# Framework for Body Sensor Networks

*Sameer Iyengar*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-154
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-154.html

November 5, 2009

# Framework for Body Sensor Networks

Sameer Iyengar

**Framework for Body Sensor Networks**
by Sameer Iyengar

**Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Professor Alberto Sangiovanni-Vincentelli
Research Advisor

Date

* * * * * *

Professor Ruzena Bajcsy
Second Reader

Date

**Acknowledgements**

It is impossible to be successful as a graduate student without a wide support network. I am extremely grateful for mine. I must thank my advisor, Alberto Sangiovanni-Vincentelli, for giving me the the freedom and inspiration to explore my interests and ideas, possibly the most valuable thing a researcher could ask for. In addition, to my mentors, Professor Ruzena Bajcsy and Professor Roozbeh Jafari, thank you for helping me focus my research and pushing me to always take that extra step.

To my colleagues here at Berkeley and in research groups at University of Texas at Dallas, Telecom Italia Research and Vanderbilt University, thank you for embodying the spirit of open and altruistic academic collaboration, through which we have all produced truly interesting results.

I have had the unique pleasure of working with a number of students and researchers from institutions around the world, notably: Philip Kuryloski, Filippo Tempia, Ville-Pekka Seppa, Po Yan and Victor Shia. Thank you for always helping even when you didn't need to, for being there to bounce ideas or just to kill time and for making research less like work and more like fun.

To my friends who have always kept me in good spirits, thank you for forcing me to take a break and for sharing the great times we've had.

And, most importantly, to my parents, it is impossible to enumerate in words all that you have given me, thank you for everything.

# Contents

# Chapter 1

# Introduction

Wireless Body Sensor Networks (also known as bodynets or Body Area Networks) have the potential to revolutionize healthcare. These networks are comprised of wearable devices with attached sensors that can measure various physiological and environmental signals. Bodynet devices communicate wirelessly with networked gateways (mobile phones, computers and PDAs) which store, analyze and communicate vital information in real-time. A bodynet can be designed to immediately alert emergency personnel to a critical situation like a heart attack or a debilitating fall. Bodynets can also help physicians catch warning signs of a disease earlier or remotely monitor the progress of a recovering surgery patient.

Body Sensor Networks free a patient from the confines of a laboratory but still enable monitoring by researchers and clinicians. Researchers can study illnesses in real world settings without intrusive video equipment. Clinicians can gain a broader view of a patient's health without increasing office visits.

### Technology

The technology behind Body Sensor Networks is not new. Wireless Sensor Networks (WSNs) are widely used in various applications such as industrial automation, environmental monitoring and military surveillance. However, recent advances in sensor technology and power consumption have made sensor networks feasible for use in wearable, real-time human monitoring. In addition, the push toward electronic medical records has created infrastructure that allows physicians to interact with electronic patient data.

A typical WSN consists of multiple nodes which communicate with a gateway or set of gateways. Each node contains a small processor, sensing devices and a wireless radio. Nodes are generally powered by batteries, but stationary nodes may also have a more reliable power source. Nodes have the ability to buffer data from sensors and perform computations using that data. Gateway devices generally have a more reliable power source, greater processing capability and can interact with outside networks.

A bodynet is a WSN in which the nodes are primarily mounted on the body of a subject, introducing an additional set of design constraints. On-body devices must be physically small and power-efficient to allow for wearability and patient mobility. Bodynet gateways may also be mobile devices such as a mobile phone or PDA. These characteristics limit the computation capability of devices in the network. Despite their limited processing power, bodynet applications generally require a high degree of reliability, creating numerous design challenges.

### Applications

Body Sensor Networks can be used in a variety of research and clinical applications. Inertial sensors are useful when performing gait and balance analysis, especially in research on diseases which cause mobility impairment such as Parkinson's disease and Muscular Dystrophy. Inertial sensors can also be used in assisted living homes for applications like fall detection and in hospitals for monitoring patient recovery and rehabilitation. Biological sensors can monitor patient vital signs to allow researchers to study complex biological functions like metabolism and emotion. Applications of Body Sensor Networks are discussed in more depth in Section 2.3.

### Research Challenges

Body Sensor Networks provide interesting research challenges at every level of their design. At the hardware level, advances are being made in physical size, power consumption and processing capability. Advancements in MEMS technology are making new types of sensors possible. The network level poses challenges for antenna designers as well as those designing scheduling algorithms and communication protocols. At the application level, developers are trying to implement advanced, distributed algorithms while ensuring that reliability and performance constraints are met.

This work focuses on defining abstractions that will allow researchers in each area to work at maximum efficiency. Currently, the bodynet application design process requires that application developers manually provision the network and create application-specific protocols. The goal of this work is to aid developers by defining a framework that abstracts network and hardware configuration out of the application development process. Such a framework also aids those working on other areas of the network stack. If applications are developed independently of a particular communication protocol or hardware configuration, new protocols and hardware can be deployed with little application redesign. Much like abstractions in place for personal computers, the framework proposed in this work will allow for widespread adoption of bodynets by making it easer for developers to deploy real-world applications.

Characteristics of bodynets and bodynet applications are discussed in Chapter 2. This outlines and motivates the design of the Body Network Service Manager (BNSM) framework presented in Chapter 3. An implementation of the framework design is discussed in Chapter 4. Three case studies characterizing the performance of the BNSM framework are presented in Chapter 5.

# Chapter 2

# Background: Body Sensor Networks

This section outlines characteristics of Body Sensor Networks and their applications. Important bodynet design considerations are highlighted throughout the section. These considerations lead to the design of a framework for developing bodynet applications presented in Chapter 3.

## 2.1 Hardware

Body Sensor Networks are useful in many applications because they are easy to customize. Bodynet hardware can be designed to minimize complexity and maximize generality by using readily available commercial off-the-shelf components rather than custom ASIC designs. The main components of the hardware are the on-body nodes and the gateway devices. Sensing devices may be built into the node itself or attached via input ports. An overview of how bodynet hardware components interact is shown in Figure 2.1.

### 2.1.1 Nodes

A sensor node, or mote, is an embedded device containing a microprocessor, local storage and a wireless radio. Many mote platforms exist, each using different processor and radio hardware,

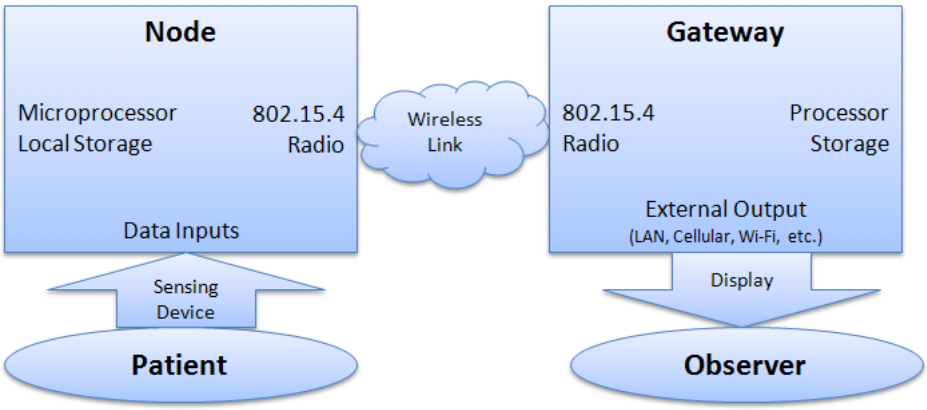


Figure 2.1: Structure of a Body Sensor Network

but device specifications and capabilities are generally quite similar [1, 2, 3]. Embedded operating systems such as TinyOS [4] and Contiki [5] have been developed for sensing devices which are unable to run, and do not require the functionality of, full-scale operating systems.

In this work, the nodes in use are the Moteiv Tmote Sky based on the TelosB platform [1]. The Tmote Sky has an 8MHz TI MSP430 microprocessor which has 10kB of RAM and 48kB of flash memory available. This microcontroller is widely used in WSN devices. The nodes communicate wirelessly using the Chipcon CC2420 radio. The CC2420 is a 250kbps, IEEE 802.15.4 compliant radio operating at 2.4GHz.

The motes run the TinyOS operating system [4]. TinyOS allows developers to use nesC, a variant of C, rather than a device specific language to program the nodes. NesC uses a non-blocking split-phase programming model. To a large extent, nesC abstracts away differences in processors and radio transceivers on different mote platforms by providing a library structure for platform-specific code. Developers define program logic and packet abstractions which compile correctly for the specified platform. However, TinyOS code requires considerable manual overhead and code repetition making applications error-prone and difficult to modify. Application developers will benefit from design tools that ease the task of device code modification.

### 2.1.2 Gateway

A gateway is any device that can interact with the nodes and contains either a human user-interface or an interface to a traditional network. In most networks, a gateway contains more processing power than the sensor nodes.

To be a part of the network, gateway devices require the ability to interface with an 802.15.4 transceiver. To validate generality, three different gateway devices have been tested through the course of this work:

- **PC:** Windows, Linux and Macintosh-based PCs
- **Mobile Device:** Motorola E680i mobile phone [6]
- **Embedded Computer:** Crossbow Stargate [7]

The PC and the Stargate board communicate with the rest of the network using a node attached via the USB port. The mobile phone uses the SD card interface to communicate with an attached Intel PSI [8] board containing a custom 802.15.4 radio.

As Body Sensor Networks become more pervasive, network designers will be unable to dictate the types of devices that are used as gateways. Application developers will benefit from specifying gateway code in a way that abstracts device specific code from the application logic. In addition, a cross-platform API for gateway code would allow applications to support many types of gateway devices.

### 2.1.3 Sensors

A sensor is any device that generates a signal based on its environment. Sensors are either built into the node or are connected via an input port. The Tmote Sky contains 16 input pins that allow hardware designers to attach sensing devices.

The majority of this work focuses on applications of MEMS inertial sensors. A 3-axis accelerometer, the ST Microelectronics LIS3LV02DQ [9], and a 2-axis gyroscope, the InvenSense IDG 300

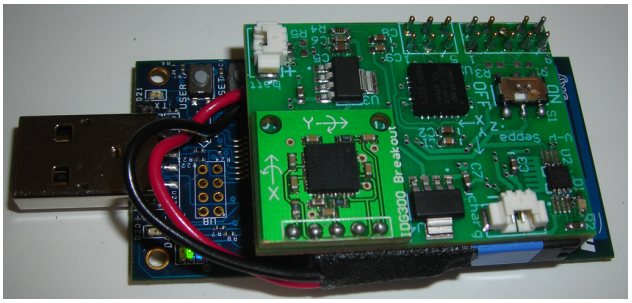

Figure 2.2: Inertial sensor board attached to Tmote Sky

[10], were integrated into a custom sensor board shown in Figure 2.2. Each component contains its own analog-to-digital converter allowing its output to directly connect to the digital I/O pins on the TmoteSky. The sensor board also contains a charging circuit and voltage regulators for the attached Lithium Ion battery.

To validate generality and study other applications, a bio-sensing board developed at the Tampere University of Technology was also tested [11]. This device uses electrodes and bioimpedance to measure vital signs such as ECG, respiration and temperature.

Code written for the node contains logic to interact with the sensors. TinyOS provides a way to abstract and define routines that developers use to query sensors in their applications. As new sensor technology is continually developed, these routines must be updated and modified. To best take advantage of the pace of hardware development, application developers must be able to easily add and update sensors attached to devices.

