DYNAMIC RECONFIGURATION OF LARGE-SCALE SENSOR/ACTUATOR NETWORKS: A DATABASE-ENABLED APPROACH

By Michael Middleton

A Thesis

Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in Engineering

Northern Arizona University

August 2017

Approved:

Paul G. Flikkema, Ph.D., Chair

James Palmer, Ph.D.

Fatemeh Afghah, Ph.D.

ABSTRACT

DYNAMIC RECONFIGURATION OF LARGE-SCALE SENSOR/ACTUATOR NETWORKS: A DATABASE-ENABLED APPROACH

MICHAEL MIDDLETON

Wireless sensor networks (WSN) provide a robust and versatile solution to distributed low-power computing over a range of diverse network topographies. One-way data aggregation from network to data center is traditionally the fundamental data acquisition feature for many wireless sensor networks. With the increasing prevalence of IoT devices and the growing number of WSN research applications, network and nodal reconfiguration to suit changing environmental conditions and sensing needs via outbound commands is not only an attractive feature, but in many cases a growing necessity. The work of this thesis comprises the implementation of a web-based application allowing end users to identify and reconfigure sets or subsets of WSN nodes based on static and dynamic WSN configuration information. This end-to-end network reconfiguration procedure relies upon an existing two-way cyber-infrastructure and a relational database to store and track network state information. Additionally, the streaming middleware solution integral to this cyber-infrastructure provides a tracking mechanism for network reconfiguration commands. This tracking ability along with the reconfiguration mechanisms not only allows end-users to enact behavioral network changes, but also provides the foundation for virtualized users to autonomously monitor and enact behavioral changes to the network in response to changing network or environmental conditions.

Acknowledgements

First, I would like to express my gratitude to Dr. Flikkema for providing me the opportunities I have undertaken these past several years. Thank you for your insight and patience as I worked towards this goal. The opportunity to work in your lab has been one of a kind and one I will not forget.

Thank you to Dr. Afghah and Dr. Palmer for agreeing to serve on my committee.

Thank you to everyone in the lab for the fun times, road trips, and late nights. Special thanks to Jonathan Knapp and Chris Porter for being an example and teaching me so much.

Finally, I would like to thank my family for all of their love and support over the last several years. Without you this would not have been possible.

Contents

List of Tables

List of Figures

To my family

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Transducers can be used to sense radiation and measure soil moisture, temperature, and other physical phenomena. Transducers are often built into devices referred to as wireless sensors, with wireless networking capabilities. Wireless sensor networks (WSNs) use wireless networking technologies to enable groups of wireless sensors to communicate with each other and connect to the Internet. Low power distributed computing, intelligent wireless networking, and sophisticated data management form the core of WSN technology. The integration of these technologies are enabling the development of modern applications like the Internet of Things (IoT).

Some collections of wirelessly networked sensing devices form a cyber-physical system. A cyber-physical system is one where computation and networking processes operate in conjunction with a physical system, such as a smart garden or an advanced industrial process. While there are a broad range of implementations and platforms that fall under the umbrella of WSN, they all have a particular set of design restrictions and offer fundamental baseline features. WSN devices generally lack sophisticated processing power, run on a finite power supply (typically batteries), and use a diverse range of network topologies. Due to the power constraints of many wireless sensor nodes, transmission ranges are often quite short, and networks require a base station with greater networking capabilities, enabling access to the rest of the Internet; this is typically accomplished via a cellular or satellite connection.

WSNs allow researchers, scientists, and analysts to embed transducers into the environment which manage and perform sampling duties on a diverse range of sensors. Additionally, actuators allow for changes to be imposed upon the environment at given scheduled times or in reaction to certain events. In a WSN, the data collected from the various devices is aggregated and routed through the network to a base station where it is accessible to applications or archived at a data center. Depending on the diversity of the data types, the sophistication of the sensors, or the high-level data management schema, there is a diverse range of operations such as data conversions, checksum error detection, and stream creation that must be performed. These operations, data conversion for instance, might be performed at the sensor node, at the base station, or even at the data center.

1.1.1 Difficulties of Network Reconfiguration

The continued persistent functionality of a WSN requires that it have a certain level of autonomy. Basic WSN operation requires data aggregation from the nodes at the data center by whatever infrastructure is in place. There is also, however, a need for direct network interaction allowing a user to observe and enact behavioral changes within subsets of a network or the network as a whole. Researchers rely upon WSN to provide the means for extended monitoring or experimentation that span lengthy periods of time; changing circumstances may require for users to change the configuration of WSN devices based upon changing experimental needs or environmental conditions. For example, if a researcher studying the effects of soil moisture at a particular geographic location predicts a rain event, he or she might want to reconfigure the behavior of the WSN devices to gather soil moisture readings at an increased sampling rate to achieve greater data resolution.

There are a number of different design implementations for heterogeneous wireless sensor network control and reorganization, such as TinyOS [1] and MagnetOS [2]. These employ systems that allow modification of the sensors' behavior. It can be difficult to perform reconfigurations at the device level when WSNs are in remote or difficult to access locations such as rural areas or dangerous industrial facilities. Providing a user interface for researchers which accommodates making such modifications would expand the functionality of the network in potentially new and unique ways.

1.2 Statement of the Problem

When physical access to a device is needed to perform device or network modification, this can place a significant burden on researchers by necessitating a large number of hours of travel time to remote locations or the traversal of dangerous environments to access the devices, such as in an industrial setting. The wireless nature of WSN systems offers a natural mechanism for users to communicate with the devices remotely.

Enabling users to remotely change the behavior of WSN nodes would allow researchers to rapidly tailor the behavior of the network to changing needs or environmental conditions. The creation of a web-based application which provides users with real-time network state information and the ability to enact changes would significantly reduce the amount of travel time, increase the amount of control a user has to rapidly reconfigure WSN nodes, and allow the mechanisms necessary to automate the reconfiguration process.

An agent is a software entity that can utilize the same network device reconfiguration capabilities that a person would have, though its actions can be defined by software that can respond to changes in network conditions. An example is an agent which monitors the power consumption of WSN devices in the network and reconfigures the devices to conserve energy. For example, if a specific WSN node is low on battery voltage, an agent might send the node commands that lower a transducer sampling rate.

The specific objectives of this work are as follows:

- Implementation of an application which allows users to remotely identify and select sets of WSN nodes based on static and dynamic search criteria;
- Implementation of an interface allowing users to enact behavioral changes in individual or groups of nodes; and
- Generalization of the reconfiguration software such that agents can monitor and reconfigure network nodes.

This thesis presents a system for both users and automated agents to reconfigure arbitrary subsets of nodes in large, complex collections of WSNs using (1) a database paradigm for selection of node subsets based on both static and dynamic data, and (2) an interface that allows both users and automated agents to identify subsets and issue valid reconfiguration commands.

1.3 Summary of Contents

Chapter 2 summarizes the academic literature discussing reconfiguration of sensor networks and the various platforms where reconfiguration has been implemented. This chapter also discusses the challenges associated with reconfiguring wireless sensor networks and the design features that researchers have utilized to overcome these challenges.

Chapter 3 discusses the WSN platform and cyber-infrastructure named WiSARDNet.

Chapter 4 explains the approach used to design network reconfiguration software with the use of a relational database.

In chapter 5, the software modules developed in this work are discussed in detail.

In chapter 6, automated network reconfiguration is discussed in detail.

The final chapter summarizes the results of the research and software implementation. Additionally, conclusions are discussed, followed by opportunities for future work.

Chapter 2

Literature Review

2.1 Introduction

WSN technology has matured such that it is a core component of the developing Internet of Things (IoT). The IoT describes the way in which networking capabilities formerly restricted to more traditional computers are being extended to physically-embedded devices, allowing for the acquisition and transfer of data on a much larger scale. WSN are geographically clustered networked devices equipped with one or more transducers which can range from a simple temperature probe to an implanted medical device. In general, these devices have limited computational resources and operate on supplies with limited power and energy. Due to these limitations, developers need to consider the design constraints of WSN applications. Many WSN devices are constrained such that their computational resources may not support a complete operating system (OS), which have been fundamental to traditional computing systems. The lack of a traditional OS may complicate the development or limit the functionality of an application.

The range of devices being added to the IoT which generate data grows daily. Even though extending the operational lifespan and overcoming computational limitations remains at the forefront of the challenges facing WSN technology, numerous developments are enabling greater WSN functionality. Some of the features being developed within the domain of WSN are intelligent routing and communication protocols, fault tolerance and network health monitoring, service oriented design, and application execution frameworks. In recent years, developers and researchers have streamlined many of these features and have created new abstractions for WSNs. These features allow developers to overcome many of the hurdles facing modern WSNs.

2.2 WSN Execution

The functionality necessary for WSN devices includes two critical features: distributed sensing and wireless data aggregation. WSN nodes have the ability to sample data from transducers and the ability to enact changes through the use of actuators. Ideally, researchers would be able to automate their experiments through an application program that interacts with individual or groups of WSN nodes. Clever use of abstraction to manage the complexities of the WSN will dictate how an application is executed; an application might run as an executable binary directly on a sensor node's hardware or perhaps within an abstraction of the network at a higher layer in the form of scripted events or command sequences. WSN hardware is often extremely restrictive in its computational resources, and therefore application deployment and execution must be carefully designed.

Ivester et al. in [3] describe ISEE, an interactive execution framework for monitoring network services and specifying operation of applications executing on a WSN node using a design paradigm known as service oriented architecture (SOA). Network monitoring functionality is an important set of features, especially as experiments increase in complexity. The generalization of these features into modular services allows them to be used with greater versatility and independent of specific application code. Researchers who have developed other modern WSN platforms have also adopted the SOA paradigm of implementing WSN functionality. Hammoudeh et al. in [4] describe a SOA approach to WSN fault tolerance and inter-node cooperation.

Another category of WSN execution paradigms includes those that use a virtual machine (VM). A virtual machine emulates a particular hardware architecture by translating instructions to another hardware architecture, thus allowing software designed for a particular platform to execute atop a different platform. Two of the more prominent WSN reconfiguration approaches that utilize virtual machines are Maté [5] and MagnetOS [2]. Maté is a virtual machine that is created to overcome many of the challenges of implementing heterogeneous network execution on resource-constrained wireless nodes. The fundamental feature of Maté is a bytecode interpreter. Programs and applications, even those with minimal complexity, can be hundreds of kilobytes in size. The virtual machine takes high-level operational instructions and organizes them into capsules that can be smaller than 100 bytes in size. The Maté byte code interpreter running on individual nodes allows for the execution of capsules that are disseminated throughout the network. This allows for a scalable approach for specifying network execution on a distributed system of resource constrained nodes. Similarly, the creators of MagnetOS use a virtual machine to specify network-level execution. These researchers use a version of the Java Virtual Machine (JVM) designed to appear as though it is operating on a single computer; however, the computer is actually a WSN. This abstraction of the WSN as a platform which can run the JVM allows for Java application code to be written by developers. The Java objects produced by the JVM can be sent throughout the network to different physical hosts, enabling the distributed execution of a single Java application. This is performed by a byte-code level translation into instructions that the nodes can execute. Maté and MagnetOS can compress large programs into small device operations, due to the abstractions which they were able to make in their design. Though a complete virtual machine abstraction of a WSN is not utilized in the work of this thesis, the core principles of energy and resource aware network execution through abstraction is one of the primary design goals.

Flikkema et al. [6] describe the design and implementation of a cyber-physical system which utilizes an ultra low power sensing platform, streaming data middleware, and a control-oriented approach to performing complex ecological experiments. Several core principles from the previous works such as energy-aware network execution and abstracted application development through high-level programming languages have been designed into this platform. An important distinction between previously described works and the platform used in this thesis is that the WSN cyber-infrastructure is a network of networks. An individual cluster of nodes at a geographic location form a WSN, but behavior needs to be tracked and controlled across multiple WSN locations. This platform forms the foundation of the work of this thesis and will be described in greater detail in Chapter 3.

As the complexity of WSNs and diversity of applications increase, so increases researchers' need to reconfigure specific hardware components or WSN properties. A simple sensing network might only support unidirectional communication where the nodes pass their data to a base station. More sophisticated sensing networks support bi-directional communication where nodes both send and receive information. Bidirectional communication is a critical feature for any WSN system to support remote device reconfiguration and reprogramming. Depending on the size of a WSN supporting a particular application, a change could affect one node or even an entire cluster of nodes, depending on the type or magnitude of the change. For example, a researcher monitoring temperature fluctuations over a large geographic area with a cluster of WSN nodes might want to increase the sampling rate to observe fine-scale environmental responses to a weather event. The difficulty of this task depends on how device reconfiguration was designed into the WSN platform and accompanying cyber-infrastructure.

Reprogramming and reconfiguration are often used interchangeably, as they might have the same effect on a researcher's experiment, but there is an important distinction between them. According to some interpretations, like e.g. [7], if changes are made exclusively to the software components, then the device has been reprogrammed. Alternatively, if there is some change in the device hardware, the device is said to have been reconfigured. For the work of this thesis, we will define reprogramming as involving transfer of new source code or application software to one or more devices, whereas reconfiguration implies that software parameters or hardware functionality can be altered without the addition of new program code. Ivester et al. [3] describe the role of control information in a WSN as a means for reconfiguration. Control information describes commands or network state information that are injected into a WSN, causing change in one or more nodes. Software called Ismanage used in [3] accompanies ISEE by providing several features, the most important of which is the ability to disseminate control information into the WSN for the purpose of reprogramming and reconfiguration. This approach is similar to the reconfiguration methodology used in WiSARDNet, the platform used for this thesis.

Eronu et al. [7] summarize many of the issues and challenges surrounding WSN reconfiguration. Given that WSN hardware is often limited in power availability, the energy overhead involved in WSN reconfiguration is a major concern, especially when the number of nodes to be reconfigured increases [7]. The authors of [8] describe a wireless data collection protocol named ORiNoCo, designed specifically to utilize the existing bidirectional messaging capabilities of a WSN for the purpose of energy efficient network reconfiguration. Efficient network reconfiguration is also a core feature of the Maté and MagnetOS platforms discussed in the previous section [5], [2]. Nodes that run the Maté byte-code converter execute capsules which contain small control sequences. In this way, new configurations can be encoded and transmitted to a WSN node. The target node, upon receiving the encoded configuration, can decode the capsule with an intermediate software layer that translates the capsule's sequence into specific values to be stored in the device's memory. The reconfiguration of nodes in the MagnetOS platform allows for software changes to be instigated within an image that runs on the Java Virtual Machine. The virtual machine then interprets the image and sends the appropriate byte-code values to each hardware device in the network. Reconfiguration of these nodes then becomes no more difficult than executing a new application. We can see that when the sensing hardware is made available as a service to higher layers as in [3], and bidirectional communication protocols are leveraged effectively, network-wide reconfiguration of many nodes can be achieved with low energy overhead.

2.3 Relational Databases for WSN Applications

Abstractions and unique design paradigms can guide the development of complex systems in ways that are simple to understand. Storing information in a database enables access to specific data via query statements. A database abstraction can be used to simplify the complexities of data collection from a WSN [9], [10], [11]. [9] uses a WSN design paradigm where the data from each node can be aggregated at a data sink, and the sink acquires the data from each node through SQL-like queries. The nodes act like individual databases, and therefore the complexity of distributed data collection can be managed through simple queries. In this way, data is acquired on demand rather than continuously streaming towards the sink, effectively reducing the amount of data transmitted to only what is desired as opposed to all available samples. The authors in [11] utilize this approach for data acquisition as well.

Database abstractions provide data archival, organization, and access features that support a variety of applications. These features can greatly improve the efficiency and usefulness of WSN systems by providing scalable storage solutions across a variety of platforms. The division of data into multiple tables which can be managed separately and linked via foreign key constraints allows complex yet versatile access by experimenters and analysts. These ideas also pair well with modern software design paradigms such as behavioral parameterization [12], where methods can be written in such a way that they define different behaviors to be decided at runtime. Behavioral parameterization is a powerful design paradigm that will allow WSN data to be easily and accessed and utilized in the ever-changing landscape of IoT applications.

