents the distributed simulation contributions. Chapter 6 presents the distributed object tracking contributions. Finally, Chapter 7 concludes with a summary of contributions.

Chapter 2

Background

In this chapter, we present background material on classification, network simulation, and object tracking in sensor networks.

2.1 Classification

A great deal of work has focused on event detection in sensor network systems, particularly in the context of accelerometer data – also our focus. Due to their relatively inexpensive price, accelerometers are largely available for standard sensor nodes and mobile phones. By observing and analyzing accelerometer readings, rich information regarding movement, tilt, speed, and vibration can be extracted. Consider some of the representative application areas. People-centric event detection systems focus on the analysis of human movement using wearable sensor nodes. A wearable system is often able to detect walking, sitting, standing, and other human behaviors performed by the carrier [25, 28, 31]. This type of system is also applied in clinical research [20, 52], for instance, in studying the movement of Parkinson's patients undergoing particular drug therapies. Others have considered the identification of context information based on sensor movements. Nericell [57] uses a mobile phone accelerometer to identify traffic and road conditions when carried by a driver. Other application areas involve structural vibration monitoring of bridges [37, 38], buildings [71], roads [40], and even volcanos [69]. In these projects, imperceptible vibrations are collected, and events of interest are extracted and recorded.

To detect interesting events, an accelerometer must typically provide a high sampling rate.

If a 2-axis accelerometer is used with a 16-bit analog-to-digital converter (ADC), 240KB of raw acceleration data is produced if the sensor is sampled for 10 minutes at 100Hz. If a 30 byte packet payload is used, it requires at least 8000 packets to transmit this data, assuming single hop communication and zero packet loss. (In TinyOS [46], the maximum payload size is 28 bytes by default.) This is clearly not a feasible choice given the resource constraints of the target platforms.

As a result, feature extraction and/or data compression techniques are often applied [37, 57, 69]. However, these techniques often rely on time-consuming manual observation and analysis of the characteristics of the data. Further, these techniques often target data of a specific type (e.g., specific acoustic samples); solutions may not be transferable to other scenarios. Finally, for acceleration-based detection, a fixed node orientation is typically required. In contrast, as an alternative approach, machine learning and pattern recognition techniques enable automation and transferability. Machine learning techniques are used to generate classification functions based on empirical data collected during a training phase. The generated functions are used to classify incoming sensor data into one or more groups. However, traditional classification techniques are computationally prohibitive for most sensor nodes. One solution is to collect sensor readings and process the data on a PC/server [20, 29, 66]. The benefit is that well-developed machine learning algorithms can be applied to generate accurate classifiers. However, the cost of communication is high. One collateral effect is that the high cost, particularly in terms of energy, inhibits re-training, which is necessary in dynamic sensing environments.

An alternative is to use a mobile phone in place of a sensor node. The latest smartphones are equipped with 1GHz processors and 512MB of RAM [6]. Some data processing and pattern recognition techniques, such as FFTs and Markov models have been applied on these devices [53,55]. However, the typical device cost is significant. For instance, the HTC EVO 4G starts from $599 without a contract [6], compared with less than $100 [11] for typical "mote-class" platforms. Further, due to the sealed structure of smartphones, it is difficult to install new sensors to, for instance, use a more accurate accelerometer. Similar to the smartphone approach, resource-rich nodes are used in [40,56] to implement machine learning techniques in-network.

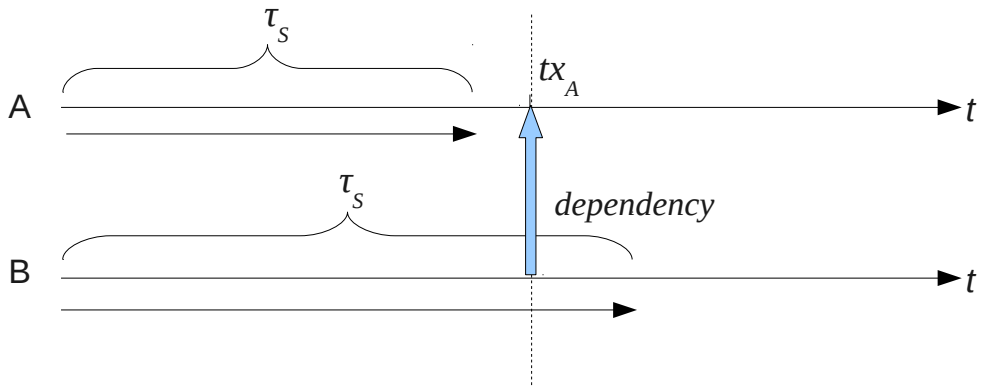


Figure 2.1: Distributed Simulation Dependencies

2.2 Network Simulation

In a typical parallel simulation system, each simulated process is executed within a unique simulation process. Due to communication between simulated nodes, temporal dependencies exist between simulated transmitters and receivers. Figure 2.1 illustrates an example of such a dependency. A and B represent two simulated nodes in a network; the horizontal lines represent simulated time. Suppose that node A performs a radio transmission at simulated time tx_A, and that node B's radio is on at tx_A. Assume that the process that simulates B runs faster than the process that simulates A. Specifically, assume that after τ_S units of physical time, node B has executed beyond tx_A, prior to node A reaching this point. Consequently, B misses the data transmission by A since it fails to read the channel at the right time. Thus, the instruction sequence at B is altered from what would be experienced in a physical run, and the simulation is erroneous. To avoid such problems, physical process synchronization is used to preserve causality paths between transmitters and receivers.

Consider a simulator that synchronizes each simulated node cycle-by-cycle; the dependency problem is easily solved. However, this strategy is inefficient, especially in distributed simulation scenarios, where synchronization incurs a physical networking delay. The state-of-the-art in parallel simulation allows simulated nodes to execute in parallel for a short interval before synchronization is performed. The interval is calculated based on the amount of time a node can execute without missing a transmission (in any possible run). As a result, all dependencies between nodes are

resolved, and no transmission will be missed. This approach is referred to as interval synchronization. Avrora [65], the most popular parallel simulator for sensor networks uses this technique.

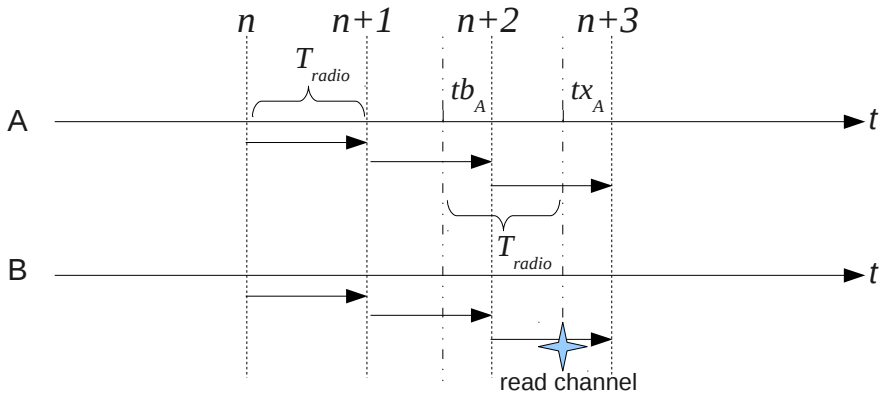


Figure 2.2: Synchronization Strategy in Avrora

In Avrora, the synchronization interval is defined by the time required for the radio chip to complete the transmission of one byte of data. Figure 2.2 summarizes this strategy. After each T_{radio} units of simulated time, a constant, all the nodes in the network are synchronized. Consider the following two cases: *(i)* Suppose nodes A and B are synchronized at the n_{th} synchronization point, and there is no transmission event between the n_{th} and the $(n+1)_{th}$ synchronization points. As a result, it is safe for both nodes to execute for another T_{radio} time. *(ii)* Suppose node A completes the transmission of a byte of data at tx_A, between the $(n+2)_{th}$ and the $(n+3)_{th}$ synchronization points. Since it will require T_{radio} time units to transmit the byte, the time at which node A starts to transmit the first bit, denoted by tb_A, must be between the $(n+1)_{th}$ and the $(n+2)_{th}$ synchronization points. Since node B is synchronized with node A at the $(n+2)_{th}$ synchronization point, it has a priori knowledge of the transmission time of A. As a result, no read channel event (a byte-based event) will be missed in the simulation.

2.3 Object Tracking

Mobile object tracking is a challenging and interesting application area for sensor networks. Representative applications include military vehicle tracking [13, 49] and wildlife tracking [36]. Typically, the target object is capable of being detected using low-level sensor signals. When a target moves across a deployed network, the sample values of nearby sensors are affected. The trajectory

of the target is reflected in the changing data trend. For example, an acoustic sensor may use the perceived volume level to indicate the proximity of an object. Existing tracking models fall into one of the following categories: *Binary proximity* sensing models provide a single bit of information, indicating whether the target is within the sensing range of a sensor. *Binary mobile* sensing models provide a single bit of information, indicating whether the target object is moving into or out of the sensing range of a sensor. *Directional* sensing models provide information about the existence of an object in a specific direction and may provide distance information, using, for example, laser or ultrasound. Finally, *proximity* sensing models provide real values related to the distance of an object to a sensor, using, for example, acoustic sensors or magnetometers.

Tracking objects in a sensor network using proximity sensors is non-trivial. First, the sensing model for proximity sensors is imperfect. Sensor readings are usually noisy and are susceptible to environmental influence. As a result, for most proximity sensors, the distance estimation metrics are not accurate. Second, since computation and storage resources are limited on typical sensor nodes, it is difficult for these devices to store and process the required data using cooperative signal processing techniques, especially if the result needs to be computed quickly. Third, the limited radio bandwidth available to typical sensor networks constrains the allowable communication patterns. Consider, for example, the Telosb Platform [3], a typical wireless sensor node with a CC2420 radio chip [9]. The device has a maximum data rate of no more than 35kbps when programmed using TinyOS [47]. Even when a high sampling rate can be achieved, the limited bandwidth constrains the interaction among nodes. Further, congestion may occur at high data rates and produce unexpected results. Fourth, as the target moves from region to region, a synchronized time reference becomes important for estimating speed and location, especially for fast moving objects. However, time synchronization increases energy and resource consumption. Finally, large-scale sensor networks require scalable tracking solutions. For example, a naive centralized approach which requires all nodes that detect an object to send sample data back to a basestation through a collection tree may not be a good fit.

Chapter 3

Related Work

In this chapter, we briefly summarize the most relevant related work.

3.1 Classification

A number of other authors have investigated event detection using classification-based techniques.

Ganti et al. describe SATIRE [25], a software architecture for wearable sensor networks that includes services for accelerometer sampling, data storage, and data transmission, as well as a web-based data portal. A Hidden Markov Model (HMM) is used to classify human activities and find possible hidden states – unobserved states under the assumption of a Markov process; the processing is performed by an upper-tier host application.

A similar approach is seen in the work of He et al. [29]. The authors use the Viterbi algorithm [24] to find the most likely sequence of hidden states in a HMM, and again, the algorithm is applied on an upper-tier host. A sliding window preprocessing scheme that computes the arithmetic mean within each interval is applied to reduce the communication overhead between nodes and the host. The Tmote Invent platform is used as a wearable device in their project, the same type of sensor used in our work.

Lorincz et al. describe Mercury [52] to sense abnormal patient activity using a wearable sensor network. Sensor nodes log all collected data to flash storage and transmit a small portion of the collected data back to a host server; five standard features are extracted on the sensor nodes.

The authors describe a throttling driver that coordinates data downloads based on configured feature thresholds and a target battery lifetime.

Borazio and Laerhoven [17] describe a wearable sensor system used to observe long-term sleep behavior. The sensor records acceleration data and ambient light, which is then combined with infrared images from an external camera to classify sleep segments and posture changes. The data is first processed using a threshold-based classifier, and then classified using a Hidden Markov Model. Sleeping postures are clustered using a Kohonen Self-Organizing Map, which iteratively updates the clusters using new sample data.

None of the above systems include in-network machine learning; event detection is delegated to a host server.

Miluzzo et al. present CenceMe [55], a smartphone application designed to detect user-centric events using audio and accelerometer data. A partially *on-phone* classification algorithm is implemented in their work. Classifier training is performed on a desktop machine, and a decision tree is generated using the J48 decision tree algorithm [70]. The generated decision tree is exported to a resource-rich smartphone, which processes raw data using a Discrete Fourier Transform (DFT) and classifies the resulting data.

Lu et al. present SoundSense [53], an event detection application that classifies daily environmental sounds using a smartphone. This application preprocesses raw sound data and uses coarse classification to classify the resulting data into groups. It then classifies each group into finer "intra-category" subdivisions. Unrecognized sounds are categorized into new classes based on a Mel Frequency Cepstral Coefficient (MFCC) feature vector [51]. The system is capable of distinguishing a number of common sounds. The resulting event information is then used in a social networking context. The classification is performed on the sampling device. It is unclear in their presentation where the training phase is performed.

Seeger et al. present myHealthAssistant [61], a health care system used to record exercise activities, such as running, cycling, and weight lifting. The system comprises a smartphone, inertial sensor, accelerometer, and heartrate sensor. The sensing unit computes basic statistics (i.e., mean, standard deviation, peaks) and transmits the results to the smartphone. The smartphone detects events using threshold-based techniques and identifies the events using a Guassian-based classifier. To identify complicated exercise activities, the authors use 3 inertial sensor units – a right knee sensor, a sensor strap around the torso, and a glove sensor.

The above three systems include in-network classification, but rely on resource-rich smartphones with orders of magnitude more resource capacity than typical resource-lean sensors.

Mohan et al. present Nericell [57], a system used to monitor road and traffic conditions using GPS, microphone, and accelerometer data from a smartphone. Accelerometer data is used to detect braking events and road bumps. A key contribution in their paper is the presentation of a 3-axis accelerometer reorientation algorithm. Machine learning is not used; the classification results stem from manual analysis of the features of the sensor data. For example, bump detection is achieved by comparing vertical acceleration readings with an acceleration threshold based on traveling speed, derived from empirical observation.

Kim et al. [40] present a classification approach used to detect military vehicles using acoustic and seismic sensors deployed on a road. The Gaussian Mixture Model (GMM) algorithm [60] is applied as a basic classifier, and the resulting likelihood measurements are processed through a decision tree generated using the Classification and Regression Tree (CART) [18] algorithm. The computation is performed on an upper-tier host. Kim et al. [37] present work focused on vibration monitoring using sensors attached to a bridge. High-rate accelerometer sampling and data transmission techniques are used in their approach. Event detection and classification are performed on a resource-rich host.

3.2 Network Simulation

Avrora [65] is an instruction-level, cycle-accurate sensor network simulator, capable of simulating multiple sensor nodes in parallel. Avrora emulates the underlying hardware platform to achieve cycle accuracy. It takes as input the compiled microcontroller code generated for the target hardware – the popular AVR architecture. Avrora's portability can be attributed to the fact that it is implemented in Java. Each simulated node is run as a separate Java thread. The event scheduler maintains a global cycle counter to represent the logical time of sensor nodes. The authors describe two problems for parallel simulation – the "send-receive" and "sampling" problems. The send-receive problem is a dependency issue caused by transmissions, as discussed in chapter 2.2. The sampling problem is an analogous issue associated with sampling the radio's RSSI register. In Avrora, interval synchronization is used to solve the send-receive problem. However, interval synchronization alone does not fully address the RSSI sampling problem since the signal attenuation model requires "state

of the air" at the point where RSSI is sampled. Since sampling RSSI is relatively rare in a common sensor node application, some efficiency can be sacrificed. The authors adopt a simple wait-for-neighbors strategy for the sampling node, which waits at the point where RSSI is sampled until all other nodes in the network report they have passed that point.

Legedza et al. [44] present two techniques to reduce the overhead associated with interval synchronization in a parallel simulation environment. The first technique, *local barriers*, provisions adjacent processes in a simulated network on the same processor, which reduces inter-process communication. The second technique, *predictive barrier scheduling*, predicts the amount of time that a process will not communicate using a combination of compile-time and runtime analysis. This enables the application of longer synchronization intervals. However, these two strategies are evaluated using several simple cases, not in a full-scale sensor network implementation.

Jin and Gupta [35] describe PolarLite, a distributed simulation framework built on Avrora, which uses a synchronization strategy similar to predictive barrier scheduling, and hence improves simulation speed. PolarLite reduces the number of synchronizations performed between processes, using Avrora's interval synchronization strategy as a starting point. The authors introduce two longer synchronization intervals based on radio initialization time and MAC backoff time, in addition to Avrora's basic strategy. Radio initialization time is defined as the the shortest interval required to turn the radio back on when the radio is off; it is 11 times the interval constant T_{radio}. When the radio is turned off, no transmission or packet reception event can be performed on the node. MAC backoff time is defined as the randomized backoff time introduced at the MAC layer when a collision is detected; it is 1 to 32 times the interval constant T_{radio}. When a collision happens, no transmission will be performed, and no data will be received inside the interval. Introducing these longer intervals, a simulated node can proceed for a longer period of time without synchronization. However, PolarLite uses the new synchronization strategy only if one of the above scenarios occurs; otherwise it will use Avrora's constant interval synchronization strategy.

A number of other authors have contributed to the state-of-the-art in sensor network simulation and parallel simulation techniques. Here we briefly describe some of the work most relevant to ours.

Before their work on PolarLite, Jin and Gupta [34] improved the interval synchronization strategy by making use of sleep time on sensor nodes. No transmission or reception can occur when the microcontroller and the radio chip are in deep sleep states, and most interrupts are disabled. In

this case, the wake-up time can be predicted; hence, longer synchronization intervals can be used. However, this strategy only works for a few special cases; it is not a general solution.

Wen et al. [68] introduce DiSenS, a distributed cycle-accurate sensor network simulator, which uses the local barrier synchronization technique introduced in [65], as well as interval synchronization. Each node periodically broadcasts its clock value; a node which reads the channel waits until all other (neighboring) nodes reach the same point. DiSenS partitions simulated nodes based on neighborhood connectivity. The nodes within each strongly-connected sub-graph are simulated on the same processor. The effectiveness of the partitioning algorithm is largely dependent on the topology of the simulated network.

Levis et al. [45] introduce TOSSIM, a simulator for TinyOS applications. TOSSIM is built into the TinyOS distribution and simulates native code, replacing hardware-dependent components with corresponding simulated components. TOSSIM is neither parallel, nor cycle-accurate; no synchronization techniques are used.

Landsiedel et al. [43] present TimeTossim, an automated instrumentation simulator, which improves the simulation fidelity of the TinyOS simulator TOSSIM [45]. In contrast to a cycle-accurate emulator, which emulates compiled instructions at the hardware-level, TimeTossim simulates cycle accuracy by estimating the number of cycles consumed by each line of source code. The estimated cycle counts are not perfect, but significantly improve accuracy compared to simulation without any cycle counts.

Shnayder et al. [62] describe PowerTOSSIM, an extension to TOSSIM [45] for simulating per-node power consumption. PowerTOSSIM consists of four components. The first instruments the simulated TinyOS application to determine the power consumption of hardware peripherals. The second statically counts CPU cycles executed by each node. The third estimates per-node energy consumption based on an energy consumption model that uses the CPU cycle count and power consumption of hardware peripherals. Finally, the fourth visualizes power consumption.

Bajaj et al. [16] introduce GloMoSim, a library for parallel simulation of large-scale heterogenous networks. GloMoSim uses a layered architecture that mirrors the OSI model. To support rapid parallel simulation, each processor simulates multiple geographically-located nodes. A variant of local barriers, this reduces inter-processor message passing.

Liu and Nicol [48] introduce SWAN, a direct-simulation environment for simulating wireless ad hoc networks, in which the actual application code can be simulated. SWAN uses various protocols

from GloMoSim. The authors explore three lookahead techniques: *(i)* short radio range, in which nodes make use of the need to receive, buffer, and retransmit a message to reach the destination node, *(ii)* conditional lookahead, where nodes make use of the finite state machine which protocols follow, and *(iii)* message reception, in which nodes make use of the need to wait for a message to be received before using the channel for sending.

In other work, Liu et al. [50] validate the fidelity of SWAN. Actual applications are compared to simulation results. The authors use five routing protocols and three radio propagation models to validate SWAN's fidelity. The authors conclude that simple radio models can support accurate simulation. However, routing protocols are sensitive to the parameters of radio models. Accordingly, the authors suggest using more complex radio models that incorporate path loss information. Additionally, empirical measurements of the environment help increase simulation fidelity by identifying the values of the parameters to radio models.

Naoumov and Gross [58] introduce an extension to ns-2 to simulate routing protocols in large ad hoc networks. To overcome the simulation speed limitation caused by increased network size, the authors investigate two ways to organize nodes: grid-based and list-based. Grid-based organization is achieved by dividing the simulation area into cells. List-based organization is achieved by using a doubly-linked list of nodes ordered based on their X-coordinate location. The authors show that list-based organization leads to improved simulation speed compared to grid-based organization, especially in large networks.

Jefferson [33] presents a "*time warp*" algorithm to synchronize distributed processes using lookahead and rollback mechanisms. In his algorithm, the state of the process is stored in a state queue each time an event occurs. Incoming and outgoing messages are recorded in an input queue and an output queue, respectively. If a late-arriving message, with a virtual time-stamp earlier than local virtual time, is received, the receiving process must rollback to a previous stored state. "Anti-messages" are sent to other processes in order to rollback side-effects caused by the transmissions sent after the rollback point. Reception of an anti-message may trigger other rollbacks, as well as additional anti-messages. However, this algorithm requires infinite storage, and the implementation complexity is high.

3.3 Object Tracking

Object tracking is a classic problem in sensor networks. Guibas [26] proposed the problem of object localization and tracking in this context. Before that, Herlihy and Tirthapura [32] introduced a self-stabilizing algorithm for maintaining a distributed queue in a network. This is a non-sequential flow problem that shares a number of similarities with distributed object tracking problems.

Since then, a number of authors have investigated object tracking algorithms in sensor networks using a range of tracking models. A variety of sensors with different capabilities and computational resources have been considered.

Aslam et al. [14] present a centralized object tracking algorithm based on particle filtering techniques. The authors assume binary sensors capable of indicating whether an object is moving into or out of the sensor range. As a result, two convex hulls of "moving out" sensors and "moving in" sensors are generated. The location of the target is estimated to be inbetween the two hulls, with a position weighted by the time duration the object was in the sensing range of each sensor. The algorithm was evaluated using simulation.

Kim [39] presents a tracking algorithm using a single bit sensor which indicates whether an object is in sensing range. The author assumes constant object speed. The location of the object is estimated by a weighting algorithm that relies on the time the object spent in the sensing range of each sensor. This algorithm was implemented using acoustic sensors on resource-lean sensor nodes. Shrivastava et al. [63] also investigate a target tracking algorithm using a binary proximity sensor. The authors show that the detection error is inversely proportional to sensor density. The authors present a linear "stabbing" algorithm to estimate the trajectory of the target, as well as its speed. The evaluation is conducted on a physical testbed using acoustic sensors. However, their approach is not scalable and does not achieve high precision. Wang et al. [67] improve the scalability of binary sensor target tracking algorithms with a distributed algorithm. Possible variations in sensor detection errors are also considered in estimating object location in their approach.

Singh et al. [64] present a target tracking algorithm for multiple targets using binary proximity sensors. The authors analyze the possibility of tracking targets of different speeds using single bit proximity information and design a multiple target tracking algorithm using particle filters. However, they only investigate the cases in a 1-D space.