## 2.2 Communication

### 2.2.1 Interference

802.15.4 devices experience interference for a variety of reasons. The 2.4GHz band is widely used for other wireless sensor networks as well as devices like cordless phones and Wi-Fi networks [12]. Absorption of 2.4GHz waves by water and the human body, in addition to interference caused by motion of wearable sensors, also affect network performance [13]. To create reliable applications despite such interference, developers often construct application-specific protocols. However, if a network is updated or needs to interact with additional devices, the corresponding protocol must be altered. Application developers can avoid this issue by using a general protocol that is flexible enough to accommodate many different types of data while still accounting for interference. By writing code so that is is protocol independent, applications can take advantage of advances in protocol design.

### 2.2.2 MAC Protocols

Many MAC protocols have been proposed and tested for wireless sensor networks. TinyOS based devices use CSMA/CD by default. However, in wireless networks the hidden terminal problem [14] affects CSMA/CD transmission reliability. Nodes may be unable to realize when it is necessary to back off. In addition, physical constraints make it such that a node that can receive data reliably

may not be able to reliably transmit data via the same link.

The reliability requirements of bodynets favor protocols that use acknowledgments or time-division to guarantee delivery, but power constraints serve to limit the amount of control messages that can be sent to provision a protocol. TSMP offers a reliable solution that combines time division and frequency hopping [15]. TDMA is effective for small networks and is used in this work because of the added advantage of its simplicity to implement.

### 2.2.3 Traffic Patterns

WSN research is divided between two types of communication patterns: centralized networks and ad-hoc networks. Centralized networks are those in which a central coordinator controls the operation of the nodes in the network. Ad-hoc networks are often self-organizing and designed with the idea that node failure is commonplace. Due to wearability constraints, bodynets have fewer sensors than traditional WSNs and each sensor is designed for a specific purpose. This architecture lends itself well to centralized networks in which the gateway has control over the specific operation of nodes in the network and node failure is the exception rather than the norm. Centralized networks have traffic patterns that are primarily either one-way or two-way.

#### One-way communication

Networks that perform traditional data collection are characterized by traffic patterns in which much of the traffic flows from the nodes to a gateway. This traffic can occur at regularly scheduled intervals or irregularly when a node meets some condition to send data, such as a threshold for an alarm. Body Sensor Networks in which large amounts of data are collected to be later analyzed offline fit this communication model. Similarly, applications which trigger alarms when critical events are detected are dominated by one-way communication after the initial configuration to set up the alarm.

#### Two-way communication

Applications which perform real-time processing may have a traffic pattern in which data flows more evenly in both directions. Such applications include adaptive networks in which the gateway will request additional information from the sensors when a particular scenario is detected. In addition, some networks work entirely based on polling, requiring the gateway to individually request data when desired. Other networks will disseminate data to the nodes for use in processing. To allow for applications requiring this type of traffic pattern, application developers will benefit from being able to reconfigure the network and request additional data during runtime.

## 2.3 Applications

The requirements of possible applications of Body Sensor Networks range from simply collecting raw data from a single sensor to highly complex distributed processing algorithms involving many nodes. This section lists many of the bodynet applications and algorithms considered during the course of this work along with design principles gathered from each.

### 2.3.1 Research Applications

#### Gait Analysis and Motion Analysis

Considerable research has been done on human locomotion [16]. Current data collection techniques commonly employ either video technology or wired sensing. Video-based techniques include standard video recording systems, camera systems that employ reflective markers and infrared camera systems. In all of these cases, highly specialized laboratory equipment is required. Subject occlusion is a limiting issue, especially in experiments involving multiple subjects. In addition, subjects are constrained to a small area. Systems that use wired on-body sensors can limit movement and cause discomfort to the subject, affecting the accuracy of data collected [17].

Body Sensor Networks can replace existing motion analysis technology allowing subjects to move freely while still being studied. To effectively replace existing systems, a bodynet used for motion analysis research must support a large number of nodes and high data throughput. Hardware should be physically small to avoid impairing subject motion. However, as most of this research is conducted via controlled experiments, data does not need to be processed in real-time.

#### Mobility Impairing Disease

Diseases such as Muscular Dystrophy and Parkinson's Disease severely limit motion. Similar to Gait Analysis research, controlled laboratory experiments are used to study patient progression and symptoms. In addition, researchers use uncontrolled experiments to assess how the disease affects a patient's day-to-day activities and their social interactions. Using real-time monitoring to assess quality of life can assist physicians in tailoring treatment to benefit a patient both mentally and physically [18]. Assessing patient quality of life requires a bodynet that can detect a patient's location and environmental sensors that detect how a patient interacts with their environment.

### 2.3.2 Clinical Applications

#### Fall Detection and Prevention

Falls are a leading cause of injury among patients over the age of 65. In addition, nearly 45% of falls occur when the patient is outside the home [19]. This suggests that on-body sensors can help prevent twice as many falling incidents as home-based environmental sensors.

Real-time systems that can detect falls can also automatically alert emergency personnel. A common solution is a manual alert system that allows a patient to call for help if needed. This is inadequate because patients are often unable to asses the severity of their condition or, if the condition is too severe, are too debilitated to use the system. Automatic detection of falls and fall severity can solve this problem. In addition, detecting joint and muscle weakening via tracking mis-steps and almost-falls can alert caretakers that a patient is at risk of falling.

A fall detection system requires high-frequency sampling and low-latency response time. In addition, the system should require little maintenance and interaction on the user's behalf. To gain widespread adoption, fall detection systems must easily integrate with existing emergency alert systems.

#### Elderly Monitoring

Elderly monitoring goes a step beyond fall detection to detect other critical events and monitor general physical health. This will enable early detection of illness, prevent injuries and help ensure overall well-being.

Assisted-living homes will increasingly be equipped with environmental sensors including floor sensors and video cameras. A bodynet should be able to interface with environmental sensors in order to offload processing to more powerful devices, combine information from environmental and on-body sensors and activate only those sensors that are necessary given the current situation. For example, a video camera can be activated only when a critical event has been reported in order to visually verify that the system has correctly detected the event. This protects patient privacy and conserves system resources.

To create such monitoring systems, bodynets should be reconfigurable in real-time allowing nodes to easily be added or removed as necessary.

#### Rehabilitation

Intensive physical therapy has been shown to assist patients recovering from injury or surgery [20]. However, quality of rehabilitation is constrained by the time and resources of physicians and therapists. Examples include patients requiring stroke rehabilitation, physical rehabilitation after hip or knee surgeries, myocardial infarction rehabilitation, and traumatic brain injury rehabilitation [17]. Wearable, electronic rehabilitation systems can improve quality of patient care while reducing the time burden on therapists.

In order to be useful, rehabilitation applications must perform local processing to distill and send relevant information to the physician. For example, after knee surgery the most important information to a physician is the knee's range of motion and knowledge about how daily activity is affecting recovery. Rehabilitation devices should also be able to give real-time feedback to the patient. For example, a system could guide the user through a set of rehabilitation exercises and give feedback regarding their progress.

### 2.3.3 Algorithms

#### Centralized Classifiers

Action recognition is often performed using decision tree or kNN classifiers [21, 22]. Such classifiers collect data from all the nodes in the network and compare it to a threshold or training set stored on the gateway. To reduce the amount of data transmitted, features of the data can be sent rather than the raw signal itself. To implement a centralized classifier, the gateway should be able to easily request raw sensor signals or signal features at desired intervals.

#### Distributed Classifiers

A distributed motion classifier that uses both local and central processing to classify an action can increase accuracy and decrease resource usage [23]. Each node locally computes and transmits a sparse linear representation of an action to the gateway. The results from all of the nodes are then combined to choose a result. To implement such a distributed algorithm, the developer should be able to easily define the local processing. The protocol must also be general enough to transmit data vectors, such as a linear representation, in an efficient manner.

## 2.4 Interaction

To gain widespread adoption, Body Sensor Networks must be easy for users to interact with. Much like consumer electronics, patients will demand products that work without requiring extensive user intervention.

### 2.4.1 Deployment

Unlike other WSNs, bodynets will likely be configured by a physician rather than a technician or the network designer. This will allow the system to meet a patient's specific needs but require that configuration and deployment be a straightforward task.

This change has two implications for application creation and deployment. First, application developers must design systems such that adding, removing and updating hardware is simple. Second, during system deployment, an application should automatically discover nodes in the network and configure the system to operate accordingly.

### 2.4.2 Interfaces

#### User Interface

Platforms should allow for easy creation of user interfaces to allow applications to provide feedback to both the subject and the observer. Application developers will benefit from a consistent model of receiving network information. This allows user interfaces to be general, extensible and reusable.

#### Existing Infrastructure

To allow access to data, Body Sensor Networks must be able to easily connect to existing infrastructure. This includes both wired and wireless Internet, cellular networks and electronic medical records databases. Using existing infrastructure aids in bodynet adoption by lowering deployment cost and time. This will make it easier to deploy bodynet technology into existing care networks. For example, emergency notification systems can consider the both most beneficial person to notify and the preferred way to connect with that person.

#### Environmental Sensors and Actuators

As building are increasingly equipped with sensing technology, bodynets must interact with environmental networks. For example, a bodynet can interact with environmental sensors by detecting the presence of a floor sensor in a room and requesting additional data from it, if necessary. A bodynet should also be able to interact with actuators. For example, a bodynet may open a door or turn on a light for a disabled patient.

#### Hardware Progress

Application software should be as hardware agnostic as possible. It should be easy to integrate new sensors as they are developed. However, a framework should not be so general that it precludes application developers from using hardware to its full potential.

## 2.5 Characteristics

### 2.5.1 Communication

#### Reliability

Data reliability is especially important for real-time applications. As many applications will reduce data dimensionality via local processing, a lost packet will translate into a larger amount of lost data. If enabled by the application, the network should be able to detect and automatically re-request any missing packets.

#### Latency

For certain healthcare applications, the time of arrival of data is especially important. For example, a critical alarm is only useful if it arrives in a timely fashion. Certain types of data or data coming from certain nodes may have a particular latency constraint. An application should be able to specify such a constraint and the network coordinator should assign communication schedules and prioritize packets accordingly.

#### Frequency

Applications in which research is a primary goal require very high data sampling rates. However, these applications may not necessarily require the data to be sent immediately, especially if the goal is to process the data offline. An application should be able to individually specify data sampling rates for each sensor.

#### Topology

Networks will likely be small but heterogeneous. Nodes will be designed and placed for specific purposes but data from each will likely be able to reach with the gateway within a small number of hops. As a result, complex routing is less important than ensuring the network operates seamlessly with a variety of device hardware configurations.

### 2.5.2 Hardware

#### Power Management

Physical constraints limit chip and battery size which, in turn, limits the amount of processing power and memory the devices have. Local processing is advantageous because thousands of instructions can be executed for the same amount of power as sending a radio message [24]. As a result, distributed processing is advantageous, but only if it requires minimal communication overhead and reduces the amount of data to be communicated.

#### Extensibility

The wide possibilities of bodynet applications require devices to work with a variety of sensors. Development of nodes with new types of sensors should be straightforward and not impact the network structure and protocol. The sensors deployed will depend on the condition of the individual

patient being monitored leading to a heterogeneous set of device types. To accommodate this, bodynets must be easily extensible with minimal manual reconfiguration.