The aim of this thesis is the use of a database abstraction to simplify the complexity of network reconfiguration by storing network state information and other parameters that specify WSN execution, and utilizing this information to generate control messages for dissemination into the network for reconfiguration. Modular software design principles such as behavioral parameterization are used extensively in this thesis to accommodate rapid network reconfiguration as well as changing design requirements of future applications. This approach is described in further detail in Chapter 4.

Chapter 3

WiSARDNet Cyberinfrastructure

3.1 Overview

This chapter describes the platform on which the work of this thesis is built. WiSARDNet is a WSN platform which consists of hardware and software that connect sensor/actuator nodes to users and their ecological experiments. Cyberinfrastructure describes computational systems with advanced data acquisition, processing, and management capabilities, according to one of the more widely used definitions of the term [13]. According to this definition, WiSARDNet can be thought of as cyberinfrastructure. Each network component is described in this chapter, as well as the data management and archival processes. The cyberinfrastructure hardware and software components can be divided into four categories: WSN, base station, real-time data center (RTDC), and end-user. Figure 3.1 shows how the hardware and software components for each of these categories fit together. The core component which links the WSN, the base station, and the real-time data center is a TCP/IP based message broker that uses the MQTT protocol. MQTT will be described in further detail in this chapter. Additionally, database archival components will be described in terms of their role in the cyberinfrastructure in this chapter. The technical specifics of the database will be described in greater detail in Chapter 4.

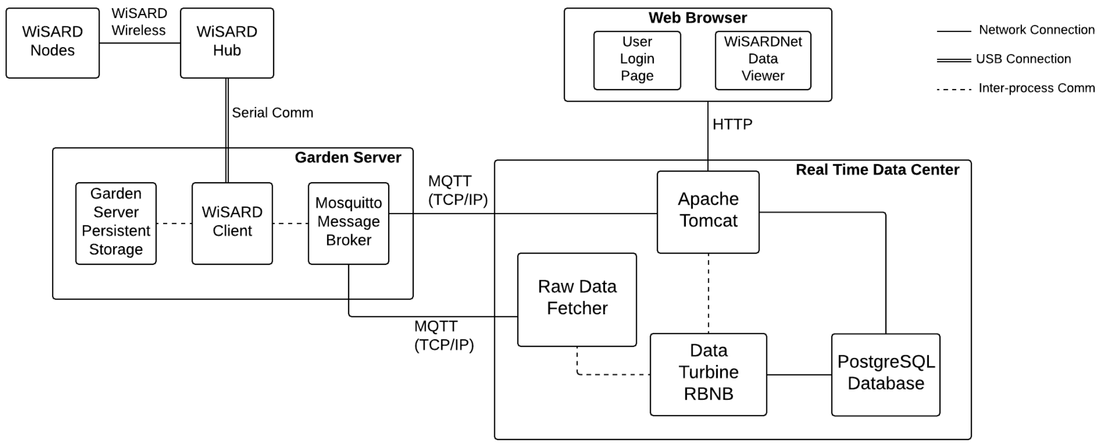


FIGURE 3.1: A high-level view of the WiSARDNet cyberinfrastructure. WiSARD networks, Garden Servers, the Real-Time Data Center, and a Web Browser are all essential in connecting data from transducers to users at their computer terminals.

3.2 WiSARD Nodes

A Wireless Sensor/Actuator and Relay Device (WiSARD) is modular, adapting easily to the different sensing and actuation needs of users. Flexibility and energy awareness are the two driving design motivations in the WiSARD development. At the heart of each WiSARD device is a central processor (CP) which governs the operation of the device and its peripherals. The CP dictates when and how tasks will be scheduled and executed, establishes and manages wireless links to other WiSARDs, facilitates the storage of sensor readings, and offloads the sensing and actuation tasks to daughterboards called satellite processors (SP).

A WiSARD can accommodate up to four different SPs. This enables each WiSARD to perform a variety of tasks specifically tailored to meet the needs of researchers and experimenters. There are three different types of SPs that a WiSARD can use: SP-ST, SP-STM, and SP-CM-STM. The SP-ST (SP - Sense Temperature) generates temperature readings by sampling Type-T thermocouples. The SP-STM (SP - Sense Temperature and Moisture) generates soil moisture and soil temperature samples. The SP-CM-STM (SP - Control Moisture - Sense Temperature and Moisture) accommodates the same sensing capabilities as the SP-STM but also provides water valve actuation capabilities. The modular design of the WiSARD allows the CP to offload these sensing and actuation operations to its SPs which can execute tasks in parallel with CP operation.

Both CP and SPs use a Texas Instruments MSP430 16-bit ultra-low-power microcontroller. This microcontroller has a variety of features and can be placed in various low-power modes. The low-power modes of operation, accompanied by energy-conscious software design, allow for a WiSARD to operate on a single pack of three AAA batteries for many weeks or even months, depending on the rate at which the devices dispatch their sampling operations. The CP also connects to a radio board which uses an Analog Devices ADF7020 RF transceiver module that operates in the 902-928 MHz license-free ISM band. When WiSARDs are powered up, they autonomously form a self-organizing and self-healing multi-hop wireless ad-hoc network [6]. A special WiSARD which is assigned the software role of Hub acts as the base station, and is connected to an embedded Linux computer referred to as a garden server.

3.3 Middleware

Middleware is a term which describes a software abstraction of the transfer of information from point A to point B. By hiding the complexities of data transport behind a simple interface that connects two pieces of software, the development of powerful data processing and management tools can be accelerated. WiSARDNet relies heavily upon the use of middleware to make the WSN data available to the rest of the world. The central architectural component that connects the garden servers to the RTDC is MQTT. MQTT is a data transfer protocol created by IBM that utilizes the publish/subscribe middleware paradigm [14].

In order for MQTT to be explained in the context of this work, a few more terms need to be defined. This work is concerned with the reconfiguration of WiSARD nodes. Each WiSARD is comprised of multiple devices. A device is a singular piece of hardware; a transducer, a CP, a SP, and a radio are each examples of a device. The set of operational parameters that control a device is called a configuration. A deployment is an instance of a device and its configuration.

Reconfiguration is the process of changing the operational parameters of one or more deployed devices.

MQTT facilitates the sharing of data between applications through the use of brokers. A broker is an application that sends and receives data to and from other applications. The transferred data are called messages. When a program sends a message to a broker, it labels the message with a topic name. The program sending the message is called a publishing client. A program that is interested in data from a publishing client can subscribe to a topic through the broker; this program is called a subscription client. All messages from publishing clients that are labeled with a matching topic will be sent by the broker to all subscription clients that have subscribed to that topic.

WiSARDNet uses Mosquitto [15], an open source implementation of a message broker that uses the MQTT protocol. A message broker installed on each garden server keeps all of the data from that site in non-volatile memory. The data is organized into archive files at regular time intervals. To transfer the data from the WiSARDs to the broker, a Java application referred to as a WiSARD client uses MQTT messages to publish data. The RTDC can then retrieve the data using a subscription client that connects to the garden server message brokers over cellular or satellite connections.

An MQTT subscription client for each garden site runs at the RTDC and is responsible for subscribing to all the data published to each broker. These subscription clients are what is referred to in WiSARDNet as a raw data fetcher

(RDF). The purpose of the RDFs is to retrieve all of the data from all of the gardens so that the data is accessible for processing, storage, or viewing. These processes are discussed in detail in the following section.

3.4 Real-time Data Center

The RTDC is a collection of compute and data servers that host a Apache Tomcat web server, a Mosquitto MQTT subscription client, an instance of an open-source data streaming middleware called Data Turbine, a PostgreSQL relational database, and all of the data management processes and applications which interact with the WSN. The RTDC has several functions, the first of which is to retrieve all of the WSN data from each of the gardens. The RTDC is a centralized destination for all of the network's data streams. From this location, applications can access, store, and modify data for their various needs, as opposed to having to interact with multiple networks individually. For instance, a process which reads in raw data streams from temperature sensors might need to calculate human-readable values from the raw sensor readings. A single data converting processor which moves data from raw transducer values to human-readable values, can easily acquire all of the raw data streams for temperature sensors at all network locations with a single fetch, rather than requiring that the process fetch the raw data from each specific garden individually. Additionally, aggregating all data at the RTDC greatly simplifies archival and backup procedures.

Another function that the RTDC performs is the execution of applications and processes which interact with the WSNs and their data. One example of an application running on a server at the RTDC might be an experiment which attempts to maintain equal soil moisture readings at two different geographic locations. The RTDC servers possess high-performance computing hardware that many applications and processes can use to interact with the networks in real-time.

3.4.1 Relational Database

Relational databases provide a flexible way to store and access large amounts of data. A relational database is composed of one or more tables; tables are 2 dimensional matrices of rows and columns. Rows and columns are referred to as records and fields, respectively. An entry in a database table occupies a single row and each field corresponding to a different attribute of that entry constitutes a new column. For example, a table which stores people might have columns for an identification number, a name, an address, and a telephone number. An entry of a new person into this table would occupy a single row, with the data matching each of the attributes would occupy the rows' intersections with each column. Formally, these columns are referred to as primary keys, as they uniquely identify database records. The database at the RTDC has numerous tables which store information about every device, garden location, and sensor reading.

In WiSARDNet, data streams are archived in a PostgreSQL table. Other tables in this database store all of the meta-data and logistical information regarding all WSN devices, as well as every data point sampled from all transducers. Additionally, there are many data streams that need to be tracked and archived such as diagnostic and control data, error reporting, and other useful information. An example of diagnostic data might be the logging of a device restart event; such events can be crucial in detecting and analyzing network performance issues.

The PostgreSQL database where all of this data is archived is physically located on its own server hardware at the RTDC, separate from the other data management and processing clients. By placing the database on its own server hardware, the database has exclusive access to dedicated processing resources to allow for the quickest possible query and insert times.

3.4.2 Data Schema

WiSARD devices interact according to the relationships that result from how they are connected together. Figure 3.2 shows a set of device relationships for an example WiSARD. The example WiSARD uses three different SP types, each one accommodating specific transducers or actuators.

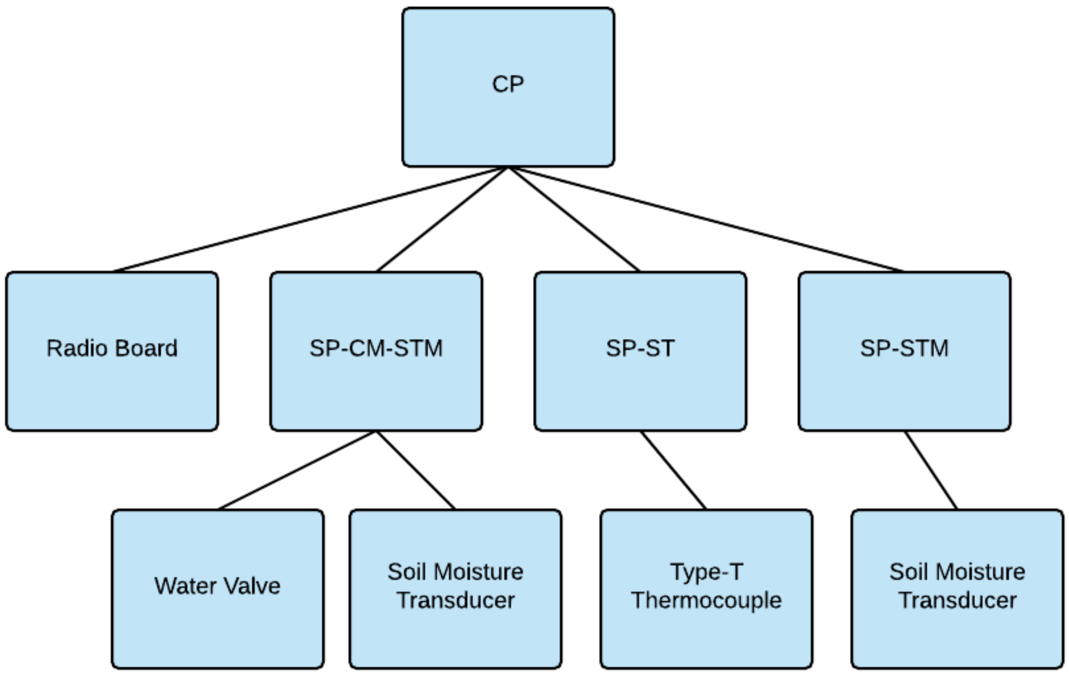


FIGURE 3.2: Example of a WiSARD's device relationship structure. This WiSARD has three SPs and each SP has a transducer. The SP-CM-STM also has a water valve that it actuates.

WiSARDNet organizes gathered data into database tables in a way that accommodates the reconfiguration of WiSARDNet devices. The organizational structures and relationships of the database tables are known as the database schema. Every piece of hardware is referred to in the schema as a device. The Device Table is therefore an important table in the schema. A record in the Device Table describes any device of any type. For instance, a radio, a soil-moisture transducer, a satellite processor, or a garden server are all types of devices that each have records in the Device Table. The configuration of a device might change at some point in time; it might be moved to another location, it might be upgraded or replaced, or it might receive a new firmware version. In this schema, the device's software parameters, hardware configuration, and state of operation are all encapsulated in a deployment. All deployments are stored as records in the Deployment Table.

When a transducer is attached to a SP, then its deployment record references the SP's deployment record in the Parent deployment field. When a change is made to any of the parameters tracked in the deployment table, the device is considered to have a new deployment, and therefore a new record is inserted into the deployment table. Both the old and new deployment records reference the device record, but the old deployment is disabled by setting the Active deployment field to *false*. Over time, many deployment records may accumulate for a particular device as changes are made. However, there should only ever be one active deployment for a device at any given time. This allows the entire deployment history of a device to be stored so that all data points from a device can be correlated to the configuration of the device at the time that the data point was created. Currently the rule of having only one active deployment for a device at any given time is not enforced by the database. Instead, this is governed by the procedures followed by a technician or researcher when creating a new deployment.

A WiSARD's device relationships naturally form a tree structure. The reason for this is because a WiSARD has exactly one CP to which all other devices connect, either directly or indirectly. This resembles the concept of a root node in a tree data structure. Likewise, each SP resembles the internal nodes of a tree since it connects to at most one CP, but can be parent to multiple transducers or actuators. Finally, the radio, the transducers, and the actuators resemble the leaf nodes of a tree since they cannot be parent to any other devices. The database schema reflects the tree structure by storing the device deployment records with parent/child relationships. This allows all of the devices of a WiSARD to be accessed using standard tree traversal algorithms on the device Deployment Table records.

Having the ability to store and track each device's configuration and related metadata in the database is of great value in the management of the networks and the various experiments. When a WiSARD is ready for deployment at a garden site, a deployment record is created with all of its configuration data, the deployment record is set to active, and the start date and time is set. As long as the device remains in this configuration, the deployment object pointing to that device will remain active. If a device is changed in any substantive way, the following actions are taken:

1. The deployment record referencing the changed device is set to inactive.
2. The current time is inserted into the stop-time field of the deployment record
3. A new deployment record is inserted into the deployment table
4. The parent field of the new deployment record references the deployment record of the parent device
5. The active status field of the new deployment record is set to active

6. The current time is inserted into the start-time field of the new deployment record

By following this procedure, device configurations are easily accessible from the database. A previous deployment exists as a single record in the Deployment Table which is trivial with regards to computational complexity and storage resources. This approach is simple, intuitive, and scales well as the number of devices deployed in the field increase. With this paradigm, a data sample is not merely a sample from a device, it is a sample from a specific deployment of a device. When a device's configuration is altered, a new deployment is created for that device and the data generated by that device references the new deployment record.

When new data from WiSARDs arrive at the RTDC from the MQTT broker, they need to be accessible for users and other services to request. Data Turbine's network accessible ring buffer data structure works well for this purpose. At the RTDC, the data streams arriving via MQTT are placed into Data Turbine's ring buffer in accordance with the stream name which identifies the device from which the sample was taken. Since the complexities of WiSARD configurations are handled via storing their configuration data in database tables, archival of sampled data in the database becomes a simple procedure. All samples from all streams are placed as individual records in a table named Data. In addition to the sampled value and the time that it was sampled, the value can be related to the specific device and deployment records via its Data Turbine stream name. In this way, data archival of acquired samples is extremely simple and all information regarding a device and its samples is easily accessible and intuitively obtained through the thoughtful design of the data schema.