Plarre and Kumar [59] present an object tracking solution based on a directional sensing

model. The authors use a combination of laser and light sensors to detect when an object passes a directional sensor. Speed and location are estimated by the time the target object spends cutting the laser line. However, this work requires special deployment of the network and only handles straight trajectories.

Liu et al. [49] present a distributed tracking system to track military vehicles. In this approach, a leader node is selected to estimate the current location of the object and computes the next estimated location using a sequential Bayesian filter (Kalman). If the next location is closer to another sensor, the leadership will be passed onto that sensor. In their implementation, acoustic amplitude sensors are used to estimate the proximity of targets, and directional acoustic sensors are deployed to provide direction information. Brooks et al. [19] present a distributed object classification and tracking framework using collaborative signal processing techniques. The authors use extended Kalman filters to compute next-step nodes. The nodes which detect the object are set to active and alert nearby nodes using prediction results. To save communication bandwidth, the authors limit the scope of relevant nodes through spatial subdivision. However, the prediction techniques require resource-rich sensor nodes.

Arora et al. [13] present a military surveillance system which comprises detection, classification, and tracking functions. To detect military vehicles and soldiers, magnetic sensors and radar units are used. After detection of a target by the classifier, the target location is decided by the centroid of the convex hull of sensors which detect the target. To limit noise, a heuristic bounding box is applied to tighten the size of the convex hull. A large number of resource-lean sensors were deployed in this project. In their later work, Kulathumani et al. [42] present an in-network object tracking and querying service for this system. The algorithm is distributed and hierarchy-free and enables target locations to be queried from any node in the network. The query time is linearly proportional to the distance between the query node and the target. Sensor nodes maintain a tracking structure which comprises small segments of paths. To query the object in the network, the *find* operation incrementally increases the query radius until the trail is intersected, then follows the trail to the object. However, it is inefficient for tracking fast moving objects because updating the trail in-network is costly.

Demirbas et al. [22] present a self-stabilizing tracking algorithm for hierarchical sensor networks. The authors assume that the network is hierarchically partitioned in space. The tracking information is propagated from lower levels to higher levels. The algorithm slows down information

propagation as hierarchy levels increase. Recent information can override misinformation at higher levels. However, the operations of hierarchy-based algorithms are energy-consumptive at cluster boundaries.

He et al. [30] present a real-time object tracking framework designed in an energy-efficient manner. The system adopts a power management protocol to switch sensor nodes to sleep when they are idle. The authors also introduce an efficient wake-up scheme and group aggregation technique to increase detection accuracy with limited delay. The end-to-end detection deadline is guaranteed by the system. However, the detection delay may be significant for capturing objects at fast speeds.

To our knowledge, the only work focused on light source tracking in sensor networks was presented by Gupta and Das [27]. Photo sensors are used as proximity sensors for detecting the light source target. They use triangulation to estimate the target position and fit the points with a straight line to estimate the trajectory. The experiments are tested on a small scale network, and it is not clear if the algorithm is scalable.

Other work introduces active tracking systems which track sensor nodes that periodically send wireless radio [73] or ultrasound signals [72]. The target object is a sensor itself which can communicate – not our scope of concern.

Chapter 4

Classification

In this section, we describe two *node-level* (in-network) classification algorithms for resource-constrained sensor nodes[1]. We focus on the use of accelerometer data to complete two different tasks: (1) detection of human activities, and (2) detection of driving events. Our work assumes the absence of a resource-rich basestation.

4.1 Multi-dimensional Bayesian Classifier Design

The first classification technique, the Bayesian classifier, uses a probabilistic pattern recognition model. The approach relies on strong independence assumptions over the input dimensions and is broadly used in a variety of applications. The approach consists of two basic stages: classifier training and data classification. The training phase involves processing training data elements tagged with their respective class designations; a classifier is generated as the output of this stage. The classification phase uses the classifier to assign incoming data elements to their respective classes.

4.1.1 Basic Principles of Bayesian Classification

Here we present an overview of Bayesian classification, adapted from [23]. A Bayesian classifier applies Bayes' rule to optimize the posteriori probability that a set of data belongs to a

[1]The inherent advantage to in-network classification is the ability to apply the techniques across a broader range of computational devices without the need for supporting high-bandwidth network and computer infrastructure. With the tremendous growth in the smartphone and wearable computing markets, the applications for such techniques are promising. Indeed, resource-lean devices for in situ classification have already proven to be commercially viable [5].

particular class by assigning a set of appropriate discriminant functions. It constructs a decision boundary between two data classes by assuming an optimal statistical model of posteriori probability $P(c_i|x)$, which denotes the probability that, given a data sample x, the sample is a member of class c_i. A sample x belongs to class c_i if and only if, for any other class c_j

$$P(c_i|x) > P(c_j|x) \tag{4.1}$$

To determine the posteriori probability $P(c_i|x)$, Bayes' rule of posteriori probability is applied,

$$g_i(x) = P(c_i|x) = \frac{P(x|c_i)P(c_i)}{P(x)} \tag{4.2}$$

where $P(c_i)$ is the priori probability of membership in c_i, $P(x|c_i)$ is the posteriori probability of x given that x belongs to class c_i, and $P(x)$ is the priori probability of sample x.

In a multi-dimensional pattern space, the posteriori probability $g_i(x)$ is a discriminant function that yields higher values for the set of data belonging to the class i. By equating discriminant functions, a decision boundary is defined. By assumption, the priori probability $P(x)$ is the same for all data samples; it can be ignored in the formulation. $P(c_i)$ denotes the percentage of samples belonging to class c_i. To determine the posteriori probability $P(x|c_i)$, Mahalanobis distance [23] is introduced, which defines a normalized distance to the class centers to measure priori probability:

$$d^2 = (x-\mu)^T \sum^{-1} (x-\mu), \tag{4.3}$$

where μ is the class center vector, T denotes transposition, and $\sum^{-1}$ denotes the inverse covariance matrix over the input data. Differing from Euclidean distance, Mahalanobis distance is based on correlations between patterns; it is scale-invariant. By assumption, the posteriori probability $P(x|c_i)$ conforms to a multivariate Guassian distribution on Mahalanobis distance. Consequently, by taking the natural logarithm (and ignoring $P(x)$), the discriminant function $g_i'(x)$ is transformed to the following form:

$$g_i'(x) = -\frac{1}{2}(x-\mu_i)^T \sum_i^{-1}(x-\mu_i) - \tfrac{D}{2} log(2\pi) - \tfrac{1}{2} log|\sum_i| + \log P(c_i) \tag{4.4}$$

where $\sum_i$ denotes the covariance matrix of class c_i. A decision boundary is generated by equat-

ing any two discriminant functions. For N measurements, class assignment is based on a series of comparisons determined by decision boundaries. If there are k distinct classes, there will be $\binom{k}{2}$ decision boundaries generated. Class assignment is based on the intersection of the generated decision boundaries.

The resource-constraints of common sensor nodes suggest the need for simplicity in constructing *node-level* classifiers. The Bayesian approach is a good fit. In the training phase, the most complex computation is the covariance matrix inversion, where the dimension of the matrix equals the dimension of the pattern space being classified. In the context of in situ sensing, the dimension of the pattern space is often no larger than 3, which is suitable for computing Bayesian classifiers *on-node*. The classifier can be generated incrementally without all training data in memory since the class center vectors and covariance matrices can be updated incrementally. The training data can be discarded after the classifier is generated. The final representation of the discriminant function contains at least 2^d coefficients, where d is the dimension of the pattern space (by the number of terms in the discriminant polynomial). Therefore, the dimension of the pattern space must typically be small. For most classification tasks, the memory resources of a typical sensor node are sufficient to store all of the coefficients. When a node is able to compute its classifier, there is no need to transfer raw samples back to the basestation, dramatically decreasing energy consumption.

4.1.2 Preprocessing

While this approach is suitable for the target hardware platforms, raw accelerometer data is often difficult to classify directly. Consider, for example, attempting to determine whether a target is walking or running based on accelerometer data collected from a sensor carried by a human target. Figure 4.1 shows training data and corresponding preprocessed data collected from a simple trial: The user carried a sensor node in his pocket, training the classifier by walking and running for several seconds. The original acceleration readings are represented in 2 dimensions in Figure 4.1a. A large number of samples are "mixed together" in the middle of the plane since the arithmetic mean of a series of vibration data settles at a fixed point, as shown in Figure 4.1b. Meanwhile, the standard deviation of the data across the classes is relatively large, scattering the readings throughout the space. Since the Bayesian classifier is based on the Mahalanobis distance from class centers, the classifier will be error-prone if it is generated on raw accelerometer readings. As seen in Figure 4.1a, the decision boundary is the tiny spot in the middle of the plane, which gives no useful information

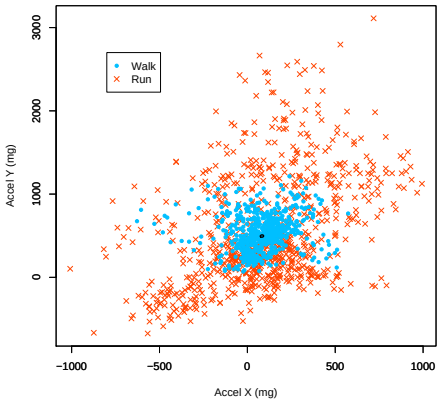
for classification.

To construct an accurate classifier, we must separate the data centers of the two classes. More precisely, the goal is to differentiate the arithmetic means and decrease the standard deviations across the two groups. For this purpose, we use *jerk*, the rate of change in acceleration:

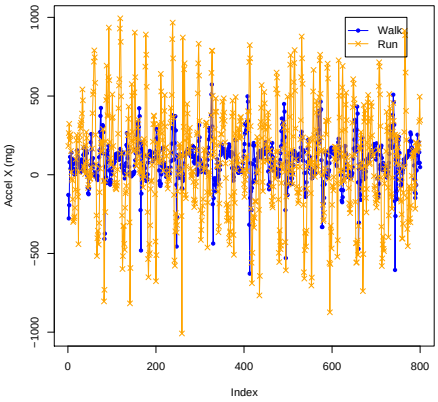
$$j = \frac{\mathrm{d}\vec{a}}{\mathrm{d}t} = \frac{\mathrm{d}^2\vec{v}}{\mathrm{d}^2t} \tag{4.5}$$

By transforming the raw accelerometer readings to absolute values of jerk, $|j|$, we can separate the arithmetic means of the two groups, as shown in 4.1d, making it possible to construct a proper decision boundary. As shown in Figure 4.1c, the new classifier forms a decision boundary, shown as a black line in the graph that cuts between the two groups of data. (Note that the unit of jerk is milli-graviton per sample, the changing rate of acceleration.)

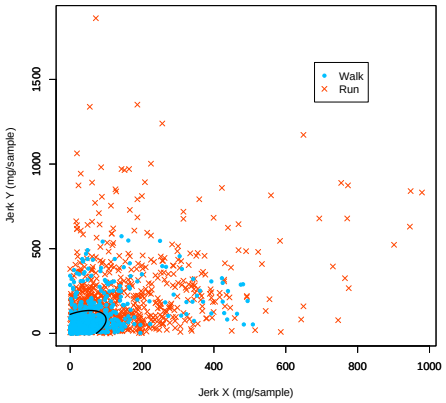
However, even if the arithmetic means are separated, high standard deviations may cause the two groups to overlap, as shown in Figure 4.1c and Figure 4.1d. The performance of the classifier can be further improved by computing the mean value of jerk over a fixed size moving window. Consider, for example, a window of size 50, corresponding to a 500 ms period when the sampling rate is 100Hz. This scenario is illustrated in Figures 4.1e and 4.1f. The standard deviation of the jerk data is smoothed, further separating the data groups. Indeed, the two groups are almost completely separated, and therefore, the resulting decision boundary correctly splits them. In general, the groups can be further separated by increasing the window size. However, the effective sampling period increases with window size. We typically choose the window size in the range of 40 to 80. During preprocessing, we store jerk data in a circular queue; the arithmetic mean of jerk can be updated in $O(1)$ time – which is suitable for the fast sampling rate of typical accelerometers. This technique presents a general method to construct classifiers on roughly periodic data with close arithmetic means and large standard deviations. It is also employed by the kd-tree classifier described in the following section. With preprocessing, both classifiers achieve accuracy of more than 85% in the case studies considered, as we will see.



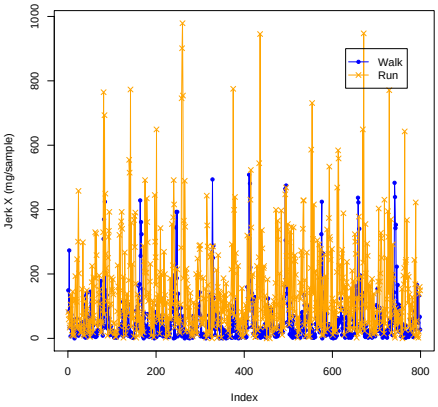
(a) Classifier (acceleration)



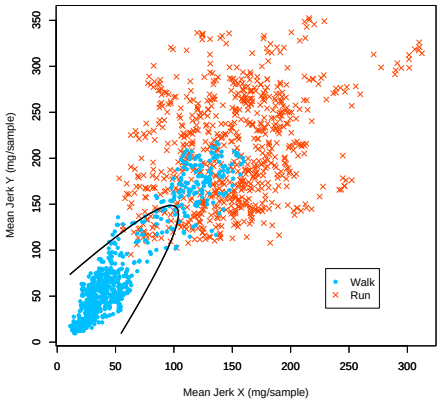
(b) Acceleration (X-axis)



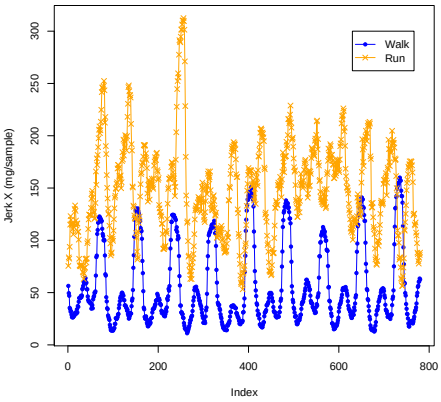
(c) Classifier (jerk)



(d) Jerk (X-axis)



(e) Classifier (jerk, sliding)



(f) Jerk (sliding, X-axis)

Figure 4.1: The Impact of Preprocessing

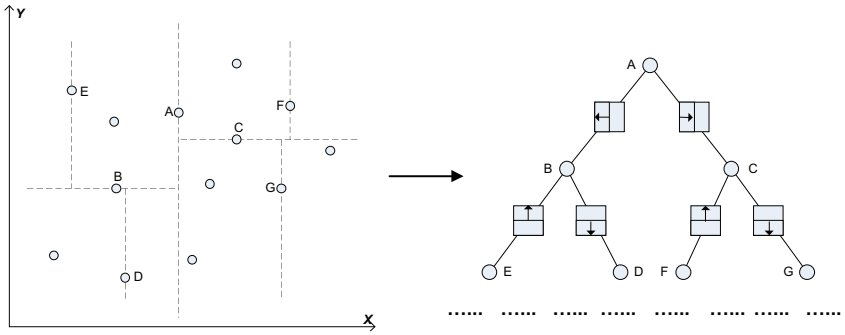


Figure 4.2: kd-Tree Data Structure

4.2 Condensed kd-Tree Classifier Design

In this section, we present the design of the condensed kd-tree classifier. We employ a condensing technique to store the tree using limited memory. The classifier training phase involves the construction of a classification tree based on pre-processed samples tagged with their respective class designations. The classification phase again uses the classifier to assign incoming data elements to their respective classes.

4.2.1 kd-Tree Data Structure

A kd-tree is a binary tree data structure broadly used to solve geometric problems, and can be used as a classifier using nearest-neighbor search [15, 41]. A kd-tree is built by alternately splitting the point set (sample set) in one of the dimensions of the pattern space. As shown in Figure 4.2, a kd-tree node splits the point set evenly into two sets, L and R, based on the median x coordinate, then y coordinate, and splits the resulting sets on x again, and so on, cycling through the dimensions. Each node corresponds to a rectangular region of the plane; child nodes again partition the parent region. A range query recursively visits all partitions of the tree that intersect the query range and returns all the visited nodes in the range; the worst-case query time is $O(n^{1-1/d})$ in d dimensions.

Tree balance is important for improving the traversal speed for random queries. A balanced kd-tree can be constructed by recursively finding the median sample within each region and splitting

on that sample. To expedite tree construction, we use fast randomized construction to insert the nodes. Our observations show that the randomness of typical accelerometer readings is sufficient for constructing a roughly balanced kd-tree by inserting the readings in sample order since the data oscillates for typical in-network sensing tasks. As a result, the randomized construction runs in $O(n \log n)$ time since the height is $O(\log n)$.

4.2.2 Nearest Neighbor Classification

We first describe the process of nearest neighbor classification in an uncondensed kd-tree. A kd-tree can be used as a classifier using *k-nearest-neighbor* search in the pattern space, where each dimension of the tree corresponds to a dimension within the pattern space. In our case, points belonging to different classes are stored in the same tree. We begin by abandoning the standard requirement of a *fixed* k-neighbor search. Instead, we introduce a dynamic neighborhood mechanism: We define D as the *neighborhood threshold*. A node j is a neighbor of node i iff the Euclidean distance from node i to node j is less than D. For a node x, we define the magnitude $m(c_i)$ of class c_i as the number of neighboring nodes associated with c_i, weighted by their respective distances from x. The likelihood of a point belonging to a class is proportional to the magnitude of the class within its neighborhood. Let $P(c_i|x)$ denote the posteriori probability that point x belongs to class c_i, and d_p denote the Euclidean distance from point x to some neighboring point p. The number of neighboring points associated with class c_i is proportional to $P(c_i|x)$, and the distance d_p is inversely proportional to $P(c_i|x)$. Intuitively, the weight can be defined as $\frac{1}{D+d_p}$, where the inverse proportionality is linear. To exaggerate the proportionality, we impose an exponent of 2 as a penalty on d_p. As a result, we must add an exponent of 2 on D. Using D^2 as the numerator, the weight function is confined to the range [1/2, 1]. We evaluate the magnitude of each class by taking the summation of neighboring nodes associated with the class, appropriately weighted:

$$P(c_i|x) \sim m(c_i) = \sum_p w(c_i) = \sum_p \frac{D^2}{D^2 + d_p^2} \tag{4.6}$$

In a kd-tree, k-nearest neighbor classification is an alternative form of a range query, which has worst-case $O(n^{1-1/d})$ running time. Classification begins with the *find-node* operation. *Find-node* performs an inexact search to find the query point in the tree; it stops at the node which defines the minimal region containing the query point. Call this node Q. Next, a breadth first

Domain	***Attribute***	***Description***	***Modified***
key_value[ ]	*int [D]*	invariant key values	Original
cond_value[ ]	*float [D]*	condensed mean values	Augmented
count[ ]	*int [C]*	count of condensed points of each class	Augmented
left	*struct kdNode **	left child pointer	Original
right	*struct kdNode **	right child pointer	Original
parent	*struct kdNode **	parent pointer	Original

Table 4.1: kd-Node Data Structure

search of the k-nearest neighbors of the query point is performed, beginning from Q. To support breadth first search, we use back links between children and parents. The search traverses the tree using normal breadth first search. To support dynamic neighborhoods, we use neighbor threshold distance D to eliminate branches that are out of range. The search process runs in $O(n^{1-1/d})$ time in d dimensions, the same as the range query. This running time is acceptable with a small number of nodes in a low-dimensional space.

4.2.3 Condensed kd-Tree

Due to the memory limitations of common sensor nodes, it is typically infeasible to construct a complete kd-tree using raw accelerometer data. For example, if we collect 2000 samples, and each tree node requires 12 bytes, the tree will consume approximately 24KB – far beyond the available memory of the MSP430 (10KB). To overcome this, we introduce a condensing technique.

To reduce the size of the tree, a *merge* operation is introduced. We merge each new node with an existing node if the new node is within the *condensing radius* of the existing node. The node structure is shown in Table 4.1: *key_value* stores the sample values originally used to define the node. *left*, *right*, and *parent* store pointers to the left child, right child, and parent, respectively. To support condensing, we augment each kd-tree node with two new fields: *count* is an array that stores the frequency of class membership for the samples merged with the node. The *cond_value* field stores the arithmetic means of all sample values represented by the node. The original sample values (*key_value*) serve as invariant keys in the condensed tree. This invariant property is necessary since *cond_value* cannot be used as a key in the search; it may change after a merge operation.

We construct the tree by inserting the data samples in sample order. The *insert* operation traverses from the root to the leaves. The new node will be merged if it is within the condensing radius of an existing node's key values. The *merge* operation updates the condensed mean values

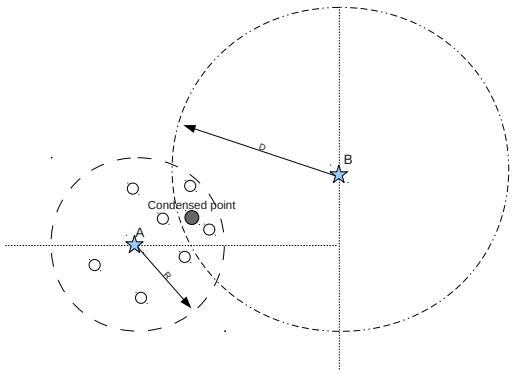


Figure 4.3: Condensed kd-Node

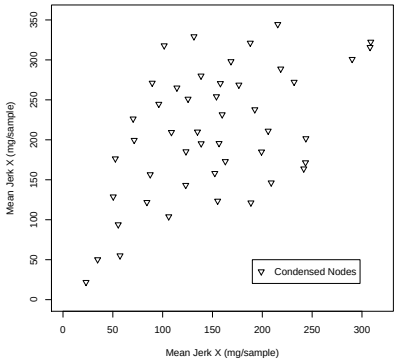


Figure 4.4: Condensed kd-Tree

stored by the existing node and updates the *count* element of the associated class. If a *merge* cannot be performed, the *insert* operation proceeds as usual. In either case, *insert* takes $O(\log n)$ time. As discussed in Section 4.3.1.2, it requires little time to search in our condensed kd-tree.

Figure 4.3 illustrates the *merge* operation. Assuming a 2-dimensional pattern space, point A denotes a pair of invariant key values in the tree; R is the condensing radius. The hollow points around A represent the values associated with A prior to insertion. The condensed values, shown as a filled circle, are adjusted after every merge. In the condensed kd-tree, the key values maintain the tree property, while the condensed values represent the mean of all values merged with the original node. Since the condensing radius limits the allowable distance of merge candidates, the condensed values will not be pulled outside the condensing radius, which avoids reconstruction of the tree.

