2014

Messaging in Scalaness: A Programming Language for Wireless Sensor Networks

Rebecca C. Norton
University of Vermont, rcnorton@uvm.edu

MESSAGING IN SCALANESS:
A PROGRAMMING LANGUAGE FOR WIRELESS SENSOR NETWORKS

A Thesis Presented

by

Rebecca C. Norton

to

The Faculty of the College of Arts and Sciences

of

The University of Vermont

In Partial Fullfillment of the Requirements
for the Degree of Bachelor of Arts with Honors
Specializing in Computer Science

May, 2014

Abstract

This paper presents the design and implementation of three types of messaging primitives for the Scalaness programming language. The main idea behind this project was to extend the current version of Scalaness to include new messaging abstractions. Scalaness is a special purpose variant of Scala for running on the hub of a wireless sensor network. The three types of messages considered here are data reports from nodes to hub via a collection tree, small values sent from hub to nodes, and larger values sent from hub to nodes. These primitives are defined in the abstract, then implemented in Scala, and the necessary specifications for translation from Scalaness command to back end Scala code are made.

Table of Contents

Chapter 1

Introduction

In this section, I will present background information on wireless sensor networks and the Scalaness programming language, explain the main idea and design of my project, and give an example of a real-world scenario in which this work could be useful.

1.1 Background and Examples

A wireless sensor network is a group of small nodes distributed over an area which collect data about their surroundings and report them to a central computer. The nodes can be small enough to be practically invisible, and they can potentially be dropped from an airplane to float down and land in an area of interest, making this type of network well suited to military surveillance and other similar applications. If secrecy is not the goal, the nodes can be bigger, up to about the size of a shoebox, and placed by hand in specific useful locations. Outside the military, wireless sensor networks are used to monitor environmental data–rainfall, temperature, soil moisture and acidity, sunlight, air and water pollution, water level in rivers and lakes, etc.

Nodes in a wireless sensor network have a limited supply of power; they must be

programmed to run as efficiently as possible. A useful technique in this situation is staged programming. The hub runs the first-stage code, meaning it does most of the work, and generates specialized second-stage code to be used by the nodes. Specialization, in this case, means adapting the code to the specific network environment (number and location of neighbors, amount of interference, and more). By ensuring that the nodes transmit exactly what they need to and no more, the specialized code saves power. Professor Christian Skalka's team is developing a language called Scalaness, which is an extension of Scala, for writing the first-stage code. The generated second-stage code is in nesT, a variation of the TinyOS language nesC. As long as the nodes run TinyOS, as is most common in wireless sensor networks, they can use the second-stage nesT code. (10)

The general organization of a wireless sensor network is shown in Figure 1.1.

The nodes in the network are small computing devices called motes: for example, TelosB or Raspberry Pi. They can be close together or spread over many miles. The network's hub consists of a more powerful computing device with its own mote connected via USB. The more powerful device runs any necessary programs that are too computationally intensive for the motes, and uses its mote as a gateway to the rest of the network.

As a sample use case, imagine a sensor system designed to measure fluctuations in air temperature over an area of land. The scientist in charge of the project writes a network orchestration program to run on the hub. The rest of the network consists of 15 TelosB motes running TinyOS, in protective casing with solar panels, small back-up batteries, external temperature sensors, and cell data modems. These motes are spread across several acres. They are programmed to take a temperature reading every hour and immediately report it to the hub. The network orchestration program gets weather reports from the Internet, and when a cloudy day threatens the solar