3.5 Summary

WiSARDNet is a WSN CI which was designed from the ground up to be modular, scalable, and accessible to users. The Mosquitto MQTT broker, the data streaming middleware Data Turbine, and the RTDC enable data sampled by sensors to be retrieved, managed, processed, and archived into a PostgreSQL database. The way in which WiSARD meta-data and configuration information is stored and accessed is critical for the development of a network configuration software which comprises the work of this thesis. How these features are utilized in the network management software is described in further detail in Chapter 4.

Chapter 4

Network Reconfiguration Software

4.1 Overview

This chapter discusses the software developed in this thesis. First, this chapter discusses the procedures involved in network reconfiguration and the features required to achieve that goal. Next, the features are discussed in terms of the software functionality that needs to be implemented. Finally, the chapter discusses how the different functionality is grouped into software modules. The specifics of how each module works is described in Chapter 5.

4.2 Approach

To achieve network reconfiguration functionality within WiSARDNet, the following features are required:

- The user needs an up to date understanding of a WSN's configuration.
- The user needs to be able to specify the desired configurations or features they want to make to a set or subset of WSN nodes.
- Users need to be able to communicate configuration changes to the WSN.
- The WSN needs to execute the new behaviors that the user specifies.
- The system needs protections that ensure continued and correct WSN operation.

As was shown in Chapter 2, there are many ways that other WSN platforms have approached network reconfiguration. For the work in this thesis, a database abstraction is used as the general approach for structuring WSN reconfiguration. The specific abstraction used is that the WSNs in WiSARDNet can be viewed as a database of dynamic heterogeneous behaviors. By viewing the WSNs from a database perspective, queries and inserts become the language that users can use to interact with the WSN. The database paradigm simplifies the complexity of WiSARDNet and guides the design of WSN reconfiguration.

In addition to the database abstraction, another abstraction is used to approach WSN reconfiguration. As described in Chapter 3, the components that comprise a WiSARD are represented in the WiSARDNet PostgreSQL database as individual devices with deployments that describe their current state. Since a WiSARD is a collection of these physical devices, it is useful to represent the device deployments as WiSARD objects. This is especially true given the number of database relationships that are managed, such as those to parent deployments, device types, sites. Figure 4.1 shows an example of how a WiSARD encapsulates multiple related deployments.

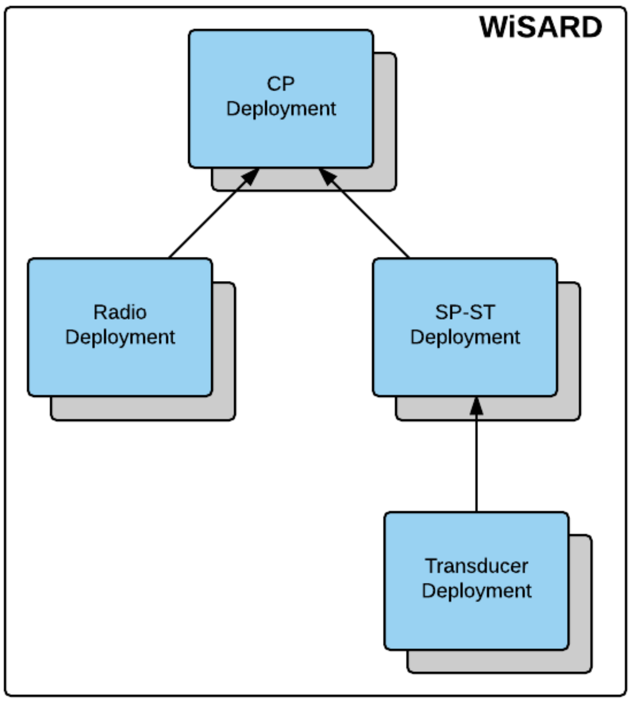
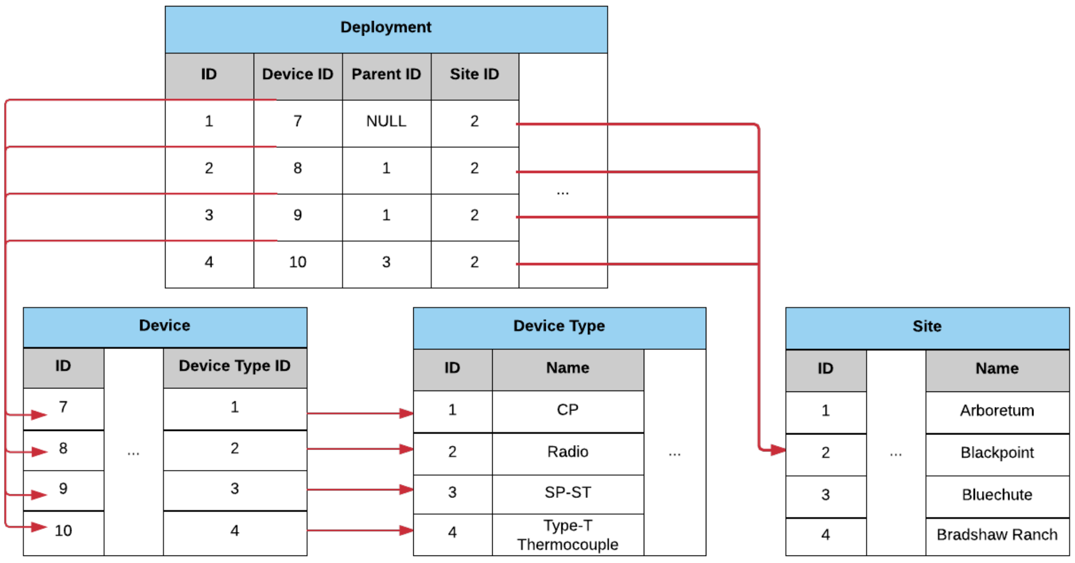


FIGURE 4.1: An example of the WiSARD abstraction used to encapsulate multiple related device deployments. Each arrow represents a deployment record's reference to its parent deployment.



Deployment				
ID	Device ID	Parent ID	Site ID	
1	7	NULL	2	
2	8	1	2	...
3	9	1	2	
4	10	3	2	

Device		
ID		Device Type ID
7		1
8	...	2
9		3
10		4

Device Type		
ID	Name	
1	CP	
2	Radio	...
3	SP-ST	
4	Type-T Thermocouple	

Site		
ID		Name
1		Arboretum
2	...	Blackpoint
3		Bluechute
4		Bradshaw Ranch

FIGURE 4.2: Database representation of example WiSARD abstraction from Figure 4.1. The arrows illustrate foreign key references between records in different tables.

Using WiSARD objects to encapsulate multiple related device deployments means that a WSN can be reconfigured by updating sets of WiSARDs rather than many individual deployments. The database abstraction of the WSNs and the WiSARD abstraction of the WSN node devices together provide the conceptual foundation for the reconfiguration features to be implemented in software. The WiSARD abstraction is implemented in code as a Java class. Each instance of the WiSARD class encapsulates all of the device deployments of a WiSARD object as shown in Figure 4.1. Instances of the WiSARD class provide methods for accessing records referenced by deployment foreign keys, similar to navigation properties in Object Relational Mapping (ORM) objects. For example, devices in a WSN are deployed at locations where a researcher would like to gather data. A Site database table was created in order to store information related to such deployment locations. One of the ORM-like methods that a WiSARD object provides is one that will return the Site record referenced by the WiSARD deployments. In addition to the WiSARD class, there are classes for encapsulating other tables in the database, including the Site Table. When a WiSARD returns a deployment's Site record, it is in the form of an instance of the Site class. The database tables and their related Java classes are discussed in further detail in the following Section.

4.3 Solution Description

Implementing WSN reconfiguration functionality into an existing software system requires that new code modules be developed that account for all of the known use cases and as many future use cases as possible. A practical example is a scenario where a researcher wants all soil moisture transducers attached to WSN nodes in a specific region to be sampled at twice their current sampling rate. Performing this procedure involves the user specifying the region and sensor types needed to produce a matching set of WSN nodes, and that he/she wants the sampling rate of those transducers to be changed. The software needs to validate that there are no conflicts between that user's requested changes and the existing operational configurations of the node. Finally, the nodes will each be signaled via command packets, execute the specified change in configuration, and will then alert the user of the changes made.

4.3.1 Implementing Functionality

Each of the features described in the previous section needs to be implemented as software functionality to be used in the WiSARDNet system. This section describes the software tasks for each feature.

4.3.1.1 Identification of WSN Nodes

Transducer samples, sensor types, WiSARD role information, and hardware metadata are all stored in database tables and are necessary to determine which nodes will need to be reconfigured. The first stage in reconfiguring a WSN is the specification of the node or nodes whose configurations will be modified. This operation begins with a user specifying search criteria for a set of WiSARD that they want to reconfigure. Next, static and dynamic WiSARD information is gathered by querying a set of PostgreSQL database tables that have all of the information that describes a WiSARD. Table 4.1 shows all of the tables that are queried when searching for WiSARDs and describes the information stored in each table. The results of these queries are then stored in instances of Java objects created for each table.

Database Table	**Description**
Device	Stores a record for each device in WiSARDNet
Deployment	Stores a record for each deployment of a device
DeviceType	Stores a record for each possible type of device
DeploymentType	Stores a record for each type of deployment
Site	Stores geographic information about each garden site
Experiment	Stores a record for each WiSARDNet experiment

TABLE 4.1: A list of the database tables containing information about WiSARDs and their configurations.

The next step in this procedure is to filter all of the queried information against the search criteria that a user has specified. Once a list of WiSARDs that match the search criteria is obtained, the user can specify a new configuration to be applied to each WiSARD in that list. The software which obtains search criteria, translates information from database tables to Java objects, and filters WiSARDs is described in further detail in the Chapter 5.

4.3.1.2 Describing the New Configuration

A WiSARD's task execution is governed by a set of configuration parameters stored within the device's non-volatile memory; this area of memory is called the task control block. Changing a device's behavior is achieved by overwriting specific values within the task control block with values that represent different behaviors.

The user must describe the changes to the system so that the correct fields in the task control block can be overwritten with appropriate values corresponding to the new behaviors. This is achieved by encoding different behavior or behavior profiles into control sequences which the WiSARDs are able to decipher.

4.3.1.3 Validating the New Configuration

As the number of configuration changes and the number of affected WSN nodes increases, the possibility of errors or resource conflicts between experiments also increases. To prevent conflicts, there is a need to validate that the intended changes will not interfere with other experiments each node is associated with. This is achieved with user access control and command validation.

User access control is the process of restricting the access of users to specific pieces of hardware or software based on a set of rules and permissions that govern the extent to which they can interact with the WSN. For example, a researcher should not be allowed to reconfigure another researcher's hardware or reconfigure the WSN in a way that will adversely affect other experiments. User access control in WiSARDNet is achieved by using database tables to store information about users and the WiSARDs they are allowed to change.

Command validation is the process of analyzing the configuration changes a user intends to make to a specified set of WiSARDs and determining whether the change is feasible and safe to make. For example, a user should not be allowed to send a command that will cause a WiSARD to misbehave or prevent it from performing its essential duties. Both user access control and command validation are essential in validating a configuration before execution.

4.3.1.4 Command Synthesis

Once the user has specified a new configuration, command packets are automatically generated. A command packet is a structure which contains a control sequence, configuration parameters, and network routing information which will allow each command to be sent to its intended WiSARD. The control sequences are defined by software classes, the command parameters are obtained from the user's configuration specifications, and the routing information is obtained from the list of WiSARD objects. Once generated, the command packets are added to MQTT messages and flushed from an MQTT publisher client at the RTDC to the MQTT broker at the destination garden server. Each garden server has a WiSARD client which retrieves command packets from the local MQTT broker and sends them into the WSN via the hub node.

4.3.1.5 Executing the Reconfiguration

When a WiSARD receives a command message from a parent node, the WiSARD parses the command message and schedules a task to execute the command. When a WiSARD executes a command, a report message is created and sent back through the network to the RTDC. The report message will confirm that the task was successfully executed or report that it failed if the task did not execute successfully.

4.3.1.6 Verifying the New Configuration

All data generated by a WSN is sent to the RTDC in the form of data streams. Each transducer reports its samples as specific streams defined by the device relationship structure discussed in Chapter 3. A WiSARD also reports diagnostic and error information as data streams in the same manner. After making a configuration change, a WiSARD reports its new configuration through a designated data stream to the RTDC.

4.3.2 Software Modules

The software implementation of the functionality described in the previous section is grouped into three distinct modules:

- Wisard_Browser_Module
- Cmd_Generation_Module
- Validation_Module

Wisard_Browser_Module contains the functionality that can identify sets of WiSARDs that match a user's search criteria. Additionally, this module is responsible for providing an up to date view of a WSN. The functionality which allows a user to describe a new WSN configuration and signal those changes to the WSN is grouped into Cmd_Generation_Module. The functionality which validates the safety of a new configurations, checks for conflicts, and implements user access control into the system is grouped into Validation_Module. Figure 4.3 illustrates the new modules added to the WiSARDNet CI, as well as the existing software systems with which they communicate.

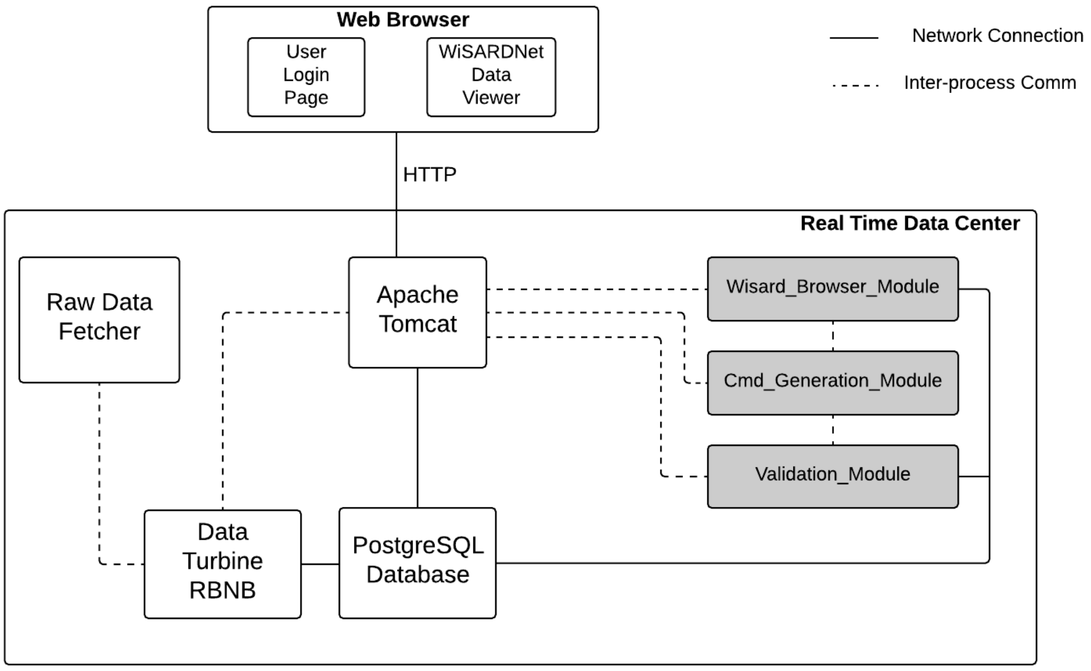


FIGURE 4.3: The software modules being added to the WiSARDNet cyberinfrastructure for WiSARD reconfiguration are shown in the shaded boxes.

Each software module contains separate functionality, but they interact and share information with each other as well as the other components of WiSARDNet. Wisard_Browser_Module interacts with the web portal user interface, it gathers information about the WSN nodes from the PostgreSQL database, and provides sets of WiSARDs to the other modules. Cmd_Generation_Module interacts with the WiSARDNet web portal to generate commands based on a user's input. Additionally, it accesses the list of WiSARDs that Wisard_Browser_Module produces. This module accesses the PostgreSQL database to enable stored behaviors to be turned into commands. Lastly, this module provides information to Validation_Module so that the commands can be verified. Validation_Module gets session variables from the web portal to authenticate users. It also takes in a set of commands from Cmd_Generation_Module, that can validate and report its results. To verify that the user is permitted to run a specified command, the user and command need to be compared against permission records stored in PostgreSQL database tables. These tables are discussed in detail in Chapter 5.