To accommodate potential "border crossings" induced by the merge operation, the classification operation must be adapted. We again use inexact search to find the node containing the most similar key values, and then begin a breadth first search to query the neighboring nodes in range. The search is based on key values. However, the associated condensed values are likely to be different, but not by a distance greater than the condensing radius R. If a condensed point has been skewed into the neighborhood threshold of a given node, while the key values are out of range, it is possible to miss a neighboring node during the search process. Figure 4.3 illustrates this situation. The condensed values for node A are represented by a filled circle in the graph; B is the invariant key of another node, and D is the neighborhood threshold. When a breadth first search reaches B (prior to A), A and its subtree will not be visited since it is out of range of B's neighborhood threshold. However, the condensed value should be counted when calculating the magnitude since it

is in range. To overcome this, we use $D + R$ as the neighborhood search range instead of D during the inexact search query. The classification still runs in $O(n^{1-1/d})$ time. Further, the magnitude function needs to consider the number of merged nodes associated with class c_i, denoted by Φ_{c_i}:

$$P(c_i|x) \sim m(c_i) = \sum w(c_i) = \sum \frac{\Phi_{c_i} D^2}{D^2 + d_p^2} \tag{4.7}$$

To improve accuracy, R should be less than D. At the same time, if R is too small, the *condensing ratio* of the tree, i.e., one minus the ratio of the condensed size to the original size, will be low. Considering the tradeoffs, we choose the condensing radius R to be less than or equal to half of the neighborhood threshold D in our case studies.

Recall the trial to detect walking and running events discussed in Section 4.1.2. Using this preprocessed dataset, we constructed a condensed kd-tree, illustrated in Figure 4.4. With a condensing radius of 7, the size of the tree is reduced by more than 95%.

4.3 Case Studies

4.3.1 Case Study 1: Human Movement

The goal of the first application case study is to identify the *walking*, *running*, and *jumping* activities of a carrier. As introduced in Chapter 2, detecting human movements is a common task for wearable sensor systems [25,28,31]. In this case study, we use Tmote Invent sensor nodes [10], each equipped with a 2-axis accelerometer. The accelerometer provides measurements in the range of $\pm 5g$ in the X-Y plane of the device. As shown in Figure 4.5, the carrier can put the sensor node into a pocket, or hang it on a neck strap. The orientation of the sensor node is not required to be vertical or horizontal. The carrier simply needs to make sure that the node stays at the same position and direction for training and detection. If the position or direction are changed, the classifier must be retrained.

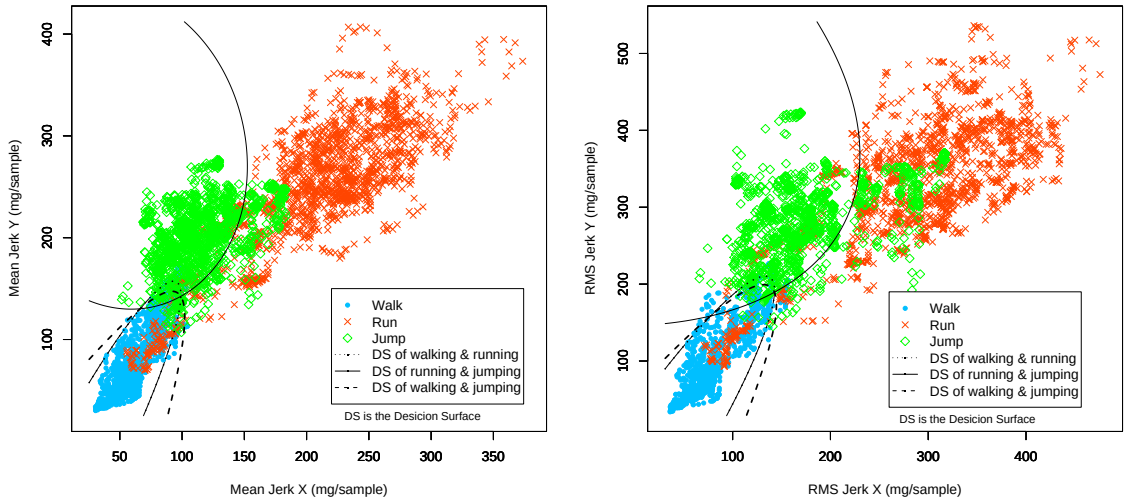
4.3.1.1 Implementation of the Bayesian Classifier

We implemented the Bayesian classifier in-network using the Tmote Invent platform [10]. The sampling rate was set to 100Hz, as recommended by the hardware manual. For each class, the training phase collects 1500 samples (15 seconds) in a sampling buffer, which consumes 6KB

Figure 4.5: Tmote Invent Carried by User

of memory. The node computes jerk using the sliding window preprocessing technique described in Section 4.1.2. The original data samples are replaced with the jerk data. When sampling for a given class is complete, the covariance and inverse covariance matrices are computed. The jerk data is then discarded. The classifier is computed when all the sampling tasks are complete. The discriminant functions are represented by a series of coefficients stored as *8-byte* doubles. Finally, the device transitions to the *online phase*, in which unknown data samples are classified. In our experiments, the classifier is triggered by the vibration detection module provided by the Tmote Invent. Once the node is triggered, it preprocesses acceleration samples through the moving window to generate jerk data. The jerk data is passed to the three generated discriminant functions, and the intersection is taken to yield the final classification result. This data may be sent back to a basestation or stored on the node.

Figure 4.6 shows a Bayesian classifier and corresponding preprocessed samples to detect walking, running, and jumping events. The window size was set to 50. We tested two different ways to compute the average through the moving window. The first uses the arithmetic mean of absolute jerk values; the second uses the root mean square (RMS). The latter is frequently used to process oscillating discrete signals since the RMS generates positive mean values for signals that oscillate between negative and positive. Although the data representations are different, the resulting classifiers are similar. The classifier in Figure 4.6a is based on preprocessed data using the arithmetic mean, and the classifier in Figure 4.6b is based on preprocessed data using the RMS. In each plot, the filled circles at the bottom-left represent jerk samples corresponding to walking events. The cross points at the top-right correspond to running events. The hollow circles inbetween these two sets correspond to jumping events. The data agrees with common sense: Acceleration is low during walking; jumping demonstrates more acceleration in the vertical axis; and running has much



(a) Bayesian classifier using arithmetic mean

(b) Bayesian classifier using RMS

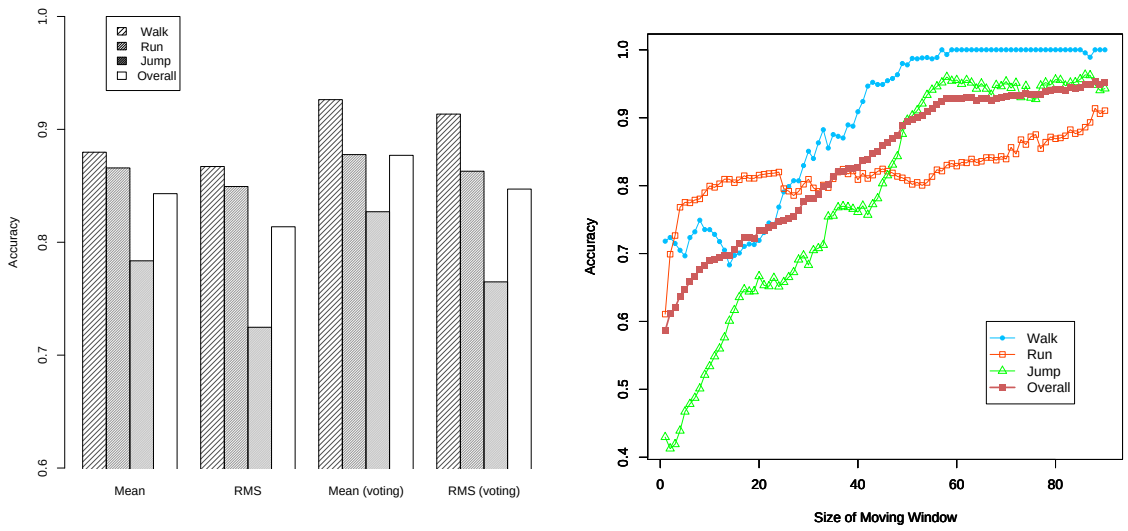
Figure 4.6: Bayesian Classifier to Classify Human Movement

higher acceleration in both axes. The decision boundary for walking and running, and the decision boundary for walking and jumping isolate the walking data well (see the two dashed lines in the bottom-left of Figures 4.6a and 4.6b). The decision boundary between running and jumping cuts vertically between the two corresponding sets of data; the majority of samples are on the correct side.

We evaluated the impact of using the RMS versus the arithmetic mean during preprocessing on the accuracy of the generated classifier. Figure 4.7a summarizes the results. In the experiment, the size of the preprocessing window was set to 50. The left two clusters of the histogram summarize the performance of the Bayesian classifier using the arithmetic mean versus the RMS. We can see that using the RMS during preprocessing does not improve the performance of the classifier.

To further improve the performance, we introduce a voting step post-classification. A vote is performed based on a series of consecutive classification results; the decision is based on the class with the highest vote. In the implementation, we use 20 consecutive classifications to perform the vote (0.2 seconds). The right two clusters in Figure 4.7a summarize the impact of the voting method on classifier accuracy. We can see that voting helps to improve classifier accuracy by eliminating some classification anomalies.

We also investigated the impact of window size on classifier accuracy (with post-classification



(a) Mean, RMS, and voting

(b) Window size

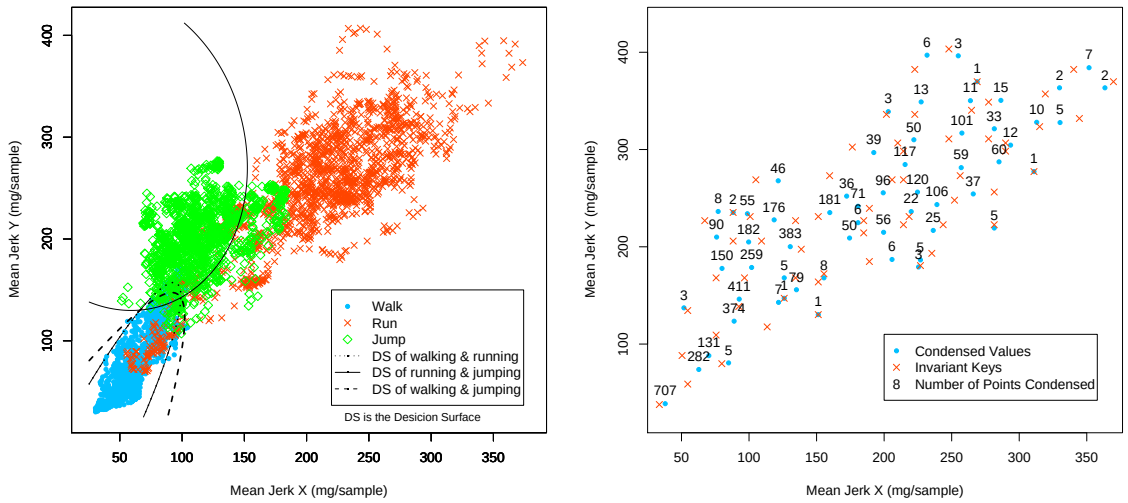
Figure 4.7: Accuracy Analysis of Bayesian Classifiers

voting); the results are summarized in Figure 4.7b. Classification accuracy is plotted for window sizes varying from 2 to 99. There are two important observations here. First, the accuracy of the classifier increases with window size up to a given point. The accuracy reaches approximately 90% in this case. However, larger windows require longer sampling times and larger buffer sizes. Second, by integrating the voting enhancement, classifier accuracy was further increased. More precisely, overall accuracy was increased by approximately 5% when the window size was smaller than 40; the improvements become less significant with larger windows.

4.3.1.2 Implementation of the Condensed kd-Tree Classifier

We next implemented the condensed kd-tree classifier on a Tmote Invent to detect human movement. The sampling rate and training time mirror the configuration of the Bayesian classifier. In the training phase, acceleration samples are again transformed to jerk samples using a sliding window; the window size is set to 50. In contrast to the Bayesian classifier, we do not store the generated jerk samples in a buffer; instead, incoming jerk samples are directly inserted in the tree. In our experiments, there are 1500 jerk samples for each class. Thus, 4500 samples are inserted into the tree. Figure 4.8a shows the preprocessed jerk samples for walking, running, and jumping events. Figure 4.8b shows the corresponding invariant keys and aggregate values stored in the condensed

kd-tree. The cross points denote the invariant keys, and the circles denote the condensed values, which have been skewed away from the invariant keys. The number associated with each node denotes the total number of samples it represents. If we constructed this tree without condensing, it would contain 4500 nodes with 89 levels. But as shown in Figure 4.8b, with a condensing radius of 7, the number of nodes decreased to 61, and the depth of the tree is reduced to 12. The tree uses only 1,464 bytes of RAM – suitable for typical sensor nodes. In this case, the condensing technique saves more than 97% in memory space. During the online phase, each jerk sample is used as a query point for the nearest neighbor search. The number of neighbors searched is dynamic based on the condensing radius and neighborhood threshold. We tested condensing radius values from 1 to 14. The neighborhood threshold was set to slightly larger than twice the condensing radius. The nearest neighbor search uses the magnitude function from Section 4.2.3 to evaluate class membership.

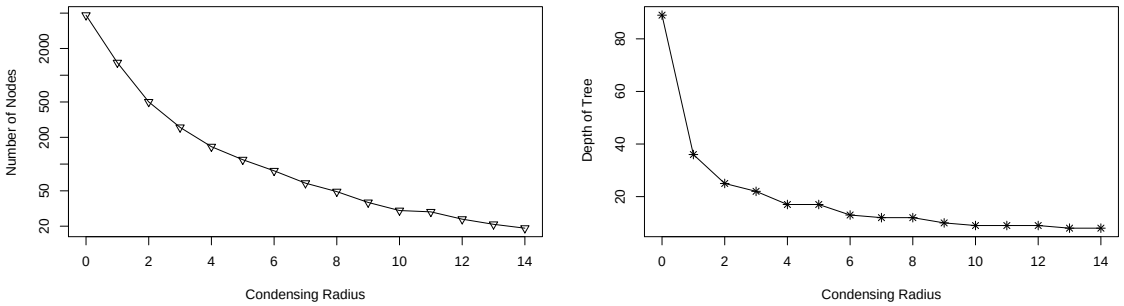


(a) Preprocessed samples

(b) Condensed tree

Figure 4.8: Condensed kd-Tree Classifier (condensing radius = 7)

Figure 4.9 summarizes the impact of the condensing radius on the size and depth of the generated tree. As shown in Figure 4.9a, the number of nodes required to represent the classifier is in excess of 1200 with a condensing radius of 1. It drops to less than 100 when the condensing radius is set to 6, and less than 30 when the condensing radius is larger than 10. (Note that the scale on the y axis is logarithmic). Figure 4.9b shows that the depth of the tree also decreases significantly with increasing condensing radius. As a point of reference, tree depth is 25 with a condensing radius of 2,



(a) Node count

(b) Tree depth

Figure 4.9: Impact of Condensing Radius on Tree Size

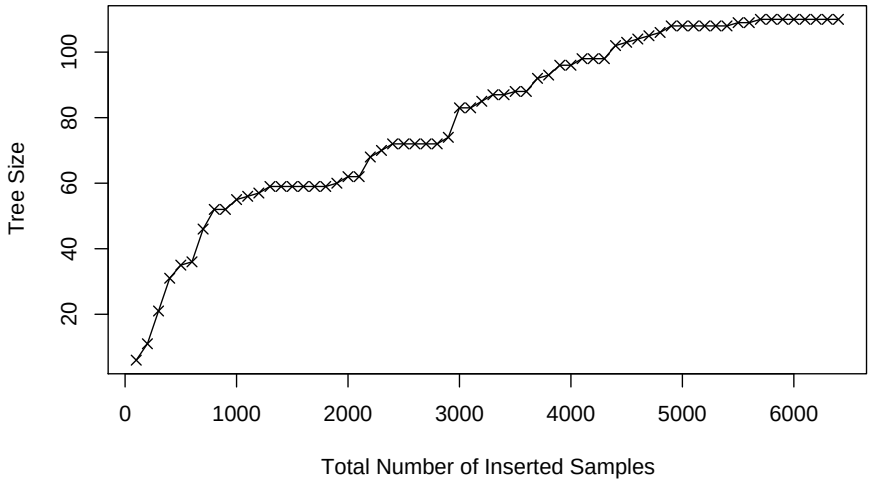


Figure 4.10: Tree Size vs Number of Inserted Samples

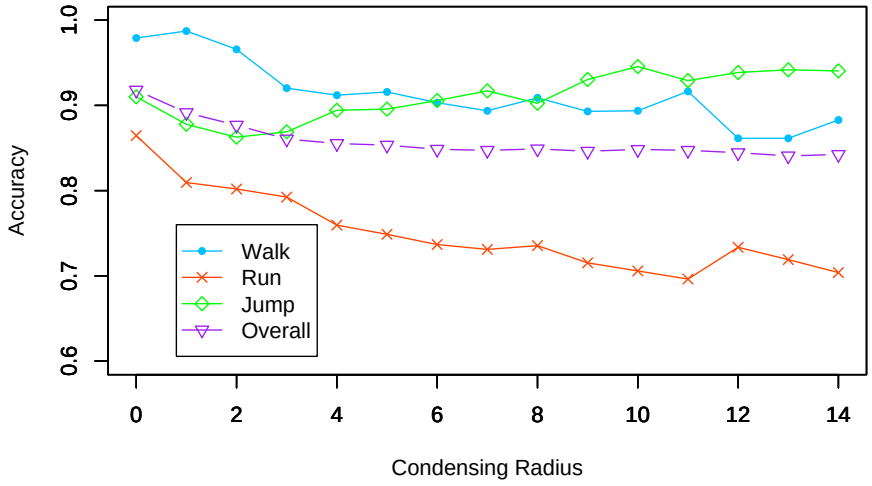


Figure 4.11: Accuracy of kd-Tree Classifier vs Condensing Radius

and it decreases to 12 with a condensing radius of 7. As a result, the speed of the k-nearest-neighbor search is dramatically increased (since the search traverses each node at most once).

We are also interested in the relationship between the size of the training dataset and the size of the resulting tree. Figure 4.10 summarizes the relationship. A total of 6,400 training samples were inserted into the tree; the tree size was recorded after every 100 samples. The condensing radius was set to 7. The graph shows that the rate of increase in tree size decreases significantly with sample count. Indeed, the tree structure becomes relatively stable beyond 5,000 samples. This feature allows the tree to be trained with large datasets. In addition, it provides the potential to accept new training data in the future to improve accuracy.

We next evaluate the relationship between condensing radius and accuracy. As shown in Figure 4.11, overall classification accuracy decreases only slightly with increasing condensing radius. The average rate of decrease in accuracy is smaller than 1% for a unit increase in condensing radius. The impact does, however, vary among classes. For example, classification accuracy for running events decreases by 21% when the condensing radius is increased from 1 to 14, but accuracy for jumping events actually increases by 9%, from 2 to 14. The explanation for this is that the tree loses granularity as the condensing radius is increased. As shown in Figure 4.8a, the data points associated with running events are inbetween the data points associated with walking events and

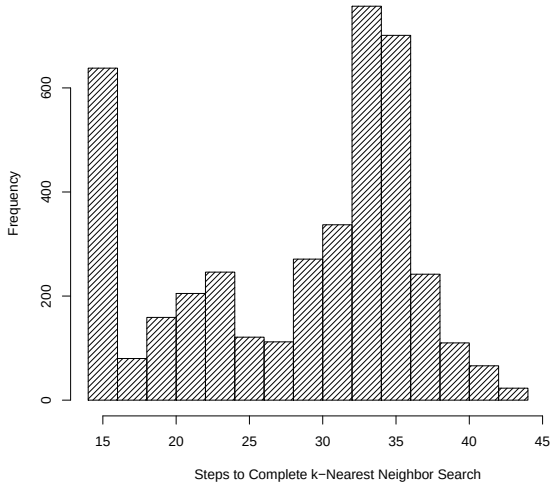


Figure 4.12: Speed of k-Nearest-Neighbor Classification in a Condensed kd-Tree

jumping events. At the boundary of two classes, a small set of data points are often "tangled" around the boundary. These points may be merged to a single condensed point. The condensed value is based on the average of all values in the region, and is therefore biased against the minority class in that region. The larger the condensing radius, the larger the number of nodes that may be merged with a condensed node, and by consequence, the larger the potential skew. The accuracy along the boundary may decrease as a result.

Considering the tradeoff between efficiency and accuracy, choosing an appropriate condensing radius is a key consideration in this approach. In this case study, values between 5 and 8 appear to be good choices.

We now consider classification speed using the condensed kd-tree. We tested 4,000 random samples in a condensed kd-tree constructed using the training data collected for this case study. The condensing radius was set to 7, and the neighborhood threshold was set to 15. Figure 4.12 summarizes the results. The X-axis denotes the number of traversal steps taken during the nearest-neighbor search; the Y-axis shows the frequency over the trial. Note that a search includes two phases – finding the target node and finding the nearest neighbors. The slowest classification takes no more than 45 steps; the fastest takes only 15. Most take 33-36 steps, or 15-16 steps. The average number of steps is 28.6. This operation is fast enough to be executed on typical sensor nodes.

We next investigate the resource consumption of the two classifiers. Table 4.2 summarizes

Classifier	*RAM Usage (bytes)*	*ROM Usage (bytes)*	*Code Size (lines)*
Bayesian Classifier	6826	2814	237
kd-Tree Classifier	219 + tree size	4720	446

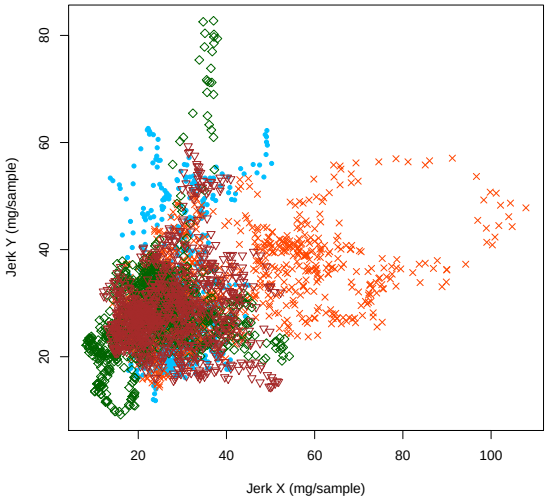
Table 4.2: Classifier Resource Usage

Sensor Nodes	*Tmote*	*Mica2*	*Iris*	*MoteStack*
ROM (KB)	48	128	128	64
RAM (KB)	10	4	8	4

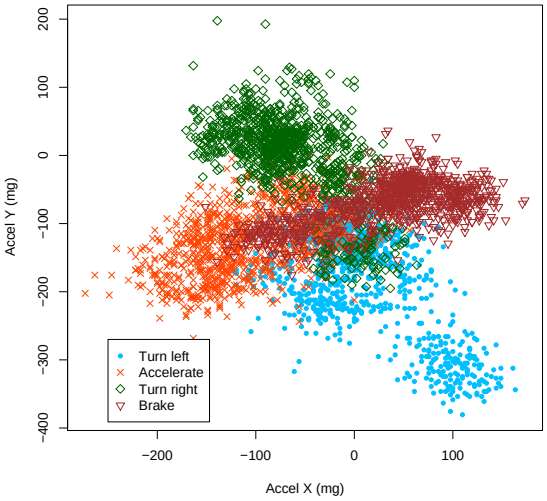
Table 4.3: Typical Sensor Node Characteristics

their utilization characteristics. The Bayesian classifier uses approximately 6.8KB of RAM, mostly to store the training data. There is no need to buffer the sample data in the condensed kd-tree approach, and the tree contents are dynamically allocated. Since we use less than 2.4KB to store the tree (usually less than 100 nodes), the kd-tree classifier requires significantly less RAM. Table 4.3 shows the computational resources available on typical resource-lean sensor platforms [2,7,8,11]. As we can see, the Bayesian classifier may not be suitable for some sensors given their limited RAM capacities. The kd-tree classifier is suitable for all of the sensors included in the table; most have sufficient resources to handle other tasks.