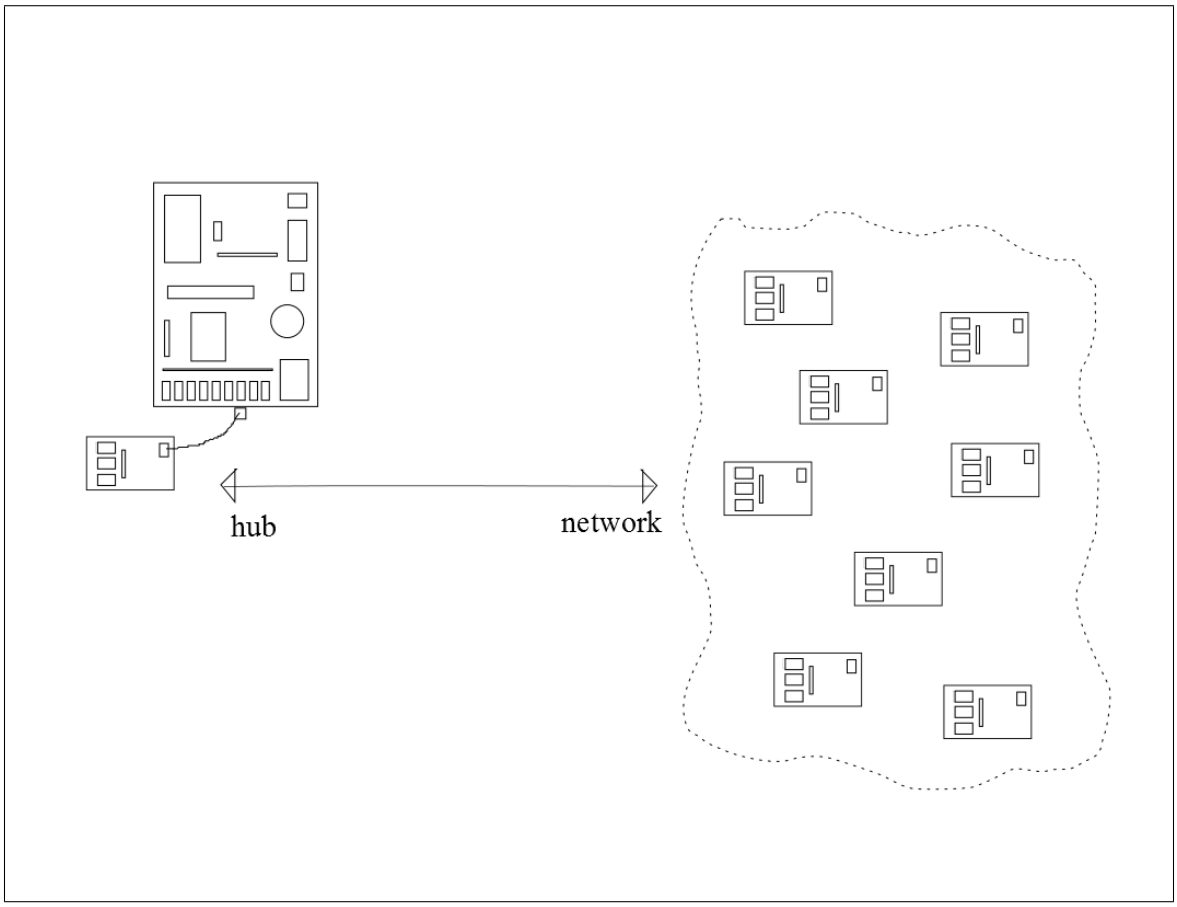

Figure 1.1: Hub and nodes in wireless sensor network

power supply, it sends a command to all of the motes telling them to reduce their reporting schedule to once every three hours. One mote is known to be in a particularly shady section, so the hub commands it to reduce its schedule even further, to once every six hours, to avoid having to shut down from lack of power. Eventually one of the scientist's team members develops a more sophisticated sampling model based on a neural network, and feeds it into the network orchestration program, which sends this model out to all of the motes. The entire network then benefits from the increase in efficiency.

In this example, we can see the need for three distinct types of communication. First, the motes need to send the data they have collected back to the hub. Next, the

hub needs to send small commands to the network, either to all of the motes or just to one. Finally, the hub needs to send a sampling model to all of the motes, which is a larger value than the previous commands. There are many other possible scenarios which would also demonstrate the requirement for these communication types.

1.2 Contributions of This Thesis

This thesis will focus on developing messaging capabilities for Scalaness to make it a language that fully supports communication between a Linux hub and a TinyOS network. This will be an important step in the growth and improvement of Scalaness. There are many situations in which the hub and nodes of a wireless sensor network need to communicate with each other, which can be generally organized into three categories: small values being sent from hub to nodes, larger values being sent from hub to nodes, and data being reported from nodes to hub.

My project includes three stages of programming language design: language-level abstractions, implementations of these abstractions, and specifications for a parser. I will first describe the types of communication that we have decided to add to Scalaness at the language level: i.e., the messaging functions that can be called and what they do. I will present the semantics of each new communication function. I will then consider how these functions should be implemented, present Scala code that demonstrates their practicality and correctness, and create a specification for parsing the new Scalaness functions into back-end Scala code. My implementations will take communication up to the connection between hub and network, as shown in Figure 1.1: creating full network connectivity should be a straightforward "plug-and-play" project for future researchers. Finally, I will summarize the main points of the paper and present several ideas for future extensions of this work.

Chapter 2

Concepts and Foundation

When deciding which messaging primitives to use and how to model them, some things to take into consideration include:

- Addressing modes: unicast vs. broadcast vs. anycast
- Timing: asynchronous vs. synchronous communication
- Channels: streaming vs. fixed-length messages vs. packets

Addressing modes make it possible to send messages to the specific node or nodes they are meant for. Unicast addressing is for communication with a single node, and broadcast addressing is for communication with all the nodes in a network. One use for a unicast message would be to command a node that is experiencing power issues to reset its power supply. Broadcast messages carry information that all nodes need, such as if a new node has been added to the network. Anycast addressing is for messages that need to get to any one or more nodes in a set. An example would be data reporting through a collection tree: each node sends its data to any node that's closer to the root than it is, and all the data eventually reaches the root.

Timing of communication is also important. The difference between asynchronous and synchronous messaging is like the difference between an email and a phone call. Asynchronous messages are sent, received, and responded to whenever the sender and receiver see fit. They can perform other tasks between messages. Synchronous messaging occupies the full attention of both sender and receiver until the exchange is finished.

Messages can be sent as long streams of data over a streaming socket, which requires synchronicity. Alternatively, they can be sent as fixed-length types like byte arrays, or as single small packets. Both of these options allow for asynchronous communication.

The following is a survey of works in the literature of computer science which have formally addressed these issues, and which inspire and inform our language design.

2.1 Survey of Related Work

The Pi Calculus

The pi calculus models synchronous point-to-point communication. The actors are called processes, and they communicate over channels. An important fact is that in addition to passing data over channels, one can also pass channels over channels. This is useful if, for example, the sender wants to specify a channel for the recipient to respond over. The pi calculus does not include confirmation or guarantees that, for example, a message will arrive looking exactly the same as when it was sent, but some of its extensions do (see The SPi Calculus).

Example:

$\bar{a}\langle x\rangle.P$ represents a process that sends value x along channel a, and once x has

been received by the intended process, behaves as the process P.

$a(x).Q$ represents a process that waits to receive value x on channel a, and once it receives x, behaves as the process Q.

$\bar{a}\langle x\rangle.P|a(x).Q$ represents these two processes running in parallel: the first process sends x on channel a, the second process receives x on channel a, and they continue on as processes P and Q respectively. Note that the first process waits until the second process has received the message before continuing, since this is synchronous communication. (8)

The Asynchronous Pi Calculus

The asynchronous pi calculus is a variant of the pi calculus. It uses an asynchronous send mechanism, meaning that the sender does not block messages from others while waiting for a response after sending. This calculus models asynchronous, point-to-point communication. It behaves as though processes are passing messages through unordered buffers, also called bags. This asynchronous send is generally preferred over models which use explicitly declared buffers, for reasons of elegance and simplicity.

Example:

The syntax for the asynchronous pi calculus is the same as for the pi calculus; the difference lies in the implementation. $\bar{a}\langle x\rangle.P$ represents a process that sends value x into a buffer on channel a, and then behaves as the process P. It does not wait for x to be received before moving on.