In summary, the functionality is grouped into three distinct software modules: Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module. These modules need to be able to interact with the different components of the WiSARDNet CI as well as with each other. The software implementations of the modules are discussed in Chapter 5.

Chapter 5

Software Module Implementation

5.1 Overview

This chapter explains how each of the three modules discussed in the previous chapter are implemented in software. First, Wisard_Browser_Module is discussed. Cmd_Generator_Module is discussed next, followed by Validation_Module. Validation_Module is discussed last because it has the broadest impact on the other modules and the system as a whole.

5.2 WiSARD Browser

Wisard_Browser_Module is a collection of software components which address four major objectives.

- Allow a user to specify WiSARD search parameters through a user interface

- Search the WiSARDNet PostgreSQLdatabase for WSN state information
- Produce a set of WiSARD data objects matching the search criteria specified by the user
- The module should integrate with the existing WiSARDNet software

Figure 5.1 shows the flow of execution for this module. Each procedure shown is described in this chapter.

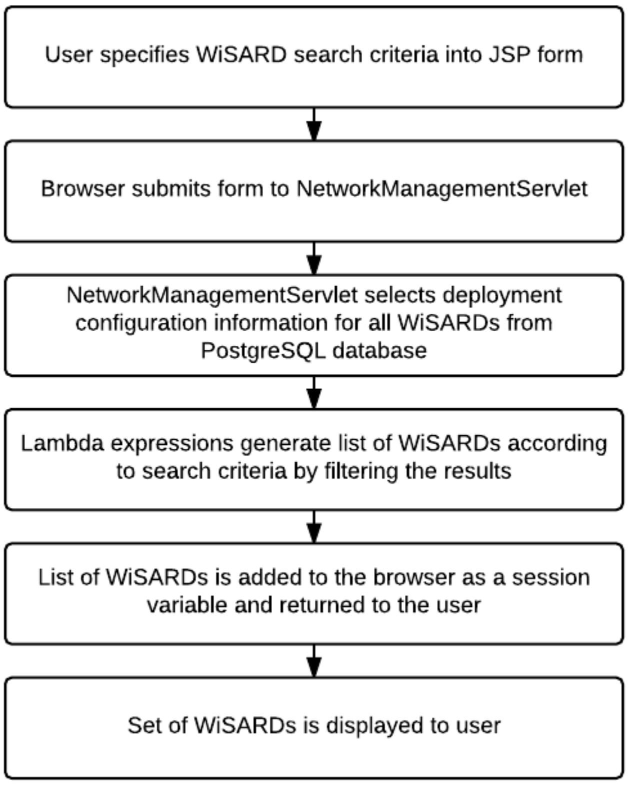

User specifies WiSARD search criteria into JSP form
Browser submits form to NetworkManagementServlet
NetworkManagementServlet selects deployment configuration information for all WiSARDs from PostgreSQL database
Lambda expressions generate list of WiSARDs according to search criteria by filtering the results
List of WiSARDs is added to the browser as a session variable and returned to the user
Set of WiSARDs is displayed to user

Figure 5.1: A flow diagram showing the actions performed by Wisard_Browser_Module to obtain a list of WiSARDs based on user specified search criteria.

5.2.1 Obtaining the User's Search Criteria

A user of this system needs an interface that allows them to send and receive information to Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module. To create this interface, JSP (JavaServer Pages) and Java servlet technologies are both used. A JSP page presents information to a user through a web browser and gathers user input necessary for back-end operations. The JSP page sends information to and from a servlet called NetworkManagementServlet, which is a java class that implements methods that respond to Get and Post requests from a web browser. Because NetworkManagementServlet is an instance of a Java class, it has the ability to interact with instances of other classes. JSP/servlet interaction is the communication pipeline which allows the user to interact with these three software modules. Figure 5.2 illustrates how NetworkManagementServlet is used to send and receive information between a user's browser to the appropriate software module.

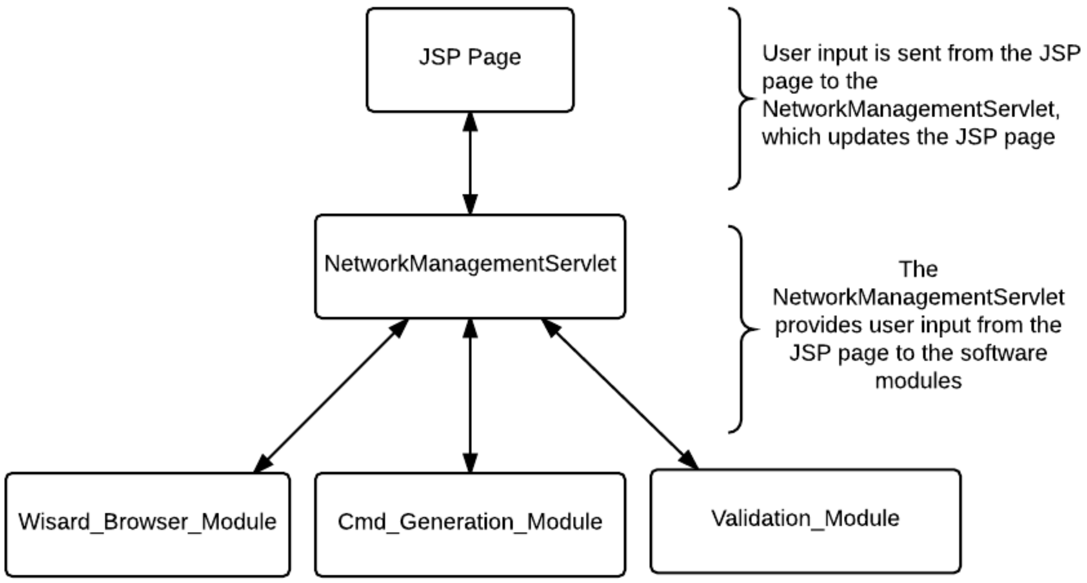

JSP Page
User input is sent from the JSP page to the NetworkManagementServlet, which updates the JSP page
NetworkManagementServlet
The NetworkManagementServlet provides user input from the JSP page to the software modules
Wisard_Browser_Module
Cmd_Generation_Module
Validation_Module

FIGURE 5.2: NetworkManagementServlet relays information between the user interface and Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module.

There are nine attributes that the WiSARDs can be against in the selection process.

- Garden Site
- Garden State
- WiSARD role
- SP type
- Associated Experiment
- Transducer type
- Deployment Type

- WiSARD Network ID
- Device Serial ID

A JSP web page provides a simple interface for the user to populate an HTML form. Figure 5.3 shows the interface which allows a user to specify the different search criteria. Each criteria has an information button in the user interface that describes the selected option in detail. Figure 5.4 shows a screen capture of the user interface with one of the information buttons.

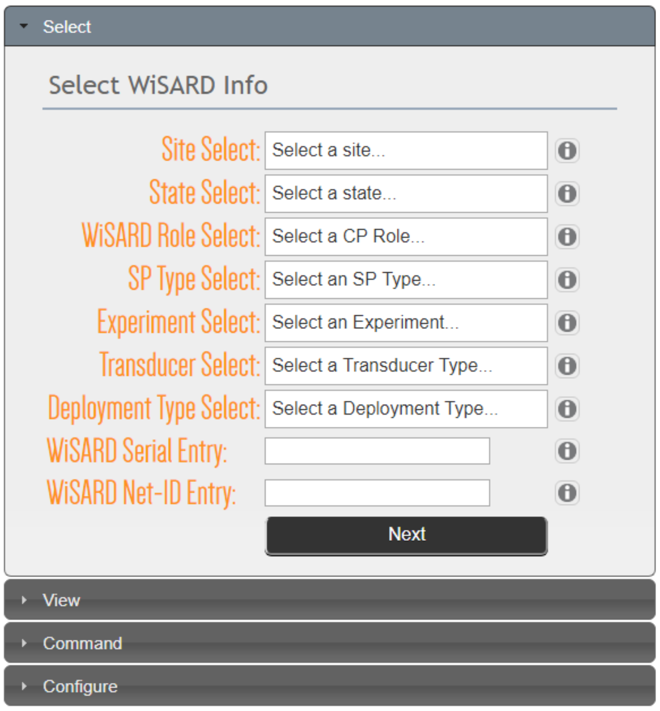
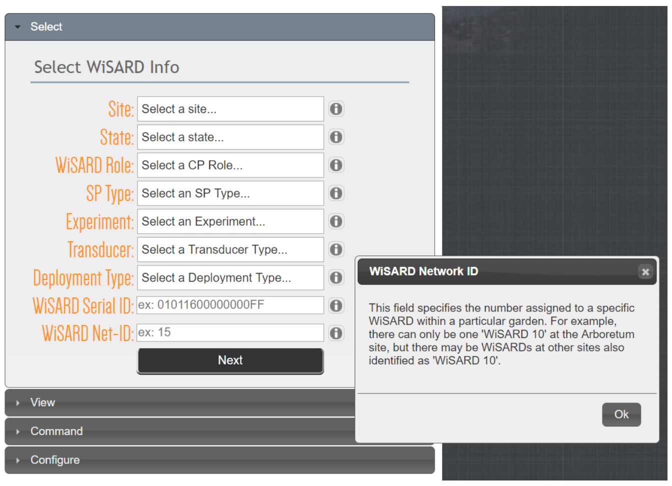


FIGURE 5.3: A screen capture of the Wisard_Browser_Module user interface page where a user specifies WiSARD search criteria. Wisard_Browser_Module queries the PostgreSQL database and creates WiSARD objects. The objects are then filtered against those search criteria to obtain a selection of WiSARD objects for a user to reconfigure.


Select
Select WiSARD Info
Site: Select a site...
State: Select a state...
WiSARD Role: Select a CP Role...
SP Type: Select an SP Type...
Experiment: Select an Experiment...
Transducer: Select a Transducer Type...
Deployment Type: Select a Deployment Type...
WiSARD Serial ID: ex: 01011600000000FF
WiSARD Net-ID: ex: 15
Next
View
Command
Configure
WiSARD Network ID
Ok

This field specifies the number assigned to a specific WiSARD within a particular garden. For example, there can only be one 'WiSARD 10' at the Arboretum site, but there may be WiSARDs at other sites also identified as 'WiSARD 10'.

FIGURE 5.4: A user can click on the information buttons next to each search criteria. Each button opens a box which describes the search criteria.

These attributes are static or dynamic pieces of information that together represent the current state of the WSN nodes. Traditionally, this information can be obtained from a database query that can be written into a function or a method. Researchers and other users of WiSARDNet often have changing or new experiment requirements. It is important for this software to be flexible and support changes to the WiSARDs and their configuration information. For example, if new WiSARD features are added, resulting in new fields in a database table or entirely new tables, the new features should not prompt extensive modification of the existing code. A new field or table would simply translate into new member variables in the WiSARD class.

To implement the querying functionality for Wisard_Browser_Module that was discussed in 4.3.1.1, there were two possible design alternatives for the filtration that would result in a list of WiSARDs: PostgreSQL or Wisard_Browser_Module application code. PostgreSQL supports a wide variety of query features that enable sophisticated data processing and filtration of queried data sets. If achieving the fastest possible query speed was of great importance for this module, then all of the searching and filtration of WiSARDs from the user's search criteria should be handled by complex PostgreSQL queries. While query time is important, the benefits that can be gained at the sacrifice of processing speed make the other alternative desirable. Chapter 2 briefly explained some of the ways in which modern software development techniques such as behavioral parameterization can add functionality to software. For Wisard_Browser_Module, a functional data processing approach allows for development flexibility at the expense of performance overhead. This does not impact the performance of the system in a significant way because the amount of WiSARD deployment configuration information compared to the total amount of data stored in the database is trivial.

WiSARD objects translate from fields in database tables to software entities through the use of intermediate Java classes. A Java class with member variables that correspond to the different attributes that define a WiSARD's configuration, and methods to translate higher-level queries into database queries and results provides a foundation for objects to be created and passed from module to module. To achieve the desired flexibility, Wisard_Broswer_Module performs a single fetch to the database and retrieves all of the state information for all of the WiSARDs. It then formats the information from the query into member variables of WiSARD objects before adding them to a list data structure. All of this functionality is built into WiSARD class methods.

Once NetworkManagementServlet has obtained a list of all WiSARD objects, it then filters that list against the attributes which the user specified in his/her form submission. Utilizing functional data processing techniques supported by lambda expressions in Java 8, obtaining a list that matches a user's search criteria is simple and flexible. Below is an example of the way in which lambda expressions are used in this software to produce powerful results with a very small amount of code.

```
return wisards.where((Wisard w) -> ''Arboretum''.equals(w.getSite()));
```

This statement returns an ArrayList named `wisards` that contains WiSARD objects. The `where()` method iterates over every entry in the ArrayList and only returns those not filtered out against the logic in the lambda expression. The lambda expression in this example compares the WiSARD site against a string containing the site name `Arboretum`. Any WiSARD in the ArrayList that does not have `Arboretum` as its site name will not be added to the returned ArrayList.

With one line of code, a data structure of WiSARD objects can be filtered based on its member variables that would otherwise have required a specific database query. Additionally, values can be added to the same expression to create more complex data filtration without requiring that the underlying functionality be altered. This functionality provides a quick and versatile way to support changing project requirements for managing networks of WiSARDs without being restricted by specific database queries.

When a list of WiSARDs has been filtered to match a user's search, NetworkManagementServlet appends it to the session as a variable where it is accessible to the JSP page at the user's browser. The JSP page displays the WiSARDs and their deployment information for the user. From this point a user can decide to continue with reconfiguring the WiSARDs that this search has returned, or return to the form and specify a different set of WiSARDs. Should the user choose to continue on with the set of WiSARDs they have selected, the data structure is accessible within the browser session, ready to send to Command_Generation_Module.

5.3 Command Generation

Cmd_Generation_Module and Wisard_Browser_Module, are similar in that they both rely on user input being sent from the user interface to NetworkManagementServlet. The user will utilize the same JSP page's user interface as Wisard_Browser_Module to specify the change he/she would like to apply to the selected WiSARDs (Figure 5.5). Each menu option corresponds to a different configuration change that can be selected. Different commands require the user to specify certain parameters which are necessary to describe the new configuration. Depending on which command the user selects, the user interface will present different fields to enter the appropriate parameters. Figure 5.6 shows the fields that are shows when valve actuation command are selected.

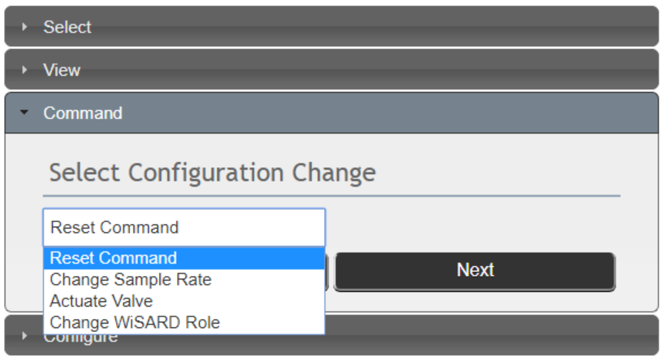


FIGURE 5.5: A screen capture of the Cmd_Generation_Module user interface that allows a reconfiguration command to be chosen. Once one is selected, Cmd_Generation_Module creates command objects for each WiSARD selected by the user. Each command object synthesizes a command packet that will produce the desired reconfiguration.