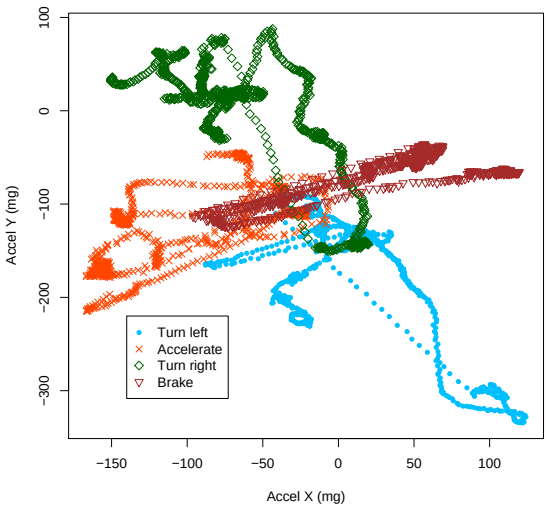
Generally, the kd-tree classifier appears to be better than the Bayesian classifier along several key dimensions, including accuracy, storage efficiency, and future training potential. Clearly the classification speed of the Bayesian classifier is good since only polynomial functions are involved. However, as discussed above, the classification speed of the condensed kd-tree is fast enough for typical sensing scenarios. The accuracy of the kd-tree classifier is slightly better than the Bayesian classifier. Further, the kd-tree classifier uses less memory by avoiding the use of a large buffer for storing temporary samples. Table 4.2 shows the resource utilization of the two classifier applications. Since we use less than 2.4KB to store the tree (usually less than 100 nodes), less memory is used for the kd-tree classifier. Most interesting, the tree can be further trained without reconstruction. Finally, the number of nodes increases slowly with an increase in samples. In the case study, the condensed kd-tree is the preferable choice.



(a) Preprocessed jerk



(b) Acceleration samples



(c) Preprocessed acceleration

Figure 4.13: The Impact of Preprocessing (Driving Events)

4.3.2 Case Study 2: Driving Events

In the second case study, we explore the detection of driving events, again using the accelerometer on the Tmote Invent. A similar effort is discussed in [57], which focuses on detecting road conditions, but without the use of a formal classifier. Our goal is to detect four basic actions: *accelerating*, *braking*, *turning left*, and *turning right*. The hardware setup is similar to the last case study. The Tmote Invent is installed facing up inside the car; again, the orientation of the node is irrelevant. The sampling period in the training phase was set to 2 seconds (200 samples); 5 training periods were performed for each event.

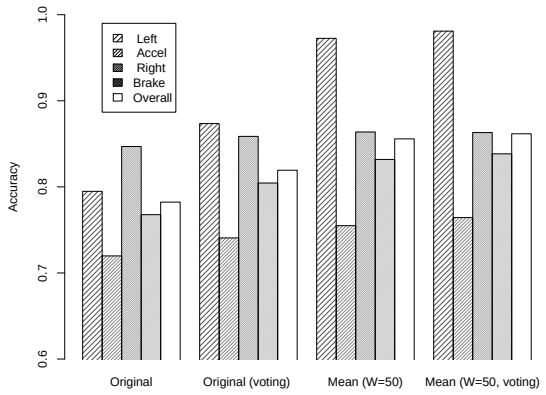
We applied the same preprocessing techniques and classification approaches as in the first case study. However, we discovered that our preprocessing techniques were not suitable for the driving scenario since the absolute value of jerk is similar for most of the target classes. Figure 4.13a shows the preprocessed jerk samples and the corresponding Bayesian discriminant functions. Many of the data points are tangled together in the left part of the graph, especially for data points associated with turning and braking events. As a result, neither of the classifiers can correctly classify the preprocessed data. We analyzed the problem and found that the most significant difference among the samples from different classes is the *direction* of acceleration, which is not considered in our first approach. Hence, instead of using absolute jerk, a scalar, we used the original data, which contains direction information, to construct the classifier. Figure 4.13b shows the original acceleration samples with the generated Bayesian discriminants. Since the acceleration directions of the four classes are different on the X-Y plane, the sample points are scattered in four parts of the graph. As the graph shows, six decision boundaries ($\binom{4}{2}$) are formed by the generated discriminant functions. These decision boundaries correctly partition the sample points for each pair of events. As a result, the generated classifier is capable of classifying the events with high accuracy using the original data. The left two clusters of Figure 4.14a summarize the accuracy of the Bayesian classifier across the event pairs using the original acceleration samples. The overall accuracy is approximately 78%. Using the voting method introduced in Section 4.3.1.1, the overall accuracy improves to 82%.

To further improve the accuracy of the classifier, we introduce an alternative preprocessing technique. Recall that computing the arithmetic mean across a moving window decreases the standard deviation of the data, "smoothing" the irregular vibration within the original sample set. Without converting the acceleration samples to jerk, we directly process the data through a moving

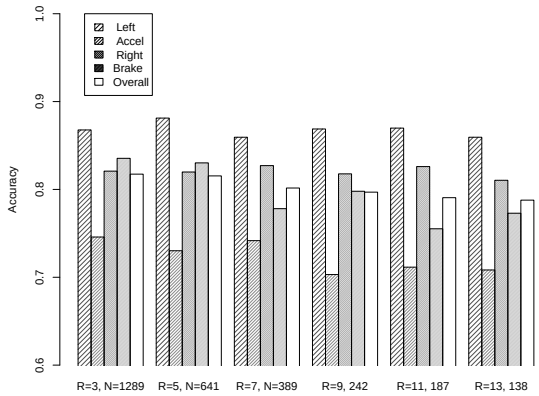
window. As a result, the overlapping areas between different classes are significantly reduced in size. Figure 4.13c shows the preprocessed acceleration samples, along with the generated Bayesian discriminants. After preprocessing, the acceleration samples appear as smooth curves. The right two clusters of Figure 4.14a summarize the accuracy of the classifier generated from the preprocessed acceleration samples. The accuracy of the Bayesian classifier rises to 86% with a window size of 50, improved by 10% compared to the unprocessed case. We next applied the voting method to the classification results, but the improvement was insignificant.

Figure 4.14b summarizes the accuracy of the kd-tree classifier generated from the original sample set, without preprocessing. The accuracy of the classifier is good; however, by preprocessing the acceleration samples, the performance can be improved. Figure 4.14c summarizes the accuracy of the kd-tree classifier generated using the preprocessed acceleration data. Classification accuracy is improved by 5% on average, yielding overall accuracy above 85% with a condensing radius less than or equal to 11. By reducing the standard deviation, the samples are less scattered in the pattern space. Consequently, the tree size is reduced. As shown in Figures 4.14b and 4.14c, for each condensing radius, the tree size is significantly larger without preprocessing. As a point of reference, the tree size is reduced from 187 in Figure 4.14b to 74 in Figure 4.14c, with a condensing radius of 11, and the overall accuracy is still above 85%.

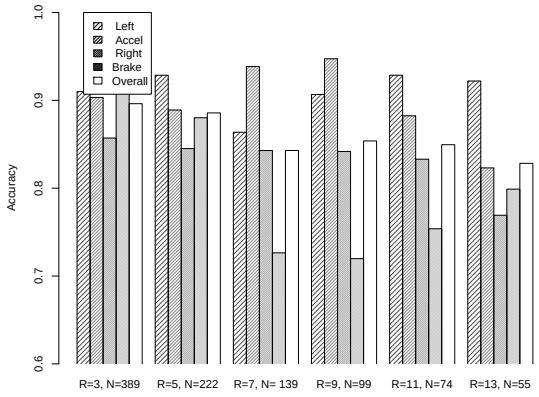
Based on the case studies considered, using jerk samples to construct a classifier is more effective for scenarios where the difference among events is based largely on undirected vibration. In these scenarios, the arithmetic means of the classes formed based on the original samples are similar, and the standard deviations are typically large. Using preprocessing to transform these samples to jerk, the arithmetic means can be separated, and the standard deviations decreased. In contrast, it is preferable to construct a classifier using acceleration data if the main difference among events is the direction of acceleration. The sliding window preprocessing technique can be applied in both scenarios to improve the performance, but without computing jerk samples in the second case.



(a) Bayesian classifier



(b) kd-Tree (no preprocessing)



(c) kd-Tree (acceleration preprocessing)

Figure 4.14: Accuracy of Bayesian and kd-Tree Classifiers

Chapter 5

Network Simulation

In this section, we present the design, implementation, and evaluation of *SnapSim*, a parallel simulator that uses probing and backtracking to reduce synchronization overhead and maximize simulation performance.

5.1 Algorithm Design

While interval synchronization improves performance significantly, it is still costly, especially for distributed simulators, where delays in the physical network may be large. For example, T_{radio} is 3072 clock cycles for a Mica2 node; its microcontroller, an ATMega128, runs at 7,372,800 Hz. Thus, 2400 synchronizations are performed per simulated second. If the physical networking delay is 200 microseconds per message, there will be at least 480 milliseconds of synchronization delay introduced per simulated second! Although radio initialization time and MAC backoff time can be exploited to reduce synchronization frequency [35], it is still not the most efficient strategy. First, the synchronization interval remains small – 11 times T_{radio} for the radio off interval, and 1 to 32 times T_{radio} for the MAC backoff interval [65]. Second, these strategies are performed only if the radio is turned off or message congestion is experienced in the MAC layer.

We reconsider the synchronization problem, defining an *optimistic simulation* as a parallel simulation of the network without any synchronization. First, suppose we are given a priori knowledge of all transmissions within a simulated run. Suppose further that it is possible to dynamically set the simulation speed of each processor. For any simulated time t, we define the *global nearest*

transmission time as the earliest simulated transmission across all nodes in the network at or after time t. We define a *valid* simulation as a run in which no simulated node reaches a global nearest transmission time before the node(s) that transmit at those times. In effect, such a run imposes a partial ordering on the communication dependency graph, ensuring that all transmission events are simulated prior to their corresponding reception events.

But of course, a perfectly optimistic simulator is not possible since instruction sequences cannot be predicted statically. However, if the *next* transmission time can always be determined, the number of synchronizations can be reduced to the number of transmissions (as we discuss next).

5.1.1 Probing and Backtracking

The central algorithmic idea is to use probing executions to find the nearest transmission time for each node in the network, and to backtrack nodes that have passed the next global nearest transmission time. The probing execution is isolated; communication primitives are ignored. A probing execution may proceed beyond the global nearest transmission time, thereby missing a read channel event. Thus, some nodes may need to backtrack and re-execute to correct for the missed transmission. To restore a node to a correct former state, a snapshot must be taken that includes the aggregate simulation state of the node.

We assume the following: The network consists of a fixed set of N processes. Each simulated node is emulated by a single physical process; the processes are distributed. Each process is capable of suspending and resuming execution of the simulated node. The simulation is cycle-accurate and the simulated microcontrollers have the same frequency. Every physical process maintains a local simulation clock based on the number of cycles executed. Nodes do not have global knowledge, except for the size of the network. Messages may be broadcast through the (physical) process network. Each process is capable of taking a simulation snapshot and restoring the simulated node to a previous snapshot point.

Before presenting the design details, we briefly summarize the basic principles. First, we define a *wave* of execution as the execution sequence spanning between consecutive global nearest transmission times. (We consider time 0 as the first global nearest transmission time.) Snapshots are taken at each global nearest transmission time. A basic task of the algorithm is to probe the end of the current wave and backtrack any processes which proceed beyond the end of the wave before it was found. Those processes will be replayed, enabling the reception of otherwise

lost events. Initially, each process creates a snapshot before the simulation begins. Next, each process starts a probing execution, ignoring the possibility of message reception. Each time a local transmission is encountered, the process notifies the other processes. The minimum received time becomes a candidate for the global nearest transmission time. If the node's current time is greater than the candidate nearest transmission time, it broadcasts its current time and waits for the final global transmission time, since it is unnecessary to continue probing execution when another node has an earlier transmission event. Each process counts the number of processes it has heard from in the current wave. If the count is less than the network size, the process waits at the candidate transmission point; otherwise, the minimum candidate time becomes the global nearest transmission time. We call the last broadcast message in a wave the *notify* message; all the processes in the network resume execution when it is received. If a process transmits or waits at the global nearest transmission time, it saves a snapshot and resumes execution without backtracking. Otherwise, the process backtracks to its last snapshot at the beginning of the wave before resuming execution. When it executes to the new global nearest transmission time, it saves its snapshot and continues probing of the next wave.

To formally define the probing and backtracking algorithm, we present the *action system* program in Algorithm 1.

Each process j in the algorithm maintains $clock.j$, a local counter that counts the number of cycles executed. Variable $timeNr.j$ stores the most recent global nearest transmission time, and $timeCnd.j$ stores the next candidate global nearest transmission time, which is the minimum of all the transmission times identified within the current wave. Within each wave, each node records the number of nodes that wait at or beyond a candidate transmission time, denoted by $count.j$. Variables $transmitting.j$ and $wait.j$ are Booleans that indicate whether the node transmits within the current wave and whether the node is in a waiting state, respectively.

Five functions are introduced to simplify the presentation. *executeInst.j* executes an instruction on node j without handling reception events. The execution does not write to the simulated wireless channel since probing execution does not guarantee the validity of the simulation. *broadcastClock.j* broadcasts the current time of simulated node j throughout the physical network. Once a node broadcasts, it transits to the *wait* state. *saveSnapshot.j* saves a snapshot of the simulated node j, and *restoreSnapshot.j* restores the simulated node j to the last stored snapshot state. *writeChannel.j* writes a byte of data to the wireless channel.

Algorithm 1 Probing and Backtracking Algorithm

VAR
$clock.j : \mathbb{N}$ {simulated time of j}
$timeNr.j : \mathbb{N}$ {global nearest transmission time}
$timeCnd.j : \mathbb{N}$ {candidate global nearest transmission time}
$count.j : \mathbb{N}$ {probing completion count}
$transmitting.j : \mathbb{B}$ {does j transmit in the current wave?}
$wait.j : \mathbb{B}$ {is j in the wait state?}
FUNCTION
$executeInst.j$: {probing execution of instruction at j}
$broadcastClock.j$: {broadcast the current simulated time}
$saveSnapshot.j$: {save snapshot}
$restoreSnapshot.j$: {restore snapshot}
$writeChannel.j$: {write to (simulated) wireless channel}
INITIALLY
$clock.j, timeNr.j, count.j = 0$;
$timeCnd.j = \infty$;
$transmitting.j, wait.j = false$;
ASSIGN
if $!wait.j$ **then**
  if $clock.j = timeNr.j$ **then**
    $saveSnapshot.j$;
  end if
  if $clock.j < timeCnd.j$ **then**
    $executeInst.j$;
  else
    $count.j := count.j + 1$;
    $broadcastClock.j$;
    $wait.j := true$;
  end if
end if
□
if *simulated transmission event* $\bigwedge !wait.j$ **then**
  $count.j := count.j + 1$;
  $timeCnd.j := min(clock.j, timeCnd.j)$;
  $transmitting.j := true$;
  $broadcastClock.j$;
  $wait.j := true$;
end if
□
if *receive broadcast message from k* **then**
  $count.j := count.j + 1$;
  $timeCnd.j := min(time.msg_k, timeCnd.j)$;
end if
□
if $count.j = N$ **then**
  $timeNr.j := timeCnd.j$;
  $count := 0; timeCnd.j := \infty$;
  $wait.j := false$;
  if $clock.j > timeNr.j$ **then**
    $restoreSnapshot.j$;
  else if $transmitting.j$ **then**
    $writeChannel.j$;
  end if
  $transmitting.j = false$;
end if

Initially, *clock.j*, *timeNr.j*, and *count.j* are set to 0; *timeCnd.j* is set to infinity; and *transmitting.j* and *wait.j* are set to false. There are four rules for this program.

The first rule checks whether the process is in the *wait* state. If not, it compares the current clock value with *timeNr.j*. If they are equal, the process has reached the global nearest transmission time and saves its snapshot. (Since *clock.j* and *timeNr.j* are set to 0 initially, each process will save a snapshot at the beginning of the simulation.) Next, it compares the current clock value with the candidate next global nearest transmission time, *timeCnd.j*. If the current time is earlier than the candidate time, the process executes the next instruction at node j (which increments *clock.j*). Otherwise, the process is already beyond a candidate transmission, and it increments *count.j* by 1, broadcasts a message containing the current time of the simulated node, and sets *wait.j* to true.

The second rule is performed when a new local transmission is identified at node j while it is not in the *wait* state. In this case, the process increments *count.j* and updates *timeCnd.j* if the current clock value is earlier. Next, the process sets *transmitting.j* to true, which indicates that the node has detected a transmission point. Next, node j's clock value is broadcast to all other processes, and *wait.j* is set to true. A node can detect only one transmission and broadcast only once within each wave (by rule 1 and 2).

The third rule is performed when a process receives a broadcast message from another process k. In this case, the process increments *count.j* and updates *timeCnd.j* if the received time is earlier.

The fourth rule is performed when *count.j* is equal to the network size N, indicating that the new global nearest transmission time is globally known. As a result, *timeCnd.j* becomes the new global nearest transmission time, *timeNr.j*; *count.j*, *timeCnd.j*, and *wait.j* are reset. (*transmitting.j* will also be reset, at the end of the rule.) If the current time of the simulated node is later than *timeNr.j*, it backtracks to the last saved snapshot. Otherwise, if the node has a transmission point at *timeNr.j*, it writes the data to the wireless channel. If the current time equals *timeNr.j*, a new snapshot will be saved when the first rule is selected later; no backtracking will be performed.

5.1.2 Algorithm Proof

We assume the execution of each process is deterministic; otherwise, a replay could result in an altered execution sequence. Each time a process is rolled back, the instruction sequence is

identical to the original execution sequence (before the global nearest transmission time was passed). The instruction-level interpreter ensures determinism after rollback. To achieve full determinism at the hardware level, in the implementation of the simulator, the random seed is tied to the program counter, enabling replay of (identical) random sequences.

Consider a single wave of execution. Define GNT, the global nearest transmission time, known to all processes, as before. Define the last global nearest transmission time as $LGNT$. When a new GNT is agreed upon by all processes, GNT becomes $LGNT$. Note that GNT and $LGNT$ are monotonically increasing. Time 0 is considered the first $LGNT$. The safety property of the algorithm is that for any process j, $clock.j \geq LGNT$. Before we prove this property, we observe that if a process j is rolled back, it will be rolled back to $LGNT$, since (1) a new snapshot will be saved only when a process passes GNT, by Rule 1 of Algorithm 1; and (2) $LGNT$ is updated (based on the former GNT) only when all processes agree upon (the new) GNT, implying that for every process j, $broadcastClock.j$ is invoked when it passes GNT, by Rule 1 and Rule 2 of Algorithm 1. As a result, each process must have passed $LGNT$ because the next GNT must be found after $LGNT$ by our determinism assumption. Otherwise, the new GNT should be found for the last execution. Hence, the last snapshot must have been saved by every process j at $LGNT$.

To prove the safety property, there are two possible cases to consider. First, if a process has been rolled back to $LGNT$ after finding the new GNT (since the clock is monotonically increasing), $clock.j \geq LGNT$. Second, if the process is not rolled back after finding the new GNT, it must have local time $clock.j \geq GNT > LGNT$.

The progress property for our algorithm is that $GNT = k \rightsquigarrow GNT > k$, which ensures that the algorithm always proceeds. Assume that there are infinite transmissions (in an infinite execution). Since the program counter is monotonic, and $clock.j \geq LGNT$ (by our safety property), when a new GNT is found ($GNT > LGNT$), GNT becomes $LGNT$. Hence, $GNT = k \rightsquigarrow GNT > k$. If there is a finite number of transmissions in an execution, by the time of the last transmission, all the preceding GNT values can be proved monotonic as before. If there is no transmission afterwards, no dependencies exist, and no synchronizations are necessary.

5.1.3 Case Analysis

To clarify the operation of the algorithm, we consider several representative scenarios. Figures 5.1 – 5.4 show 4 basic execution scenarios in a network consisting of two nodes. Larger networks

are analogous. The horizontal timeline in each graph denotes simulated time (i.e., cycle count); tx denotes the simulated time of a transmission event. The horizontal curly brackets denote physical execution time (e.g., microseconds), labeled by τ_S; Δ represents an unknown networking delay. The diagonal-filled bars denote probing execution, and the hash-filled bars denote re-execution after backtracking.

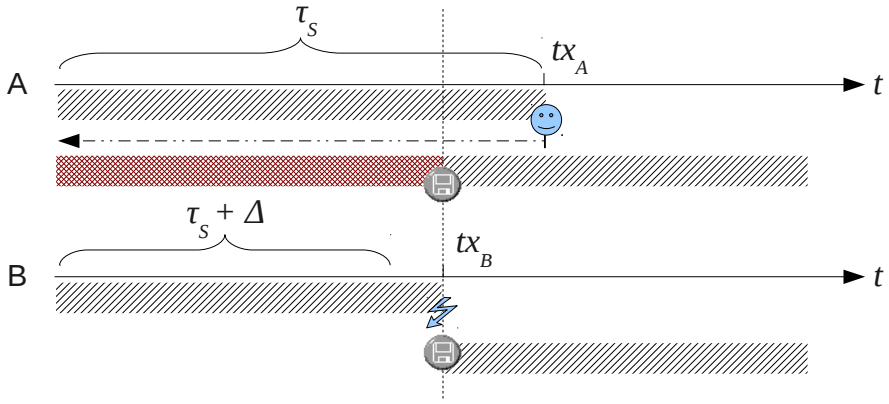


Figure 5.1: Algorithm Analysis: Case 1

Consider Figure 5.1. Assume that after τ_S units of physical time, process A has reached a transmission at simulated time tx_A, where it broadcasts this transmission time and waits; process B receives the broadcast at $\tau_S + \Delta$, where Δ is the networking delay from A. At this point, B has not reached its transmission time tx_B, which is smaller than tx_A. As a result, when B eventually reaches tx_B, tx_B becomes $timeNr$, and B sends the notify message since it has already received the transmission time, tx_A, from A (i.e., $count = N = 2$). Since tx_B is equal to $timeNr$, B resumes execution; node A backtracks to the last snapshot and resumes execution. A will save a snapshot when it passes tx_B.

Next consider Figure 5.2, which uses the same transmission times (i.e., tx_A is greater than tx_B). In this run, B executes to tx_B in τ_S physical time and waits. When receiving the broadcast at $\tau_S + \Delta$, A has not reached tx_B, nor its own transmission event. In this case, A continues to execute until it reaches tx_B. As a result, tx_B becomes the global nearest transmission time. Next, the notify message from A is received at B, and both A and B save snapshots and resume execution. No backtracking is necessary.