$a(x).Q$ represents a process that retrieves value x from a buffer on channel a, and then behaves as the process Q.

$\bar{a}\langle x\rangle.P|a(x).Q$ represents these two processes running in parallel: the first process sends x into a buffer on channel a, the second process receives x from that buffer

some unspecified amount of time later, and they continue on as processes P and Q respectively. (2)

Communicating Sequential Processes

CSP is a process calculus that describes systems in terms of processes that only interact with each other through the passing of messages. Processes can be sequential, as the name suggests, but they can also be defined to run in parallel. Like in the pi calculus, communication in CSP is synchronous and between only two processes at any given time. The sending process cannot send its message until the receiving process is ready to take it. There are also primitives to represent choice, of which CSP has two forms: nondeterministic and deterministic.

Example:

$a \rightarrow P$ represents a process that waits for event a to happen, and then behaves as the process P.

$(a \rightarrow P) \Box (b \rightarrow Q)$ represents deterministic choice: this process waits for either event a or event b to happen, and then behaves as process P or process Q depending on which one it was.

$(a \rightarrow P) \sqcap (b \rightarrow Q)$ represents nondeterministic choice: if both events a and b happen at the same time, this process randomly decides which path to take and then behaves as process P or process Q according to that decision. (6)

Calculus of Communicating Systems

CCS was developed by the same theorist as the pi calculus, and around the same time as CSP. It is also based on the idea of processes communicating through channels, synchronously and with one sender and one receiver, but it uses a different syntax than

the other two calculi. Also, while CSP has both nondeterministic and deterministic choice, CCS does not distinguish between the two; it includes only one primitive for choice.

Example:

$a.P_1$ represents a process that performs action a and then behaves as the process P_1. Note the difference from CSP: here, instead of passively waiting for event a to happen somewhere in its environment, the process actively performs action a.

$P_1 + P_2$ represents choice: this process can proceed as either P_1 or P_2. No specification is given as to how the choice is made.

Therefore, $a.(P_1+P_2)$ represents a process that performs action a and then behaves as either P_1 or P_2. (7)

Calculus of Broadcasting Systems

CBS is a calculus based on CCS, with the important difference that it models broadcast communication instead of point-to-point. In CBS, there is no private communication. When a process sends a message, all other processes in the system receive it instantly. They can then decide to do something based on that message or to just discard it. If two processes attempt to send messages at the same time, the conflict is resolved nondeterministically. In this situation there is no delay between when a message is sent and when it is received, and there is no way to introduce delay or use buffers, so communication is synchronous.

Example:

$w!p$ represents a process that sends message w and then behaves as process p. It ignores every message it hears, since it is not listening for anything. Note that this process only continues after its message has been received by the other members of

the network, since this is synchronous communication.

$v?w!p$ represents a similar process, but this one listens for message v before sending message w and becoming process p. Any messages other than v will be discarded by this process.

Choice is represented by the operator &: the process $x?p\&w!q$ will send message w and become process p unless it receives message x, in which case it will send nothing and become process q. (9)

The B-Pi Calculus

This is a variant of the pi calculus called b-pi which, like CBS, models synchronous broadcast communication. The sending process broadcasts a message to the entire group. All of the other processes have previously decided whether or not to listen on the sender's specific channel. Each process receives the broadcast message if it is listening and ignores it otherwise. B-pi's semantic rules describe the broadcast mechanism by saying that after a message is picked up and read by a receiving process, that message continues to exist for other processes to receive. This is in direct contrast to the point-to-point semantics of the pi calculus, CSP, and CCS, where once a message is read by the receiving process it disappears.

Example:

$\bar{x}y.p$ represents a process that sends y on channel x and then behaves as process p.

$x(y).q$ represents a process that receives y on channel x and then behaves as process q.

$\bar{x}y.p||x(y).q||x(y).r$ represents three processes running in parallel: the first process sends y on channel x, the second process receives y on channel x, and since commu-

nication here is broadcast, the third process is also able to receive y on channel x. They then continue on as processes p, q, and r respectively. (4)

The SPi Calculus

The spi calculus is an extension to the pi calculus meant for describing and analyzing cryptographic protocols. As in the pi calculus, processes communicate over channels, but here channels can be restricted so that only certain processes are able to use them. The goal is to establish two main properties: integrity and secrecy. This means that messages will not be corrupted in transit, and that they are visible only to those processes who are authorized to read them.

Example:

c_{AB} represents a channel c which is restricted for use by processes A and B.

We can define process A to be $\overline{c_{AB}}\langle M\rangle.A$ and process B to be $c_{AB}(x).0$. This means that process A sends message M along the secure channel c_{AB} and then continues as process A, and process B receives some message (which in this case will be M) on the secure channel c_{AB} and then terminates.

Because c_{AB} is defined to be available only to A and B, a third process D will not be able to read any messages sent over this channel. (1)

Mobile Ambients

This approach models processes that move between ambients (defined as bounded places where computation happens). The focus is less on communication and more on security. Processes can only interact with each other if they are located within a common boundary, and security is considered to be the ability or inability to cross boundaries. A process can be defined to have capabilities such as "enter n," "exit n,"

or "open n," where n is the name of an ambient.

Example:

$n[P]$ represents an ambient named n, inside which is running a process named P. P can represent one process or the parallel composition of several processes.

$n[in\ m.P]$ represents an entry instruction that tells ambient n to enter a sibling ambient m. After this transition is performed, n becomes a child of m. Similarly, $n[outm.P]$ represents an exit instruction that tells ambient n to exit its parent ambient m.