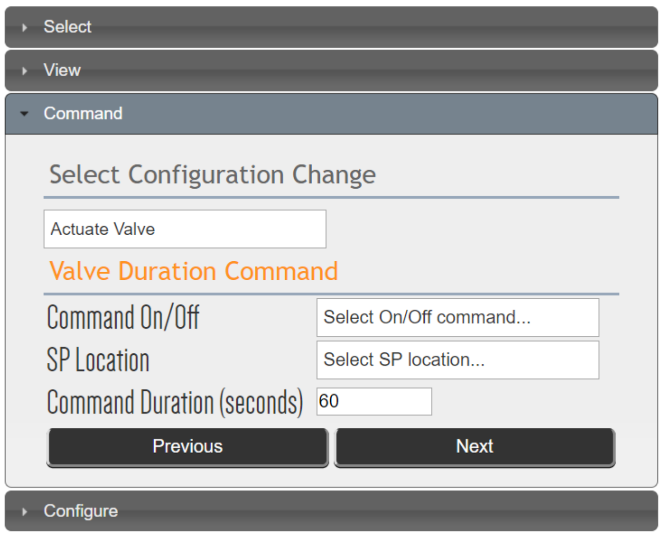


FIGURE 5.6: Each command class requires different parameters be specified by the user. A JavaScript function in the user interface renders different HTML fields in the user interface depending on the configuration change that the user selects. This figure shows the fields that are rendered when the Actuate Valve command is chosen.

Once a user has selected an option, the process of synthesizing the appropriate commands can proceed. Figure 5.7 shows the flow of execution for this module.

Each procedure shown in the diagram is described in this section.

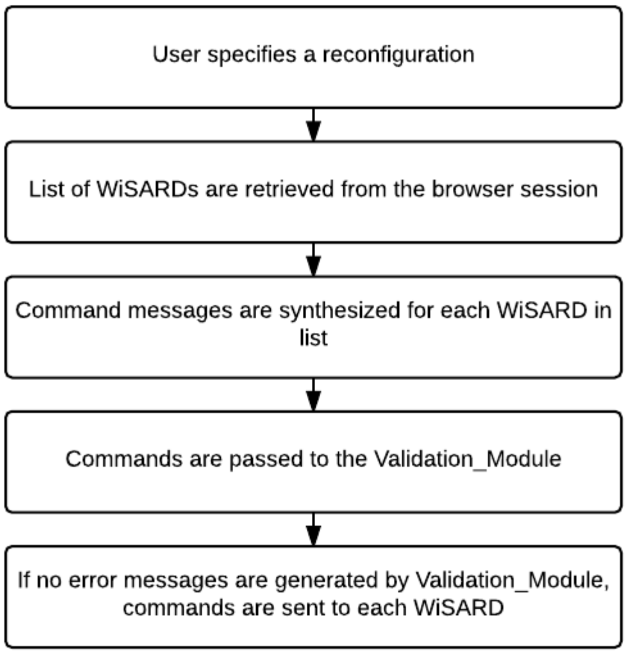

User specifies a reconfiguration
List of WiSARDs are retrieved from the browser session
Command messages are synthesized for each WiSARD in list
Commands are passed to the Validation_Module
If no error messages are generated by Validation_Module, commands are sent to each WiSARD

FIGURE 5.7: A flow diagram showing each procedure Cmd_Generation_Module makes to synthesize WiSARD command packets that will perform a user specified reconfiguration.

To perform a reconfiguration in WiSARDNet, a specific command is generated and sent to each WiSARD that the change will affect. A command in the WiSARDNet communication protocol consists of multiple components. A hub address and a destination address describe which WSN the command should go to, and which WiSARD in the WSN that the command is intended for.

These potential reconfigurations include a soft reset on the device, updating a sampling rate, changing a software role or ID, or actuating a water valve. The list of commands that are possible with a WiSARD continues to grow with each new experiment and feature that WiSARDNet supports. For this reason, the of the design of command synthesis is implemented in a way that provides flexibility for future developments. In WiSARDNet, a command consists of a payload, an expiration, and a checksum. The payload is the byte stream which contains the command and its parameters the WiSARD will interpret and execute. The expiration is a value which the sender sets to prevent out of date commands from being executed. For example, if a command for some reason takes an hour to traverse the network, it may no longer be relevant and should not be executed. The checksum allows the destination WiSARD to verify that it received the command correctly. Each of these command components are encapsulated within a command class named NetManagementCommand.

The class definition for NetManagementCommand provides the member variables and methods for every command type. However, certain commands need additional parameters which aren't provided in NetManagementCommand's class definition. For this reason, a specific class is defined for each command type. Each command class uses inheritance to extend the member variables and methods provided by NetManagementCommand. Table 5.1 lists each of the command classes that have been implemented and the reconfiguration change that they make.

Command Class	Operation
SetSamplingRate	Adjusts the sampling rate of all transducers of a given type on a WiSARD
ResetCommand	Performs a soft reset on a WiSARD
ActuateValve	Actuates a latching solenoid water valve to an open or closed state
ChangeRole	Changes the software role of a WiSARD

TABLE 5.1: A list of each of the command classes that have been implemented.

To perform a reconfiguration, a command object is instantiated from the command class definition that corresponds to the user's specified configuration. The change that the user specifies will be enacted for all of the WiSARDs in the list that was added to the session by NetworkManagementServlet. Cmd_Generation_Module loops through each WiSARD and chooses the correct command parameters to include in the payload. Once the command objects have been successfully generated they are sent to Validation_Module in a data structure, similar to the way that Wisard_Browser_Module provides the WiSARD list to Cmd_Generation_Module.

5.4 Validation

Before commands can be sent out to reconfigure WiSARDs, there are two sets of checks which the commands must pass. First, Validation_Module confirms that the user that submitted the commands has appropriate permissions to reconfigure the selected WiSARD nodes. Second, the module assess the commands for their impact on the system, and ensures that none of the changes will cause a WiSARD to misbehave. WiSARDs may have hardware that is shared between multiple experiments, and reconfiguring that WiSARD could have adverse effects on one or more experiments. The procedures that this module performs are shown in Figure 5.8. Each of these procedures is described in this section.

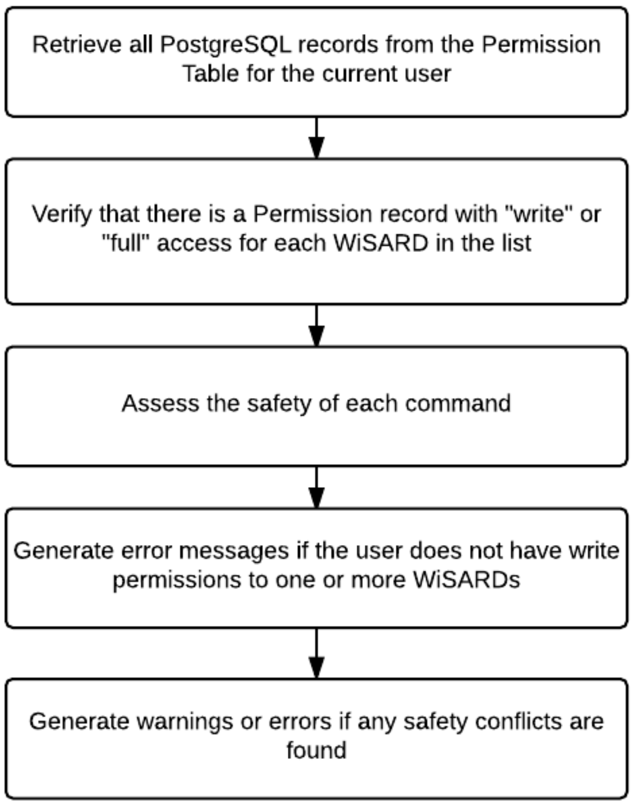

Retrieve all PostgreSQL records from the Permission Table for the current user
Verify that there is a Permission record with "write" or "full" access for each WiSARD in the list
Assess the safety of each command
Generate error messages if the user does not have write permissions to one or more WiSARDs
Generate warnings or errors if any safety conflicts are found

FIGURE 5.8: A flow diagram showing each procedure taken by the Validation_Module to validate commands and user access prior to the reconfiguration of WiSARDs.

5.4.1 Permissions and User Access Control

The PostgreSQL database schema prior to this thesis had certain tables which could have been used for user access control, but the schema was not robust. To implement user access control in a flexible way, a better approach was needed. Two core abstractions of the existing database schema are used to design the permissions system for WiSARDNet. The first abstraction is that anyone to whom a permission might be granted is considered a permission entity. The second abstraction is that anything which would need its access restricted is to be considered a permission resource. After applying these two abstractions to the database, the Person and Organization tables are defined as permission entities. Likewise, the Deployment, Site, and Experiment tables are defined as deployment resources. With these two abstractions, granting permission simply becomes a mapping between a permission entity and a permission resource with a value designating the level of access the entity has to the resource.

Two access levels are used to denote permissions to resources: read and write. Table 5.2 describes what each permission level allows. Note that an entity with write access can override changes made by other entities with write access, but a warning message is generated to inform the user of the scenario. In this case, the user should proceed at their own discretion, and the permissions should be granted by an administrator with this in mind. In the greater context of the user access control paradigm, additional access levels could be added to allow delete privileges. Even though that use case doesn't directly correlate to the reconfiguration software, the paradigm still supports that feature for other applications. These two permission levels are sufficient to regulate how users interact with the WiSARDs.

This approach to user access control also provides a flexible way to manage other hardware or software resources which might be incorporated into WiSARD-Net in the future. If a new type of resource is added to the system which is described by a new table, then adding a foreign key constraint from Permission-Resource to the new table is all that is needed for a record in Permission to grant permissions to that new PermissionResource.

Permission Level	Description
read	An entity can view a resource and all information regarding its deployment configuration
write	An entity has access to modify and reconfigure a resource

Table 5.2: A description of each access level an entity can be granted to a resource.

To implement user access control with this approach, three tables were added to the database. First, a table called PermissionEntity was created to reference the Person and Organization tables. Next, a table called PermissionResource was created to reference the Deployment, Experiment, and Site tables. Lastly, a table called Permission was created with foreign key constraints that reference the PermissionEntity and PermissionResource primary keys. Each record in Permission specifies an access level value that a PermissionEntity record has to a PermissionResource record. Figure 5.9 shows these tables and their foreign key constraints. Each foreign key constraint references the primary key of the table that the arrow points to. The unnamed table on the right shows how the schema can be used to manage new resources in the future.

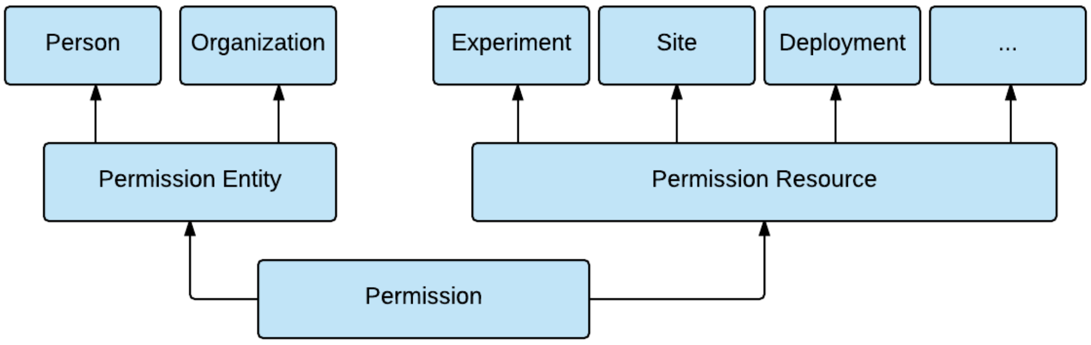


FIGURE 5.9: A diagram that shows how the three new permission tables are used to create a user access control schema.

When Validation_Module receives a data structure of command objects from the command generation module, the software in Validation_Module first checks that there are no user access violations in any of the commands. This is done by iterating over each command object and performing a lookup to the permission table. The objective is to gather all of the Permission records whose permission entity matches the user that created the commands, and verify that a permission resource with write access is present, for the WiSARD in question. If the user is found to have insufficient permissions to execute a command, an error message is generated and added to a queue of messages that is displayed once Validation_Module has iterated over every command. All commands with insufficient permissions are rejected, and will not be sent to the WiSARDs.

The validation logic for the reconfiguration software currently grants permissions to WiSARDs. However, the database schema supports a more granular level of detail. A record in the permission resource table could reference the deployment of a device, meaning that access to a WiSARD could be restricted down to individual transducers.

5.4.2 Safety Validation

If all of the commands pass the user access permission checks, then the module performs a configuration safety test on each command. The difficulty of the safety check comes from the different types of commands that can be executed, and the diversity of experiments and their needs. It is for this reason that the command class was designed such that each command type will implement its own safety logic. This way, when a new command is developed, the designer can build in a method which will perform a safety check within the context of that method. Figure 5.10 shows the structure of these command classes and how they inherit and override behaviors.

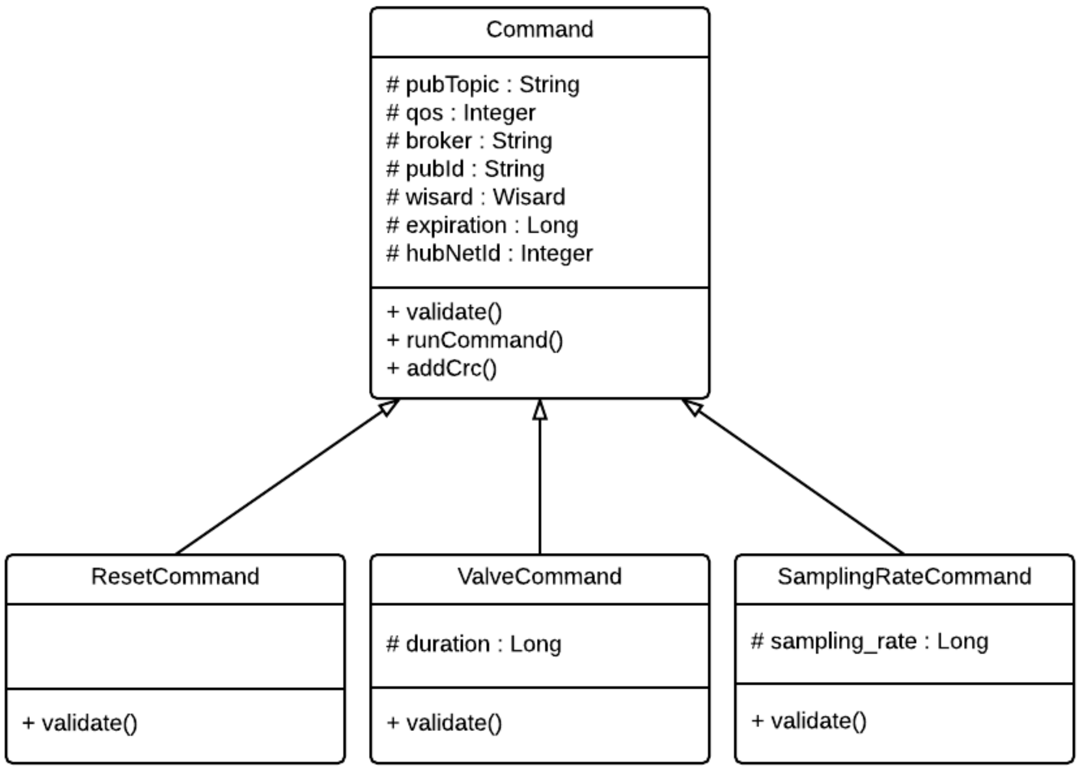


FIGURE 5.10: A class diagram showing how new commands inherit from their parent class, yet implement their own validate method.

A practical example of a safety validation method would be a case where a particular command is designed to increase the sampling rate of a particular transducer type on a WiSARD. Here, an appropriate safety check would ensure that the new sampling rate is a reasonable value. For example, if the user enters a non-numeric value, a negative value, or a value corresponding to a sampling rate which the WiSARDs cannot physically perform, those should result in a failed safety check where the command is rejected. Another example is a command to reconfigure a WiSARD that one or more experiments are using. If a configuration change is specified for a WiSARD associated with an experiment, a warning message is generated and added to a queue that is shown to the user. In any scenario where multiple users have write access to a WiSARD, the command is not rejected but the user will be warned of the conflict through a message.

5.5 Summary

Each of the three software modules developed for this thesis serves a distinct role in enabling WiSARD reconfiguration. Wisard_Browser_Module is a flexible module that enables users to search for and select a group of WiSARDs based on a range of network state information. Cmd_Generation_Module enables a user to select a change he/she would like to make to the set of selected WiSARDs, and automatically generate the commands necessary to implement those changes. Validation_Module performs user access control and network safety checks to ensure that all reconfigurations are made in a way that eliminates conflicts. The design and implementation of these three modules provides great flexibility to developers who will work on the WiSARDNet platform in the future.