### 2.5.3 Processing

Given power constraints and communication frequency requirements, local processing that reduces data dimensionality is incredibly advantageous. To promote reusability, local processing functions should be defined in a modular manner. Processing functions should have access to local sensor data and allow the application to specify additional parameters at run time. Local processing is additionally advantageous because, in order to be useful, clinical bodynets will need to distill information rather than overloading a physician with data.

### 2.5.4 Local Storage

Many nodes will have local solid-state storage available for use. This storage can hold pre-programmed data or data sent by the network coordinator. For example, a distributed classifier would require each node to store a local training set. Nodes can also use this storage for sensor data. This data can be bulk transferred or used in computation.

## 2.6 Advantages of a Framework

It is not easy nor necessary for bodynet developers to consider all the characteristics outlined in Section 2.5 during the design of each application. These considerations can be embodied in a framework.

The current process of application development requires the developer to generate custom software based on the devices and manually provision the entire network stack. Many developers working in a particular sub-class of bodynet applications create and reuse their own custom framework code for their applications. For example, the SPINE framework was designed for those interested in performing feature computation on inertial sensors [25]. It is possible to improve on this idea by creating a framework that is suitable for the general class of bodynet applications.

A framework allows the developer to focus on the application logic and abstract network provisioning and local processing. It also provides a structure with which new devices can be deployed into a network with minimal reconfiguration effort. The Body Network Service Manager (BNSM) framework outlined in Section 3 provides an API for applications to make service requests without worrying about manually configuring the network. In addition, it provides a code generator to make it easy to change sensing and local processing on individual nodes.

# Chapter 3

# BNSM: An Application Development Framework

## 3.1 Background

Given the advantages of a Body Sensor Network framework outlined in Section 2.6, the BNSM (Body Network Service Manager) framework was developed to simplify the task of bodynet application development. It encapsulates much of the functionality that application developers require (and must often manually replicate) in their applications.

BNSM creates an abstraction layer, the Network Service Manager (NSM) between the application and the nodes. An overview of BNSM is shown in Figure 3.1. An application makes service requests to the NSM, which coordinates the the network and responds by generating events. Internally, the NSM sends commands to the nodes which coordinate communication as well as local sampling and processing.

## 3.2 Abstractions

### 3.2.1 Request/Event Model

The natural interaction pattern between an application and the network is one in which the application makes a request and processes the network's response upon arrival. The same request and event model is chosen for abstracting the internals of the network provisioning required to complete a task.

Section 3.3 outlines possible service requests that an application can make. To use these services, the application must simply specify the relevant request parameters to the NSM via the NSM API.

The NSM responds to service requests with events. An event-based model is used because a request may asynchronously trigger many events. as the response time of a particular service will vary based on requirements. Example events include: arrival of data, network status updates and notifications regarding errors. An application must define an event handler to process incoming events.

An error notification is an event signaling that something has affected the normal service runtime. For example, a service notification may be generated when the NSM cannot fulfill a request

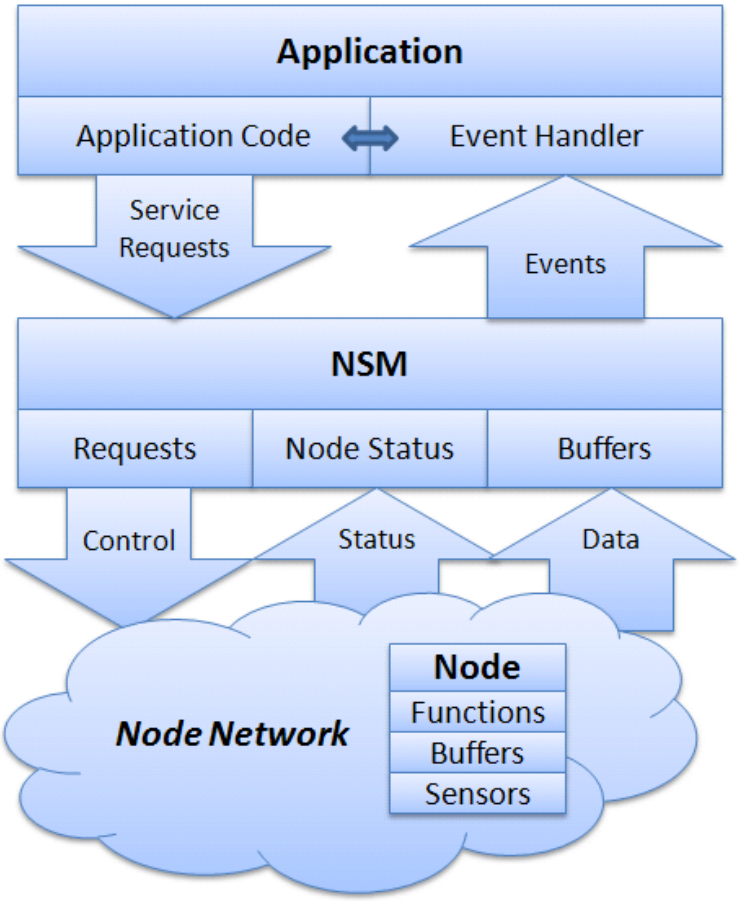


Figure 3.1: Body Network Service Manager Overview

due to node failure, network congestion or unsatisfiable latency constraints. Error notifications behave like exceptions, if not handled, they may cause the entire application to terminate.

### 3.2.2 Functions

Functions allow developers to define local processing in a structured manner. The flexibility to define all processing as a function avoids the need for special-case functionality to be included in the framework code. This promotes code reusability and makes the framework easier to maintain. A function may return an array of values or none at all.

Any local processing the developer desires can be defined as a function. There are three common uses of functions: data processing, communication control and data storage. The most common use of a function is to process data. The simplest data processing function just returns raw data. More complex functions perform additional computation using the data. Functions can also be used to generate alarms or specify logic to control communication. This is accomplished by returning a value only when a particular condition is satisfied. If a function returns no value, data will not be transmitted to the gateway. This allows developers to encapsulate communication logic in a processing function and not modify additional code dealing with packet assembly and transmission.

A function can also store and retrieve data from local storage. This is useful if a function needs to refer to existing data to generate a new result.

## 3.3 Application Services

This section outlines the services that the NSM makes available to the application. This includes the ability to discover nodes in the network, make sampling and processing requests, disseminate data and query network status.

### 3.3.1 Discovery

The discovery process detects which nodes are active in the network. This allows a single application to support multiple node configurations by dynamically adjusting to the set of nodes that are discovered. In addition, this prevents errors that arise from applications that assume certain nodes are active, as power, network and physical constraints may render some nodes unavailable.

During the discovery process each node announces its identifier and type. The NSM returns this information to the application. An application may also desire quality of service information, such as an indicator of link quality and battery life of the node. This information is useful for making adjustments to the network to account for battery failure and excessive network congestion.

BNSM uses a user-defined type system for nodes. During the hardware configuration process, API functions are generated for each node type. Given a node type, the application knows which sensors and processing functions are available on the node. Type-checking of application requests avoids errors that stem from attempting to activate non-existent sensors or processing functions.

Some WSN service discovery systems allow for a more general discovery process in which the node announces its capabilities, allowing new capabilities to be added to nodes and automatically be discovered by an application. As BSN applications generally focus on a specific purpose, it is reasonable to assume the application developer will know the types of sensing and processing to be performed in the network *a priori*. This reduces complexity as well as transmission overhead in the setup process.

Care must be taken to minimize packet collision during the discovery process. The NSM cannot assign a communication schedule to the nodes until after they have all been discovered, requiring a different communication protocol to be used during discovery. Particular implementation details of the discovery process are discussed further in Section 4.3.2.

### 3.3.2 Requests

Requests are used to activate local processing. To do so, a request must specify parameters and requirements for the desired functionality. Corresponding quality of service constraints for the request may also be specified, if required by the application. Once a request is made, the NSM assigns the request a unique identifier that both the application and the NSM use to reference the request in future interaction.

To allow applications to run multiple requests simultaneously, requests are made in groups. After a series of requests are submitted to the NSM, the application can activate the set of requests to begin processing. At this point the NSM will assign a communication schedule that will allow the nodes to respond to the request. Activating requests in groups also allows the NSM to wait

until all requests have been made before creating a communication schedule, avoiding unnecessary processing.

Once the network is in operation, requests may be added or removed in a similar fashion. The application indicates a series of requests to be removed and may submit additional requests to be activated. Upon activation, the NSM will assign an appropriate communication schedule.

#### Required Parameters

Local processing is abstracted by functions, which are described in Section 3.2.2. Functions allow the node to process or store data. A request must specify the local function to execute and any parameters to the function.

As most functions will process a vector of sensor data, the request must specify which sensors' data to send to the function. In addition, the request should specify the sensor sampling rate as well as a window size and window shift to select the correct segments of data. The window size is the number of samples to pass to the function. The window shift is the number of samples by which to shift the window for the next segment of data. For example, a window size of 10 and a shift of 10 would create data windows containing samples 0 to 9, 10 to 19, and so on. A window size of 10 and a shift of 5 would create data windows containing samples 0 to 9, 5 to 14, 10 to 19 and so on. The function will be executed whenever there is a window of data ready.

#### Optional Parameters

An application may specify a latency requirement which is used in the node scheduling process. This is especially useful for time-critical applications where a delayed response is equivalent to no response at all. Latency constraints may be expressed as a value range. The NSM will attempt to assign the ideal-case schedule, but has flexibility if that is not possible. As security of data is also important, a request may specify an encryption function to use for any data generated by that request.

### 3.3.3 Data Dissemination

Applications may require the ability to push data to the nodes. This data may be used to reprogram the nodes, be used as part of a computation, or simply be stored for access during future processing.

To send data, the application must specify the data to be pushed and the desired node destinations. The developer should also define a local processing function that will store or process the received data. This function will be executed once the data transfer is complete.

### 3.3.4 Status

At any given time the application may query the NSM for the status of a particular node or request. Node status includes information regarding active requests as well as quality of service information such as battery life and link quality. In the case of a request, the NSM will return whether or not a request is active as well as information regarding the communication schedule assigned to the request.

## 3.4 Nodes

At the most basic level, a node can perform four tasks: send or receive packets, sample sensors, store data and perform computation. Accordingly, the management of node tasks is controlled by four components: a Communication Manager, Sampling Manager, Buffer Manager and Function Manager.

### 3.4.1 Communication Manager

The Communication Manager (CM) receives and responds to commands from the NSM. Upon receiving a command, the CM signals the other local components to fulfill the request. Requests may generate values to be sent back to the gateway. These values accumulate in the Send Buffer. During the communication slot assigned by the NSM, the CM combines pending values in the Send Buffer into a Data Packet and sends the packet to the gateway.

#### Commands

There are 4 types of commands the CM must process:

**Discovery** The NSM issues a Node Discover command in order to find nodes in the network. The CM responds with a Node Announcement. The Node Announcement contains a unique identifier for the node and the node type.

**Function Activation** To activate processing, the NSM must relay the service request parameters supplied by the application. This includes the processing function, sensors, sampling rate, window size and window shift. This information is used to allocate buffers, initialize sensors and set up timers.