Next consider Figure 5.3, again the same setting. tx_A is greater than tx_B, and in τ_S physical time, B reaches tx_B and waits. However, when A receives the broadcast from B, it has already passed

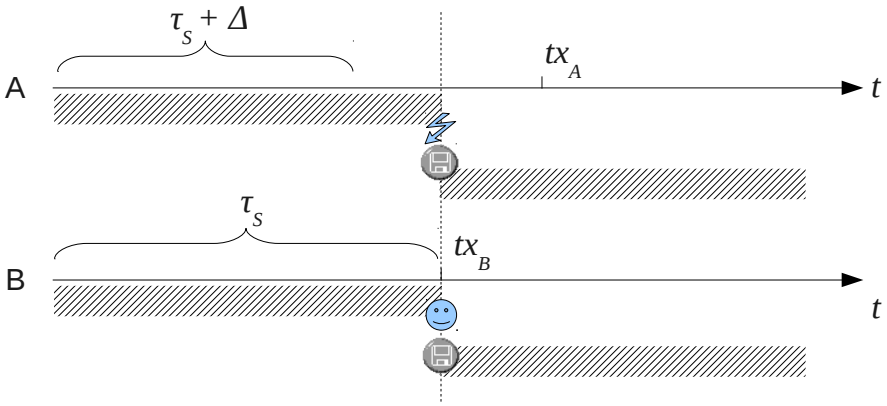


Figure 5.2: Algorithm Analysis: Case 2

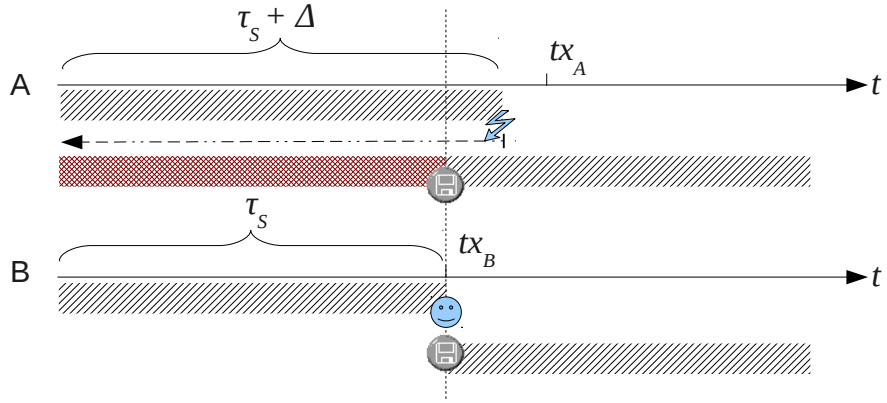


Figure 5.3: Algorithm Analysis: Case 3

tx_B, but has not reached its transmission time tx_A. Since A cannot find an earlier transmission time than tx_B, it stops searching, and tx_B becomes the global nearest transmission time. After A notifies B, B resumes execution; A backtracks to the last snapshot, resumes execution, and saves a new snapshot when it passes tx_B.

Figure 5.4 shows another possible case. Suppose that nodes A and B transmit at the same time ($tx_A = tx_B$), and as a result, $timeNr$ is the transmission time for both nodes. No matter which node reaches $timeNr$ first, no backtracking is required since the first sender will notify the second, which will wait at tx_A/tx_B. Another possible case is that there are no transmissions in the network ($tx_A = tx_B = +\infty$). In this case, no synchronization and backtracking will be performed; the simulator will execute at its maximum speed.

The probing and backtracking algorithm is based on the idea of optimistic execution. To

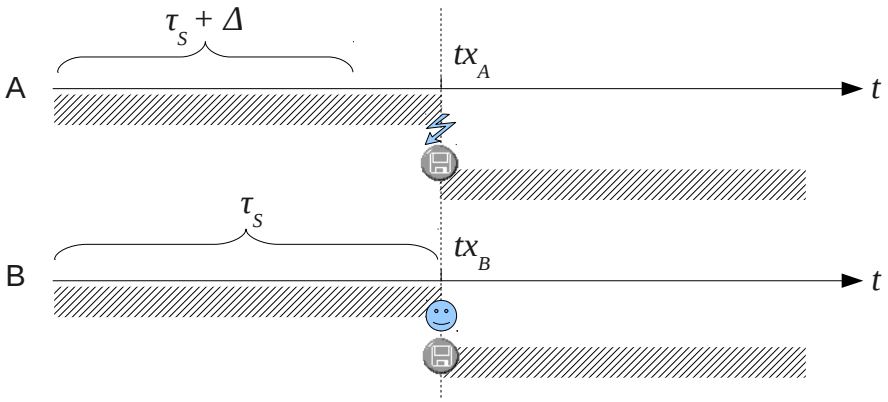


Figure 5.4: Algorithm Analysis: Case 4

reduce synchronization overhead as much as possible, it probes the global nearest transmission time and backtracks nodes that have passed this time, to the last saved snapshot. New snapshots are saved at the global nearest transmission time. The number of synchronizations is limited to a minimum, assuming no prior knowledge of the global nearest transmission time. From the start of a wave, at most two rounds of execution will be performed. The algorithm compensates slower processes, since if they reach the global nearest transmission time later than the transmitting process, no backtracking will be performed at these processes. As a result, the speed of snapshotting/backtracking and the technique used to restore nodes to a valid state become the main implementation challenges.

5.2 System Implementation

SnapSim is implemented using Avrora [65] as the foundation. In our implementation, the interval synchronizer is removed, and a distributed wireless channel is implemented at each process. The channel simulates the wireless communication medium and is updated by each transmission. A vector clock is also maintained at each process and records the reported transmission times received from other nodes. Each node maintains a snapshot that contains a copy of the node's simulation state at the last global nearest transmission time. In the following sections, we discuss key implementation details.

5.2.1 Snapshots

A snapshot includes the full simulation state of a node; it is, conceptually, a clone of the process simulation thread. A wave starts from the last snapshot time and ends at the new snapshot time. By the correctness of the algorithm, a snapshot always represents a valid simulation state. Since a node never backtracks to earlier waves, the existing snapshot is replaced when a new snapshot is saved. In detail, a snapshot in our design comprises the following state: *(i)* all state related to the microcontroller, including memory, registers, flags, etc.; *(ii)* the state of the interpreter interpreting the instructions; *(iii)* the states of all SPI and ADC devices connected to the microcontroller, including the radio; and *(iv)* the state of all auxiliary components for simulation– for instance, the event clock, device state monitor, etc. As a result, the snapshots are relatively large, and the associated checkpointing / backtracking overhead would be large if we saved a full copy of the snapshot for each node. To improve efficiency, we take the following steps to minimize the time (and space) required to save and load a snapshot:

(1) Known invariants and constants are not saved in a snapshot. As an example of the former, variables that represent assembly instructions loaded on the microcontroller are invariant if the microcontroller is not reprogrammed. As a result, the snapshot size can be reduced.

(2) For peripheral devices– for instance, an SPI device– if the device is not accessed in the wave since the last snapshot, the previous saved state will be kept; no new save is required.

(3) For large arrays of elements, an incremental snapshotting strategy is used since only a subset of elements typically change from snapshot to snapshot. More precisely, a large array is divided into N smaller partitions, and an *immaculacy* bitmask of N bits tracks whether the partitions have been modified since the last snapshot. If a partition is untouched in a given wave, no save action will be performed on the partition. When a snapshot is restored, or a new snapshot is saved, the immaculacy array is reset. One example application of this strategy in our simulator concerns the array that represents the RAM of the microcontroller, which is large.

By taking the above steps to reduce the size of each snapshot, execution of the save and restore methods consumes less than 1.5% of the total running time in our evaluation.

5.2.2 Distributed Resources

In SnapSim, a vector clock is introduced at each process to maintain the local knowledge of simulated time at other processes. The vector clock is updated when a local nearest transmission time message is received or a local instruction is performed. When a node has received a broadcast message from all nodes in the network, the lowest value of the vector clock becomes the global nearest transmission time. We simulate the communication between distributed (physical) processes by imposing an adjustable networking delay when a process broadcasts. As a result, we are able to evaluate the performance of the simulator in a range of (physical) networking scenarios.

In SnapSim, every thread maintains its own wireless channel, updated whenever a transmission event occurs. Both the transmitted data and the corresponding simulated time are recorded. The latter information is required since when another node reads the channel, it compares its clock time and the data timestamp to decide what will be read. The channel is modeled as a FIFO queue.

5.2.3 Probing and Backtracking

The probing and backtracking algorithm is implemented as described in Section 5.1. The *wait()* and *notify()* functions of Java *Thread* are used to support the *wait* and *notify* functions in the algorithm. We identify a transmission event as the cycle where the last bit of a byte of data is written to the TX register of the radio chip; as a result, a *transmit* event will be fired on the event clock. When a node passes the current global nearest transmission time, a new snapshot needs to be saved by calling the *saveSnapshot()* method on the *Retraceable* interface of the interpreter; it recursively saves object state bottom-up in the object graph. When a restore is needed, the *restoreSnapshot()* method will be called; the state of the simulated node is restored to the snapshot taken at the last global nearest transmission time. In addition, to restore to a correct state, it is necessary to record the position where the *saveSnapshot()* function was called within the interpreter, since several functions in the interpreter may update the clock beyond the time where a snapshot needs to be taken. For example, if *saveSnapshot()* is performed when the interpreter executes a sleep function, when *restoreSnapshot()* is called, the program needs to be restored to the same function and continue to simulate the node.

5.2.4 Finding Exact Snapshot Points

The backtracking algorithm requires that each snapshot function be performed at the exact point when a transmission occurs. However, capturing the state of a simulated node at the exact cycle of a global nearest transmission event is non-trivial. Although the simulator is cycle-accurate, a given cycle may not "appear" during simulated execution since many instructions consume more than 1 cycle. For example, assume a snapshot is required at cycle 1000. However, an instruction executes at cycle 998 and consumes 4 cycles, as shown in Figure 5.5. When should the snapshot be taken (which must be correct and retraceable)?

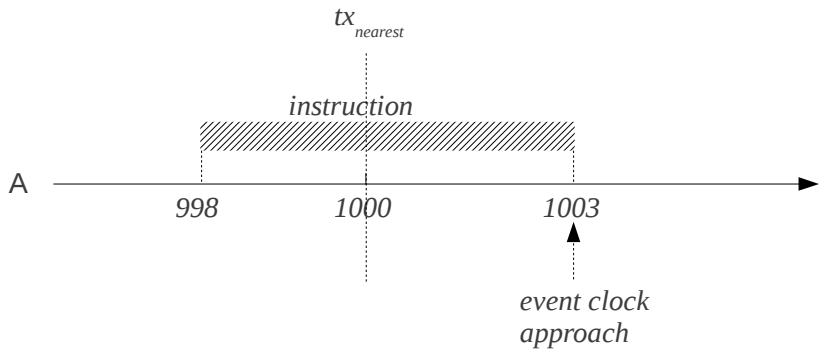


Figure 5.5: Hidden Cycles

To solve this problem, we exploit the event clock mechanism of Avrora. The event clock is used to update the cycle clock after each instruction is executed. Events – for instance, read channel events on the radio– are triggered based on the event clock. When the event clock reaches the scheduled time of an event, the event is fired. As a result, from the view of the event clock, events fire at their scheduled times after the executing instruction is complete. The fired event does not have an impact on the result of the last executed instruction since instructions are atomic, and it preserves the time sequence of events. We add a snapshot event to the event clock once the global nearest transmission time is obtained, and it fires when the node re-executes to the global nearest transmission time. It saves the snapshot on the cycle of the event, after the instruction that crosses the global nearest transmission time. When saving the snapshot, the snapshot event in the event clock is skipped. When the snapshot is restored, the simulator begins from the point where the

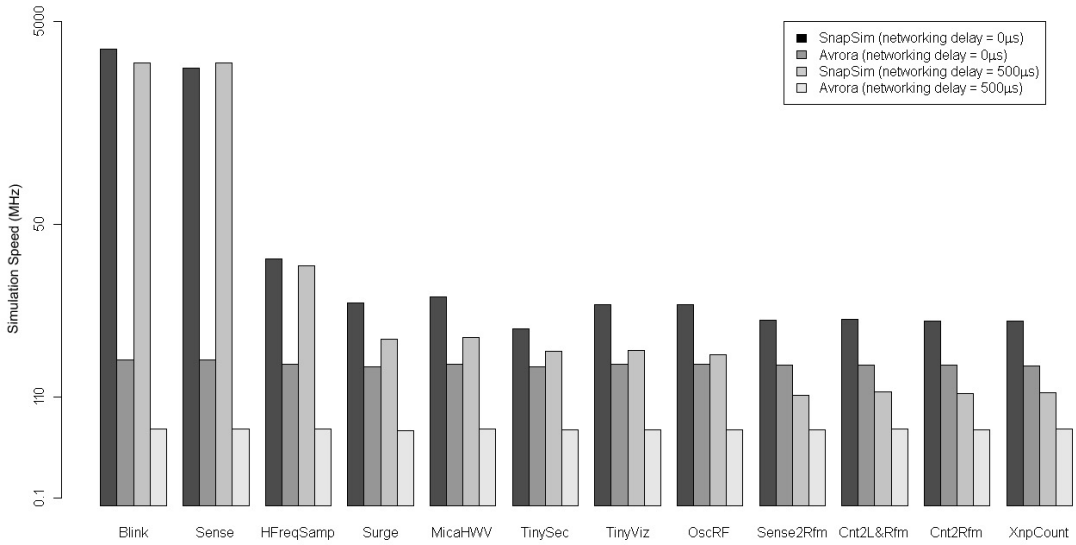


Figure 5.6: Performance of Representative TinyOS Applications

event clock is updated.

5.2.5 Networking Delay

The (physical) networking delay refers to the total time required to encode a broadcast packet at a sender process, transmit the encoded packet, and decode the packet at a receiver process. To make it adjustable for evaluation purposes, the physical networking delay is simulated in SnapSim. The networking delay is usually hundreds of microseconds in a LAN, depending on the network bandwidth and traffic, and usually more than 1 millisecond across the Internet. We use the Java Native Interface (JNI) to issue a Linux system call to *usleep()*. As a result, a simulated networking delay with accuracy on the order of tens of microseconds can be configured. The networking delay is assumed to be symmetric between nodes.

5.3 Evaluation

In this section, we present experimental results for SnapSim, using Avrora as our baseline for comparison[1]. The experiments were performed on a desktop PC with an Intel Core i7 820 processor and 8GB RAM. The JVM heap size was set to 512MB. To limit I/O impacts, the data monitors used to display simulation results were disabled in both simulators.

We assume the simulated sensor network is a complete graph; the results might represent a cluster within a larger network. For our test cases, we use the TinyOS [46] applications in the Avrora test suite. Figure 5.6 summarizes the simulation speed of SnapSim compared to Avrora in a network of 8 nodes, across the test suite. The applications are ordered by bitrate (i.e., transmitted bytes of data per second), from left to right. For each application, The left two bars represent the simulation speeds of SnapSim and Avrora, respectively, when no networking delay is introduced. The right two bars represent the simulation speeds of SnapSim and Avrora, respectively, with a networking delay of 500 microseconds, a typical LAN delay. The Y-axis denotes the simulation speed in MHz; the axis is logarithmically scaled. For both scenarios, it is clear that SnapSim runs significantly faster than Avrora across all applications. For applications with a very low bitrate (or no transmissions), SnapSim is up to 4000 times faster with a networking delay, and 1150 times faster without a networking delay. For most mid-range applications (i.e., HFreqSamp to OscRF), SnapSim is at least 5.5 times faster with a networking delay, and 4 times faster without a delay. For the application with the highest bitrate, XnpCount, SnapSim is more than 2.3 times faster with a networking delay, and 2.6 times faster without a delay.

As seen in the figure, the network bitrate is the key factor that affects the performance of SnapSim since the number of synchronizations depends on the number of bytes transmitted over the network. Avrora runs at a relatively stable rate due to its use of a fixed interval synchronization strategy.

To further evaluate the performance of SnapSim in different scenarios, we consider 3 typical TinyOS applications with varying bitrates: *(i)* ***Sense***, a basic sensing application, which does not use the radio; *(ii)* ***Surge***, a typical tree-based sensor network application, which transmits at a relatively low bitrate; and *(iii)* ***XnpCount***, another tree-based sensor network application, which transmits at a high bitrate. In a network with 8 nodes, the bitrates of the above three applications

[1] Although PolarLite [35] may be a better baseline for comparison, its source code is not opensource, and is hence unavailable.

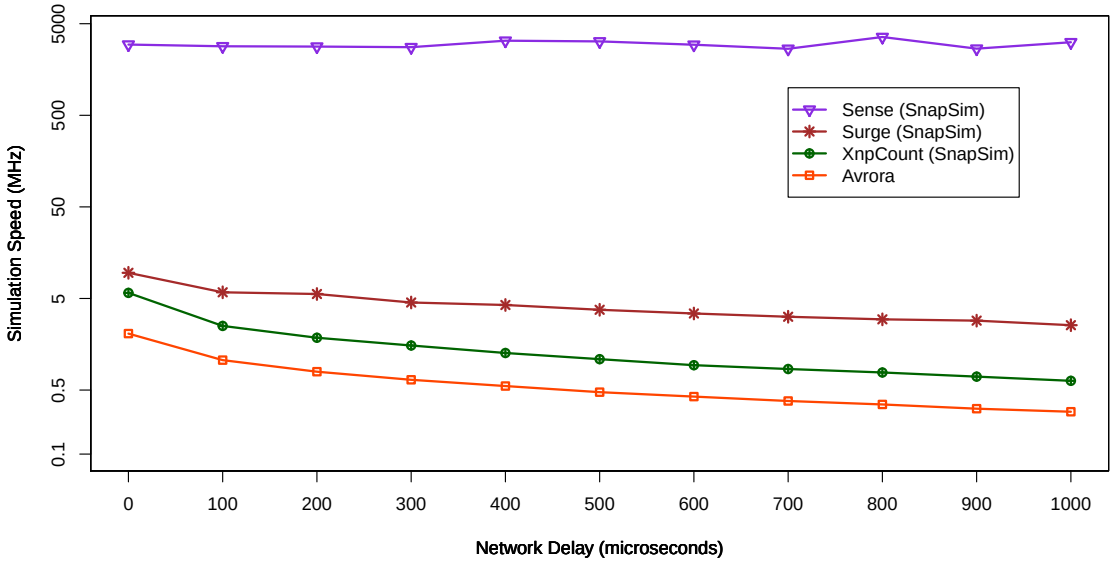


Figure 5.7: Performance vs Delay (network size=8)

are approximately 0 bits/s, 1820 bits/s and 9500 bits/s. Figure 5.7 shows the performance of SnapSim and Avrora as the networking delay is increased, again in a network of 8 nodes. Since Avrora maintains a relatively stable speed across varying bitrates, only the curve of Surge is shown for Avrora. (The other applications run at similar speeds; the differences could not be seen in the graph.) Again, the y-axis represents simulation speed on a logarithmic scale. The networking delay is tuned from 0 to 1000 microseconds. Notice that omitting the networking delay does not mean there will be no synchronization delay; synchronization among threads also introduces a delay, although usually smaller than the networking delay. The graph reveals three key observations: *(i)* SnapSim runs faster than Avrora, with or without a networking delay. *(ii)* SnapSim simulates the Sense application at the fastest simulation speed, stably, since no synchronizations are performed. *(iii)* The Surge and XnpCount applications slow down as the networking delay increases, for both simulators. The rate of decrease is close to linear. Since the networking delay is introduced via synchronization, the number of synchronization points decides the simulation speed. Our solution reduces the number of synchronizations to an optimal level, yielding higher performance.

We next investigate the scalability of our algorithm. Figure 5.8 summarizes the impact of network size on simulation speed with a networking delay of 500 microsceconds. Again, SnapSim

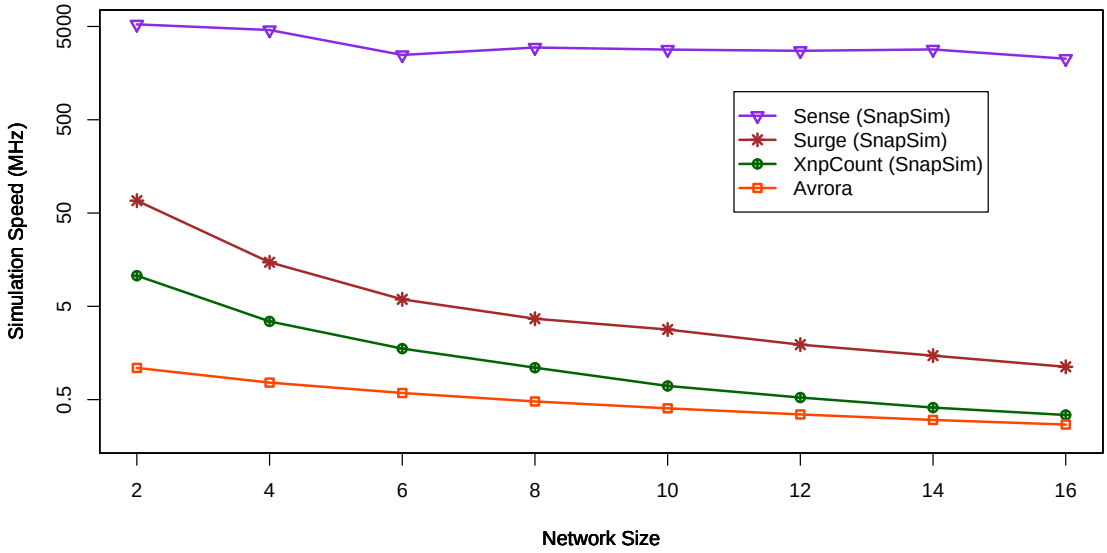


Figure 5.8: Performance vs Size (networking delay=500μs)

is faster than Avrora for all three programs. Sense runs at the highest speed due to the absence of synchronization overhead. Surge and XnpCount achieve fast simulation speeds when the network size is small. As network size increases, the data transmitted in the network increases, increasing the number of synchronizations, which slows down the simulation. For example, the simulation speed of XnpCount in a 16 node network is only 26% faster than Avrora. In total, 1983 bytes of data are transmitted per second network-wide, which is close to 2400 interval synchronizations per second (as in Avrora). However, very few wireless sensor network applications reach such a high bitrate (at steady state). Most applications work in a low duty-cycle mode to save energy. Moreover, as network size increases, the network is usually multi-hop, not completely connected as we have assumed. The network size in our experiments can be treated as a cluster size.

Figure 5.9 summarizes the impact of network size on simulation speed in the absence of networking delay. SnapSim is still faster than Avrora in most cases, even when the simulator is centralized on a single desktop. However, as seen in the bottom-right of the graph, SnapSim simulates XnpCount slower than Avrora when the network size is larger than 12. In those cases, the number of synchronizations is very close to the number of intervals in Avrora, and the more complicated synchronization mechanism of SnapSim leads to slower simulation speed.