Chapter 6

Automated Reconfiguration

6.1 Overview

This chapter discusses automated network reconfiguration and how it is supported by the work of this thesis. First, the concept of automated reconfiguration and the motivation for its use is described. Then the design and implementation of the automated reconfiguration software in the work of this thesis is described. Finally, an example is shown that demonstrates the practical uses of this software.

6.2 Requirements

The work in this thesis allows network reconfiguration through user interaction. User interaction can sometimes be inadequate because of slow response time, a lack of understanding of system behavior, or human error. Creating a system that can automatically respond to incoming sensor data (or calculations derived from incoming sensor data) by reconfiguring the network overcomes many limitations encountered by requiring user input.

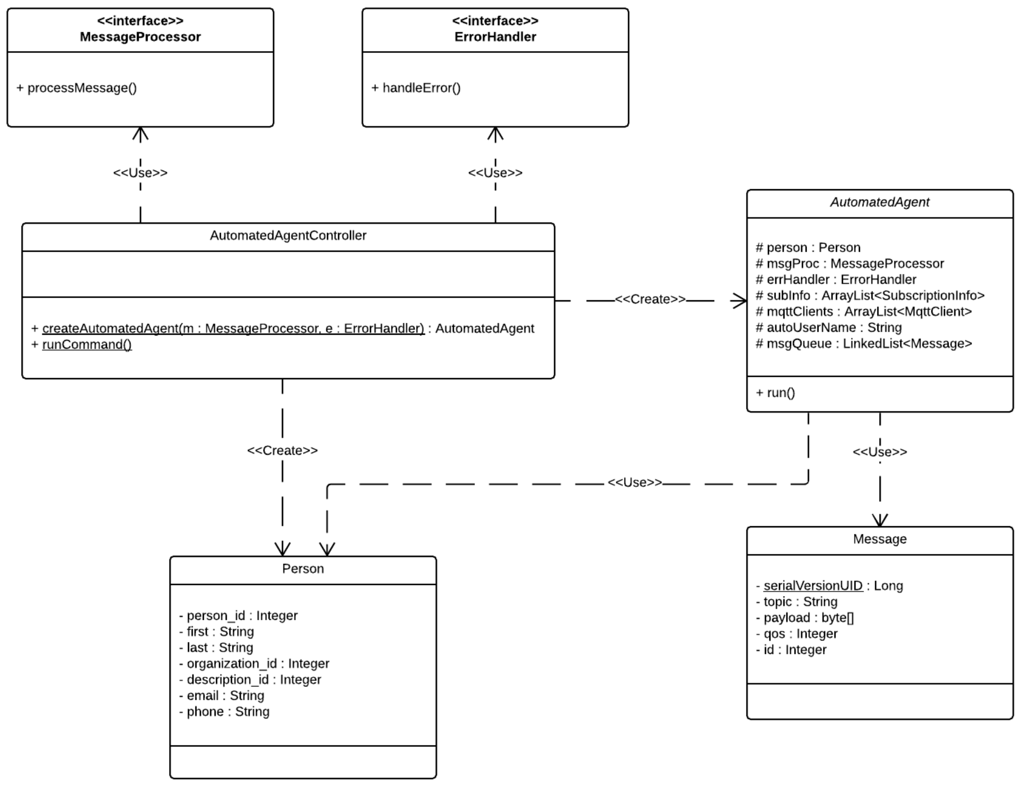
The software applications designed in this work to autonomously monitor and reconfigure WiSARDs are called automated agents. Agents must fulfill three main requirements. First, the agents must adhere to the user access and safety validations that govern the rest of the system. More specifically, the developers of automated agents should not be allowed to intentionally or unintentionally bypass, improperly implement, or override the security provisions. Second, they must be able to receive and handle WiSARD data streams as well as any exceptions that may occur. WiSARD data streams are accessible to applications by using an MQTT subscription client. These data streams are called messages. Finally, the agents should be able to interact with the other CI systems. For instance, an agent should be able to interact with the PostgreSQL database using the query methods that the WiSARD class provides.

6.3 Design

The automated agents are defined by a Java class designed to address these requirements. The first requirement of preventing agents from circumventing system security is solved by separating the validation logic from the Agent Class. If the validation logic is external to the Agent Class, then agents will not be able to dictate their own security privileges. The second requirement of receiving and handling data streams and handling exceptions specific to each agent is solved by the design decision that each agent has message handling logic and exception handling logic that can be specific to that agent. The final requirement of allowing an agent to interact with the other parts of the CI is solved by using the functionality in Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module. For instance, an agent can use the WiSARD Class discussed in Chapter 5 as a means of interacting with the database.

To implement this design, the AgentController Class is used and encapsulates much of the functionality described in the previous chapters, primarily validation and safety. The controller class will log in an agent, validate that it has sufficient privileges to access the data streams it uses, and then create an agent object. The agent class implements the Java runnable interface, allowing the agent to be put into a thread and executed. The design and implementation of the agents is generalized, so that there is common code from agent to agent. Every agent uses the same logic to listen for and handle incoming messages. Message processing and error logic that distinguishes one agent from another are passed to the agent as parameters in the form of lambda expressions (anonymous functions).

Using inheritance in object-oriented software design is typically a good way to reuse code. Inheritance has been used extensively in the code described in the previous chapters. However, in the case of designing the automation behavior, using aggregation over inheritance resulted in a cleaner design. Aggregation is the object-oriented design concept where the functionality of an object is determined by the aggregation of other objects, as opposed to implementing or inheriting the functionality directly. This approach is accomplished with the use of two interfaces referred to as the MessageProcessor and the ErrorHandler. These interfaces allow the operational logic for each agent to be passed in as lambda expressions to an Agent object during its creation. Figure 6.1 is a UML class diagram that shows how agents are implemented using aggregation.



<<interface>> MessageProcessor
+ processMessage()
<<interface>> ErrorHandler
+ handleError()
<<Use>>
<<Use>>
AutomatedAgentController
+ createAutomatedAgent(m : MessageProcessor, e : ErrorHandler) : AutomatedAgent
+ runCommand()
<<Create>>
AutomatedAgent
# person : Person
# msgProc : MessageProcessor
# errHandler : ErrorHandler
# subInfo : ArrayList<SubscriptionInfo>
# mqttClients : ArrayList<MqttClient>
# autoUserName : String
# msgQueue : LinkedList<Message>
+ run()
<<Create>>
<<Use>>
<<Use>>
Person
- person_id : Integer
- first : String
- last : String
- organization_id : Integer
- description_id : Integer
- email : String
- phone : String
Message
- serialVersionUID : Long
- topic : String
- payload : byte[]
- qos : Integer
- id : Integer

FIGURE 6.1: A UML class diagram showing the classes and interfaces necessary to instantiate AutomatedAgent objects. The arrows describe the relationships between the four different classes as well as the two interfaces.

AutomatedAgentController, AutomatedAgent, Person, and Message are each a class. MessageProcessor and ErrorHandler are functional interfaces. A functional interface defines a type for a lambda expression. This is necessary so that variables can be created to reference lambda expressions. The text on each of the arrows describes the relationship between the classes and interfaces used in the automation software. The AutomatedAgentController class `<<uses>>` the MessageProcessor and ErrorHandler interfaces to store lambda expressions that implement the logic for processing messages and for handling errors, respectively.

The AutomatedAgentController class `<<create>>` instances of both the Person class and the AutomatedAgent class. An AutomatedAgentController takes in the two lambda expressions, creates a Person object, and passes those along into the constructor of AutomatedAgent class to create new agents.

Each AutomatedAgent object can then use its Person object, the functionality in the Message class, and whatever lambda expressions that were passed in as the MessageProcessor and ErrorHandler. A detailed example of the software classes showing how this software works is described in the next section.

6.4 Example Agent

Using the design described in the previous section, there are many different ways that an automated agent can be used to manage WiSARDs. The following code snippets demonstrate how this software is used in practice by showing the class definitions for AutomatedAgentController and AutomatedAgent, the definitions for the MessageProcessor and ErrorHandler functional interfaces, as well as an example program which demonstrates how to use these classes.

Below is a simplified class definition for the AutomatedAgentController class that handles authentication, creation, and execution of the agents.

```
/*
 * class definition for automated user controllers which create, authenticate,
 * and return automated user objects
 */
public class AutomatedAgentController{
   public static AutomatedAgent createAutomatedAgent(String username, String password, ArrayList<
     ↪ SubscriptionInfo> subscriptions, MessageProcessor msgProc, ErrorHandler errHandler){
      // logs in user, validates permissions to subscriptions, creates person and
     ↪ automated user
   }

   public static runCommand(NetManagementCommand nmc, AutomatedAgent agent){
      // validates user's permission to run command and performs safety validation
      nmc.runCommand();
   }
```

The code snippet shown below contains the definitions for the functional interfaces that allow lambda expressions to be passed to the agents that define their behavior. A type is a notation that specifies how variables or objects should be interpreted and used. As described in the previous section, functional interfaces specify a type that allows lambda expressions to be saved to variables and used by the agents. In the case of these interfaces, the types are MessageProcessor and ErrorHandler.

```
/*
 * Classes who implement this interface must define a method to handle
 * messages
 */
public interface MessageProcessor{
   public void processMessage(Message msg);
}

/*
 * Classes who implement this interface must define a method to handle
 * exceptions
 */
public interface ErrorHandler{
   public void handleError(Exception e);
}
```

The snippet shown below is the class definition for AutomatedAgent. In the constructor, all necessary information to establish connections to MQTT message brokers as well as define the agent's behavior is passed in as a set of parameters. The method `run()` establishes its subscriptions with the MQTT brokers it will be listening to, and then waits for messages to arrive. Once messages arrive, they are given to `processMessage()` whose functionality was passed into the constructor as a lambda expression. Additionally, any exceptions are passed to `handleError()` whose functionality was also passed to the constructor as a lambda expression.

```
/*
 * class definition for automated users which encapsulates a person and message
    ↪ processor
 * objects, as well as an array list of subscription objects. This will allow
    ↪ automated
 * agents to monitor data streams and enact changes upon the WiSARDs they have
    ↪ sufficient
 * permissions for
 */
public abstract class AutomatedAgent implements runnable, MqttCallback{
   protected Person person;
   protected MessageProcessor msgProc;
   protected ErrorHandler errHandler;
   protected ArrayList<SubscriptionInfo> subInfo;
   protected ArrayList<MqttClient> mqttClients;
   protected String autoUserName;
   protected LinkedList<Message> msgQueue;

   public AutomatedAgent(Person p, ArrayList<SubscriptionInfo> s, String autoUserName,
    ↪ MessageProcessor m, ErrorHandler e){
      this.person = p;
      this.msgProc = m;
      this.errHandler = e;
      this.subInfo = s;
      this.autoUserName = autoUserName;
      this.mqttClients = new ArrayList<MqttClient>();
      this.msgQueue = new LinkedList<Message>();
   }

   public void run(){
      try{
         // create connections for all subscription clients and add to an ArrayList
         for(SubscriptionInfo subscription : subInfo){
            MqttConnectOptions connOpts = new MqttConnectOptions();
            connOpts.setCleanSession(false);
            connOpts.setConnectionTimeout(180);
            MqttClient subClient = new MqttClient(subscription.broker, subscription.gsCommonName
    ↪ + "/" + autoUserName);
            subClient.subscribe(subscription.subTopic);
            mqttClients.add(subClient);
         }

         // while thread is running, listen for messages
         while(!Thread.currentThread().isInterrupted()){
            synchronized(msgQueue){
               Message m = msgQueue.peekFirst();
               if(m != null){
                  msgProc.processMessage(m);
                  msgQueue.removeFirst();
               }
            }
         }
      } catch(Exception e){
         errHandler.handleError(e);
      }
   }

   @Override
   public void connectionLost(Throwable arg0) {
      // empty
   }

   @Override
   public void deliveryComplete(IMqttDeliveryToken arg0) {
      // empty
   }

   @Override
   public void messageArrived(String arg0, MqttMessage arg1){
      synchronized(msgQueue){
         msgQueue.add(new Message(arg0,arg1));
      }
   }
```

The final code snippet below shows a simple example of how to utilize the classes defined above in an agent that automatically reconfigures a WiSARD. This example program creates an agent that toggles between two different sampling rates every 12 hours. The `main()` method shown at the bottom of this example uses the AutomatedAgentController class to instantiate an AutomatedAgent object with the `createAutomatedAgent()` method. Note that no instance of the AutomatedAgentController class created. This is because the method is declared to be static. Likewise, `runCommand()` is also static. The Java Virtual Machine (JVM) allows static methods to be called without creating an instance of the class that defined them.

```
public class MyAutomatedAgent TimerTask{
   public static volatile long interval = 300; // 5 min default sampling rate
   public static volatile AutomatedAgent agent = null;
   public static volatile Wisard wisard;
   public static volatile int hub;
   public static volatile byte taskID_hi;
   public static volatile byte taskID_lo;
   public static volatile byte fieldID;

   // The run method for the example program
   public void run(){
      toggleInterval();
      scheduleTask(12);
   }
```

```
    // A method which generates and executes the commands for the reconfiguration
    public void toggleInterval(){
       interval = interval == 60 ? 300:60; // toggles between 1 and 5 minute rates
       SetSamplingRate cmd = new SetSamplingRate(interval);
       AutomatedAgentController.runCommand(cmd, agent);
    }

    // A method which schedules reconfiguration events
    public static void scheduleTask(int numberOfHours){
          // get current time
          LocalDateTime currentTime = LocalDateTime.now(Clock.systemUTC());
          Calendar calendar = Calendar.getInstance();

          // get target date time
          calendar.setTime(currentTime);
          calendar.add(Calendar.HOUR_OF_DAY, numberOfHours);
          LocalDateTime target = calendar.getTime();

          // schedule task
          Timer timer = new Timer();
          timer.schedule(new MyAutomatedAgent(), target);
    }

    public static void main(String[] args){
       // check that all input arguments were given
       if(args.length < 6){
          throw new IllegalArgumentException("You must provide site, wisard ID, hub ID, taskID_hi
 ↪ , taskID_lo, and fieldID");
       }

       // store the passed in command line arguments
       String site = args[0];
       int wisardID = Integer.parseInt(args[1]);
       try{
          wisard = Wisard.getByID(siteID, wisardID);
       } catch(Exception e){
          throw new IllegalArgumentException("WiSARD " + wisardID + " was not found at site " +
 ↪ site);
       }
       hub = Integer.parseInt(args[2]);
       taskID_hi = Integer.parseInt(args[3]);
       taskID_lo = Integer.parseInt(args[4]);
       fieldID = Integer.parseInt(args[5]);

       // empty list of subscriptions since this open-loop example does not care about
 ↪ sensor readings
       ArrayList<SubscriptionInfo> subscriptions = new ArrayList<SubscriptionInfo>();

       // an example of how to use the AutomatedUserController to create an
 ↪ AutomatedUser
       agent = AutomatedAgentController.createAutomatedAgent("username", "password", subscriptions,
 ↪  (Message msg) -> {
          // Do Nothing - this open-loop example does not care about sensor reading
 ↪ values and is never called
       }, (Exception e) -> {
          // prints exception to std error
          System.err.println(e);
       });
       new Thread(agent).start();

       // schedule the first reconfiguration event
       scheduleTask(12);
    }
}
```

6.5 Greenhouse Experiment

To demonstrate the ability of the automated agent framework described in this chapter, an experiment was devised and an agent created to automatically manage the experiment parameters.

6.5.1 Setting up the Experiment

A WiSARD network was deployed to a site at Northern Arizona University. This site is located within a greenhouse containing buckets with various soil types. Each bucket has holes in the bottom along with a mesh fabric, allowing water to drain out of the buckets, but not the soil. Over each bucket is a water line with a 1/2 gallon per hour drip emitter that regulates the amount of water a bucket receives from a water line. A WiSARD with a SP-CM-STM controls the watering events by actuating a latching solenoid water valve. Additionally, a separate WiSARD with a Decagon 5TM transducer is monitoring the dielectric permittivity of the soil at a depth of 20cm. Dielectric permittivity is an important measurement that can be used to determine the volumetric water content (VWC) of a soil. Finely ground cinders from Northern Arizona were chosen as the soil for this experiment.