**Begin Processing** This command assigns the communication schedule and notifies the node to start querying the sensors. At this point data will accumulate in the Send Buffer. When it is the current node's turn to transmit, the CM will assemble and send a packet using data in the buffer.

**Receive Data** When the network is pushing data to the nodes, adequate buffer space must be allocated. When the transfer is complete, the correct processing function should be executed by the Function Manager.

**Missing Packet** In the event that that an expected packet is not received by the NSM, it will request that the packet be re-sent. The correct data will be retrieved from the buffer and will be re-transmitted.

### 3.4.2 Buffer Manager

The Buffer Manager allocates a set of circular buffers to store sensor data. It provides the functionality for other components to easily add and retrieve data while internally moving pointers according to the specified window size and window shift.

### 3.4.3 Sampling Manager

The Sampling Manager manages a set of timers to control when sensors are polled. When a sensor has data ready, it copies the values into the correct buffer. A request may require data from multiple sensors. Once each sensor has been sampled enough times to generate a window of data, the Sampling Manager will signal that the request is ready for computation by the Function Manager.

### 3.4.4 Function Manager

The Function Manager executes functions with the requested parameters. If the function returns data, the Function Manager adds header information about the corresponding request before storing the value into the Send Buffer.

## 3.5 Network Service Manager

### 3.5.1 State Variables

The NSM must maintain certain information:

**Requests** The NSM must keep track of all request parameters. In addition, the NSM stores scheduling information and constraints for each request so it can detect if constraints are not being met.

**Nodes** The NSM maintains information about each node in the network and continually updates their connectivity status to ensure that each node remains connected. The NSM may also maintain power information, if it can be provided by the nodes, to assist in determining the feasibility of requests.

### 3.5.2 Idle

When there are no pending requests, the network is considered idle. Nodes should periodically report their status to the NSM to ensure that connectivity is maintained. This makes it easier to carry out requests when they do arrive.

### 3.5.3 Requests

#### Node Connectivity

If a request requires a connection to a set of nodes, the NSM must ensure that all the nodes are active before proceeding. Presumably the connection to all the nodes has been kept alive during idle time. If there are nodes for which status information is unavailable, the NSM should send a beacon to the node as a last attempt to make a connection.

If a connection cannot be established, the application has three options: 1) cancel the service request, 2) request a replacement for the unavailable node, or 3) request data from the remaining nodes. The NSM must generate a service notification to determine the course of action. If the application chooses not to cancel the request, it can additionally specify what to do if the unavailable node rejoins the network during runtime. The application can choose whether to add the rejoined node or to continue without it.

#### Setup

The NSM is responsible for allocating transmission time slots to avoid packet collision and meet any specified latency requirements (see Section 3.5.7). During setup, the NSM may find that the request is invalid because the requested sensor on the node cannot return data. This may be because of incorrect addressing or because of a malfunctioning sensor. In this case, the NSM must generate an error notification and allow the application to decide how to proceed.

#### Runtime

The NSM buffers data as it arrives from the nodes. When all the data for a request has arrived, the NSM generates a data event. This event is then processed by the application's event handler.

The NSM is also responsible for detecting and reporting when the network is unable to meet service requirements. For example the network may be unable to meet a latency requirement because it may be too stringent given the amount of time it takes to execute the requested function with the specified parameters.

Ideally, the NSM would be able to determine infeasible request parameters before initiating a request. However, the nature of functions makes it difficult to estimate the running time that will occur for all input possibilities. As it is unreasonable to assume that the application will always make valid requests, the NSM should generate an error notification if a request cannot be fulfilled. The application has two options: either cancel the service request or attempt to continue with different parameters. To continue the request, parameters to be changed could include the number of requests, the latency constraints or the function used, among other possibilities.

### 3.5.4 Sending Data

#### Setup and Transfer

The connection setup process to send data is similar to that of a data request. A communication schedule must be generated, accounting for other active requests and each node must allocate sufficient buffer space. Given the limitation of packet length in wireless radios, the data will likely be fragmented into a number of packets. Considerable header information can be saved by sharing a transmission schedule, removing the need to transmit addressing and length information with every packet.

Packet errors are likely to occur when sending a large amount of data. Transfers should be accompanied by a checksum to verify that the data was transferred correctly in addition to the standard packet checks performed by the NSM. If the NSM is unable to confirm that the data was successfully received by a particular node, it may retry the transmission if the latency requirement allows. If not, it must generate an error notification. If the node is only able to receive part of the data due to communication or storage constraints, the NSM should generate an error notification indicating how much of the file was received. The application can decide to proceed, cancel or allow additional time for re-transmission.

#### Post-transfer

Once the transfer is complete, nodes signal that data has been successfully received and wait for confirmation before executing the specified local processing function. This is due to the fact that

all of the nodes may not have successfully received the data. In this case, the application may not want the processing function to be executed. Once the confirmation is received, the nodes proceed to execute the function and confirm its success.

**Runtime considerations**

If the request is canceled, the nodes should be signaled to discard any data. To account for the event that a node drops out during the data push process, the node should time out and assume that the connection to the network has been lost if there is an excessive delay during the data transfer or while waiting for confirmation to execute the post-transfer processing function.

Transmission cost can be saved by determining which packets are redundant among nodes. These packets can be broadcast or multi-cast to avoid sending multiple packets with the same information.

### 3.5.5 Service Termination

During service operation the application can request to terminate the service request. The NSM must stop transmission to and from the nodes. If nodes are in the process of receiving data, they should be instructed to discard the received information.

### 3.5.6 Node arrival/departure

If a node leaves the network, the NSM must trigger an error event in which the application can choose to request a replacement or cancel the service request. The application may also specify what to do in the event that the missing node rejoins the network. The two options are to either continue using the replacement node or switch back to the original node.

### 3.5.7 Assigning a Schedule

When a set of requests is activated, the NSM must generate a TDMA communication schedule. The schedule assigns a fixed number of consecutive slots to each node. This section formulates the schedule generation problem.

Given a set of nodes $N$, each node, $n \in N$ has a set of requests, $R_n$. A request, $r$, has the following properties: a sampling period, $t_r$, a window size $w_r$, a window shift $s_r$, an upper bound on the number of values generated by the request $v_r$ and an optional latency constraint $l_r$.

A communication slot has a fixed length of $p$ seconds[1]. A single packet may be sent during a communication slot and a packet has the capacity to send up to $c$ values.

Let $\sigma_n$ be the number of communication slots assigned to node $n$ and let $k$ be the number of communication slots assigned in a particular frame.

The value generation rate of request $r$ is:

$$V_r = \frac{v_r}{t_r s_r} \tag{3.1}$$

The value generate rate, $V_n$ (values/second) of node $n$ is the sum of the value generation rate of all requests to that node:

[1] In practice, communication slot length is measured on the order of milliseconds rather than seconds.

$$V_n = \sum_{r \in R_n} V_r \tag{3.2}$$

For a request to be feasible, the rate at which values are generated must be less than or equal to the rate at which values can be sent:

$$V_n \leq \frac{\sigma_n c}{kp} \tag{3.3}$$

Define the fraction of slots assigned to node $n$, $f_n$:

$$f_n = \frac{\sigma_n}{k} = V_n \frac{p}{c} \tag{3.4}$$

For a request to be feasible, the sum of all $f_n$ must be less than 1 or, equivalently, that the sum of each node's value generation rate must be less than the transmission capacity per second:

$$\sum_{n \in N} f_n \leq 1 \Longrightarrow \sum_{n \in N} V_n \leq \frac{c}{p} \tag{3.5}$$

As a result, given the value generation rate for each node it is possible to determine if a feasible schedule can be generated. However, this only determines whether there is enough capacity to send all the data. This capacity must be allocated in an efficient manner by deriving integers $\sigma_n$ and $k$ from $f_n$. These values must not be unreasonably large such that there is not enough storage space for a waiting node to buffer data while another node transmits data. In addition, the latency constraints of each request must be met.

If each node, $n$ has a maximum buffer size of $b_n$ values, a feasible schedule can be assigned, ignoring latency constraints, if there is enough buffer space for $n$ to store values while the other nodes are sending. The following condition must be satisfied for each $n \in N$:

$$kp \leq b_n \tag{3.6}$$

A slot assignment algorithm is shown in Algorithm 1. $k$ is chosen based on the capacity of the node with the smallest buffer capacity. Slots are assigned to a node by taking the ceiling of $f_n * k$. If the total number of slots assigned is still less than $k$, the assigned schedule is valid and fulfills buffering constraints.

**Input**: Set of nodes, $N$, set of node buffer sizes, $b$, slot length, $p$
**Output**: Number of slots, $\sigma$, to assign to each node or $false$ if no schedule possible
$k = \min_{n \in N} b_n$;
**foreach** $n \in N$ **do**
  $\sigma_n = \lceil f_n * k \rceil$;
**end**
**if** $\sum_{n \in N} \sigma_n \leq k$ **then**
  return $\sigma$;
**else**
  return $false$;
**end**

**Algorithm 1**: Slot assignment algorithm

Including latency constraints in schedule assignment is slightly more complex but follows a similar pattern. This method is effective but leaves room for optimization, as this algorithm may not be able to find all valid slot assignments.

### Function Return Values

While a function may return a variable number of values depending on the parameters, an upper bound on the number of values returned, $v_r$, must be determinable by the request parameters. This allows the NSM to provision enough communication bandwidth for the function data. One downside of this method is that enough communication bandwidth will be assigned to the function to handle the case when the function does transmit data, however, if the specified communication conditions are not satisfied, this bandwidth will go unused.

There are a few potential solutions to this: One solution involves having the developer specify a probability of condition satisfaction. Another solution has the node signal the NSM that the condition has been satisfied and that a new communication schedule should be allocated to account for the additional bandwidth. However these solutions introduce additional overhead and complexity with marginal benefit.

# Chapter 4

# BNSM: Implementation

## 4.1 Overview

The implementation of BNSM is split into two parts: the device code and the gateway code. The device level implementation uses TinyOS nesC code and a Java API is provided on the gateway for the creation of applications. Developers specify hardware configuration in a set configuration files and use a code generator to generate files that enumerate the necessary constants to access individual nodes, sensors and processing functions.

## 4.2 Configuration

Setup of the network is performed via a set of configuration files which specify the types of devices, sensors and local processing functions to be used. The configuration files are specified using YAML, a data serialization language [26], because of its readability and low overhead.

### 4.2.1 Device

The properties of each device in the network are specified in an individual configuration file. A sample configuration file for an inertial sensor is shown in Listing 4.1. The file has three parts:

**Name** A unique name for the device. This name is used as the node type by the NSM and the application.

**Sensors** A list of each sensor on the device. The configuration of a particular sensor is specified through an additional configuration file described in Section 4.2.2. The entry for each sensor should correspond to the sensor configuration file name. For example the configuration for the `voltage` sensor would be found in `voltage.yaml`.

**Functions** A list of the local processing functions to be pre-programmed on the device. The code generator will look for a TinyOS component with the same name. For example the entry for the `Mean` function will look for a TinyOS component `MeanC`. Function implementation is discussed further in Section 4.3.6.