A simple form of communication is local anonymous message passing within an ambient. $\langle M \rangle$ represents some process releasing M "into the ether" of the surrounding ambient, and $(x).P$ represents another process receiving x and becoming P. The parallel composition of these two actions is $(x).P|\langle M \rangle$. (3)

Representational State Transfer

REST is an architectural style for designing network applications. Its focus is on speed and simplicity, and in practice it usually uses HTTP for communication within the network. The World Wide Web is a REST-based architecture. In REST, interactions occur between clients and servers through connectors (caches, tunnels, etc.) using operations (GET, POST, PUT, DELETE). Each source of specific information is called a resource, and has a unique identifier. A representation of a resource can be sent through the network to allow a client to view the current state of that resource, or even to modify it. In practice, representations are usually written in XML. An important principle of REST is layering: if a request must pass through several connectors to reach its destination, each connector only knows about its part of the journey, i.e. tunnel AB passes the request from point A to point B without knowing or

caring that the request's final destination is point D. REST is also stateless, meaning that clients and servers do not have to remember anything about their interactions; instead, each request contains all the information necessary for processing it.

Example:

In the World Wide Web implementation of REST, the information for a GET request is included in the URL passed from the client to the server. It has this form: (protocol)/(host)/(resource)/(optional parameters). To request the bar resource from foo.com, the client asks for http://www.foo.com/bar. To request the first item in a list of items, the client might ask for http://www.foo.com/item?id=1.

Parameters may be search indices, keywords, etc. There cannot be any spaces between or within the parameters; otherwise the request will not read whatever comes after the space. (5)

Chapter 3

Communication

In this chapter, I will outline the basic messaging patterns to be implemented in Scalaness. Then I will propose three communication abstractions based on those patterns. I will explain the features of each abstraction, followed by its syntax.

3.1 Desired Communication Patterns

As discussed in the introduction, wireless sensor networks encounter several different situations requiring communication between the hub and the nodes. The hub needs to be able to send two types of messages to the nodes: The first type is commands, which can be small values like integers corresponding to a command table, to tell the nodes to make changes in how they are operating. The second type is larger values, such as machine learning models.

The nodes need to be able to send one type of message to the hub: reported data, which can be in fixed size integers sent via collection tree protocol at any time.

The details of these desired patterns are presented in Figure 3.1.

These three communication patterns lend themselves to three different abstrac-

Pattern	Form	Example
Small values	16-byte array	Command to change sampling rate
Larger values	Any size byte array	Machine learning model for intelligent sampling rate changes
Data reporting	Any type that implements the Message interface, such as PrintfMsg	Data collected from external sensors

Figure 3.1: Communication patterns

tions, which will be described in the next section.

3.2 Proposed Communication Abstractions

In this section I will describe the features and syntax of each of the three abstractions. I will also introduce the TinyOS protocols that we will use as a foundation for each type of messaging.

3.2.1 Small Values

The hub sends small values, such as control statements, to the nodes. Again, it can use either unicast or broadcast addressing. This type of communication will be asynchronous, and it will use the TinyOS Dissemination protocol to provide guaranteed reliability. Asynchronicity is the best choice here because it allows the hub to send messages whenever it wants and then move on to other tasks, rather than wasting

```
object SmallValuesExample{
  val command : Int = 2
  val value : Int = 1
  val moteID : Int = 5
  smallsend(command,value,moteID)
}
```

Figure 3.2: Sample usage of small values

time waiting for a response. These messages will be one fixed size (16 bytes). An important part of this implementation will be flow control: since messages sent by Dissemination take some time to reach every member of the network, the hub must wait until each message has been received by every node before sending another one. Otherwise, the nodes at the far edge of the network could miss some messages. Dissemination is the standard protocol for this type of task in TinyOS; although it might be possible to write a replacement that solves this problem of latency in network consensus, it would be outside the scope of this project. Flow control is the ideal way to deal with the latency inherent in using Dissemination, because acknowledgments (telling the sender when a message has been received) are not part of the protocol.

Syntax: **smallsend(c,v,a)**

c is the command identifier. **v** is the value of the message. **a** is the address to which the message will be sent. All three arguments are integers. They will be packaged into a byte array before being sent off via Dissemination, since that's what the protocol requires for formatting.

Suppose that in your network the command to change on/off status is represented by the integer 2, the value for on is 0 and off is 1, and you want to tell the mote with ID number 5 to power off. Example code for this situation is shown in Figure 3.2.

3.2.2 Larger Values

The hub sends larger values, such as machine learning models, to the nodes - values that are too large for Dissemination. Again, it can use either unicast or broadcast addressing. To ensure that the entire message gets across without interruption or loss, this type of communication will use a streaming socket, and by using TCP as the underlying protocol it will provide guaranteed reliability. Since a streaming socket is used, this type of communication is synchronous. These messages will be byte arrays and can be any size. We expect larger values to be sent relatively infrequently, so it makes more sense to establish a new socket session for each message and close it afterward than to drain resources by leaving sockets open all the time.

Syntax: **largesend(v,a)** to send the large value **v** over a new socket to address **a**. The large value must be formatted as a byte array. The address is an integer.

Suppose you have developed a machine learning model that fits in a byte array of size 64. You want to send it to all the motes in your network. In your addressing scheme, mote IDs start at 1, and 0 represents a broadcast. Example code for this situation is shown in Figure 3.3.

3.2.3 Collection Tree Listening

The hub listens for data being reported from the nodes through a collection tree. It will use a synchronous communication model for this purpose, and just listen constantly, blocking until it receives any reports. Although this disallows other actions in the same thread while blocking is taking place, it is appropriate for this situation. We envision that its typical use case will be a listening loop in a separate thread spawned for this purpose. The hub will automatically establish itself as the root of the tree when the system loads; we simply need one command to tell it to start

```
object LargerValuesExample{
  val address : Int = 0
  val model : Array[Byte] = createModel
  largesend(model,address)

  def createModel : Array[Byte] = {
    val modelA : Array[Byte] = new Array[Byte](64)
    // Fill this array with the model before returning it
    return modelA
  }
}
```

Figure 3.3: Sample usage of larger values

listening. The messages returned by the listener can be any type that implements the Java Message interface, such as PrintfMsg.

Syntax: **listen()** to listen for data from the collection tree.