The objective of the experiment is to maintain the dielectric permittivity at a depth of 20cm between an upper and a lower threshold. The Decagon 5TM transducer measure dielectric permittivity in a raw version which needs to be converted to the true dielectric permattivity. The conversion is

$$\varepsilon_a = \varepsilon_{raw}/50$$

The Decagon 5TM can measure raw dielectric permittivity on a scale of 1 (completely dry) to 4000 (completely wet). To maintain the maximum sampling resolution, the control is maintained by acting upon raw transducer measurements as opposed to the final values for dielectric permittivity. The agent controls the levels by turning the water valve on and off such that the dielectric permittivity is kept between 400 and 520. Since the transducer is buried at a depth of 20cm, there is a latency between when the moisture exits the waterline and when it is sampled by the transducer. There is also network latency both when sending a command to a WiSARD and receiving a sample value from the WiSARD. For these reasons, threshold values of 410 and 480 were chosen to compensate for the described latencies. The lower threshold requires less compensation than the upper threshold due to slow rate at which it dries.

6.5.2 Agent Implementation and Results

The specific agent used to control this experiment follows the framework described in the previous section. The control logic which this agent follows is shown in 6.2. The agent is a multi-threaded command line Java program. The main thread of the agent program creates the agent object using the classes and methods described in the previous sections of this chapter. The agent is then executed in a separate thread; it receives and processes messages from the sampling WiSARD and sends out reconfiguration commands using the ActuateValve command class mentioned in chapter 5.

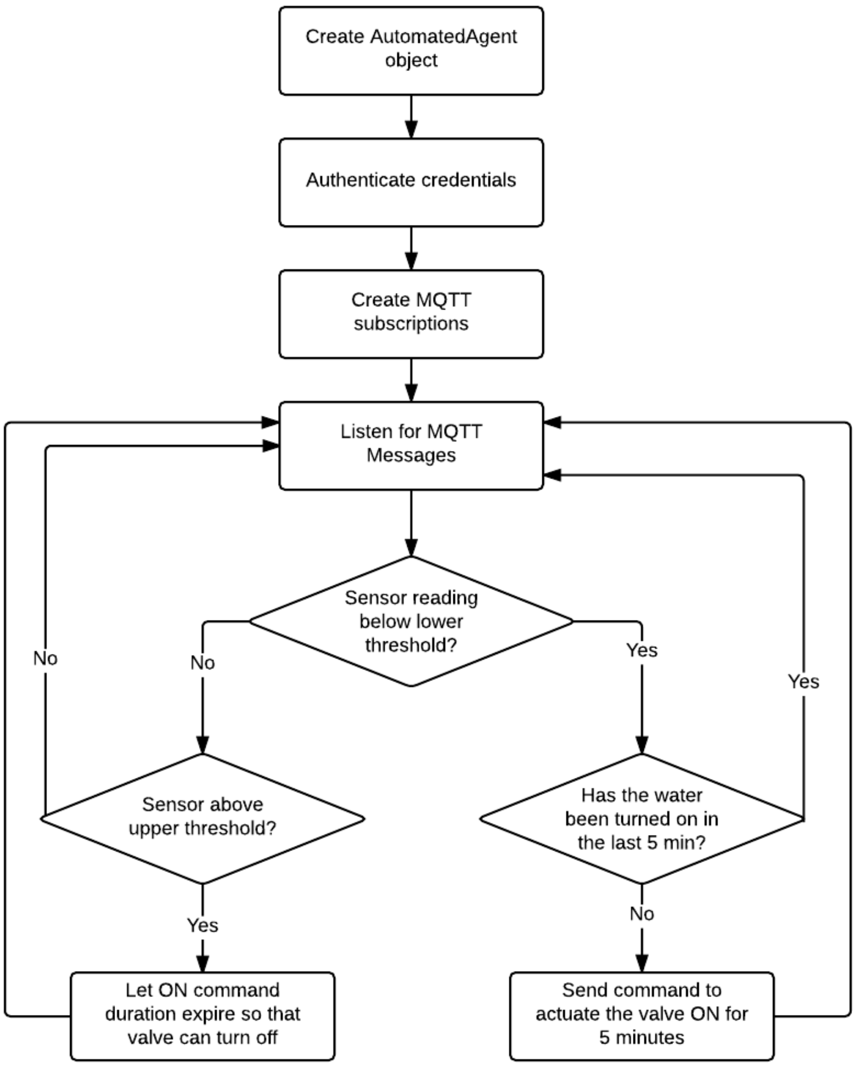

Create AutomatedAgent object
Authenticate credentials
Create MQTT subscriptions
Listen for MQTT Messages
Sensor reading below lower threshold?
No
Yes
No
Yes
Sensor above upper threshold?
Has the water been turned on in the last 5 min?
Yes
No
Let ON command duration expire so that valve can turn off
Send command to actuate the valve ON for 5 minutes

Figure 6.2: The automated agent that controls the greenhouse experiment follows this logic diagram.

The core logic of the agent's control algorithm is implemented in a MessageProcessor lambda expression. The following code segment shows the MessageProcessor for this agent in its entirety. The logic in the MessageProcessor is what the agent executes whenever a new message is received. Each message is parsed and analyzed to determine whether it contains a sample from the transducer at a depth of 20cm. A WiSARD reports many streams, and samples from different transducers often are grouped into the same message in the WiSARD wireless communication protocol. If a message is obtained containing a sample from the transducer of interest, the logic in the MessageProcessor will use the measurement to determine whether the valve should be turned on.

```
MessageProcessor msgProc = (Message msg) -> {

  int net_id_hi = msg.getPayload()[8];
  int net_id_lo = msg.getPayload()[9];
  int net_id = ((net_id_hi << 8) | (net_id_lo & 0xFF));
  int valve_status = 0;

  // if message is from the valve wisard
  if(net_id == 0x0005){
    // if the packet has a stream 5
   if(msg.getPayload()[19] == 0x05){
     logLine("Messaged received from wisard " + net_id);

     // then this byte is the valve status
     valve_status = msg.getPayload()[21];
     logLine("Valve status was changed to: " + valve_status);
     logLine("------------------------------------------------------------");
   }
  }

  // is message is from the sensing wisard and a full data packet
  if(net_id == 0x0003 && msg.getPayload().length > 40){

  // get sensor reading
  int sp_mod = msg.getPayload()[14];
  int stream = msg.getPayload()[35];
  int sample_hi = msg.getPayload()[37];
  int sample_lo = msg.getPayload()[38];
  int sample = ((sample_hi << 8) | (sample_lo & 0xFF));

  // only make decisions when we get a new reading from our sensor
  if(sp_mod == 3 && stream == 5){
    logLine("Messaged received from wisard " + net_id);
   logLine("SP: " + sp_mod + " stream: " + stream + " reads " + sample);

   // if we are in upslope where we want to keep watering...
   if(risingState){
      // if soil is still not wet enough, keep watering
      if(sample < upper_limit){
        if(last_cmd_time==0 || System.currentTimeMillis()-last_cmd_time>
         180*1000){
         ActuateValve cmd = new ActuateValve(wisard, 0, (byte)1, 3600,
            true, 300);
         AutomatedAgentController.runCommand(cmd, agent);
         last_cmd_time = System.currentTimeMillis();
         logLine("Sample below upper threshold, sending another on command " +
            "to keep watering");
        }
        else{
          logLine("Sample below upper threshold, but not enough time since " +
            "last cmd to send a new one");
        }
      }
      // otherwise, the soil has gotten wet enough — turn off valve
      else{
        risingState = false;
        logLine("Sample at or above upper threshold, letting duration expire");
      }
   }
   // otherwise, we are in the downslope where we want it to keep drying
   else{
      // if soil has dried too much, turn on water and change state
     if(sample < lower_limit){
      ActuateValve cmd = new ActuateValve(wisard, 0, (byte)1, 3600, true, 300);
      AutomatedAgentController.runCommand(cmd, agent);
      risingState = true;
      last_cmd_time = System.currentTimeMillis();
      logLine("Sample below lower threshold, sending an on command to start watering");
     }
     else{
       if(last_cmd_time == 0 || System.currentTimeMillis() - last_cmd_time > 180*1000){
        logLine("Sample above lower threshold, letting duration expire");
      }
      else{
        logLine("Sample still above lower threshold, but not enough time since " +
         "last cmd to send a new one");
      }
     }
   }
   logLine("-------------------------------------------------------------------");
   }
  }
};
```

If the dielectric permittivity level is determined to be less than the lower threshold value, the ActuateValve command class is used to synthesize a command message which will instruct the WiSARD to actuate the valve into the open state for a five minute duration. When the time parameter in a valve duration command message expires, the WiSARD will actuate the valve into the closed position, stopping the flow of water into the bucket. If another valve actuation command is sent to a WiSARD while there is still time remaining from the previous command's duration, the value of the remaining time is overwritten with the

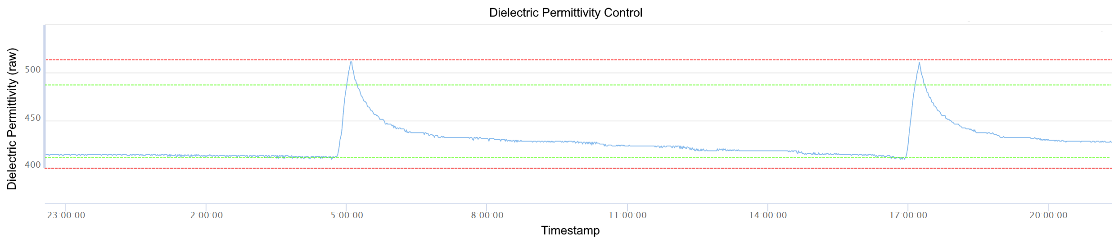


FIGURE 6.3

duration parameter of the new command message. The software in this Message-Processor continually sends valve actuation command messages with five minute durations for as long as the transducer reports readings less than the upper threshold. If the upper threshold is passed, then the duration is allowed to expire and no more messages are sent until the samples once again fall below the lower threshold.

Figure 6.3 shows the raw dielectric permittivity samples from the Decagon 5TM transducer while the agent was active. The green dashed lines indicate the threshold values of 410 and 480 used to actuate the valves. The red regions indicate the periods where the valves were open and allowed water to flow into the soil. The agent ensured that the raw dielectric permittivity never exceeded 520 nor did it fall below 400.

6.6 Summary

Automated reconfiguration is a valuable feature that allows for greater control over a WSN. The automated agent software framework described in this chapter allows automated reconfiguration to be utilized in WiSARDNet. The greenhouse control experiment demonstrates the ability of an automated agent to control an experiment on a live WiSARD network by utilizing the software modules described in Chapter 5. A user can utilize this software framework to create their own automated agents. Appendix **??** provides instructions on how to create an automated agent.

Chapter 7

Conclusion and Future Work

7.1 Overview

This work introduces new functionality into the WiSARDNet platform that enhances the usability of the system for researchers. Its design also provides a foundation for more features to be added in the future. This chapter summarizes the accomplishments of this thesis and assesses the impact each component has on the WiSARDNet platform and its users. Finally, this chapter discusses opportunities for future work.

7.2 Conclusions

This work presents a design and implementation for adding network reconfiguration capabilities to the WiSARDNet platform according to the requirements established in Chapter 1. Each design requirement is listed below along with a description of how it was accomplished in this work.

- **Implementation of an application that allows users to remotely identify and select sets of WSN nodes based on static and dynamic search criteria.** Wisard_Browser_Module fulfills this objective by allowing users to specify information that corresponds to a variety of WiSARD attributes that are both static and dynamic in nature. This module produces a set of WiSARDs that match the search criteria entered by the user.

- **Implementation of an interface allowing users to enact behavioral changes in individual or groups of nodes.** This objective is satisfied by Cmd_Generation_Module that generates commands to reconfigure WiSARDs based on changes by the user. The extent to which a user can reconfigure the WiSARDs is governed by a user access control and network safety check that is performed for every command that is generated.

- **Generalization of the reconfiguration software such that agents can monitor and reconfigure network nodes.** This is satisfied with the implementation of automated agents that are able to utilize Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module to reconfigure WiSARDs in reaction to different events that can be specified for each agent. Additionally, automated agents are subject to the same user access control and validation rules as human users.

7.3 Future Work

WiSARDNet is a platform that supports complex distributed ecological experiments. In this work, the addition of user access control, remote reconfiguration, and the ability to automate reconfiguration adds valuable functionality to the WiSARDNet platform. It also provides a foundation supporting additional capabilities.

7.3.1 Stored Behaviors and Profiles

The central paradigm of this thesis is that a WSN can be viewed as a database of heterogeneous behaviors, and that network reconfiguration can be achieved using a database approach. The process of network reconfiguration utilizes a command generator to build messages that invoke WSN configuration changes. A natural extension of this functionality would be for a user to have access to reconfiguration profiles, or groups of reconfiguration commands that produce a more complex defined behavior for a WiSARD. These profiles could be stored in a PostgreSQL data table, and could be selected and deployed to a set of WiSARDs.

One example is a low power profile that reduces the sampling rates of all transducers in a WiSARD. This profile could also change the WiSARD's role from a node that relays messages to other nodes into a leaf node that does not need to expend energy communicating with other nodes. Once operational, a user would have access to this profile that could then be executed via the commands it encapsulates. Additionally, profiles could be governed by the user access control system, due to the generalized nature of the user access control schema. An extension of the PermissionResource table to include profiles would allow a permission entity to be granted access to specific profiles.

7.3.2 OTA Reprogramming

Currently, new firmware versions must be physically uploaded to the WiSARDs because there is no over-the-air (OTA) reprogramming capability for the WiSARDs. The software implemented for this thesis could be expanded to support OTA reprogramming. The compiled source code for a new firmware version could be broken down into a series of messages, where each message payload contains a numbered piece of the new firmware. These messages could be sent sequentially to the intended WiSARD; once the WiSARD has received all of the messages, it could begin to overwrite the existing firmware in flash memory with the new firmware.

Implementing these features would not be a trivial task. OTA reprogramming would require expanding the command generation with features that would support breaking down a file containing a new firmware into many messages, numbering these messages, and sending these messages to the intended WiSARD. Additionally the firmware on WiSARDs would need to be upgraded to support these behaviors as well. The WiSARD would need to be able to recognize that each message is a piece of a new firmware, and would need to store that piece of the message. Additionally, the WiSARD would need to verify that it has received all of the messages correctly, assemble the firmware pieces from each message into the entire file, and then overwrite the existing firmware with the new version. The process of overwriting the old firmware with the new one would also need to have a way to revert back to the old firmware if the new firmware was unable to be correctly downloaded or stored.

OTA reprogramming would require all of these new features, but the work of this thesis provides a foundation for these features to be built upon.

7.3.3 Creation of Complex Automated Agents

The addition of automated agents to the WiSARDNet system provides a way for the WSNs to be monitored and reconfigured in real time based on different events that a developer or researcher must specify for each agent. The use case described in Chapter 6 is a fairly simple example which manages the sampling rate of a WiSARD based on the time of day, but the system supports complex agents that monitor and reconfigure large numbers of WiSARDs in response to many different events. The agents are designed to have access to all of the monitoring and reconfiguration features that a human user does, so any improvements or additions to the modules such as the ones described in the previous sections would also benefit the agents in expanding their capabilities. Everything is in place for a user to build a complex agent by following the example code in Chapter 6.

7.3.4 Improved User Interface

As is the case with many software systems, there are plenty of opportunities for improvements to be made to the user interface. The user interface for the WiSARD Browser and Command Generation Module presents users with a simple way to enter WiSARD search criteria and specify changes. This can be improved with the inclusion of a geographic map that allows a user to view all of the WiSARDs that he/she has permission to reconfigure. Users with experiments spanning large geographic regions would have an easier time visualizing the effects of a reconfiguration if they had access to a user interface that displayed WiSARDs in this way.