Listing 4.1: Configuration for an inertial sensing device

```
name: inertial
sensors:
 - accelerometer_x
 - accelerometer_y
 - accelerometer_z
 - gyroscope_x
 - gyroscope_y
 - voltage
functions:
 - RawData
 - Mean
 - SVD
```

### 4.2.2 Sensor

Each individual sensor also contains its own configuration file. This information is not included in the device configuration file because multiple devices may contain the same sensor. Specifying the information in separate files reduces code duplication and promotes the ease of use of standard sensor components in device creation. The filename should correspond to the name listed in the device configuration file (see Section 4.2.1). An example configuration file for a battery voltage sensor is shown in Listing 4.2.

The configuration file has five parts:

**Component** The nesC file containing the device-level implementation of the code that polls the sensor.

**Instance** A name for the instance of the component. If two sensors belong to the same component and are given the same instance name, the code generator will avoid generating redundant code. For example, this is useful for a biaxial gyroscope which specifies a single instance of the gyroscope component for both the X- and Y-axis.

**Interface** The TinyOS interface through which to access the component. In order to avoid restricting sensors to use a particular interface (such as the `Read` interface) any interface is supported. This allows existing sensor code to be easily ported to BNSM without rewriting the interface through which sensor data is accessed.

**Initiate Read** A block of code that initiates the asynchronous reading of the sensor. This code may call methods specified in the sensor component by referencing the instance name.

**Read Done** A block of code that is called when the sensor reading is complete in order to perform any necessary post-processing of the sensor reading. When post-processing is complete, the code should specify the `@putvalue` directive with a pointer to the data and the number of data values as parameters. The code generator will replace the directive with a function call that puts the sensor value in the appropriate buffer. In this manner, sensor code is not tied to a particular buffer.

Listing 4.2: Configuration for a battery voltage sensor

```
tos_component: VoltageC
tos_instance:  Voltage
tos_interface: Read<uint16_t>

initiate: |
 call Voltage.read();

done: |
 event void Voltage.readDone(error_t result,
                             uint16_t data) {
    if (result == SUCCESS) {
      @putvalue(&data, 1);
    }
 }
```

### 4.2.3 Code Generator

The code generator uses the configuration files to generate device specific Java and TinyOS code. This prevents errors that may arise if the developer had to manually sync the Java and TinyOS code each time new capabilities are added to the network. The code generator is executed with the filenames of the configuration file for each device to be included in the network. It generates a global set of TinyOS constants described in 4.3.1. For each device, a specific Buffer Manager, Function Manager, Sampling Manager and header file is generated. In addition, the code generator creates a Java class for each node type that enumerates device-specific constants (see Section 4.5.4).

## 4.3 TinyOS

The TinyOS implementation is broken into two parts, the specific code for each type of device and the generic codebase that implements the protocol and allows developers to define local processing functions. An overview of how the various TinyOS components connect is shown in Figure 4.1

### 4.3.1 Constants

Three sets of constants are generated: Node Constants, Sensor Constants and Function Constants. Each enumerate unique identifiers for the types of nodes, sensors and functions, respectively, that are found in the system. Corresponding constants are also generated for the Java API and are described in Section 4.5.4.

### 4.3.2 Communication Manager

The Communication Manager (CM) is a static component. It includes a generated header file that defines the sensor type as one of the types specified by the Node Constants described in Section

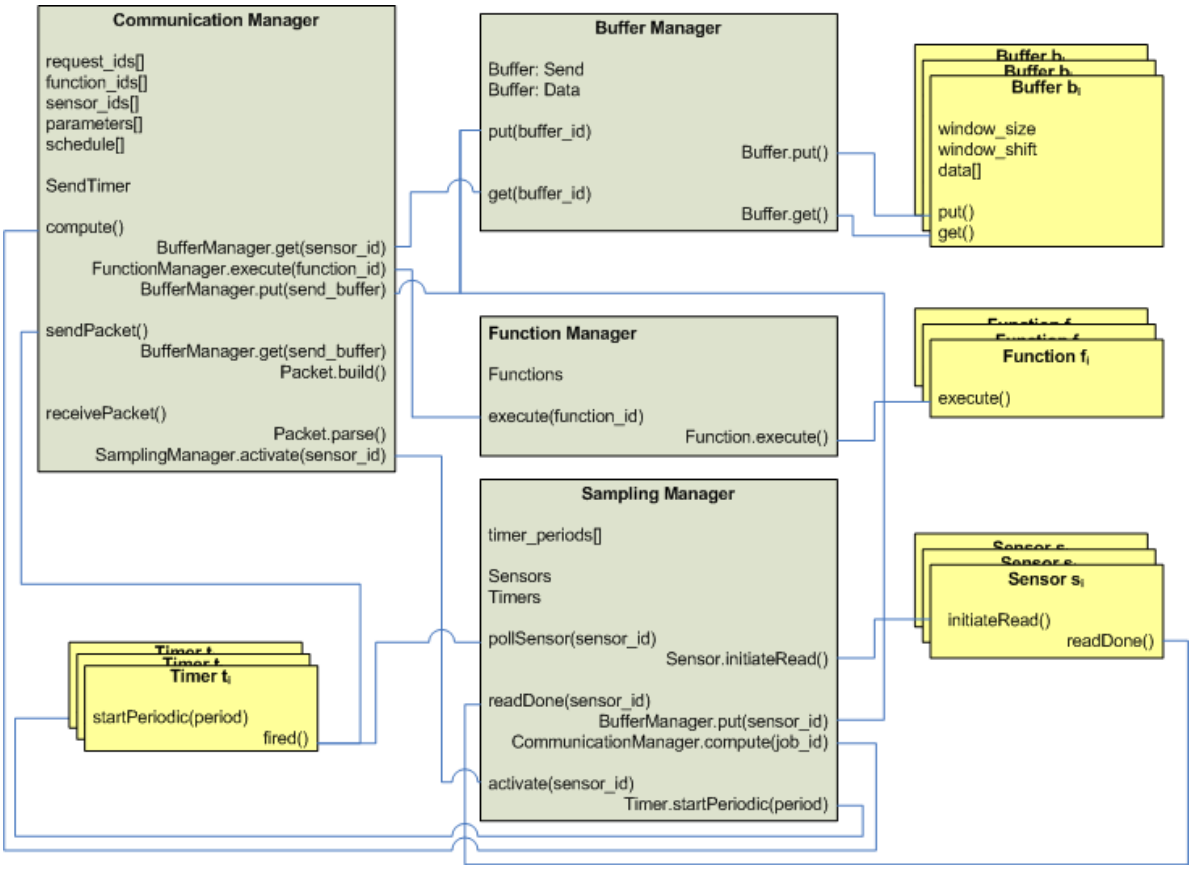


Figure 4.1: BNSM TinyOS component structure

4.3.1. The CM coordinates the other components, calling the appropriate functions in the Buffer Manager, Function Manager and Sampling Manager when required.

Internally, a request is called a job and is assigned an identifier which corresponds to the order in which the request was received. Data structures in the CM map the job identifier to the corresponding Request identifier and Function identifier.

#### Discovery

When the CM receives a Node Discover packet, it waits for a period proportional to the Node identifier before responding with a Node Announce Packet. This period defaults to 5 milliseconds and is specified in the Node Constants file. As all the nodes in the system receive the Node Discover packet at approximately the same time, if each responded immediately, considerable packet collision would occur.

### 4.3.3 Buffer Manager

The Buffer Manager is automatically generated. As TinyOS does not support dynamic memory allocation, a circular buffer is allocated for each sensor. The size of the buffer is specified in the generated header file for the device and defaults to 100 samples per buffer.

### 4.3.4 Sampling Manager

The Sampling Manager (SM) is generated from the sensor configuration files specified in Section 4.2.2. Enough timers are allocated such that each sensor may be sampled at a different frequency. However, sensors are assigned to a timer based on the sampling period. When a timer fires, it calls the function to initiate processing for each of the sensors assigned to it. This minimizes interrupt overhead as opposed to if the system were to activate multiple timers at the same frequency.

The Sampling Manager maintains the window size and window shift for each job. When enough data has been sampled to allow the corresponding function to process the data, the SM signals the CM to activate processing.

### 4.3.5 Function Manager

The Function Manager (FM) is a generated component that links in and directly calls the functions required by the device. This avoids requiring local processing functions to be included from any other component. To allow functions to be called, the Function Manager defines the `execute_function` command. Given the identifier of a function to execute, the Function Manager passes the necessary parameters to the function. This is used by the Communication Manager to activate local processing.

### 4.3.6 Functions

To implement a function, the developer must create a component that provides the `Function` interface. The `Function` interface requires the developer to specify the `execute` function. This function takes four parameters:

**data_pointers** A handle that points to an array of pointers, each pointing to an array of sensor data.

**num_elements** The number of elements in each array of sensor data. Each sensor will supply the same number of samples to the function

**parameters** A pointer to the array of parameters passed to the function.

**return_buffer** A pointer to the buffer in which the function should store its return values.

To implement a function, the developer manipulates the data, stores any number of values in `return_buffer` and then returns the number of values that were stored. The implementation of a function to calculate the mean value of a data buffer is shown in Listing 4.3

## 4.4 Packet Protocol

The BNSM packet protocol outlines the format of packets sent between a node's Communication Manager and the Network Service Manager.

Listing 4.3: Function to calculate mean value of a data buffer

```
module MeanC {
  provides interface Function;
}

implementation {
        /*
        * Calculates the Mean of the given data array
        *
        * @param data_pointers: Pointer to an array of data pointers
        * @param num_elements: Number of elements
        * @param parameters: Additional parameters to the function
        * @param return_buffer: Location to copy return value
        *
        * @return 'int16_t': 1 (number of values returned)
        */
        command int16_t Function.execute(
             int16_t** data_pointers, uint16_t num_elements,
             int16_t*  parameters, int16_t* return_buffer) {
          int16_t* data = data_pointers[0];

          uint16_t i;
          int32_t sum = 0;

          for (i = 0; i < num_elements; i++) {
            sum += data[i];
          }

          return_buffer[0] = sum / num_elements;
          return 1;
        }

}
```

| 8 | 16 | 24 | 32 |
|---|---|---|---|
| `Protocol Version` | `Packet Type` | `Source` | `Destination` | `Packet Number` |

| | | |
|---|---|---|
| `Protocol Version` | 4 bits | To accommodate for future versions of the protocol. |
| `Packet Type` | 4 bits | Indicates what kind of packet follows. |
| `Source` | 8 bits | ID of the source node. |
| `Destination` | 8 bits | ID of the destination node. |
| `Packet Number` | 8 bits | Current packet number. |

Table 4.1: Packet Header

### 4.4.1 Header

Each packet includes common header information, described in Table 4.1. Including the version number allows for revisions to this protocol to be implemented gracefully. The current version number is 1. The packet type specifies what kind of payload follows the header. Constants for packet types are specified in the `PacketConstants` header file in TinyOS and the `PacketConstants` class in Java. The source and destination are the identifiers of the sending and receiving nodes respectively. The destination may also be a reserved broadcast address, indicating that the packet is meant to be received by all nodes. Though the standard TinyOS header also specifies source and destination information, it is included in the BNSM header to make it possible to implement additional routing at the application level, if necessary. The packet number can be used to detect when a packet is dropped. As it is only 8 bits, and a node is likely to send more than 256 during the course of its runtime, this cannot be used as a unique packet identifier.

### 4.4.2 Node Discover

The Node Discover packet has a single 8-bit field that enables the NSM to send a command to the nodes. There are two possible commands: discovery and reset. The discovery command requests that nodes respond with a Node Announce Packet (see Section 4.4.3). The reset command cancels all active requests and resets the buffers.