We next evaluate the performance of the synchronization logic used in SnapSim and Avrora. We simulate Surge and XnpCount in a network of 8 nodes, in the absence of networking delay, for

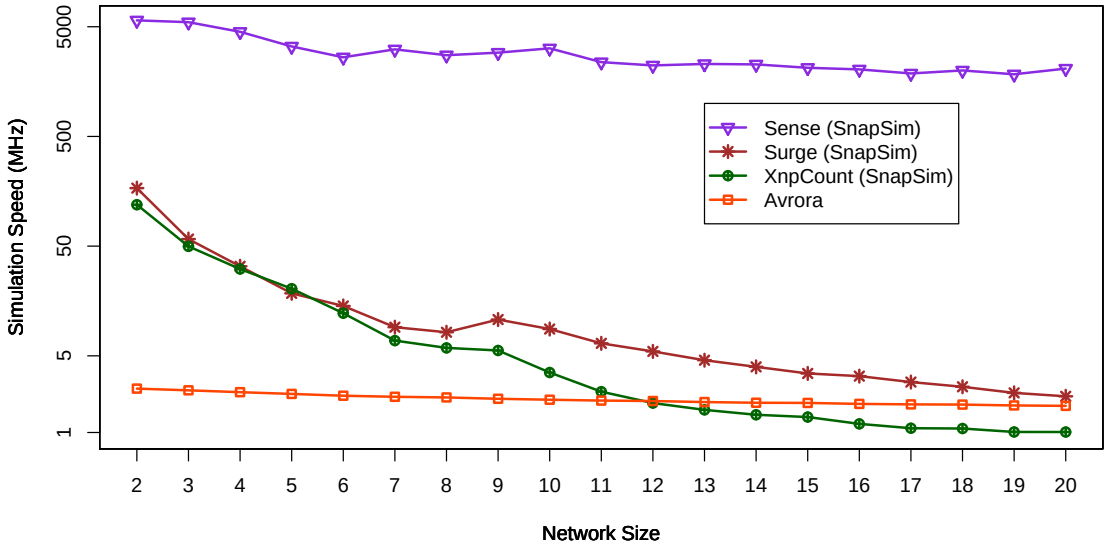


Figure 5.9: Performance vs Size (no networking delay)

Application	*Sync. %*	*Save %*	*Restore %*	*Total Running Time (ms)*	*Est. Memory Usage (KB)*
Surge (SnapSim)	19.22	0.71	0.68	87,947	52,408
Surge (Avrora)	82.36	–	–	372,218	35,659
XnpCount (SnapSim)	43.93	1.44	1.36	131,654	52,852
XnpCount (Avrora)	85.55	–	–	363,487	34,673

Table 5.1: Synchronization, Snapshotting, and Backtracking Overhead

100 simulated seconds each. We accumulate the time spent synchronizing nodes using the *NanoTime* API provided by Java. Table 5.1 summarizes the synchronization overhead. The second column of the table shows the percentage of the total running time spent synchronizing nodes. The third and fourth columns show the percentage of time consumed by the save and restore methods, respectively. The fifth column shows the total running time of each application, and the sixth column shows the estimated JVM memory usage, profiled by JProfiler [4]. As the table shows, the synchronization overhead is large for Avrora, even in the absence of networking delay. The synchronization overhead exceeds 80% for Surge, and 85% for XnpCount. In contrast, the synchronization overhead is much lower in SnapSim – approximately 20%– 40%. In our observations, a single synchronization consumes approximately 1 millisecond for Avrora, and 850 microseconds for SnapSim since synchronized blocks in Java are inefficient compared to parallel execution. Another observation is that the save and restore snapshot methods consume little running time – no more than 1.5%. As a result, SnapSim simulates much faster than Avrora in most cases.

We next evaluate the JVM heap memory usage of both simulators. SnapSim generally requires a larger heap than Avrora because of its use of snapshots. As detailed in Table 5.1, an increase of 50% heap space was witnessed. We view the increase as worthwhile to achieve better simulation performance.

Finally, we evaluate the number of saved snapshots and restored snapshots. It is clear that the number of saved snapshots is equal to the number of bytes sent by all the nodes in the network. We find that most nodes backtrack to the start of the current wave. For example, for XnpCount in a network of 8 nodes, in the absence of networking delay, only 14.3% of snapshots are not restored, and 12.6% of them involve non-transmitting processes in a wave. Synchronization is slower than simulation, and as a result, most simulated nodes have already executed beyond the possible global nearest transmission time when the broadcast is received. Moreover, the ratio grows smaller as the networking delay increases. For the same scenario with 500 microseconds of networking delay, 9.9% of non-restored snapshots involve non-transmitting processes within a wave. However, compared to the synchronization time, the cost of re-execution is small; it does not have much of an impact on performance.

Chapter 6

Object Tracking

Sensor networks typically comprise a large number of nodes with limited resources. Hence, the object tracking framework must be scalable, and at the same time, lean in its use of computation and communication resources. With these goals in mind, we present a dynamically-scoped tracking algorithm and a supporting snapshotting algorithm[1] that uses asynchronous timestamping techniques to estimate synchronized sensor readings.

6.1 Design

6.1.1 Dynamically-Scoped Tracking Algorithm

We present a dynamically-scoped tracking algorithm that forms a sequence of *event regions*, each dominated by a *leader node* that senses the highest stimuli value within a given range. The leader nodes record the sensor readings from surrounding nodes that detect the target object. Leadership is passed along the tracking path as the target moves. We describe the basic idea, present the tracking algorithm formally, and finally consider each rule in detail.

Before describing the algorithm, we make our assumptions explicit: The network of sensor nodes can be described as a connected graph $G = (V, E)$. Each node $i \in V$ is a single processor with a unique ID, able to communicate with neighboring nodes via message passing. Neighbors of a node i are denoted as $N(i) = \{j|(i, j) \in E\}$. Each node is able to read and write its variables

[1] Here we use the term *snapshot* to refer to the record of sensor values collected by neighboring nodes at a specific time.

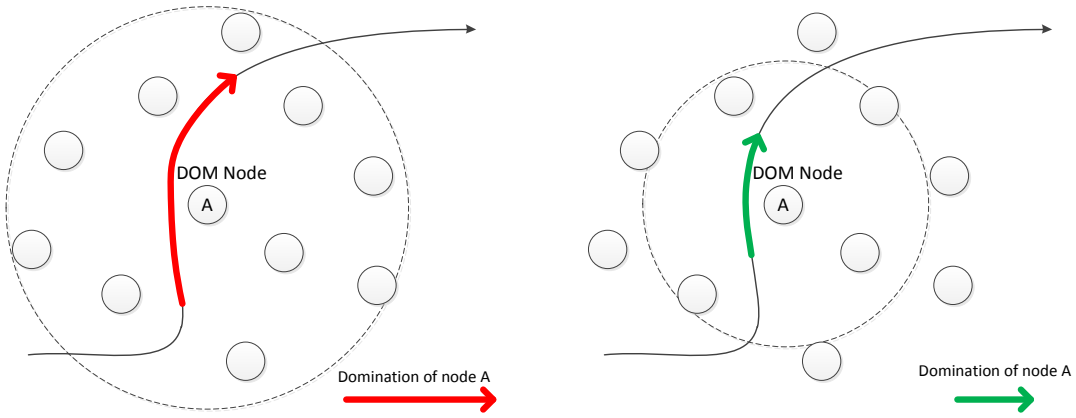
and send and receive messages; however, it is not able to access the variables of other nodes without messaging. Each node is equipped with a proximity sensor, and this sensor is sampled periodically. We assume that the distance to the target object is inversely proportional to the value of the sample. (This is common for many proximity sensors, at least probabilistically.) A node detects an object when the sampled value is higher than a specified *threshold* value. Other nodes may receive this value via radio communication.

We assume that nodes are activated by a timer with a constant period. Nodes may only transmit on each timer event to avoid message congestion. The sampling rate is at least as fast as the transmission timer rate. The recent sample values are stored in a buffer.

To simplify the presentation of the algorithm, we first assume that the network is synchronized to a global clock. (We will later abandon this assumption). As a result, sampling events can be synchronized. We assume that nodes have location information about themselves. Based on the samples and location information of neighboring nodes (included in reported messages), it is possible to compute an estimated position of the target object given that the samples are inversely proportional to the distance from the target. To enable location estimation, we assume that the network is over-provisioned. At any time t, more than two nodes are able to detect the object. The object trajectory is estimated by connecting the sequence of discrete points from estimated locations.

When an object is in the network, several nodes may detect the object at the same time. We refer to the node with the largest sample value as the *dominating node*[2]. Nodes within the *dominating range* of the dominating node are termed *dominated nodes*, and the set containing the *dominating node* and the *dominated nodes* is termed an *event region*. It is important to note that the event region may be different from the 1-hop communication range of the dominating node. The range of a proximity sensor is usually much smaller than the communication range of a typical sensor, and the network is over-provisioned to guarantee accuracy. Therefore, it is not necessary to include all the 1-hop neighbors in the event region. A smaller event region involves fewer nodes, which decreases the possibility of message congestion. A smaller event region also avoids interference with other nodes which are not related to the locally detected event – for example, another object detected in another section of the network. The trade-off of using a small event region is that if the moving object is fast enough, in one round of domination, it may "escape" from the event region before the dominating node passes the domination token to the next dominating node. Therefore,

[2]Node identifiers are used to break sample value ties to ensure a total ordering.



(a) Fast Moving Object (b) Slow Moving Object

Figure 6.1: Dynamic Scope of Event Region

the scope of a dominating node becomes a key factor that affects domination change. As a result, to track objects at different speeds, we use a dynamic scope for establishing the event region according to the speed of the moving object in the recent past. We show two examples in Figure 6.1. Node A is the dominating node, and the arrow represents the trajectory of the moving object. In Figure 6.1a, when the object is moving at a high speed (previously), a larger event region will be formed. All the nodes in the event region will be available for the next round of domination. In contrast, if the object is moving at a slower speed, as shown in Figure 6.1b, a smaller event region will be formed.

To formally define the algorithm, we present the *action system* program shown in Algorithm 2, which runs on each node.

Each node has a globally unique ID, i. The constant THR denotes the threshold of event detection; i.e., the value at which target detection is triggered. Each node maintains its location information, denoted by loc_i. As we assumed, each node has access to global time, denoted by τ, and can access its sensor reading via $x_i(t)$, which samples at time t. The dominating node identifier and dominating sample value (i.e., the maximum value in the event region), are stored in DOM_i and M_i, respectively. Variable t_{sample} denotes the global time when the last dominating value was sampled. Variable v_i denotes the estimated speed of the target object, while ρ_i denotes the radius of the event region. SS_i denotes the snapshots collected within an event region, which contains a series of snapshots over time. Each snapshot records the state of all reporting nodes captured by

Algorithm 2 Dynamically-Scoped Tracking Algorithm

CONSTANT
$THR : \mathbb{N}$ {event threshold, $THR > 0$}
VAR
$loc_i(\mathbb{N}, \mathbb{N})$ {2D location of node i}
$\tau : \mathbb{N}$ {synchronized global time}
$x_i(t) : \mathbb{N}$ {sample of i at time t}
$DOM_i : \mathbb{N}$ {dominating node of i}
$M_i : \mathbb{N}$ {max sample in the view of i}
$t_{sample} : \mathbb{N}$ {time when the dominating node last sampled}
$v_i : \mathbb{N}$ {object speed estimated by node i}
$\rho_i : \mathbb{N}$ {radius of event region}
$SS_i : Sequence$ {sequence of snapshots within event region of i}
FUNCTION
$broadcast(i, x, t, loc, \rho)$ {broadcast from dominating node}
$report(DOM_i, i, x, t, loc)$ {unicast from dominated node}
$addSnapshot(i, x, t, loc)$ {append a new snapshot to SS_i}
$computeSpeed(SS_i)$ {returns estimated speed of target}
$dis(loc, loc)$ {returns distance between nodes}
INITIALLY
$loc_i = (X_i, Y_i); DOM_i = \perp; M_i = THR;$
$v_i = v_{MAX}; \rho_i = \rho_{MAX};$
$SS_t = \phi; t_{sample} = 0;$
ASSIGN
if *timer event fires at* τ **then**
  if $x_i(\tau) > M_i \vee (DOM_i = i \wedge x_i(\tau) > THR)$ **then**
    if $(DOM_i = i) \wedge |SS_i| > 2$ **then**
      $v_i = computeSpeed(SS_i);$
      $\rho_i \propto v_i;$
    end if
    $SS_t = addSnapshot(i, x_i(\tau), \tau, loc_i);$
    $DOM_i = i; M_i = x_i(\tau);$
    $broadcast(i, M_i, \tau, loc_i, \rho_i);$
  else if $DOM_i = i \wedge x_i(\tau) \leq THR$ **then**
    $DOM_i = \perp; M_i = THR;$
    $broadcast(\perp, M_i, \tau, loc_i, \rho_i);$
  else if $DOM_i \neq i \wedge DOM_i \neq \perp \wedge x_i(\tau) > THR$ **then**
    $report(DOM_i, i, x_i(t_{sample}), t_{sample}, loc_i);$
  end if
end if
□
if $rcv.broadcast.j(DOM_j, M_j, t_j, loc_j, \rho_j)$ *event* **then**
  if $(DOM_i = DOM_j \vee M_i < M_j \vee DOM_j = \perp) \wedge dis(loc_i, loc_j) < \rho_j$ **then**
    $DOM_i = j; M_i = M_j; t_{sample} = t_j; \rho_i = \rho_j;$
  end if
end if
□
if $rcv.report.j(j, i, x_j, t_j, loc_j)$ *event* **then**
  $SS_i = addSnapshot(j, x, t_j, loc_j);$
end if
□

the dominating node. In the context of object tracking, the aggregate snapshot records the sensor readings from all nodes in the event region, which can be used to estimate the object's position.

We introduce five functions. $broadcast(i, x, t, loc, \rho)$ denotes a message broadcast from a dominating node, which contains the node ID, its recent sample value, the sample time, the location of the node, and the event region radius. In contrast, $report(DOM_i, i, x, t, loc)$ denotes a unicast message transmitted by a dominated node to a dominating node, which contains the dominating node ID, local node ID, its sample value, sample time, and node location. $addSnapshot(i, x, t, loc)$ adds snapshot information to SS_i. $computeSpeed(SS_i)$ computes the estimated speed of the object based on the current snapshot set. Since the sensors are noisy, the estimated speed is averaged through a moving window. $dis(loc, loc)$ returns the Euclidean distance between two points.

Initially, each node gets its own location (X_i, Y_i) and sets DOM_i to bottom, meaning *not dominated*. The dominating value M_i is set to THR; when a node detects a value higher than THR, it will declare event detection. v_i is set to the maximum speed, and ρ_i is set to the largest range since we do not have a priori knowledge of the target's speed. (The pre-set maximum radius is useful in capturing fast objects, if they appear.) Snapshot set SS_i is set to empty, and t_{sample} is set to 0.

The first rule executes when the periodic timer is fired, at global time τ. The sensor node compares its recent sensor reading $x_i(\tau)$ with its dominating value M_i. There are several cases to consider. If the sample $x_i(\tau)$ is greater than M_i, i becomes a dominating node. If i is already a dominating node, and its sample $x_i(\tau)$ remains greater than THR, it continues dominating. When a node is dominating and its current snapshot set contains at least 2 complete records for speed estimation, it estimates the speed of the object and estimates ρ_i, which is proportional to speed v_i. (Note that since the location estimation is not precise, ρ is typically set to be larger than the range the object could travel at the estimated speed.) Otherwise, the node will use the default value of ρ_i or the value obtained from the last dominating node, if available (rule 2). Next, the node adds a new snapshot record at time τ, and sets DOM_i and M_i to i and $x_i(\tau)$, respectively. Finally, a broadcast message is sent declaring domination in the event region.

If node i was previously dominating, but $x_i(\tau)$ is less than or equal to THR, the object is out of i's sensing range, and as a result, node i gives up domination, setting DOM_i and M_i to their initial values, and broadcasts a message indicating the node is no longer dominating.

If node i was previously dominated, and $x_i(\tau)$ is less than or equal to M_i, but greater than THR, it will report to the dominating node, DOM_i, with its sample value at time t_{sample} – the

timestamp it last received from the dominating node (rule 2).

The second rule executes when a node receives a broadcast message from a dominating node j. There are three possible scenarios if the node is within the event region range of j. First, if the dominating node was j in the last round (DOM_i is j), i still recognizes j as its dominating node. Second, if j is not equal to DOM_i, but the new dominating value from j is greater than the current value of M_i, j becomes i's new dominating node. Third, if j has already abandoned domination ($DOM_j = \perp$, Rule 1), i becomes not dominated. (Note that the object tracking process may continue if another node picks up domination in the next round.) In each case, the node sets DOM_i and M_i to DOM_j and x_j, respectively, records j's timestamp, and sets ρ_i to ρ_j for future use.

The third rule executes when a node receives a report message from a neighbor in the event region. The receiving node may not be the dominating node now, but it must have been dominating before. When it receives the report, it adds the information to the set of snapshots.

At this point, we haven't addressed how to generate a snapshot of an event region in an unsynchronized network, nor how target locations are estimated. To improve the algorithm, we describe an adaptive snapshotting technique using asynchronous timestamping and location estimation.

6.1.2 Adaptive Snapshots using Timestamping

Snapshots are triggered by a dominating node and record the sample states of neighboring nodes at times synchronized with the dominating node.

As presented, the snapshotting algorithm assumes time synchronization for recording snapshots at each time τ. However, network synchronization is computationally expensive and introduces additional communication overhead, potentially causing network delays and message congestion – especially during high data-rate tracking tasks. However, a limited form of network synchronization can be helpful in recording snapshots within an event region. Many time synchronization algorithms use timestamping triggered by a *start-of-frame delimiter* (SFD) interrupt generated by the on-board radio. The SFD interrupt is signaled at the same time on both the transmitter and the receiver (within tens of microseconds) and can be used to establish millisecond-level (or better) synchronization on nodes.

Using the SFD interrupt, we can synchronize node samples according to the timestamps received from dominating nodes. Figure 6.2 presents the process. The two horizontal arrows denote

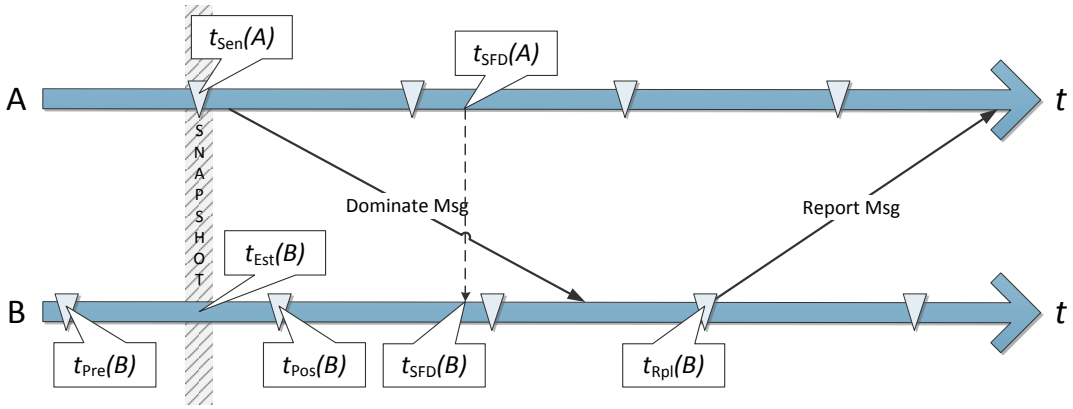


Figure 6.2: Timestamping Snapshots in an Event Region

the execution timelines of two nodes, A and B; A is the dominating node. The triangles denote timer events. Note that timer events are not synchronized. Suppose that node A samples its sensor at time $t_{Sen}(A)$ and transmits the data to node B. It often takes more than ten milliseconds for B to receive the packet, depending on network traffic and link quality. There is no way for node B to tell when A sensed the data without timestamping. However, at $t_{SFD}(A)$, the SFD interrupt fires; at approximately the same time, the SFD interrupt also fires on B at $t_{SFD}(B)$. Therefore, if we transmit the timestamp of $t_{Sen}(A)$ along with the timestamp at $t_{SFD}(A)$[3], B can compute the sensing time in its view by $t_{Est}(B) = t_{SFD}(B) - (t_{SFD}(A) - t_{Sen}(A))$. However, since A and B are not synchronized, there may be no sensing value at time $t_{Est}(B)$. Since the sampling period is very small, we can assume that the sample value changes linearly, and the sample $x(t_{Est})$ at t_{Est} can be estimated by Equation 6.1.

$$x(t_{Est}) = x(t_{Pre}) + (x(t_{Pos}) - x(t_{Pre})) * (t_{Est} - t_{Pre})/P \tag{6.1}$$

where $x(t_{Pre})$ and $x(t_{Pos})$ are samples just before and after t_{Est}, respectively. P is the sampling period. To access the previous samples, a fixed size queue is used to buffer the samples. As a result, it is possible to estimate the sample values across the event region to generate a synchronized snapshot at a given time specified by the dominating node, without introducing additional communication overhead.

To keep track of the duration of each dominating round, we transmit the dominating du-

[3]Transmission of timestamp t_{SFD} is supported in hardware.

ration within the dominating node's broadcasts. The next dominating node, which was dominated in the last event region, receives the duration of the last dominating node, and the time of sensing as well. The time difference between dominating nodes is added to the received duration, since they are not synchronized. Therefore, an accurate accumulated duration of object detection can be generated along the path.

Another challenge in snapshotting is the rate of snapshot generation. Since the timer period is short, if we transmit the dominating message and record a snapshot on every timer event, as shown in Algorithm 2, the battery will be consumed quickly, especially for slow moving objects, which may stay in the network longer. In this case, nodes may transmit many redundant samples given the high sampling rate. To reduce the number of transmissions without losing information, we dynamically adjust the snapshotting rate according to the target's movement. Besides using a constant snapshotting rate, which is usually much lower than the timer event rate, we also record a snapshot in the following two scenarios. First, when a new event region is established, a snapshot is taken immediately. Second, when a previously inactive node detects the object in the event region, a snapshot is recorded since the change of event region indicates the movement of the object. In addition, to notify the dominating node, a "join" message is sent to the dominating node by the newly detecting node. Third, if the rate of change in the samples collected by the dominating node is higher than a given threshold in the past round, it indicates the object may move fast in the region, thus a snapshot is taken.

6.2 System Implementation

We implement the algorithm in the context of tracking point light sources. To track high-speed objects, we use a high sampling rate (i.e., 100Hz). We adopt a typical state machine design. Figure 6.3 presents a simplified automata that captures the high-level behavior of the system.

A node state changes only if a periodic timer event is signaled, denoted by a solid edge, or a broadcast message is received, denoted by a dashed edge. Each node begins in the START state, where it computes the ambient light level, which may vary from one node to another. Later, when detecting a light source, each node compares its reading with its ambient value and transmits the difference. This supports tracking in environments with complex ambient conditions.

Next, the node transits to IDLE; it transits back to START periodically to update its

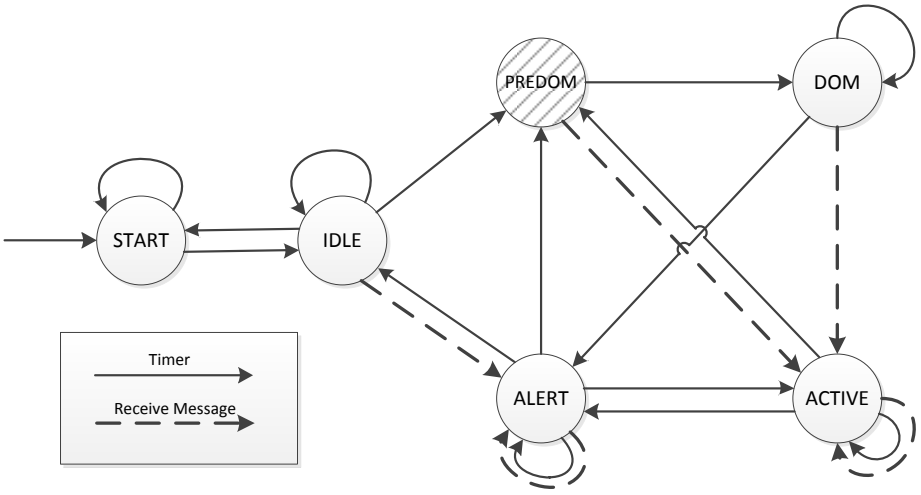


Figure 6.3: System Model

ambient light calculation. We define the event detection threshold THR as a constant threshold value plus the ambient light detected by the node. The node will remain in the IDLE state while its sensor reading is less than THR.