Usage for this could be a loop that retrieves the messages returned by the listener and writes them to a database. This could run in a separate thread to keep the main thread free for other activity. Example code for this situation is shown in Figure 3.4.

```
object CollectionExample{
  listen()
  var m : Message = null
  while (true){
      if (!CollectionTreeListener.queue.isEmpty){
        // Retrieve message from queue and write to a SQL database
        m = CollectionTreeListener.queue.dequeue
        // This syntax is from Squeryl (squeryl.org), one of many options
        val done = reportedData.insert(new DataItem(m))
    }
  }
}
```

Figure 3.4: Sample usage of collection tree listening

Chapter 4

Implementation

The current implementation of Scalaness/nesT, which can be found at <https://github.com/pchapin/scala>, has many useful features, but is missing abstractions for sending and receiving messages. The main idea behind this project was to extend the current implementation to include new messaging abstractions. My goal was to define each messaging concept using existing features and protocols found in Scala, Java, and TinyOS, building a functional back end behind my streamlined new messaging commands for Scalaness. I then specified what a parser would need to do to translate each of these Scalaness commands into back end Scala code. For each implementation, there were certain Scala features that proved useful, such as multithreading for the small values code.

In the rest of this section, I will describe the underlying protocols for each implementation, give the details of my implementation, and, finally, explain my evaluation process.

4.1 Small Values

As mentioned in the introduction, sending small values is useful for situations where the hub needs to command one or more nodes to perform a task such as resetting the power supply or changing the format of reported data.

4.1.1 Underlying Protocols

TinyOS provides a protocol called Dissemination which is well suited to the task of sending small values in fixed-size messages over a network. It distributes each message to every node: anycast addressing is straightforward, and multicast or unicast addressing simply requires some nodes to receive the message, pass it on, and then ignore it. The cost is small, though, compared to the great benefit of guaranteed reliability, which Dissemination provides. It is the TinyOS standard protocol for fixed-size messaging. The format for a message is as follows: [command name field — value field — recipient ID field]. As mentioned in the section on proposed communication abstractions, there is some latency inherent in sending information to the network this way; the hub must delay sending a new message until enough time has passed to be sure the old message has reached every node. How long this delay needs to be is dependent on the size of the network. More information about Dissemination can be found at `<http://tinyos.stanford.edu/tinyos-wiki/index.php/Dissemination>`.

4.1.2 Implementation

For sending small values over the network, we defined the command **smallsend(c,v,a)**, where **c** is the command identifier, **v** is the value of the message and **a** is the address to which the message will be sent.

To implement this **smallsend** method for small values, we made use of Scala's

```
// Spawn Preparer
new Thread(new Preparer(drop, queue)).start()
// Spawn Launcher
new Thread(new Launcher(queue, delay)).start()
// Send message to Preparer
drop.put(msg)
```

Figure 4.1: Excerpt from main thread in small values script

multithreading feature. I wrote a script in which the main thread reads information from the command line, a Preparer thread takes the message and adds it to a queue, and a Launcher thread retrieves messages from the queue and sends them off to the network via Dissemination. Figure 4.1 shows how the main thread sets up the other threads and sends a message (the rest can be found in the appendix).

This threaded approach allows the main thread to send off messages whenever it wants without worrying about flow control, and ensures that no messages get lost in transit. For the passing of messages between the main thread and the Preparer, I defined a synchronized Drop, which functions like a mailbox: when the main thread puts a message in it, the Preparer is notified and picks it up. For the passing of messages between the Preparer and the Launcher, I defined a synchronized Queue, which is similar to the Drop but holds multiple messages in first-in-first-out order.

The user of this script provides a command identifier, a value, and an address, which are all integers, but for a message to be send via Dissemination it needs to be a fixed-size byte array (16 bytes is a typical small size). I decided to use the Drop to package the given information into the necessary format. Java provides a conversion tool called a ByteBuffer, into which you can put integers, strings, etc. I was able to import it into Scala and use it to create a byte array containing both the value of the message and the address to which it needs to be sent. Figure 4.2 shows the

```
val b : ByteBuffer = ByteBuffer.allocate(16)
b.order(ByteOrder.nativeOrder)
// Create byte array message containing c, v, and a
b.putInt(c)
b.putInt(v)
b.putInt(a)
val msg_array : Array[Byte] = b.array()
```

Figure 4.2: Excerpt from Drop class in small values script

construction of that byte array.

As explained previously, the main concern here is flow control. The specific number of seconds to wait between messages is defined when the script is started; the Launcher thread takes care of flow control by regulating how frequently messages are taken from the Queue and sent to the network. Figure 4.3 demonstrates the delay in action.

```
Thread.sleep(1000 * delay) // wait the specified number of seconds
if (queue.isEmpty)
  break
var m = queue.take()
```

Figure 4.3: Excerpt from Launcher class in small values script

4.1.3 Evaluation

To verify this implementation's correctness, I wrote a test harness that listens for the output that would ordinarily be sent to the network and prints it out instead. Each time a message was sent, it displayed the contents of the message (i.e. the byte array containing the command identifier, value, and address) and a timestamp. This allowed for verification of the flow control delay as well as the message data: if I had

specified that the script should wait 5 seconds between messages, I could check that the timestamps were at least 5 seconds apart. I ran trials with several different delays and several different messages, and they were all successful.

4.2 Larger Values

Some messages that need to be sent from the hub to one or more nodes do not fit the size restriction for small values. These larger values can include things like machine learning models to help the network operate more efficiently.

4.2.1 Underlying Protocols

Dana Desautels, another student working with Professor Skalka, has demonstrated that it is possible to use Berkeley Low-Power IP (BLIP) to communicate with the network over TCP. BLIP is a TinyOS tool for implementing various IPv6-based protocols and forming multi-hop networks of nodes. All you need is one or more higher-powered devices, such as laptops, to use as edge routers, and your motes. A set of motes connected to one edge router is called a subnet. More information about BLIP can be found at `<http://tinyos.stanford.edu/tinyos-wiki/index.php?title=BLIP_Tutorial&oldid=5642>`.