### 4.4.3 Node Announce

The Node Announce packet shares the same format as the Node Discover packet. It allows a node to specify its type via a single 8-bit field. The node identifier is automatically specified by the header.

### 4.4.4 Function Activate

The Function Activate packet, described in Table 4.2, specifies information necessary to activate a local processing function and corresponding sensing, if any. The request identifier, function identifier, sampling period, window size and window shift are specified in a straightforward manner. After this, the number of sensors, $N$, to be associated with the function is specified. The first $N$ parameters are then assumed to be sensor identifiers, each to be polled at the sampling period specified. The rest of the parameters are treated as data parameters. If a function requires additional parameters, it can specify the number of parameters to follow in an additional packet.

| 8 | 16 | 24 | 32 |
|---|---|---|---|
| Request Id | Function Id | Sampling Period | Window |
| Shift | Num Sensors | Param 1 | Param 2 |
| Param 3 | Param 4 | Param 5 | Additional Params |

| | | |
|---|---|---|
| Request ID | 8 bits | The ID of the function activation request. |
| Function ID | 8 bits | The ID of the function to activate. |
| Sampling Period | 8 bits | Interval in milliseconds that each sensor should be polled. |
| Window Size | 8 bits | Number of samples to use as an input array for the function. |
| Window Shift | 8 bits | Number of samples to offset when generating the next window. |
| Num Sensors | 8 bits | Number of sensors included in the parameter list. |
| Param 1...5 | 8 bits each | If the parameter refers to a sensor, Sensor ID. Otherwise, the parameter value. |
| Additional Params | 8 bits | If 5 parameters are not sufficient, the number of parameters that will follow in another packet. |

Table 4.2: Function Activate Packet Format

### 4.4.5 Activate Processing

The Activate Processing packet, described in Table 4.3, assigns a communication schedule and signals that a node should begin processing all pending Function Activate requests. The first field specifies the total number of nodes. The following fields specify the number of slots assigned to each node ordered by node identifier. If only the number of nodes is specified, it is assumed that a single slot is assigned to each node.

### 4.4.6 Data

A data packet combines up to five Data Values into a single packet and sends them. The format of a Data Value is shown in Table 4.4. The first 16 bits are used for information about the data

| 8 | 16 | 24 | 32 |
|---|---|---|---|
| Num Nodes | Slots 1 | Slots 2 | Slots 3 |
| Slots 4 | Slots 5 | Slots 6 | Slots 7 |
| Slots 8 | Slots 9 | Slots 10 | Slots 11 |
| Slots 12 | Slots 13 | Slots 14 | Additional Nodes |

| | | |
|---|---|---|
| Num Nodes | 8 bits | Number of nodes in the network. |
| Slots 1...14 | 8 bits each | Indicates the number of communication slots assigned to the corresponding node. |
| Additional Nodes | 8 bits | If the network contains more than 14 nodes, the number of nodes for which slot information will follow in another packet. |

Table 4.3: Activate Processing Packet Format

| 8 | | 16 | 32 |
|---|---|---|---|
| `Request Id` | `isFirst` | `Fragment` | `Data Value` |

| | | |
|---|---|---|
| `Request Id` | 8 bits | The ID of the request this Data Value corresponds to. |
| `isFirst` | 1 bit | Boolean indicating whether or not this is the first fragment pertaining to the request. |
| `Fragment` | 7 bits | For the first fragment, this section indicates the total number of fragments. For all subsequent fragments, this contains the fragment number. |
| `Data Value` | 16 bits | Contains the actual data that was requested. |

Table 4.4: Data Value Format

and the last 16 bits contain the actual data. The corresponding request identifier is specified with each Data Value. Since a function can generate more than one value, the first Data Value contains the total number of values to follow for the specified request. Subsequent Data Values specify their fragment number. This allows the NSM to verify that the information for an entire request has been received.

## 4.5 Java

The Java API provides methods for developers to create applications that interact with the network.

### 4.5.1 Elements

**Node** A `Node` object encapsulates a node's identifier and its type. It also includes methods to request quality of service information, if available.

**BaseStation** The `Basestation` object is created by the application to specify connection parameters for the Network Manager (see Section 4.5.2) to communicate with the 802.15.4 radio attached to the gateway.

**Request** A `Request` object encapsulates the parameters of a request outlined in Section 3.3.2. The Request class generates a unique request identifier for each request created.

**Event** The `Event` object returns data associated with a particular request. It references the corresponding request identifier.

### 4.5.2 Network Manager

The Network Manager is the core of the gateway code. The application can use the following Network Manager methods to interact with the network.

**discoverNodes** This method allows the application to discover the nodes in the network. The Network Manager sends a Node Discover packet and then waits for a specified amount of time for all the nodes to respond. This delay defaults to 2 seconds but can be modified in the `Constants` class.

**activateRequest** An application constructs a `Request` object and then passes it to this function. The Network Manager sends a Function Activate packet to the corresponding node and stores the request information locally to use when validating the schedule.

**beginProcessing** When this method is called, the NSM to validates and generates a communication schedule and activates the nodes via an Activate Processing packet.

### 4.5.3 Application Interface

An application must implement the `BNSMApplication` interface which allows the Network Manager to communicate with the application.

**discoveryComplete** This method gives the application a map of node identifiers to their corresponding `Node` objects. The application can use this information to activate requests.

**eventReceived** When the NSM has received all fragments of a particular request, it creates an `Event` object to return to the application.

### 4.5.4 Constants

Each node type has a corresponding generated class that encapsulates the local processing functions and sensors that are possible to activate on that node. These constants are specified through a `Sensor` enumeration and a `Function` enumeration. By using these enumerations when constructing `Request` objects, a developer can avoid attempting to activate invalid sensors on a particular node. In addition, the generated `Nodes` class encapsulates constants that refer to each available node type.

# Chapter 5

# Case Studies

## 5.1 Event Detection and Reconfiguration

### 5.1.1 Overview

Bodynet resources can be conserved by activating processing only when necessary, such as when a particular event is detected. This case study considers a fall detection application. Threshold based fall detection algorithms are prone to false alarms from daily activities [27]. Once a patient has fallen, it is useful to analyze additional signals, such as vital sign data, to detect the severity of the fall. However, constantly collecting vital sign data is likely unnecessary and wastes resources. The BNSM function abstraction provides the ability to construct systems that can be quickly reconfigured based on detected events.

A simple fall detector was constructed by polling the Z-axis of the accelerometer at 20 Hertz. If the maximum value over a 20-sample window exceeds 2g, an event is sent to the gateway indicating that a second sensor should be activated to return additional information about the situation.

Listing 5.1 shows how a local processing function can return a value only if it exceeds a certain threshold. In this example, the threshold value is hard coded at 2000 (to represent 2g) but a parameter could be used to allow the threshold to be dynamically specified.

Listing 5.1: Function that returns a value only if it exceeds a threshold

```
command int16_t Function.execute(
    int16_t** data_pointers, uint16_t num_elements,
    int16_t*  parameters, int16_t* return_buffer) {
  int16_t threshold = 2000;

  call Max.calculate(data_pointers, num_elements, return_buffer);

  if (returnValues[0] > threshold) {
    return 1;
  } else {
    return 0;
  }
}
```

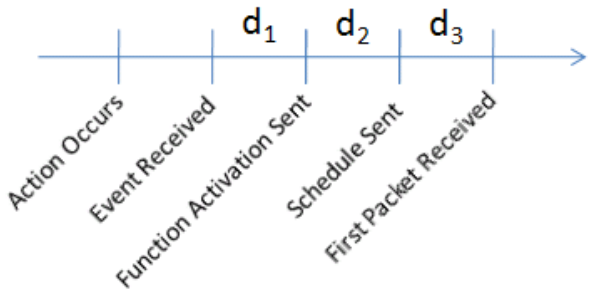


Figure 5.1: Event detection and reconfiguration timeline

| | Mean | Std. Dev. |
|---|---|---|
| $d_1$ | 13.5 | 5.5 |
| $d_2$ | 15.6 | 5.9 |
| $d_3$ | 117.8 | 13.1 |
| $d_1 + d_2 + d_3$ | 146.9 | 9.9 |

Table 5.1: Network reconfiguration latency (see Figure 5.1)

The Max function is called to calculate the maximum value over the data window and this value is copied into `return_buffer`. The Function Manager uses the return value of the function to indicate the number of values that the Communication Manager should read from the buffer. If the value exceeds the threshold, the Function Manager will return 1 signaling that the function has generated a value and that Communication Manager should add the value to the buffer of values to be sent. However, if the threshold is not met, the Function Manager will return 0 and the Communication Manager will know that the function did not generate any values that need to be sent to the gateway.

The chain of events is shown in Figure 5.1. $d_1$, $d_2$ and $d_3$ represent the duration between the start of each event in the timeline. The Java application waits for an event from the fall detection request. Once the request is received, the application creates a request for data from a second node. The request is then activated in order to generate a new schedule. Once the nodes receive the new schedule, data from the second node begins to arrive at the gateway.

### 5.1.2 Latency

Table 5.1 outlines the amount of time elapsed for each action. The mean total time to detect an event, reconfigure the network and begin receiving data is 146.9 milliseconds. There are two degrees of variability: schedule generation and sensor initialization time. In this case, schedule generation is very simple as there are only two nodes at the network each sending data a relatively low data rate with no latency constraints. A sensor may take some amount of time before its values stabilize, the latency for receiving data shows when the first packet will arrive at the gateway but the time to receive useful values will vary by sensor.

### 5.1.3 Comparison

Other query processing systems, such as TinyDB, offer the capability to reconfigure networks in real-time with similar performance [28]. However, the main advantage in BNSM is the ease with which custom processing can be defined. TinyDB allows developers to define "aggregates", similar to functions in BNSM, which perform computation on a series of data. However, the process of defining new aggregates is far more cumbersome [29]. The BNSM function interface allows local processing functions to perform complex computation and return multiple values by defining a TinyOS component and adding it to the appropriate configuration file (see Section 4.3.6). TinyDB requires the developer to modify multiple parts of the TinyDB framework code. In addition, the developer must create a custom reader on the gateway to parse any data from any aggregate that returns more than one value.

## 5.2 Real-Time Classification

Many classification algorithms have been developed to perform real-time action recognition [30, 22, 31]. Some have been implemented only in simulation, others have been implemented in wired networks and others have been tested in actual bodynets. This case-study investigates the comparison of simulated and real-time performance of a simple motion classifier. The goal is to show how BNSM simplifies real-time implementation, allowing algorithm developers to verify that their algorithms are feasible for real-time performance. In a real-time classifier, it is harder to segment data and ensure consistency of inputs. This example shows that classification accuracy can be severely impacted by inaccurate segmentation. In addition, processing should produce a result without excessive delay, as inputs are continuously entering the system. This example shows that, given a correct segmentation algorithm, a simple classifier can quickly produce accurate classification results.

### 5.2.1 Data Collection

A set of actions were designed to mimic dumbbell exercises. The actions are: 1) Curl Up, 2) Curl Down, 3) Lateral Raise Up, 4) Lateral Raise Down, 5) Front Raise Up, 6) Front Raise Down as well as four static positions: 7) Side, 8) Raised Lateral, 9) Raised Front and 10) Curled Up. Fifty trials of each of each action were performed with the accelerometer node placed in the palm of the hand and held like a dumbbell. Data was sampled from accelerometer X-, Y- and Z-axes at 40 Hertz.