We introduce a temporary state, PREDOM, entered when a node detects an object and tries to dominate the event region by broadcasting its value. When a target appears in the network, several nodes may detect the object, but typically not at the same instant. As a result, the node which detects the object first may not be the one with the largest sample value. PREDOM is introduced to stabilize the event region; only one dominating node will be selected.

If a pre-dominating node doesn't receive a message with a higher value before the next timer event, it transits to the DOM state, and dominates the event region. At this point, the node begins to broadcast its domination and records the snapshots received within the event region according to the object's speed. The node remains in the DOM state until one of two cases occurs. First, if the sensed value is lower than THR, it transits to the ALERT state. Second, if it receives a higher value from a pre-dominating node in the event region, it transits to the ACTIVE state.

The ACTIVE state is reached when a node is dominated and has a sensor reading higher than THR. ACTIVE nodes will report their values to the dominating node after receiving dominating broadcasts. When a PREDOM or DOM node receives a value higher than its value, it transits to the ACTIVE state. Analogously, when an ACTIVE node has a sensor reading higher than the

dominating value, it transits to PREDOM, and may later become a DOM node. When an ACTIVE node obtains a value below THR, it changes to the ALERT state.

The ALERT state indicates that a node is in an event region, but contains a value less than THR. It may report to the dominating node if it was ACTIVE at the time it received the last dominating broadcast. If it has a value higher than THR, but less than the dominating value, it becomes ACTIVE; otherwise, if its value is higher than the dominating value, it transits to PREDOM. If it hasn't received a dominating message after a given period, it becomes IDLE.

The transition sequence can be complicated. Consider the network containing 4 nodes shown in Figure 6.4. The nodes are labeled from A to D; we assume they are within a single event region. The horizontal bars denote the state of each node over time. Each triangle denotes an unsynchronized timer event, fired periodically.

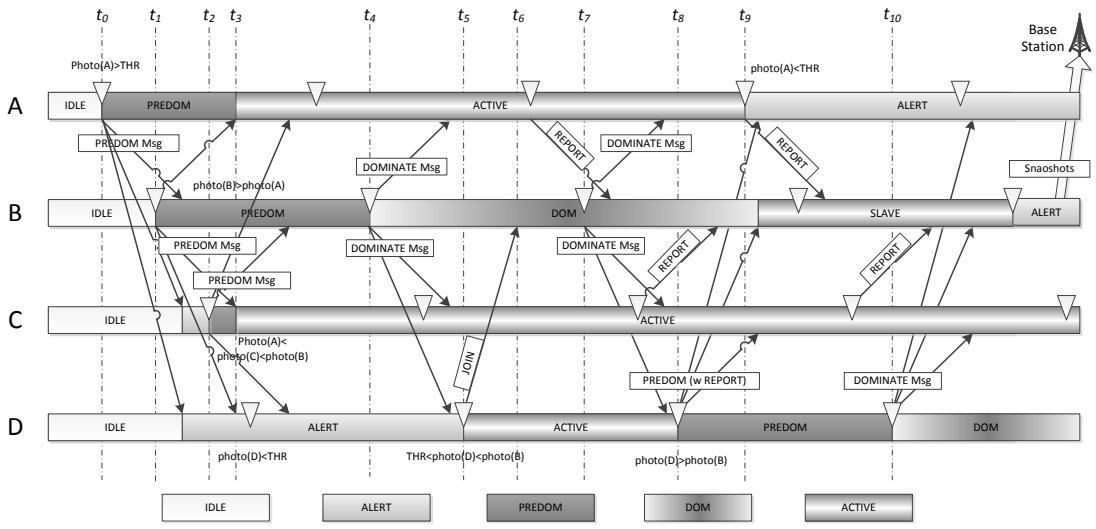


Figure 6.4: Example of Interactions Among Nodes

Initially, all nodes are in the IDLE state. At time t_0, node A detects an object when its timer fires, moves to the PREDOM state, and broadcasts its message. It takes some time for transmission and reception to complete. Before B receives the message from A, it detects the object at time t_1, with a value higher than A's value. B also transits to the PREDOM state and broadcasts. When C and D receive the PREDOM message from A, they set themselves to ALERT. When C detects the object with a higher value than A at time t_2, it also moves to the PREDOM state and broadcasts. However, it is preempted when it receives a higher value from B at t_3. Only one node will succeed

in the bid for domination; the others will become ACTIVE. To improve stability, a node must have a value higher than the previous proposed value, plus a constant threshold (to avoid domination oscillation).

Assume B becomes the dominating node at time t_4 when its timer fires again, and starts to broadcast its dominating value. All the ACTIVE nodes that receive the dominating message must report their values to the dominating node when their timers fire. Assume node D obtains a value higher than THR, but less than B at t_5; it becomes a new ACTIVE node and joins the event region via a join message. When dominating node B receives the message at t_6, it broadcasts again at t_7, and records a new snapshot when reports are collected. At time t_8, when D has a value higher than B, it moves to the PREDOM state and broadcasts. Since node D has received a dominating message from B, it must still report its value at time t_7; the PREDOM broadcast also contains a report value for B. Similarly, node A must report to node B at time t_9, even though it is in the ALERT state, since it was ACTIVE at time t_7, when B requested a report. The captured snapshots at node B may be sent back to the basestation through a collection tree when B enters the IDLE state. D will become the new dominating node at time t_{10}.

The upper-bound on tracking speed is constrained by the timer rate and the packet transmission rate. If an object is moving fast enough, it may leave an event region before domination has been established (when the next timer fires); thus, the dominating node is unable to transit from PREDOM to DOM. In such a case, the object may "escape" the event region without being tracked.

6.2.1 Tracking Structure

Figure 6.5 illustrates the snapshot data structure used at each dominating node, a three-layer tracking list. The first layer contains information about the target. Each identified target is assigned a unique *track ID*. The track ID comprises two parts, the ID of the first dominating node in the tracking process (*ID*), and a local counter (*track step*). The track ID is passed along as the object moves. It allows multiple objects moving in the network at the same time, as long as the active event regions never overlap (at the same time). *track step* is a simple counter. Since a node may be selected as a dominating node multiple times during a tracking process, the tracking step is used to disambiguate the domination sequence. Each tracking record maintains a pointer to a snapshot structure (level 2), and a pointer to the next tracking record.

A snapshot records the sensing values within an active event region at the time of request

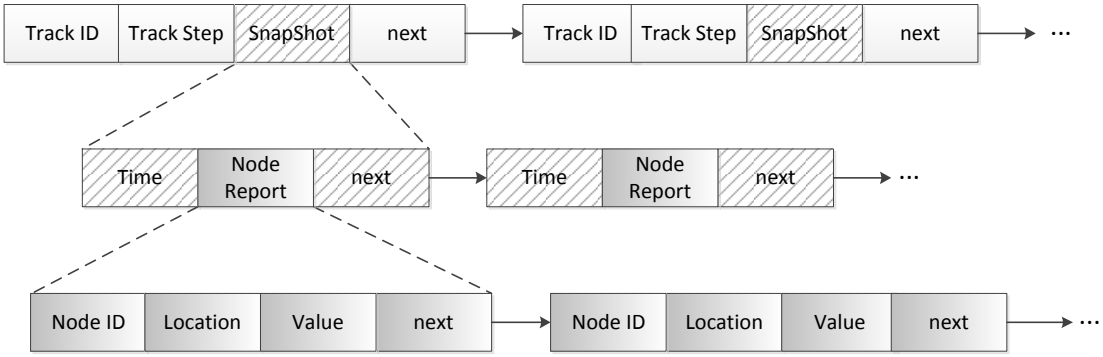


Figure 6.5: Tracking Structure

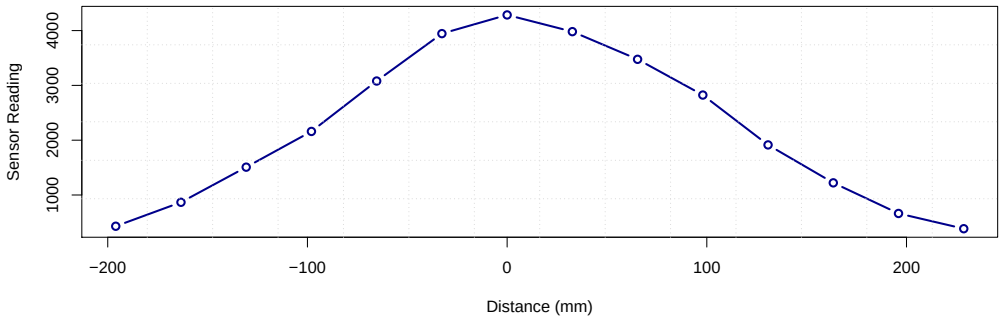


Figure 6.6: Photo Strength vs Distance

by the dominating node. Each snapshot record records the time when the snapshot was requested, beginning with the first dominating node. Finally, it includes a linked chain of node reports. Each report contains the reporting node ID, node location, and report value. The chain is used to estimate the target's location; the time-stamps are used to estimate speed and trajectory.

6.2.2 Location Estimation

The attenuation of a typical point source of light is non-linear and difficult to analyze. Measured intensity varies with distance, angle, and reflection. To analyze these issues, we mounted a point light source on a tripod at a distance of 1.5 meters from a wall. Fourteen sensors were deployed side-by-side on the wall. The light source was pointed at the middle of the line. Figure 6.6 shows the measured light intensity versus distance (from center). The readings are modulo the associated ambient light levels. The attenuation is almost linear with distance – but the slope

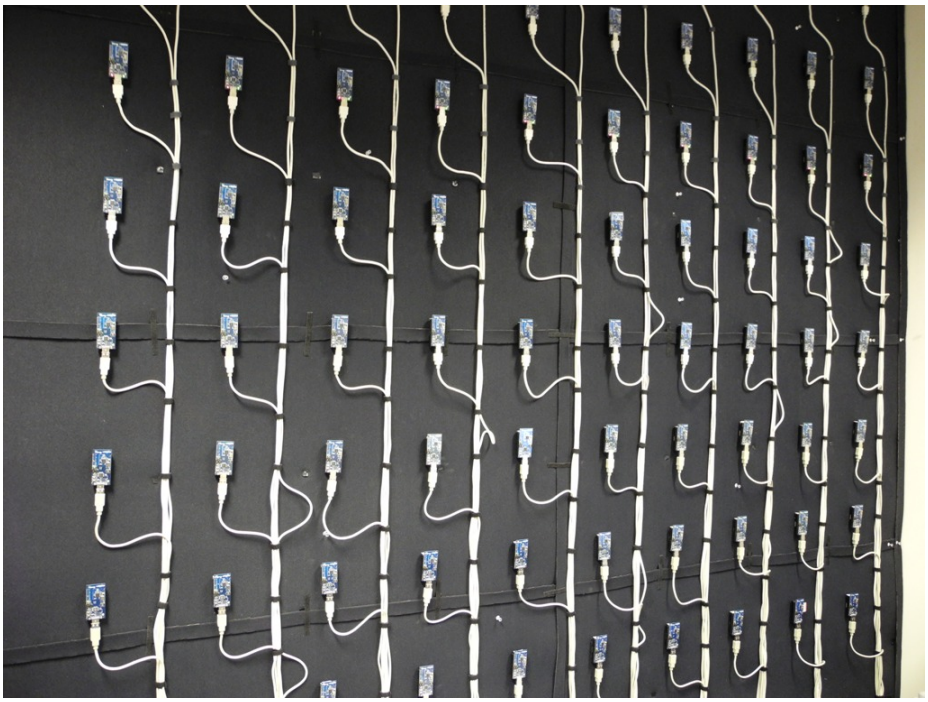


Figure 6.7: NESTbed

decreases at the center and edge areas. Based on these observations, we adopt a linear-weighted location estimation approach. We compute the weighted average on both the X and the Y axes, according to Equation 6.2.

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i[X_i, Y_i]}{\sum_{i=1}^{n} x_i} \tag{6.2}$$

where n is the number of reporting sensors, x_i is a reading from sensor i, and vector $[X_i, Y_i]$ is the 2D coordinate of i.

6.3 Evaluation and Analysis

We evaluated the tracking implementation on the Clemson University NESTbed system [21]. The testbed consists of 80 Telos nodes with on-board photo sensors (Hamamatsu S10871). The nodes are deployed on a wall-mounted, semi-regular grid of 10 columns and 8 rows, as shown in Figure 6.7. For evaluation purposes, we used a common flashlight as the light source. We didn't require a fixed distance or angle, and did not control the ambient light in the room.

Consider the mobile light trajectory generated by FlashTrack in Figure 6.8. The target was moving from the bottom-right corner of the testbed to the top-left corner. The estimated trajectory

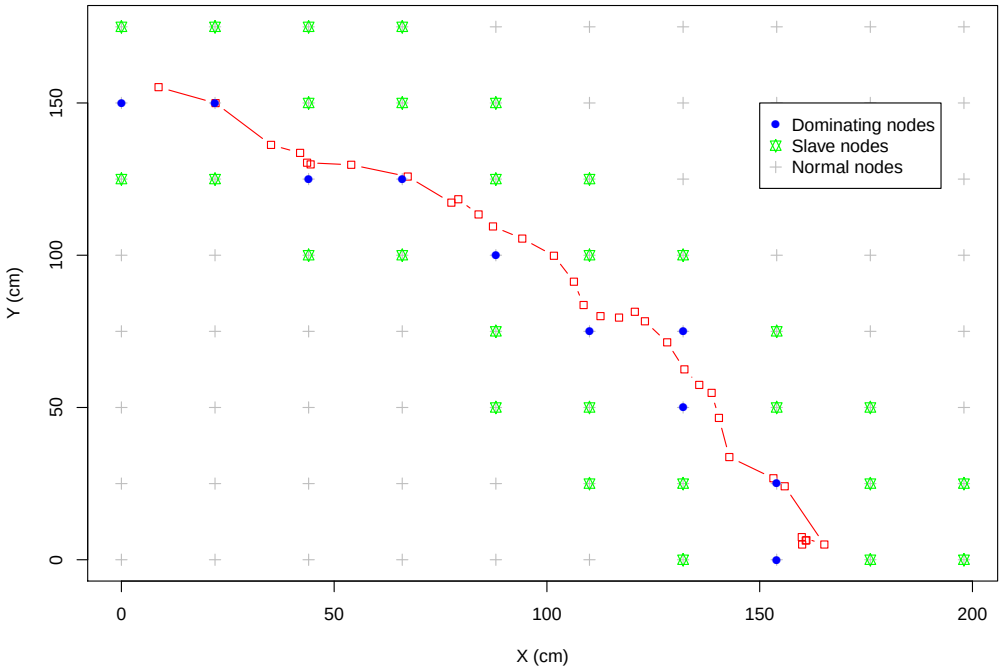


Figure 6.8: Object Trajectory and Dominating Regions

was generated by connecting the raw estimated locations streamed from the network. In the plot, the "+" signs denote the position of the sensor nodes deployed in the grid. The blue dots denote the dominating nodes, and the green markers denote the active nodes along the target's path. It is clear that the dominating nodes are selected close to the target's path, and the active nodes are neighboring sensors which detect the target. On average, a dominating node dominates 3 to 4 active nodes when at the edge of the network, and 5 to 8 active nodes otherwise. The actual event region is larger; other nodes are in the alert state.

To record ground truth, an infrared recording system was developed: A Wiimote [1], which includes an infrared camera, was installed at a fixed point in front of the testbed as a recording device. The Wiimote provides high resolution, high frequency infrared point tracking. We installed a 65mW infrared laser on the flashlight, which provides a reflected tracking point for the Wiimote, as shown in Figure 6.7. The tracking point location is streamed by Bluetooth to a client application and recalibrated to match the testbed's coordinates .

We evaluate the FlashTrack implementation with objects of varying speeds: (i) slow, $0 - 1.5(m/s)$; (ii) moderate, $1.5 - 4(m/s)$; (iii) fast, $4 - 10(m/s)$; (iv) out of range, $> 10(m/s)$. The cell size is approximately 25 centimeters; as a result, slow objects travel less than 6 cells per second, moderate speed objects travel less than 16 cells per second, and the upper bound is 40 cells per

second.

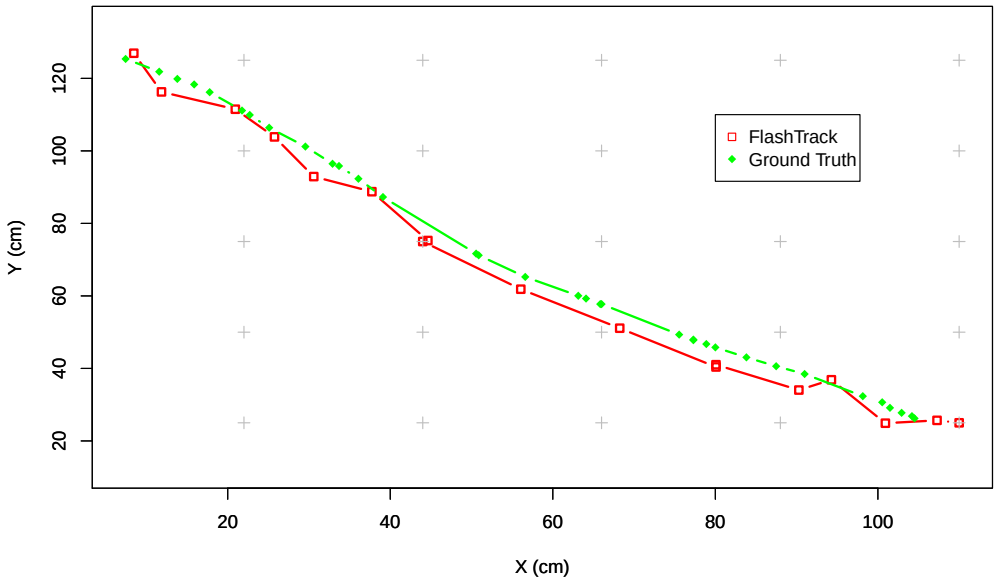


Figure 6.9: Straight Line Trajectory (moderate speed)

We first investigated a straight line trajectory at a moderate speed. Figure 6.9 shows the trajectory computed by FlashTrack and the ground truth provided by the Wiimote. The object traversed the network in approximately 1.3 seconds. The average snapshot rate was 13.14 snapshots per second. The figure shows that the trace estimated by FlashTrack matches the ground truth trajectory well. The maximum location estimation error is less than 4 cm. However, because of sensor noise and limited deployment density, the raw data generated a zig-zag trace. This can be improved by curve fitting during post-processing, which is not the concern of this thesis.

We also investigated the speed estimates provided by FlashTrack for the same trajectory, compared with ground truth. We computed the speed of the object by averaging the speed estimates through a filter of 4 samples. Figure 6.10 presents the estimated object speed over time. Like the trajectory estimates, the speed trends between the two sets match well. However, it is not precise at each step, particularly, at the beginning of the trajectory, where the estimated speed is slower than ground truth. The explanation is that the object was first detected at the edge of the network (bottom-right in Figure 6.9), where there are no active nodes beneath the first dominating node; the estimation result is biased due to asymmetric node coverage. The average speed computed by

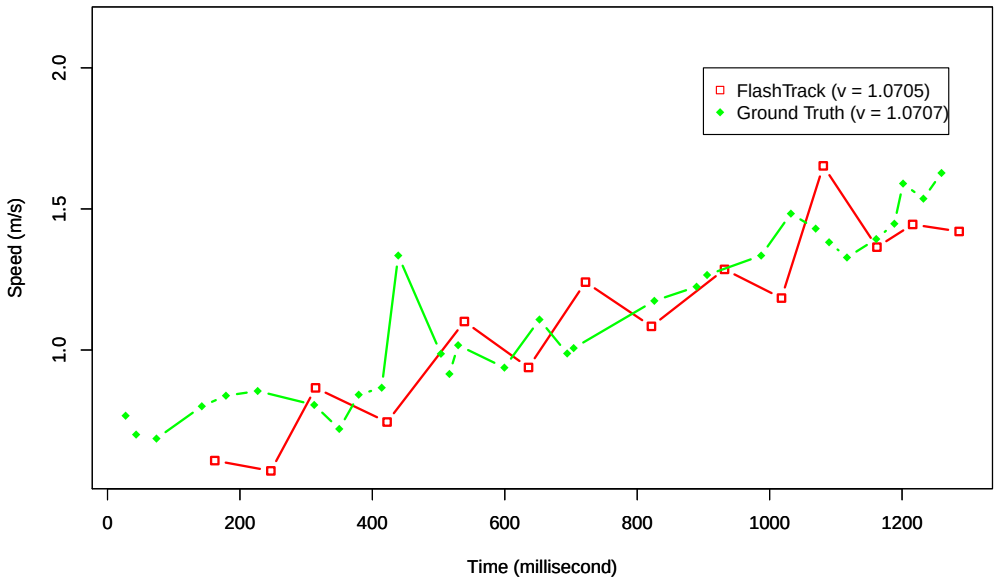


Figure 6.10: Speed of Trajectory (moderate speed)

FlashTrack differs from ground truth by less than 1% in this case, on average.

Figure 6.11 shows an example of a long trajectory with a slower speed. The target persists for more than 5 seconds in the network. More than 60 estimated points were generated, and the snapshot rate dropped to 11.1 snapshots per second. Also, a small event region was generated for each dominating node since the speed of the object was slow; as a result, fewer nodes are included in the computation. Still, the estimated trajectory roughly matches the ground truth trajectory, though the raw curve is noisy. Figure 6.12 shows the speed comparison with ground truth. Although the speed estimation fluctuates, the average speed estimation is accurate – the error is less than 3%, on average.

We also investigated the performance of our framework for fast mobile objects. First, we captured a moving object with a speed of more than 2.5 meters per second, which appears in the network for approximately 650 milliseconds. The trajectory is shown in Figure 6.13. As the figure shows, the estimated trajectory correctly captures the object trace; 11 points are captured, and the snapshot rate reaches 16.1 snapshots per second. The average speed estimation error is less than 5% on average, when compared to the ground truth result.