4.2.2 Implementation

For sending larger values over the network, I defined the command **largesend(v,a)**, where **v** is the value of the message and **a** is the address to which the message will be sent.

Scala provides full support for using sockets in your programs, which was useful

```
// Open socket
val socket = new Socket(addr, 7)
// Create streams for writing and reading
val out = new ObjectOutputStream(
  new DataOutputStream(socket.getOutputStream()))
val in = new DataInputStream(socket.getInputStream())
// Send message
out.writeObject(m)
out.flush()
// Close socket
out.close()
in.close()
socket.close()
```

Figure 4.4: Excerpt from send function in larger values script

here. I wrote a script that opens a socket to the given address, sends the message, and closes the socket. In Dana's work, port number 7 is used for communication with the network; I used the same port here for consistency. An excerpt from the send fuction in this script is given in Figure 4.4.

4.2.3 Evaluation

To verify this implementation's correctness, I wrote a test harness: another process that runs on the same machine, listens on the specified socket, and prints the message to show that it was received properly. I ran trials with several different messages successfully. To communicate with the network, all that needs to be done is to replace the localhost address with the address of the edge router. BLIP will get the message the rest of the way. For addressing, the default is that the edge router is called fec0::64, and each mote is addressed such that mote a is called fec0::a.

```
// Queue for passing messages to the program
val queue : MQueue[Message] = new MQueue[Message]
// When a message is received, add it to the queue to be picked up
def messageReceived(to : Int, message : Message){
  queue += message
}
```

Figure 4.5: Excerpt from collection tree listener script

4.3 Collection Tree Listening

When motes in a wireless sensor network need to report data they've collected to the hub of the network, they can use a collection tree to do so easily and efficiently.

4.3.1 Underlying Protocols

My CollectionTreeListener class is designed to connect with a single mote, which acts as the root of the collection tree, and collect and return any data that comes over that connection. I based the CollectionTreeListener on a TinyOS Java tool called PrintfClient. It is assumed that the mote controls the flow of messages so that the listener receives one at a time.

4.3.2 Implementation

For listening for data being reported from the network, I defined the command **listen()**.

To implement this, I wrote a simple script using the TinyOS MessageListener interface. The excerpt in Figure 4.5 shows that a queue is used to make messages available to the main program.

4.3.3 Evaluation

For this implementation, I confirmed that the existing Java listener in TinyOS can be imported into Scala and used for the purpose of listening on a socket for collection tree data. I also confirmed that my CollectionTreeListener maintains the same basic functionality as the existing listener. To initiate communication with the network, all that needs to be done is to connect a mote to the computer running this program.

4.4 Transformations from Scalaness to Scala Implementations

The following is a specification for the translation of the new messaging commands from Scalaness into working Scala code. Although implementing a parser for these commands is outside the scope of this project, this section provides definitions for everything that this parser would need to do.

Following the conventions of programming language theory, a translation from Scalaness syntax e to Scala syntax S is written as:

$\|e\| = \mathrm{S}$

4.4.1 Sending Small Values

When the program is launched, these imports are performed:

```
import scala.collection.mutable.Queue
import scala.util.control.Breaks._
import scala.util.control.BreakControl
import scala.runtime.ScalaRunTime
import java.nio.ByteBuffer
```

```
import java.nio.ByteOrder
import java.lang.NumberFormatException
```

These global variables are created:

```
// Define flow control delay - 2 is used for testing
val delay = 2
// Create Queue
var queue = new Queue
// Create Drop
val drop = new Drop()
```

And these threads are initialized:

```
// Spawn Preparer
new Thread(new Preparer(drop, queue)).start()
// Spawn Launcher
new Thread(new Launcher(queue, delay)).start()
```

Then, as the program runs, each instance of $smallsend(c, v, a)$ is translated as follows:

$\|smallsend(c, v, a)\| =$ `drop.put(c,v,a)`

From there, the message will be sent to the Preparer thread via the Drop, to the Launcher thread via the Queue, and printed by the Launcher thread to be picked up by the mote and sent to the network. Because the underlying protocol is Dissemination, each mote in the network will receive the message, but only the mote or motes specified in the address field will keep it.

4.4.2 Sending Larger Values

When the program is launched, these imports are performed and these definitions are made:

```
import java.io._
import java.net._
// Associative array for looking up addresses
val addresses = Map((1,InetAddress.getByName("localhost")))
def largesend(m: Array[Byte], a: Int) = {
  try {
    // Get address
    val addr = addresses(a)
    // Open socket
    val socket = new Socket(addr, 7)
    // Create streams for writing and reading
    val out = new ObjectOutputStream(
      new DataOutputStream(socket.getOutputStream()))
    val in = new DataInputStream(socket.getInputStream())
    // Send message
    out.writeObject(m)
    out.flush()
    // Close socket
    out.close()
    in.close()
    socket.close()
  }
  catch {
    case e: IOException =>
      e.printStackTrace()
  }
}
```

Then, as the program runs, each instance of $largesend(v, a)$ is translated as follows:

$\|largesend(v, a)\| =$ `largesend(v,a)`

This allows for a new socket connection to be made each time a new address and corresponding message are given.