A data collection application was built using BNSM. It provides a GUI interface to start and stop trials and save data to a file. The core of the application are the three requests specified to activate the triaxial accelerometer. For example, the request for the accelerometer X-axis is shown in Listing 5.2.

### 5.2.2 Fixed Window

#### Simulation

The average number of samples collected in each trial ranged from 81 to 169, with 122.8 being the average. Given this information, the first 120 samples (3 seconds) of data are used to represent the morphology of the signal. For trials that are shorter, the last sample is repeated to create 120

Listing 5.2: Request for raw data from X-axis of accelerometer

```
int window = 1;
int shift = 1;
int samplingPeriod = 25; //40 Hz
Request x = new Request(node,
        AccelNode.Function.RAW_DATA.intValue(),
        AccelNode.Sensor.ACCEL_X.intValue(),
        samplingPeriod, window, shift);
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 732.57 | 1538.1 | 2545.8 | 1722.6 | 2586.1 | 1756.1 | 2838.5 | 2716.6 | 2542.6 | 2820.3 |
| 2 | 1468.7 | 539.77 | 1610.9 | 2521.6 | 1649.9 | 2627.9 | 2948.3 | 2655.5 | 2492.2 | 2881 |
| 3 | 2456.4 | 1565.7 | 358.96 | 2812.9 | 819.65 | 3006.5 | 2580.1 | 3198.9 | 3034.4 | 2477.3 |
| 4 | 1599.8 | 2502.5 | 2826.4 | 419.08 | 3012 | 793.44 | 2158.7 | 3434.1 | 3251.1 | 2083.4 |
| 5 | 2514.6 | 1634.3 | 845.16 | 3007.3 | 421.71 | 3068 | 2775.9 | 2910.4 | 2729 | 2893.9 |
| 6 | 1628.5 | 2593.6 | 3005.7 | 776.24 | 3057.8 | 367.65 | 2288.9 | 3243.4 | 3040.8 | 2466.1 |
| 7 | 2725.7 | 2897.8 | 2564.1 | 2124 | 2752.4 | 2260.4 | 63.517 | 4515.2 | 4306.8 | 983.36 |
| 8 | 2626.7 | 2619.1 | 3197.1 | 3428.5 | 2896.4 | 3243.7 | 4525.3 | 137.57 | 336.79 | 4635.8 |
| 9 | 2408.3 | 2416.1 | 2998.5 | 3209.7 | 2685.9 | 3011.2 | 4291.2 | 308.48 | 108.54 | 4425.5 |
| 10 | 2698.2 | 2823 | 2453.7 | 2040.6 | 2864.6 | 2432.1 | 968.23 | 4616 | 4437 | 52.513 |

Table 5.2: Mean distance of trial to the center of each class

samples. This is representative of if the person had held the movement for the remaining samples. If the trial is longer, the remaining samples are simply not used.

Each trial is split into 12 windows of 10 samples each. A window is characterized by the mean value of the data in the window. The center of each action class is represented by a 12-point morphology vector, created by using the mean value of all trials.

For each class, the Euclidean distance of each trial to the center of every class was calculated to yield the distance matrix shown in Table 5.2. The mean distance from a trial to the center of its own class is 320.2. The mean distance from a trial to all other classes is 2337.4. The mean distance from a trial to the next closest class is 883.8. This suggests that the using Euclidean distance is a reasonable method to classify an action assuming the start and end of the trial are known.

The classification algorithm calculates the distance from a trial to each class and selects the class with the minimum distance. If this distance is greater than 1000, the trial is rejected as unable to classify. To simulate the performance of the classifier in a real-time environment, where the start and end of an action is not known, trials are created where the data was offset. To create a trial with a positive offset $t$, the first $t$ of 12 morphology points are discarded and the last point is replicated $t$ times and added to the end of the vector. This simulates beginning the classification on a data frame that begins $t$ windows after the action start. A negative offset is constructed in a similar fashion: points from the end of the vector are discarded and points at the beginning of the vector are replicated. The results for classification accuracy for various offsets is shown in Figure 5.2. From this, we can see that classification accuracy drops sharply for an offset of more than 2 in either direction. This makes sense because the morphology is designed to represent the

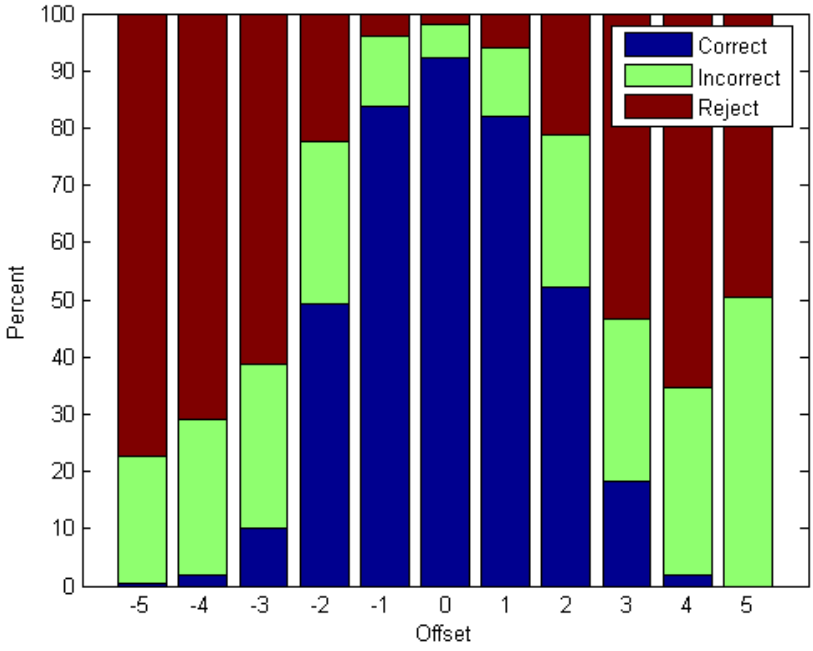


Figure 5.2: Simulated Classification Accuracy

shape of the signal from start to finish.

### Real-Time

The real-time classifier was made by modifying the data collector, as shown in Listing 5.3. The new request reflects the window size of 10 samples and the processing function that calculates the mean value of the window.

Listing 5.3: Request for mean value from X-axis of accelerometer