Finally, we investigated an even faster object - which has a speed of more than 7 meters

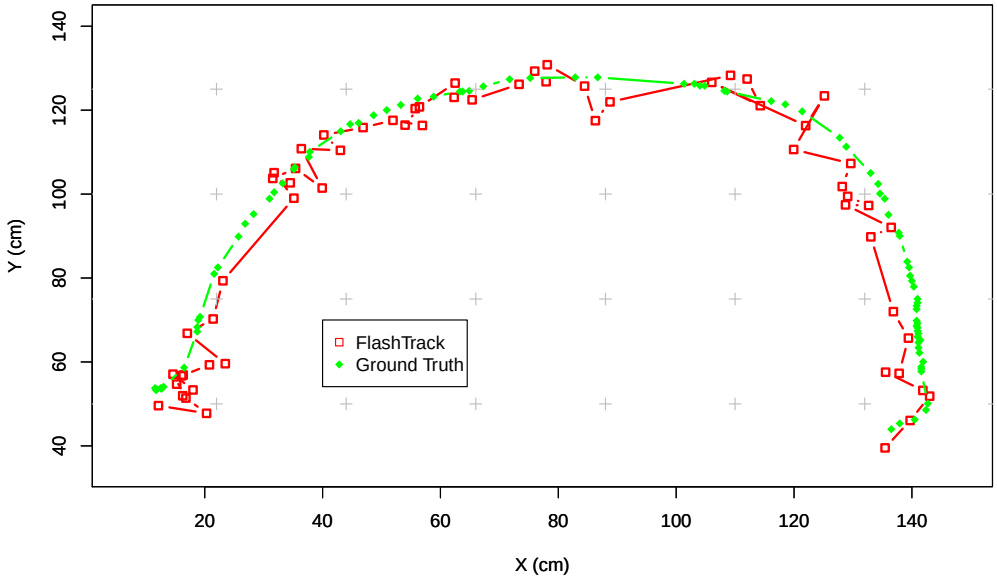


Figure 6.11: Long Curve Trajectory (slow speed)

per second. The object passes through the network in approximately 200 milliseconds, as shown in Figure 6.14. In this case, it only requires 35 to 40 milliseconds for the object to move from one node to another, which is barely sufficient for the sensor to establish an event region. The figure shows that the algorithm still captures the correct target trace, and the speed estimation error is approximately 7%. The snapshot rate reaches 21.3 snapshots per second, due to the speed of the object. When an even faster object ($> 10m/s$) passes through the network, the object may escape from the event region before the event region is established. Therefore, the algorithm failed to capture a complete trace at this speed.

In summary, our tracking framework captures the trajectory the mobile object in-network, achieves high fidelity, dynamically adapts to object speed, and is capable of capturing both fast and slow-moving objects.

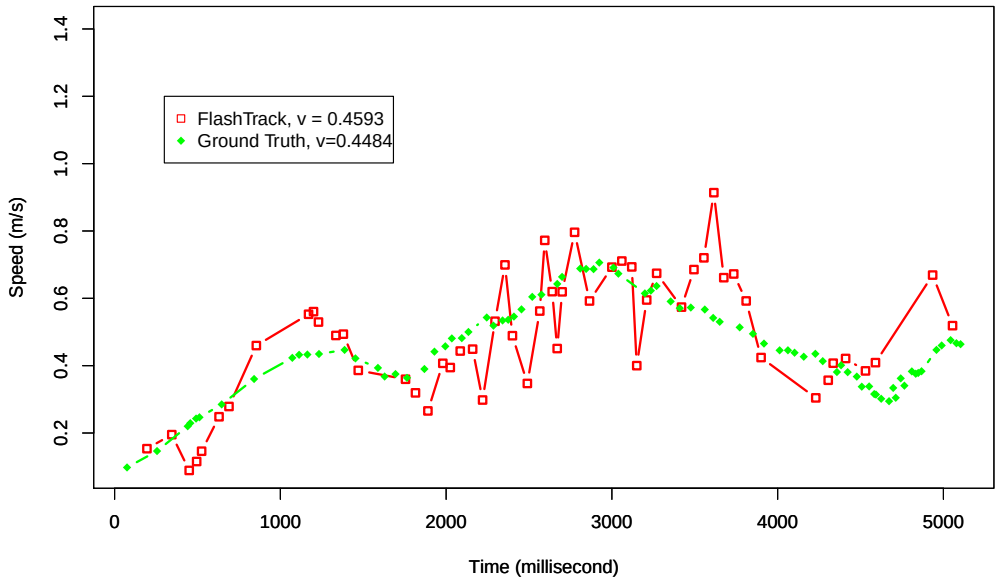


Figure 6.12: Speed of Trajectory (fast)

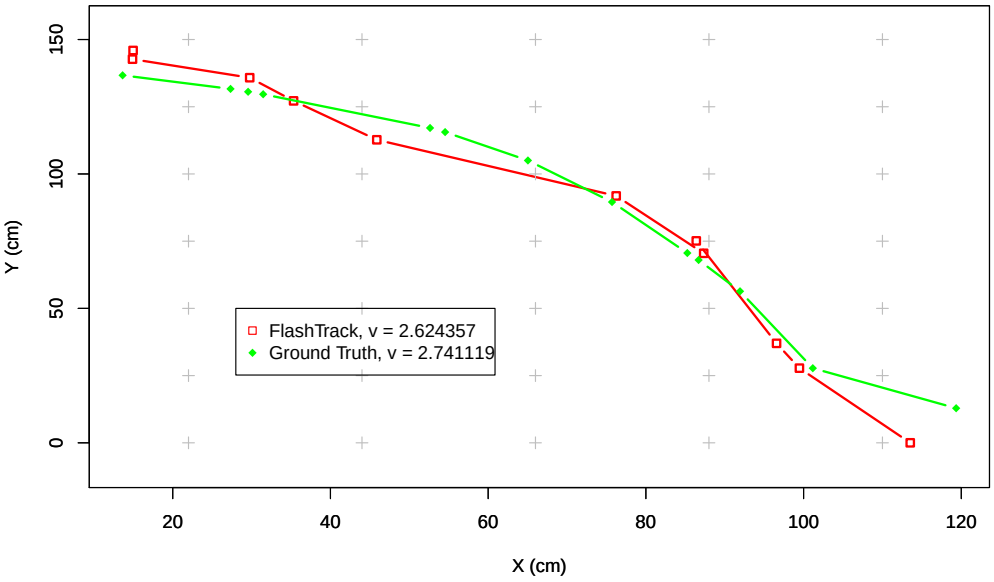


Figure 6.13: Curve Trajectory (fast speed)

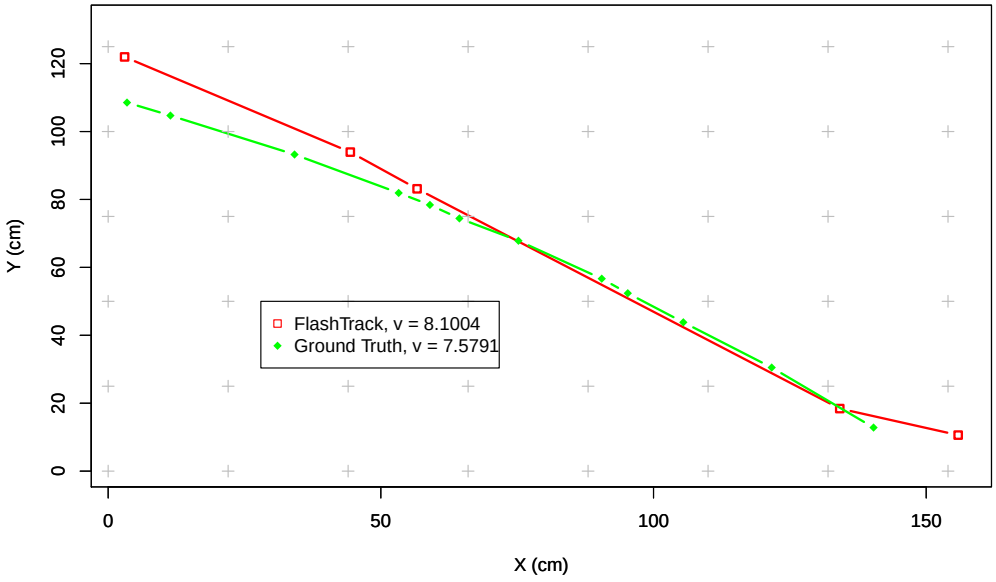


Figure 6.14: Straight Line Trajectory (fastest speed)

Chapter 7

Conclusion

Wireless sensor networks consist of large numbers of energy-constrained sensors with limited computational resources. Numerous applications across a variety of areas show the promise of this technology. However, due to the nature of the hardware, fast and efficient classification and target tracking are challenging in this context. In contrast to traditional desktop applications, sensor implementations require light-weight and energy-efficient design techniques. Tracking fast moving objects requires high sampling and data rates, which conflicts with the resource constraints of common sensors. Further, due to the large number of nodes in typical sensor networks, parallel simulation is usually applied, where speed is the key performance concern.

7.1 Challenge Summary

7.1.1 Classification

The main challenge addressed by our work centers on the mismatch between computationally intensive classification techniques and resource-constrained sensor nodes. We present a generic, *node-level* classification framework for resource-constrained sensors, such as the popular Tmote platform [10], with an MSP430 microprocessor. We present two different classification techniques and a preprocessing technique for accelerometer data. In the first design, we use a multi-dimensional Bayesian Classifier, which is relatively light-weight and suitable for resource-constrained devices. In the second design, we use an innovative classifier, a condensed kd-tree, which can reduce the number

of leaves in a regular kd-tree by 90.0%. Using k-nearest neighbor search in the condensed kd-tree, we classify incoming events in $O(n^{\frac{1}{2}})$ time for the 2-dimensional case, where n denotes the tree size. The preprocessing technique uses a sliding window to smooth the accelerometer readings in $O(1)$ time; this improves the performance of the classifiers significantly. A post-classification voting method further improves accuracy. Both classifiers yield high classification accuracy in our case studies and reduce communication overhead and energy consumption when compared to the raw data collection approach. Moreover, since each classifier is generated in-network, re-training is energy-efficient.

7.1.2 Network Simulation

In this component of the dissertation, the main challenge addressed by our work centers on reducing the synchronization overhead in parallel simulation systems. Without using traditional interval synchronization, which introduces significant synchronization overhead, we present an optimistic distributed synchronization algorithm that uses probing execution and backtracking. Jefferson [33] has argued that general rollback is too complex and inefficient to implement. We show, however, that our design is capable of reducing the number of synchronization cycles significantly and improves performance by 2 to 10 times for representative WSN applications, compared to the interval synchronization algorithm.

7.1.3 Object Tracking

Finally, in the third focus area, the main challenge addressed by our work centers on designing a fast and efficient in-network tracking framework which detects and tracks moving objects. The framework is based on a distributed algorithm which tracks a dynamically-scoped event region in the network. A leader is elected for every event region. The leader adaptively constructs the event region according to the speed of the target, receives reports from nodes in the event region, and computes snapshots of the states of the nodes. The tracking framework is adaptive; the sampling rate is adjusted based on the speed of the tracked object. Our framework is hierarchy-free, scalable, and does not require time synchronization. The object tracking task is executed on-node and is capable of capturing fast moving objects in-network.

7.2 Contribution Summary

7.2.1 Contribution 1

While machine learning techniques offer a number of advantages in the context of sensor-based event detection, their application presents a challenge for in situ scenarios. Existing techniques are resource-intensive, precluding direct implementation on mote-class platforms. We explore both existing and new techniques to support training and classification on resource-lean devices.

We first consider Bayesian classification, a traditional machine learning method, and show that when this approach is carefully implemented, the computational complexity is suitable for resource-constrained sensor networks. The multi-dimensional discriminant functions are computed and stored on-node efficiently. The classification result is generated from multiple discriminant sub-results. We next consider condensed kd-tree classification, a novel classification method that uses an enhancement of a standard kd-tree as the underlying representation structure. A dynamically-scoped nearest neighbor search technique is used to classify incoming data samples based on the distance-weighted categorizations of corresponding neighbors. The representation allows developers to adjust the tree size to accommodate memory-limited hardware without a significant loss in classifier accuracy. Next, we consider pre-processing enhancements to both classifiers. Finally, the performance of both classifiers and the corresponding enhancements are evaluated in the context of two representative case studies.

The classification methods presented support fast classification, achieve high accuracy, require little memory, and support in-network retraining. They are suitable for a wide range of sensor network applications.

7.2.2 Contribution 2

While parallel simulation increases the potential scale of sensor network simulations, simulation speed is constrained by the synchronization overhead introduced by common interval synchronization techniques. We introduce a new probing and backtracking technique to reduce the synchronization overhead.

We present a probing and backtracking algorithm, which optimizes the number of synchronizations to match the number of transmission dependencies. The algorithm synchronizes processes only if a transmission is detected. Each process finds the nearest global transmission time using

probing execution; processes that have passed the global nearest transmission time are backtracked. To enable backtracking, a snapshot of each process is saved at the last global nearest transmission time. A semi-formal proff as well as case analyses are provided.

The prototype implementation, SnapSim, is based on Avrora. The experimental results show that SnapSim is faster than Avrora in most cases. Considering a networking delay of 500 μs, SnapSim is 8 times faster than Avrora for typical TinyOS applications like Surge, when the network size is 8. Without a networking delay, SnapSim is approximately 4 times faster than Avrora.

7.2.3 Contribution 3

While there has been significant work in target tracking using a range of techniques, there is a lack of efficient and accurate techniques for tracking fast moving objects in-network. Therefore, we propose a new light-weight tracking framework, which is adaptive based on target speed.

The tracking framework is based on a hierarchy-free, distributed tracking algorithm which tracks mobile objects using a dynamically-scoped event region. Each event region is dominated by a leader node which detects the maximum sensor value in the event region. Each dominating node coordinates the tracking process in its event region; the scope is based on estimated target speed. A snapshotting algorithm is used for capturing the information in each event region; the rate is also adjusted based on object speed. To avoid expensive time synchronization, an SFD-based timestamping technique is used to obtain synchronized sensor reading estimates.

We implement the tracking framework in the context of light source tracking and evaluate it on a large-scale testbed. We track an infrared laser using a Wiimote to record ground-truth in our experiments. The experimental results show that the tracking implementation efficiently and accurately tracks the trajectory of light objects, even for very fast point sources.

Bibliography

[1] Controllers at nintendo : Wii : Console. www.nintendo.com/wii/console/controllers.

[2] Corssbow iris. www.dinesgroup.org/projects/images/pdf_files/iris_datasheet.pdf.

[3] Crossbow telosb. www.willow.co.uk/TelosB_Datasheet.pdf.

[4] ej-technologies. http://www.ej-technologies.com/.

[5] Fitbit. http://www.fitbit.com/.

[6] Htc evo 4g. http://www.htc.com/us/products/.

[7] Mica2. http://bullseye.xbow.com:81/Products/productdetails.aspx?sid=174.

[8] Motestack. http://www.intelligentriver.org/.

[9] Texas instrument cc2420. http://www.ti.com/product/cc2420.

[10] Tmote platform. http://www.moteiv.com/.

[11] Tmote sky. http://www.sentilla.com/.

[12] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Communications Magazine*, 40(8):102–114, 2002.

[13] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605 – 634, 2004.

[14] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus. Tracking a moving object with a binary sensor network. pages 150–161. ACM Press, 2003.

[15] H. S. Baird, M. A. Moll, and C. An. Document image content inventories. 2007.

[16] L. Bajaj, M. Takai, R. Ahuja, and R. Bagrodia. Simulation of large-scale heterogeneous communication systems. In *The IEEE Military Communication Conference*, pages 1396–1400, Washington DC, USA, october–november 1999. IEEE.

[17] M. Borazio and K. Van Laerhoven. Combining wearable and environmental sensing into an unobtrusive tool for long-term sleep studies. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pages 71–80, New York, NY, USA, 2012. ACM.

[18] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and regression trees. 1984.

[19] R. Brooks, P. Ramanathan, and A. Sayeed. Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE*, 91(8):1163 – 1171, aug. 2003.

[20] T. R. Burchfield and S. Venkatesan. Accelerometer-based human abnormal movement detection in wireless sensor networks. In *HealthNet '07: Proceedings of the 1st ACM SIGMOBILE international workshop on Systems and networking support for healthcare and assisted living environments*, pages 67–69. ACM, 2007.

[21] A. Dalton and J. Hallstrom. An interactive, server-centric testbed for wireless sensor systems. Technical Report CUDSRG-08-06-01, Clemson University (Dependable Systems Research Group), 2006.

[22] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In T. Higashino, editor, *Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 900–900. Springer Berlin / Heidelberg, 2005.

[23] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* Imperial College Press, 2001.

[24] G. Forney. The viterbi algorithm. In *Proceedings of the IEEE*, volume 61, pages 268–278. IEEE, 1973.

[25] R. K. Ganti, P. Jayachandran, T. F. Abdelzaher, and J. A. Stankovic. Satire: a software architecture for smart attire. In *MobiSys '06: Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 110–123. ACM, 2006.

[26] L. Guibas. Sensing, tracking and reasoning with relations. *Signal Processing Magazine, IEEE*, 19(2):73 –85, mar 2002.

[27] R. Gupta and S. R. Das. Tracking moving targets in a smart sensor network. In *In Proceedings of IEEE VTC Fall 2003 Symposium*, pages 3035–3039, 2003.

[28] N. Györbíró, A. Fábián, and G. Hományi. An activity recognition system for mobile phones. *Mob. Netw. Appl.*, 14(1):82–91, 2009.

[29] J. He, H. Li, and J. Tan. Real-time daily activity classification with wireless sensor networks using hidden markov model. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 3192 –3195. IEEE, 2007.

[30] T. He, P. Vicaire, T. Yan, L. Luo, L. Gu, G. Zhou, R. Stoleru, Q. Cao, J. A. Stankovic, and T. Abdelzaher. Achieving real-time target tracking usingwireless sensor networks. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:37–48, 2006.

[31] C. R. Henk and H. Muller. Context awareness by analysing accelerometer data. In *The Fourth International Symposium on Wearable Computers*, pages 175–176. IEEE, 2000.

[32] M. Herlihy and S. Tirthapura. Self stabilizing distributed queuing. In J. Welch, editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 209–223. Springer Berlin / Heidelberg, 2001.

[33] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, July 1985.

[34] Z.-Y. Jin and R. Gupta. Improved distributed simulation of sensor networks based on sensor node sleep time. In *In International Conference on Distributed Computing in Sensor Systems*, volume 5067 of *Lecture Notes in Computer Science*, pages 204–218. Springer Berlin / Heidelberg, 2008.

[35] Z.-Y. Jin and R. Gupta. Improving the speed and scalability of distributed simulations of sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 169–180, Washington, DC, USA, 2009. IEEE Computer Society.

[36] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGPLAN Not.*, 37:96–107, October 2002.

[37] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Wireless sensor networks for structural health monitoring. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 427–428. ACM, 2006.

[38] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 254–263. ACM, 2007.

[39] W. Kim. On target tracking with binary proximity sensors. In *In Proc. IPSN*, pages 301–308. IEEE Press, 2005.

[40] Y. Kim, S. Jeong, D. Kim, and T. S. López. An efficient scheme of target classification and information fusion in wireless sensor networks. *Personal Ubiquitous Comput.*, 13(7):499–508, 2009.

[41] C. Knigge. The intrinsic fraction of broad-absorption line quasars. *Monthly Notices of the Royal Astronomical Society*, 386:1426–1435(10), May 2008.

[42] V. Kulathumani, A. Arora, M. Demirbas, and M. Sridharan. Trail: A distance sensitive wsn service for distributed object tracking. In K. Langendoen and T. Voigt, editors, *Wireless Sensor Networks*, volume 4373 of *Lecture Notes in Computer Science*, pages 83–100. Springer Berlin / Heidelberg, 2007.

[43] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *The* 7^{th} *International Conference on Information Processing in Sensor Networks*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[44] U. Legedza and W. E. Weihl. Reducing synchronization overhead in parallel simulation. In *The* 10^{th} *Workshop on Parallel and Distributed Simulation*, pages 86–95, Washington, DC, USA, 1996. IEEE Computer Society.

[45] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *The* 1^{st} *International Conference on Embedded Networked Sensor Systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[46] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In W. Weber, J. M. Rabaey, and E. Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer (Berlin, Heidelberg), 2005.

[47] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An operating system for sensor networks.* Springer Berlin Heidelberg, 2004.

[48] J. Liu and D. Nicol. Lookahead revisited in wireless network simulations. In *The 16th workshop on Parallel and Distributed Simulation*, pages 79–88, Washington DC, USA, may 2002. IEEE Computer Society.

[49] J. Liu, J. Reich, and F. Zhao. Collaborative in-network processing for target tracking. *EURASIP J. Appl. Signal Process.*, 2003:378–391, January 2003.

[50] J. Liu, Y. Yuan, D. Nicol, R. Gray, C. Newport, D. Kotz, and L. Perrone. Simulation validation using direct execution of wireless ad-hoc routing protocols. In *The 18th workshop on Parallel and Distributed Simulation*, pages 7–16, New York NY, USA, may 2004. ACM.

[51] B. Logan. Mel frequency cepstral coefficients for music modeling. In *International Symposium on Music Information Retrieval*, 2000.

[52] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 183–196. ACM, 2009.

[53] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2009.

[54] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM, 2002.

[55] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 337–350. ACM, 2008.

[56] D. Minnen and T. Starner. Recognizing and discovering human actions from on-body sensor data. In *In Proc. of the IEEE International Conference on Multimedia and Expo*, pages 1545–1548, 2005.

[57] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336. ACM, 2008.

[58] V. Naoumov and T. Gross. Simulation of large ad hoc networks. In *The 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, pages 50–57, New York NY, USA, september 2003. ACM.

[59] K. Plarre and P. R. Kumar. Tracking objects with networked scattered directional sensors. *EURASIP J. Adv. Signal Process*, 2008:74:1–74:10, January 2008.

[60] D. Reynolds and R. Rose. Robust text-independent speaker identification using gaussian mixture speaker models. *Speech and Audio Processing, IEEE Transactions on*, 3(1):72 –83, 1995.

[61] C. Seeger, A. Buchmann, and K. Van Laerhoven. myhealthassistant: A phone-based body sensor network that captures the wearer's exercises throughout the day. In *The 6th International Conference on Body Area Networks*, Beijing, China, 11/2011 2011. ACM Press, ACM Press. Best Paper Award.

[62] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *The 2^{nd} International Conference on Embedded Networked Sensor Systems*, pages 188–200, New York NY, USA, november 2004. ACM.

[63] N. Shrivastava, R. M. U. Madhow, and S. Suri. Target tracking with binary proximity sensors: fundamental limits, minimal descriptions, and algorithms. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 251–264, New York, NY, USA, 2006. ACM.

[64] J. Singh, U. Madhow, R. Kumar, S. Suri, and R. Cagley. Tracking multiple targets using binary proximity sensors. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 529–538, New York, NY, USA, 2007. ACM.

[65] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *The 4^{th} International Symposium on Information Processing in Sensor Networks*, Piscataway NJ, USA, April 2005. IEEE Press.

[66] Y. Wang, M. Martonosi, and L.-S. Peh. Predicting link quality using supervised learning in wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 11(3):71–83, 2007.

[67] Z. Wang, E. Bulut, and B. Szymanski. Distributed target tracking with imperfect binary sensor networks. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1 –5, 30 2008-dec. 4 2008.

[68] Y. Wen, R. Wolski, and G. Moore. DiSenS: scalable distributed sensor network simulation. In *The 12^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 24–34, New York NY, USA, march 2007. ACM.

[69] G. Werner-Allen, S. Dawson-Haggerty, and M. Welsh. Lance: optimizing high-resolution signal collection in wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 169–182. ACM, 2008.

[70] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, second edition, 2005.

[71] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 13–24. ACM, 2004.

[72] S. Yi and H. Cha. An active tracking system using ieee 802.15.4-based ultrasonic sensor devices. In X. Zhou, O. Sokolsky, L. Yan, E.-S. Jung, Z. Shao, Y. Mu, D. Lee, D. Kim, Y.-S. Jeong, and C.-Z. Xu, editors, *Emerging Directions in Embedded and Ubiquitous Computing*, volume 4097 of *Lecture Notes in Computer Science*, pages 485–494. Springer Berlin / Heidelberg, 2006.

[73] G. V. Zàruba, M. Huber, F. A. Kamangar, and I. Chlamtac. Indoor location tracking using rssi readings from a single wi-fi access point. *Wirel. Netw.*, 13:221–235, April 2007.