Another opportunity for improvement is a user interface that lets a user build an automated agent visually, rather than requiring that he/she develop an agent by programming it directly. This would make automating network reconfiguration much more accessible to researchers without sufficient Java programming experience. This would require generalizing the agent code and allowing boiler-plate features to be specified and populated from either a form or some other type of user input. Additionally, allowing users to edit and modify agents from this user interface would also be beneficial.

7.3.5 Integration of Other IoT Platforms

IoT devices are growing in number every day, and because of this there are many other WSN platforms and management solutions being developed and used, such as Amazon Web Services (AWS IoT). AWS IoT is a software platform which provide cloud management services to IoT devices. There is great potential for the work of this thesis to be adapted for use in another WSN platform such as AWS IoT.

Integration of the work of this thesis into another platform would not be a trivial task, but there are a couple of foundational similarities between AWS IoT and parts of WiSARDNet that present an opportunity for future work that is worth exploring. For example, AWS IoT uses the MQTT as a middleware solution for connecting IoT devices to the AWS cloud. Portions of the WiSARDNet publish/-subscribe clients (Raw Data Fetcher and WiSARD Client) could be adapted to allow garden servers to connect to the AWS cloud as opposed to the WiSARDNet RTDC. Once accomplished, Wisard_Browser_Module, Cmd_Generation_Module, and Validation_Module could then be adapted to provide their WiSARD reconfiguration functionality with the tools that the AWS IoT platform provides.

The largest obstacle to an integration such as this is the reliance that WiSARDNet has on a relational database. AWS IoT provides database services to networked devices using DynamoDB, a non-relational database. Many of the abstractions used to manage WiSARDs and experiments in WiSARDNet would not translate easily to a non-relational database paradigm and would require a different approach. Additionally, Wisard_Browser_Module and Validation_Module were designed in such a way that they take advantages of the relational database paradigm used in WiSARDNet. Using these modules in a platform like AWS would require the code be modified to use a different paradigm.

Once the code has been modified to support the new paradigm however, automated agents could be easily implemented to reconfigure WiSARDs using AWS IoT event triggers which allow for actions to be taken in response to the analysis of incoming data streams. The event trigger paradigm used in AWS IoT is similar to the design approach taken for automated agents in this work, and therefore the control logic could translate without extensive reorganization.

Appendix

The framework provided in this thesis allows a user to create their own automated agent to reconfigure WiSARDs or control experiments. The creation of an automated agent for use with WiSARDNet requires Java programming experience as well as familiarity with a command line interface. Additionally, a WiSARDNet system administrator will need to create an account for the user making the agent.

First, you will need to create a new Java project in the editor of your choice. Eclipse was used in the work of this thesis and is recommended for the creation of automated agents. To utilize the automated agent framework developed in the work of this thesis, you will need a few open source support libraries to be added to your project.

- The PostgreSQL JDBC Driver
- The Java Commons Codec
- The Eclipse Paho MQTT Driver

These libraries will allow you to connect to the WiSARDNet PostgreSQL database, connect to a WiSARDNet MQTT broker, create publish and subscribe clients, and parse message packets. Additionally, you will need to include the WiSARDNet Helpers, Interfaces, and Utilities support libraries in your project. These support libraries contain a variety of Java classes and interfaces that enable interaction with the different WiSARDNet systems. These include the AutomatedAgent and AutomatedAgentController classes discussed in Chapter 6.

To create an agent, you will need to create Java file that defines a class. There are three critical components your class will need to have.

- A main method
- A MessageProcessor
- An ErrorHandler

The main method is needed so that the agent will be runnable as a command line program. The entire class file for the greenhouse experiment agent described in Chapter is shown in the code sample below. This class file provides an example for how each of these software pieces can be approached.

```java
package Agents;

import java.sql.SQLException;
import java.sql.Timestamp;
import java.time.LocalDateTime;
import java.util.Calendar;

import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
import org.apache.commons.codec.binary.Hex;

import java.util.ArrayList;
import interfaces.ErrorHandler;
import interfaces.MessageProcessor;
import helpers.SubscriptionInfo;
import helpers.Wisard;
import utilities.Message;
import utilities.ActuateValve;

public class GreenhouseAutomatedAgent{
   public static volatile long interval = 300; // 5 min default sampling rate
   public static volatile AutomatedAgent agent = null;
   public static volatile Wisard wisard;
   public static volatile boolean risingState = false; // assumption is valve is closed
   public static int upper_limit = 480;
   public static int lower_limit = 410;
   public static long last_cmd_time;

   // main method
   public static void main(String[] args) throws SQLException{

      // check that all input arguments were given
      if(args.length < 2){
         throw new IllegalArgumentException("You must provide username and password");
      }

      // make a new wisard object (for the wisard we are sending the cmd to)
      wisard = new Wisard(0, "", "", "", 5, "", "", "");

      String username = args[0];
      String password = args[1];

      ArrayList<SubscriptionInfo> subscriptions = new ArrayList<SubscriptionInfo>();
      subscriptions.add(new SubscriptionInfo("#", "tcp://bio17.bio.nau.edu:1883", "greenhouse",
     ↪ 2));

      MessageProcessor msgProc = (Message msg) -> {

         int net_id_hi = msg.getPayload()[8];
         int net_id_lo = msg.getPayload()[9];
         int net_id = ((net_id_hi << 8) | (net_id_lo & 0xFF));
         int valve_status = 0;

         // if message is from the valve wisard
         if(net_id == 0x0005){
            // if the packet has a stream 5
            if(msg.getPayload()[19] == 0x05){
               logLine("Messaged received from wisard " + net_id);

               // then this byte is the valve status
               valve_status = msg.getPayload()[21];
               logLine("Valve status was changed to: " + valve_status);
               logLine("--------------------------------------------------------------------");
            }
         }

         // is message is from the sensing wisard and a full data packet
         if(net_id == 0x0003 && msg.getPayload().length > 40){

            // get sensor reading
            int sp_mod = msg.getPayload()[14];
            int stream = msg.getPayload()[35];
            int sample_hi = msg.getPayload()[37];
            int sample_lo = msg.getPayload()[38];
            int sample = ((sample_hi << 8) | (sample_lo & 0xFF));

            // only make decisions when we get a new reading from our sensor
            if(sp_mod == 3 && stream == 5){
               logLine("Messaged received from wisard " + net_id);
               logLine("SP: " + sp_mod + " stream: " + stream + " reads " + sample);

               // if we are in upslope where we want to keep watering...
            if(risingState){
               // if soil is still not wet enough, keep watering
               if(sample < upper_limit){
                  if(last_cmd_time == 0 || System.currentTimeMillis() - last_cmd_time >
 ↪ 180*1000){
                     ActuateValve cmd = new ActuateValve(wisard, 0, (byte)1, 3600, true, 300)
 ↪ ; // on cmd
                     AutomatedAgentController.runCommand(cmd, agent);
                     last_cmd_time = System.currentTimeMillis();
                     logLine("Sample below upper threshold, sending another on command to keep
 ↪ watering");
                  }
                  else{
                     logLine("Sample below upper threshold, but not enough time since last cmd
 ↪ to send a new one");
                  }
               }
               // otherwise, the soil has gotten wet enough - turn off valve
               else{
                  risingState = false;
                  logLine("Sample at or above upper threshold, letting duration expire");
               }
            }
            // otherwise, we are in the downslope where we want it to keep drying
            else{
               // if soil has dried too much, turn on water and change state
               if(sample < lower_limit){
                  ActuateValve cmd = new ActuateValve(wisard, 0, (byte)1, 3600, true, 300);
 ↪ // on cmd
                  AutomatedAgentController.runCommand(cmd, agent);
                  risingState = true;
                  last_cmd_time = System.currentTimeMillis();
                  logLine("Sample below lower threshold, sending an on command to start
 ↪ watering");
               }
               else{
                  if(last_cmd_time == 0 || System.currentTimeMillis() - last_cmd_time >
 ↪ 180*1000){
                     logLine("Sample above lower threshold, letting duration expire");
                  }
                  else{
                     logLine("Sample still above lower threshold, but not enough time since
 ↪ last cmd to send a new one");
                  }
               }
            }
            logLine("------------------------------------------------------------------");
         }
      }

   };

   ErrorHandler errHandler = (Exception e) -> {
      // prints exception to std error
      e.printStackTrace();
   };

   // an example of how to use the AutomatedUserController to create an
 ↪ AutomatedUser
   agent = AutomatedAgentController.createAutomatedAgent(username, password, subscriptions, "
 ↪ webportal", msgProc, errHandler);
   new Thread(agent).start();
}

public static void logLine(String msg){
   System.out.println(msg + " (" + new Timestamp(System.currentTimeMillis())+ ")" );
}
```

```
}
```

The main method needs username and password Strings as command line input parameters; this is necessary for the agent to be authenticated. Within the main method, you need to define an ArrayList data structure of SubscriptionInfo objects. These objects must contain the connection information necessary to subscribe to a set of WiSARD streams from a particular MQTT broker. You will need to make one SubscriptionInfo object for every broker you wish to receive data from. Next, you will need to define a MessageProcessor and an ErrorHandler. The MessageProcessor is a lambda expression which contains the logic that an agent will use to process MQTT messages. The ErrorHandler is a lambda expression that defines how the agent will handle the different types of exceptions that can occur.

To complete your agent, you will need to place two function calls at the end of your main method. First, you will need to call the createAutomatedAgent static method from the AutomatedAgentController class, and pass it the username, password, SubscriptionInfo ArrayList, MessageProcessor, and ErrorHandler variables as input parameters. The method signature in the AutomatedAgentController class file shows exactly how these parameters need to be passed. The results of this static method call should be assigned to a variable. In this example, the variable `agent` is used to reference the created object. Finally, A new `Thread()` object needs to be created and passed `agent` as a parameter and the `start()` method needs to be called. Once all of these things are in the class file, the agent program can be finalized and run.

Once the code is written for the automated agent, the files and support libraries need to be exported as a runnable JAR file. If prompted by your development editor, the required libraries should be packaged with your code into the JAR file.

Bibliography

[1] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, “Tinyos: An operating system for sensor networks,” *Ambient intelligence*, vol. 35, pp. 115–148, 2005.

[2] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. Kim, B. Zhou, and E. G. Sirer, “On the need for system-level support for ad hoc and sensor networks,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. 2, pp. 1–5, 2002.

[3] M. Ivester and A. Lim, “Interactive and extensible framework for execution and monitoring of wireless sensor networks,” in *2006 1st International Conference on Communication Systems Software & Middleware.* IEEE, 2006, pp. 1–10.

[4] M. Hammoudeh, S. Mount, O. Aldabbas, and M. Stanton, “Clinic: A service oriented approach for fault tolerance in wireless sensor networks,” in *Sensor Technologies and Applications (SENSORCOMM), 2010 Fourth International Conference on.* IEEE, 2010, pp. 625–631.

[5] P. Levis and D. Culler, "Mat: A tiny virtual machine for sensor networks," *ACM Sigplan Notices*, vol. 37, no. 10, pp. 85–95, 2002.

[6] P. G. Flikkema, K. R. Yamamoto, S. Boegli, C. Porter, and P. Heinrich, "Towards cyber-eco systems: Networked sensing, inference and control for distributed ecological experiments," in *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on.* IEEE, 2012, pp. 372–381.

[7] E. Eronu, S. Misra, and M. Aibinu, "Reconfiguration approaches in wireless sensor network: Issues and challenges," in *2013 IEEE International Conference on Emerging & Sustainable Technologies for Power & ICT in a Developing Society (NIGERCON).* IEEE, 2013, pp. 143–142.

[8] A. Reinhardt and C. Renner, "Remote node reconfiguration in opportunistic data collection wireless sensor networks," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a.* IEEE, 2014, pp. 1–3.

[9] S. Madden, "Database abstractions for managing sensor network data," *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1879–1886, 2010.

[10] O. Diallo, J. J. Rodrigues, M. Sene, and J. Lloret, "Distributed database management techniques for wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 604–620, 2015.

[11] L. D. Chagas, E. P. Lima, and P. F. R. Neto, "Real-time databases techniques in wireless sensor networks," in *Networking and Services (ICNS), 2010 Sixth International Conference on.* IEEE, 2010, pp. 182–187.

[12] R.-G. Urma, M. Fusco, and A. Mycroft, "Java 8 in action," 2014.

[13] C. A. Stewart, S. Simms, B. Plale, M. Link, D. Y. Hancock, and G. C. Fox, "What is cyberinfrastructure," in *Proceedings of the 38th annual ACM SIGUCCS fall conference: navigation and discovery.* ACM, 2010, pp. 37–44.

[14] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S - a publish/-subscribe protocol for wireless sensor networks," in *3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE 2008)*, January 2008, pp. 791–798.

[15] "Mosquitto: An open source mqtt broker," https://mosquitto.org/, 2012.

[16] T. AbuHmed, N. Nyamaa, and D. Nyang, "Software-based remote code attestation in wireless sensor network," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE.* IEEE, 2009, pp. 1–8.

[17] A. Antola, A. L. Mezzalira, and M. Roveri, "Ginger: a minimizing-effects reprogramming paradigm for distributed sensor networks," in *Robotic and Sensors Environments (ROSE), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 88–93.

[18] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, "Mote runner: A multi-language virtual machine for small embedded devices," in *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on.* IEEE, 2009, pp. 117–125.

[19] P. D. Felice, M. Ianni, and L. Pomante, "A spatial extension of tinydb for wireless sensor networks," in *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on.* IEEE, 2008, pp. 1076–1082.

[20] D. He, C. Chen, S. Chan, and J. Bu, "Sdrp: A secure and distributed reprogramming protocol for wireless sensor networks," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 11, pp. 4155–4163, 2012.

[21] C.-M. Hsieh, Z. Wang, and J. Henkel, "Eco/ee: Energy-aware collaborative organic execution environment for wireless sensor networks," in *2012 IEEE Wireless Communications and Networking Conference (WCNC).* IEEE, 2012, pp. 1998–2002.

[22] C.-H. Hsueh, Y.-H. Tu, Y.-C. Li, and P. H. Chou, "Ecoexec: An interactive execution framework for ultra compact wireless sensor nodes," in *2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON).* IEEE, 2010, pp. 1–9.

[23] H. Hu, J. He, and J. Wu, "Distributed processing of approximate range queries in wireless sensor networks," in *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on.* IEEE, 2015,

pp. 45–51.

[24] C. Jardak, P. Mhnen, and J. Riihijrvi, "Spatial big data and wireless networks: experiences, applications, and research challenges," *IEEE Network*, vol. 28, no. 4, pp. 26–31, 2014.

[25] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu, and B. Xu, "An iot-oriented data storage framework in cloud computing platform," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1443–1451, 2014.

[26] Y.-S. Kang, I.-H. Park, J. Rhee, and Y.-H. Lee, "Mongodb-based repository design for iot-generated rfid/sensor big data," *IEEE Sensors Journal*, vol. 16, no. 2, pp. 485–497, 2016.

[27] S. Sarangi and S. Kar, "A scriptable rapid application deployment framework for sensor networks," in *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on.* IEEE, 2013, pp. 1035–1040.

[28] M. Szczodrak, O. Gnawali, and L. P. Carloni, "Dynamic reconfiguration of wireless sensor networks to support heterogeneous applications," in *2013 IEEE International Conference on Distributed Computing in Sensor Systems.* IEEE, 2013, pp. 52–61.

[29] R. Tompkins, T. B. Jones, R. E. Nertney, C. E. Smith, and P. Gilfeather-Crowley, "Reconfiguration and management in wireless sensor networks," in *Sensors Applications Symposium (SAS), 2011 IEEE.* IEEE, 2011, pp. 39–44.

[30] V. Vaidehi and D. S. Devi, "Distributed database management and join of multiple data streams in wireless sensor network using querying techniques," in *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on.* IEEE, 2011, pp. 583–588.

[31] J. L. Wilder, V. Uzelac, A. Milenkovic, and E. Jovanov, "Runtime hardware reconfiguration in wireless sensor networks," in *2008 40th Southeastern Symposium on System Theory (SSST).* IEEE, 2008, pp. 154–158.

[32] L. Rockoff, *The language of SQL.* Nelson Education, 2010.