```
int window = 10;
int shift = 10;
int samplingPeriod = 25; //40 Hz
Request x = new Request(node,
        AccelNode.Function.MEAN.intValue(),
        AccelNode.Sensor.ACCEL_X.intValue(),
        samplingPeriod, window, shift);
```

The classifier GUI allows the user to indicate the starting and ending points of the movement. The real-time results are shown in Figure 5.3. Determining the end of a movement via user input introduces a non-negligible source of error. Accuracy does not drop as sharply for an offset of 1 as it does in the simulated graph, possibly caused by a time lag of the user indicating that the

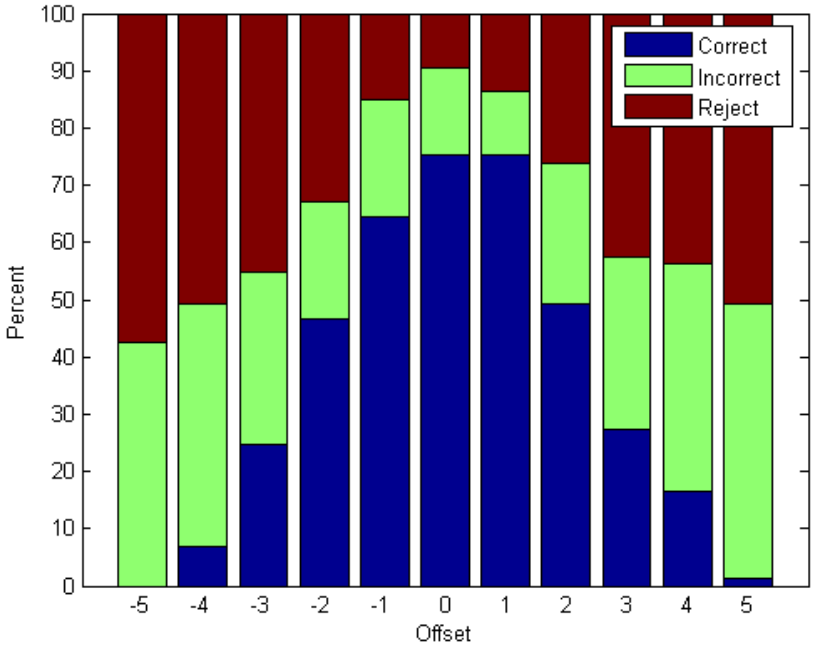


Figure 5.3: Real-Time Classification Accuracy

movement is finished. A good segmentation algorithm may be able to overcome this, but this graph shows that a poor segmentation algorithm will greatly affect classification accuracy.

**Analysis**

In general, the real-time classifier is slightly less accurate than the simulated results. Table 5.3 compares the simulated and real-time accuracy for the most accurate offset of each movement. For some movements, the most accurate offset is consistently one window prior or later to the end of the movement indicated by the user. This confirms human error introduced by having the user specify the movement duration. These classification results show that accurate motion classification can be achieved with a relatively simple algorithm given accurate action segmentation. Implementing the classifier in real-time also shows that simulation-only classification results can be misleading as they do not necessarily consider all factors involved in real-time implementation.

## 5.3 Deployment

Integrating and deploying Body Sensor Networks is a largely unexplored area as many applications are still in their infancy. This section outlines a case study in integrating a bodynet with a Wi-Fi based routing network for deployment in the home.

The CareNet system (see Figure 5.4) is a two-tier routing system developed by Vanderbilt and UC Berkeley [32]. It is designed to easily deploy Body Sensor Networks in a home or building. Each instrumented room contains a Stargate board with an attached 802.15.4 receiver and an optional video camera. The Stargate boards communicate with the central home computer using an ad-hoc

| Action | Simulated | Real Time |
|---|---|---|
| Curl Up | 78 | 73.3 |
| Curl Down | 90 | 78.6 |
| Lateral Raise Up | 98 | 90 |
| Lateral Raise Down | 94 | 100 |
| Front Raise Up | 94 | 83.3 |
| Front Raise Down | 100 | 100 |
| Side | 100 | 100 |
| Raised Lateral | 100 | 100 |
| Raised Front | 96 | 100 |
| Curled Up | 100 | 100 |

Table 5.3: Simulated vs. Real-Time Accuracy

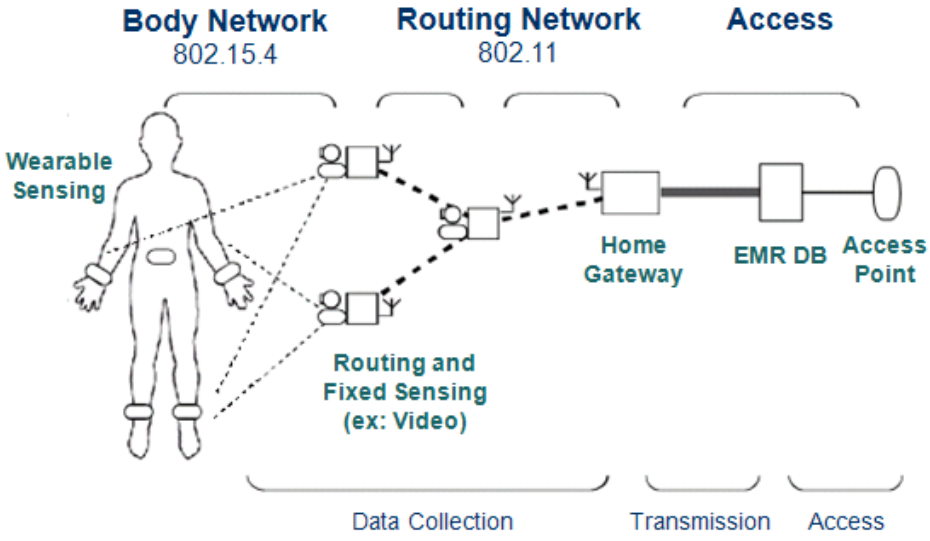


Figure 5.4: CareNet Architecture

802.11 network. During operation, when an individual is near a particular Stargate, the CareNet system activates the attached video camera. The system timestamps and sends the packets from the individual bodynet as well as corresponding video data to the central computer. An individual is free to move around the house and data will arrive the central PC as if the communication happened directly. The sensor and video data collected can be played back using a viewer shown in Figure 5.5.

This system was deployed in the homes of four elderly patients to collect data about their daily activities. The subject was instructed to do a set of controlled movements and then allowed to continue about their daily movements in a normal fashion. The signal data was then correlated with video data to aid researchers in developing unsupervised motion recognition systems.

This case study demonstrates the design considerations involved in deploying an end-to-end body sensor network. The CareNet deployment increases lifetime of the on-body nodes as packets are routed to the gateway over Wi-Fi rather than through an 802.15.4 mesh network, saving on communication and power consumption. Bandwidth limitations and 802.15.4 interference issues

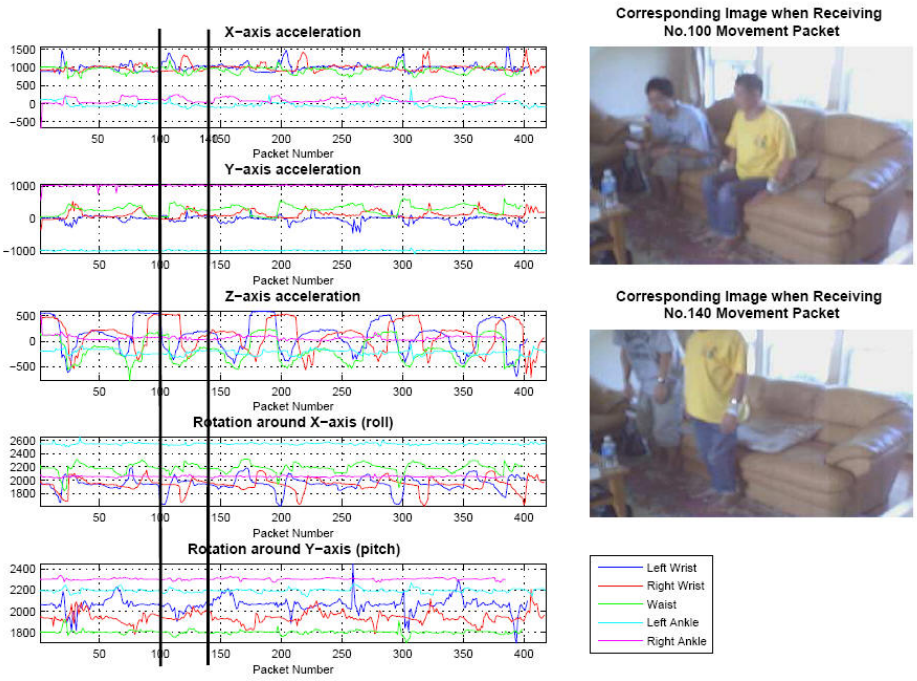


Figure 5.5: CareNet Data Viewer

are avoided by using the more-reliable 802.11 network to forward both data and video packets. Deployment of the system is easy because it can connect to any computer with an 802.11 connection without requiring additional hardware to be attached to the machine.

# Chapter 6

# Conclusion and Future Work

This project has successfully shown the design, implementation and effectiveness of a framework for Body Sensor Networks. This framework significantly improves the development process of bodynet applications.

BNSM has been released as an open-source project to allow its development to continue and its userbase to grow[1]. Future work includes further development and refinement of the framework functionality. In particular, functionality allowing the gateway to disseminate data and request node status has yet to be fully implemented. However, more important than additional functionality, the major future task lies in validation via implementation of real applications. Body Sensor Network research projects at the University of California at Berkeley, University of Texas at Dallas, Cornell University, Vanderbilt University and Telecom Italia Research plan to use elements of the BNSM framework to create real-time applications.

With this project and numerous others across the world, the futuristic vision of real-time wearable healthcare monitoring is quickly becoming a reality.

[1] Available online at `http://bnsm.sourceforge.net`

# References


[1] *Tmote Sky Data Sheet*, Moteiv Corporation, 2006. [Online]. Available: http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf

[2] *MICAz Data Sheet*, Crossbow. [Online]. Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf

[3] *SmartMesh M2510 Data Sheet*, Dust Networks. [Online]. Available: http://www.dustnetworks.com/docs/M2510.pdf

[4] 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 *Ambient Intelligence*, 2005, pp. 115–148. [Online]. Available: http://dx.doi.org/10.1007/3-540-27139-2_7

[5] A. Dunkels, "Programming Memory-Constrained Networked Embedded Systems," Ph.D. dissertation, Swedish Institute of Computer Science, Feb. 2007. [Online]. Available: http://www.sics.se/ adam/dunkels07programming.pdf

[6] *E680i Manual*, Motorola.

[7] *Stargate Data Sheet*, Crossbow Technology. [Online]. Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_Datasheet.pdf

[8] T. Pering, P. Zhang, R. Chaudhri, Y. Anokwa, and R. Want, "The PSI board: Realizing a phone-centric body sensor network," in *4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007)*, vol. Volume 13. Springer Berlin Heidelberg, 2007, pp. 53–58. [Online]. Available: http://www.springerlink.com/content/g26742734141r167/

[9] *LIS3LV02DQ Data Sheet*, ST Microelectronics. [Online]. Available: http://www.st.com/stonline/products/literature/ds/11115.pdf

[10] *IDG 300 Data Sheet*, InvenSense. [Online]. Available: http://www.invensense.com/shared/pdf/IDG_300_Datasheet.pdf

[11] V.-P. Sepp, J. Visnen, P. Kauppinen, J. Malmivuo, and J. Hyttinen, "Measuring respirational parameters with a wearable bioimpedance device," in *13th International Conference on Electrical Bioimpedance and the 8th Conference on Electrical Impedance Tomography*, vol. Volume 17. Springer Berlin Heidelberg, 2007, pp. 663–666. [Online]. Available: http://www.springerlink.com/content/h0k54844264264v1/

[12] M. S. Hansen and S. Sta, "Practical evaluation of ieee 802.15.4/zigbee medical sensor networks," Master's thesis, Norwegian University of Science and Technology, 2006.

[13] R. C. Shah, L. Nachman, and C. yih Wan, "On the performance of bluetooth and ieee 802.15.4 radios in a body area network," in *BodyNets '08: Proceedings of the Third International Conference on Body Area Networks*, 2008.

[14] J. Shin, U. Ramachandran, and M. Ammar, "On improving the reliability of packet delivery in dense wireless sensor networks," *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pp. 718–723, 13-16 Aug. 2007.

[15] *Technical Overview of Time Synchronized Mesh Protocol (TSMP)*, Dust Networks. [Online]. Available: http://www.dustnetworks.com/docs/TSMP_Whitepaper.pdf

[16] J. Perry, *Gait Analysis : Normal and Pathological Function.* Delmar Learning, January 1992.

[17] E. Jovanov, A. Milenkovic, C. Otto, and P. de Groen, "A wireless body area network of intelligent motion sensors for computer assisted physical rehabilitation," *J Neuroeng Rehabil*, vol. 2, Mar 2005. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=552302

[18] G. T. Carter, J. J. Han, R. T. Abresch, and M. P. Jensen, "The Importance of Assessing Quality of Life in Patients with Neuromuscular Disorders," *American Journal of Hospice and Palliative Medicine*, vol. 23, no. 6, pp. 493–497, 2007. [Online]. Available: http://ajh.sagepub.com/cgi/reprint/23/6/493.pdf

[19] A. Kochera, "Falls among older persons and the role of the home: An analysis of cost, incidence, and potential savings from home modification," March 2002. [Online]. Available: http://www.aarp.org/research/housing-mobility/accessibility/aresearch-import-788-IB56.html

[20] E. Taub, G. Uswatte, and R. Pidikiti, "Constraint-Induced Movement Therapy: a new family of techniques with broad application to physical rehabilitation–a clinical review."

[21] R. Jafari, R. Bajcsy, S. Glaser, B. Gnade, M. Sgroi, and S. Sastry, "Platform design for healthcare monitoring applications," *High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, 2007. HCMDSS-MDPnP. Joint Workshop on*, pp. 88–94, 25-27 June 2007.

[22] D. Karantonis, M. Narayanan, M. Mathie, N. Lovell, and B. Celler, "Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 10, no. 1, pp. 156–167, Jan. 2006.

[23] A. Yang, R. Jafari, P. Kuryloski, S. Iyengar, S. S. Sastry, and R. Bajcsy, "Distributed segmentation and classification of human actions using a wearable motion sensor network," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-143, Dec 2007. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-143.html

[24] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 188–200.

[25] *Spine White Paper*, Telecom Italia. [Online]. Available: http://spine.tilab.com/papers/2007/WhitePaper.pdf

[26] O. Ben-Kiki, C. Evans, and I. Net, *YAML Aint Markup Language (YAML) Version 1.1*, 2008. [Online]. Available: http://yaml.org/spec/cvs/current.html

[27] J. Chen, K. Kwong, D. Chang, J. Luk, and R. Bajcsy, "Wearable sensors for reliable fall detection," *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pp. 3551–3554, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1617246

[28] S. R. Madden, "The design and evaluation of a query processing architecture for sensor networks," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, BERKELEY, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=1123678

[29] E. Shvets, *Extending TinyDB: Creating Custom Aggregates*, 2003. [Online]. Available: http://www.sabanciuniv.edu/mdbf/comnet/eng/SensorWeb/pdfler/tinyos/tinydbagg.pdf

[30] M. Mathie, B. Celler, N. Lovell, and A. Coster, "Classification of basic daily movements using a triaxial accelerometer," *Med Biol Eng Comput*, vol. 42, pp. 679–687, Sep 2004.

[31] J. Lee and I. Ha, "Real-time motion capture for a human body using accelerometers," *Robotica*, vol. 19, no. 6, pp. 601–610, 2001.

[32] S. Jiang, Y. Xue, Y. Cao, S. Iyengar, R. Bajcsy, P. Kuryloski, S. Wicker, and R. Jafari, "Carenet: An integrated wireless sensor networking environment for remote healthcare," in *BodyNets '08: Proceedings of the Third International Conference on Body Area Networks*, 2008.