4.4.3 Listening for Collection Tree Data

At the beginning of the program, the following imports are performed to provide the necessary support for the listener:

```
import java.io.IOException
import net.tinyos.message._
import net.tinyos.tools._
import net.tinyos.packet._
import net.tinyos.util._
// Scala Queue type renamed as MQueue to avoid conflict with TinyOS Queue
import scala.collection.mutable.{Queue=>MQueue}
```

After the program is launched, the command *listen*() is translated as follows:

$\|listen()\| =$

```
object CollectionTreeListener extends MessageListener{
  // Interface to the mote
  var mote : MoteIF = new MoteIF
  // Queue for passing messages to the program
  val queue : MQueue[Message] = new MQueue[Message]
  // When a message is received, add it to the queue to be picked up
  def messageReceived(to : Int, message : Message){
    queue += message
  }
  def usage() {
    System.err.println("usage: CollectionTreeListener <source>")
  }
```

```
  def main(args : Array[String]) = {
    var source : String = null
    if (args.length != 1) {
      usage()
      System.exit(1)
    }
    source = args(0)
    var phoenix : PhoenixSource = null
    if (source == null) {
      phoenix = BuildSource.makePhoenix(PrintStreamMessenger.err)
    }
    else {
      phoenix = BuildSource.makePhoenix(source, PrintStreamMessenger.err)
    }
    System.out.println(phoenix)
    var mif : MoteIF = new MoteIF(phoenix)
    this.mote = mif
    var msg : PrintfMsg = new PrintfMsg()
    this.mote.registerListener(msg, this)
  }
}
```

The queue can then be accessed through CollectionTreeListener.queue and items can be removed, as shown in the example in Section 3.2.3.

Chapter 5

Conclusion

At the beginning of this paper, I introduced the key concepts behind wireless sensor networks and the Scalaness programming language. I presented three main communication objectives and illustrated them with a sample network scenario. The communication patterns we wanted to implement were small values, larger values, and collection tree listening. Each has its own uses, advantages, and disadvantages, which I took into consideration when I designed the abstractions for these patterns. I then built an implementation of each type of messaging in Scala and defined translations between these implementations and the corresponding Scalaness syntax. When incorporated into Scalaness, my code will carry messages back and forth to the connection between the hub and the network, and the underlying TinyOS protocols will complete the picture by handling communication among nodes in the network.

In general, my goal has been to design and implement high-level messaging primitives for Scalaness. There is more to be done, but my hope is that by advancing this aspect of the language I have made contributions to the Scalaness project that will be useful both immediately and in ongoing research.

5.1 Future Work

There are several possibilities for extending this project in the future. I will explore two of them here.

The first and most natural extension is to incorporate these hub-based messaging implementations into a fully functional network communication system. Another student or researcher could write code that connects with mine at the point where messages cross from the hub to the network. While my work has demonstrated that these concepts work at a relatively high level, a logical next step would be to tailor their use to the needs of a specific network. Scalaness is a powerful tool for managing wireless sensor networks, especially with communication capabilities added.

Another possible direction that future work could explore would be adding one or more additional types of messaging. For example, one of the Scalaness team's long-term goals is to add dynamic reprogramming for the motes. Recall from the introduction that as a staged programming language, Scalaness has the ability to generate nesT code for the motes to run. When new code is generated, rather than transferring that code to the motes by hand, researchers could save time and effort if they could deploy it over the network. Automatic over-the-air reprogramming would allow near-instant adaptation to changes in the network and its environment. It would move systems based on Scalaness one step closer to full autonomy, freeing the people in charge to focus their efforts in other areas. We could add a type of messaging to Scalaness, similar to the types presented here, specifically for over-the-air reprogramming.

Appendix A

Full Source Code

SmallValues.scala

```
import scala.collection.mutable.Queue
import scala.util.control.Breaks._
import scala.util.control.BreakControl
import scala.runtime.ScalaRunTime
import java.nio.ByteBuffer
import java.nio.ByteOrder
import java.lang.NumberFormatException

object SmallValues {
  class Preparer(drop : Drop, queue : Queue)
    extends Runnable{
      override def run() : Unit =
      {
        // Receive packets from main thread and add them to queue
        breakable{
          while (true)
```

```
        {
          val message : Array[Byte]= drop.take()
        queue.put(message)
        }
      }
     }
   }

class Launcher(queue : Queue, delay : Int)
  extends Runnable{
    override def run() : Unit =
    {
      // Take packets from queue and output them
      // Mote running Dissemination can read from this output and send
      // messages to network
      breakable{
        while (true)
        {
          Thread.sleep(1000 * delay) // wait the specified number of seconds
          if (queue.isEmpty)
            break
          val m : Array[Byte] = queue.take()
          println(m)
          // For testing, print a string representation of the message
          // and a timestamp
          // val time = new java.sql.Timestamp(System.currentTimeMillis())
          // println(m.mkString + " " + time)
        }
      }
    }
  }
```

```
class Drop
{
  var message : Array[Byte] = new Array[Byte](16)
  var empty : Boolean = true
  var lock : AnyRef = new Object()

  def put(c : Int, v : Int, a : Int) : Unit =
    lock.synchronized
    {
      // Wait until message has been retrieved
      await (empty == true)
      // Toggle status
      empty = false

      // Package information into bytes
      val b : ByteBuffer = ByteBuffer.allocate(16)
      b.order(ByteOrder.nativeOrder)

      // Create byte array message containing c, v, and a
      b.putInt(c)
      b.putInt(v)
      b.putInt(a)
      val msg_array : Array[Byte] = b.array()
      message = msg_array

      // Notify preparer that status has changed
      lock.notifyAll()
    }

  def take() : Array[Byte] =
```

```
    lock.synchronized
    {
      // Wait until message is available.
      await (empty == false)
      // Toggle status
      empty=true
      // Notify initiator that status has changed
      lock.notifyAll()
      // Return the message
      message
    }

  private def await(cond: => Boolean) =
    while (!cond) { lock.wait() }
}

class Queue
{
  var queue : scala.collection.mutable.Queue[Array[Byte]] =
    new scala.collection.mutable.Queue[Array[Byte]]
  var lock : AnyRef = new Object()

  def put(x: Array[Byte]) : Unit =
    lock.synchronized
    {
      // Store message
      queue += x
      // Notify preparer that status has changed
      lock.notifyAll()
    }
```

```
  def take() : Array[Byte] =
    lock.synchronized
    {
      // Notify initiator that status has changed
      lock.notifyAll()
      // Return the message
      queue.dequeue()
    }

  private def await(cond: => Boolean) =
    while (!cond) { lock.wait() }

  def isEmpty : Boolean =
    return queue.isEmpty
}

def main(args : Array[String]) : Unit =
{
  // Define variables c, v, and a
  var c : Int = 0
  var v : Int = 0
  var a : Int = 0

  // Read in command line arguments
  if (args.length != 2) {
    println("Must have three arguments: command name, value, and address")
    System.exit(1)
  }
  try{
    c = args(0).toInt
    v = args(1).toInt
```

```
      a = args(2).toInt
    }
    catch{
      case e : NumberFormatException => println("All arguments must be integers")
      System.exit(1)
    }

    // Define flow control delay - 2 is used for testing
    val delay = 2

    // Create Queue
    var queue = new Queue
    // Create Drop
    val drop = new Drop()

    // Spawn Preparer
    new Thread(new Preparer(drop, queue)).start()
    // Spawn Launcher
    new Thread(new Launcher(queue, delay)).start()

    // Define messages to be sent - for testing
    // var messages : Array[Int] = Array(0, 1, 2)
    // Send test messages
    // messages.foreach((msg) => drop.put(0, msg, a))

    // Send information to Preparer
    drop.put(c, v, a)
  }
}
```

LargerValues.scala

```
import java.io._
import java.net._

object LargerValuesSender {
  // Associative array for looking up addresses
  // Right now it just has localhost, but can easily
  // be expanded to hold as many mote addresses as needed
  val addresses = Map((1,InetAddress.getByName("localhost")))

  def largesend(m: Array[Byte], a: Int) = {
    try {
      // Get address
      val addr = addresses(a)

      // Open socket
      val socket = new Socket(addr, 7)

      // Create streams for writing and reading
      val out = new ObjectOutputStream(
        new DataOutputStream(socket.getOutputStream()))
      val in = new DataInputStream(socket.getInputStream())

      // Send message
      out.writeObject(m)
      out.flush()

      // Close socket
      out.close()
```

```
      in.close()
      socket.close()
    }
    catch {
      case e: IOException =>
        e.printStackTrace()
    }
  }

  def main(args: Array[String]): Unit = {
      // Test message is all 1s for now
      var message : Array[Byte] = Array.fill[Byte](64)(1)
      // Send test message to receiver on localhost
      largesend(message, 1)
  }
}

object LargerValuesReceiver {
  def main(args: Array[String]): Unit = {
    try {
      // Listen for a connection on port 7
      val listener = new ServerSocket(7);
      // When a connection is made, start the ListenerThread
      while (true)
        new ListenerThread(listener.accept()).start();
      listener.close()
    }
    catch {
      case e: IOException =>
        System.err.println("Could not listen on port: 7.");
        System.exit(-1)
```

```
    }
  }
}

case class ListenerThread(socket: Socket) extends Thread("ListenerThread") {
  override def run(): Unit = {
    try {
      // Create streams for writing and reading
      val out = new DataOutputStream(socket.getOutputStream());
      val in = new ObjectInputStream(
        new DataInputStream(socket.getInputStream()));

      // Read in message
      val message = in.readObject().asInstanceOf[Array[Byte]];

      // Print first byte of message to show that it was received
      println("Received: " + message(0))

      // Close socket
      out.close();
      in.close();
      socket.close()
    }
    catch {
      case e: IOException =>
        e.printStackTrace();
    }
  }
}
```

CollectionTreeListener.scala

```
import java.io.IOException
import net.tinyos.message._
import net.tinyos.tools._
import net.tinyos.packet._
import net.tinyos.util._
// Scala Queue type renamed as MQueue to avoid conflict with TinyOS Queue
import scala.collection.mutable.{Queue=>MQueue}

object CollectionTreeListener extends MessageListener{
  // Interface to the mote
  var mote : MoteIF = new MoteIF

  // Queue for passing messages to the program
  val queue : MQueue[Message] = new MQueue[Message]

  // When a message is received, add it to the queue to be picked up
  def messageReceived(to : Int, message : Message){
    queue += message
  }

  def usage() {
    System.err.println("usage: CollectionTreeListener <source>")
  }

  def main(args : Array[String]) = {
    var source : String = null
    if (args.length != 1) {
      usage()
```

```
      System.exit(1)
    }
    source = args(0)

    var phoenix : PhoenixSource = null
    if (source == null) {
      phoenix = BuildSource.makePhoenix(PrintStreamMessenger.err)
    }
    else {
      phoenix = BuildSource.makePhoenix(source, PrintStreamMessenger.err)
    }
    System.out.println(phoenix)
    var mif : MoteIF = new MoteIF(phoenix)
    this.mote = mif
    var msg : PrintfMsg = new PrintfMsg()
    this.mote.registerListener(msg, this)
  }
}
```

Bibliography

[1] Abadi, Martn, and Andrew D. Gordon. "A calculus for cryptographic protocols: The spi calculus." *Proceedings of the 4th ACM conference on Computer and communications security*. ACM, 1997.

[2] Beauxis, Romain, Catuscia Palamidessi, and Frank D. Valencia. "On the asynchronous nature of the asynchronous pi-calculus." Concurrency, Graphs and Models. Springer Berlin Heidelberg, 2008.

[3] Cardelli, Luca, and Andrew Gordon. "Mobile ambients." *Foundations of Software Science and Computation Structures*. Springer Berlin/Heidelberg, 1998.

[4] Ene, Cristian, and Traian Muntean. "A Broadcast-based Calculus for Communicating Systems." *IPDPS*. (2001).

[5] Fielding, Roy T., and Richard N. Taylor. "Principled design of the modern Web architecture." *ACM Transactions on Internet Technology* (TOIT) 2.2 (2002): 115-150.

[6] Hoare, Charles Antony Richard. "Communicating sequential processes." *Communications of the ACM* 21.8 (1978): 666-677.

[7] Milner, Robin. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[8] Milner, Robin. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[9] Prasad, Kuchi VS. "A calculus of broadcasting systems." *Science of Computer Programming* 25.2 (1995): 285-327.

[10] Skalka, Christian, et al. *Scalaness/nesT: Type Specialized Staged Programming for Sensor Networks*. Submitted for publication Feb. 2013.