Simple Sensor Syndication

Final Report

Marianne Graham   Wade Simmons

June 22, 2006

Abstract

Wireless sensor networks are useful for collecting large amounts of data from an environment. Usually, all of the data are collected in a database and analyzed offline. However, by the time the data is evaluated, it is outdated and may not be useful. The goal of this project is to make wireless sensor networks more interactive by allowing scientists to define data events of interest and receive notification for only those events online.

To accomplish this task, we wrote software for the wireless sensors and the data collection server. The sensors in BB 154 collect information on light, temperature, and humidity. The software defines when a sensor should send back data and possibly what data to send. The application on the server decides when events occur, such as considerable changes in temperature or light readings, and publish an RSS feed for the client. Scientists can define events and subscribe to the RSS notifications through a web interface. The final deliverable is a demo which involves walking into a sensor network with a laptop, specifying interesting events, and then monitoring those events in real time. This demo will be submitted to ACM SenSys 2006, one of the top conferences in sensor networks.

The software for the sensors is written in nesC as a TinyOS component, meaning that any TinyOS application can easily include it and utilize our application. The server application is written in Python due to portability, availability of preexisting modules, and rapid development time.

Contents

1 Requirements **4**
1.1 Introduction 4
1.1.1 Purpose 4
1.1.2 Scope 4
1.1.3 Definitions, Acronyms and Abbreviations 5
1.2 Overall Description 6
1.2.1 User Characteristics 6
1.2.2 Product Overview 6
1.2.3 Constraints 7

2 Design **8**
2.1 Event-Oriented Design 8
2.2 Sensor Side Design 9
2.2.1 Components 9
2.2.2 Packets 9
2.2.3 Tasking the Network 10
2.2.4 Building a Routing Tree 10
2.2.5 Sending Data to the Server 11
2.3 Server Side Design 11
2.3.1 Networking Subsystem 12
2.3.2 Data Regions 13
2.3.3 Event Subsystem 14
2.3.4 RSS Publisher 14
2.3.5 Web Interface 14
2.4 SenSys Demo 15

3 Results **16**
3.1 Overview 16
3.1.1 Sensor Side 16
3.1.2 Server Side 16
3.1.3 Web Interface 17
3.1.4 SenSys Demo 17
3.2 Lessons Learned 18

3.2.1 Testing Drain with Low Radio Power 18
3.2.2 Web Interface 18
3.2.3 No Software is Bug-Free 18

Bibliography **18**

List of Figures

1.1 Connection and flow of the SSS application. 6

2.1 Mailman-SSS analogy 8
2.2 Incorporation of nesC components and how they communicate with each other and the server 9
2.3 Drip packets being sent and rebroadcasted through the network 10
2.4 Routing tree being build 11
2.5 Drain packets going to the base mote using multihop routing 12
2.6 SSS interface with Drain 12
2.7 Subsystems and logic flow of SSS server application 13
2.8 Reverse engineered Serial Forwarder protocol 13
2.9 Connection of web interface and SSS back-end 15

Chapter 1

Requirements

1.1 Introduction

1.1.1 Purpose

The purpose of this project is to create an application for wireless sensor networks that allows a scientist to define and subscribe to specific events that happen in the network. At this time, wireless sensor networks usually collect large amounts of data and store them in a database to be analyzed offline. However, this time delay may cause the data to become useless to the scientist. Our project will solve this problem by allowing scientists to choose what data they want to be notified about while the wireless sensor network is online.

Due to the constraints of wireless sensors, we must program them in the nesC language. Our client strongly encouraged us to use Python for the server application because it provides tools, such as the Twisted library, that make it easier to work with the sensors. The application must be easy to use, and the nesC component must be easy to include in other applications. It also must not interfere with any applications it is added to. The motes (defined in section 1.1.3) used must be *Tmote Sky*s.

1.1.2 Scope

The scope of the project has been broken down into mandatory and optional requirements. Our client wants us to complete an application for the wireless sensors and for the base station server (which stores all the data). These applications should allow a user to define events that can occur in the network and subscribe to receive notification of the events as they happen. The second mandatory goal is to create a demo for the SenSys conference in the fall. This means we must create some demonstration based on our research that will be exciting to watch and show off the full functionality of our project.

Although it is not necessary, our client strongly suggests we create a web interface to access our application. This would create a simple method for scientists to define events and subscribe to notifications.

1.1.3 Definitions, Acronyms and Abbreviations

Here is a list of definitions and acronyms that may be useful when reading this paper:

mote A wireless sensor (or mote) is a small device with a CPU, memory, radio, and usually various sensors. We will use Tmote Sky motes, designed by Moteiv. The constraints associated with the motes are limited processing power (8MHz, 16-bit CPU), minimal memory and storage (10k RAM, 48k Flash), and finite lifetime (small batteries).

nesC nesC is a programming language designed to build applications for the TinyOS platform. TinyOS is an operating environment designed to run on distributed embedded Wireless Sensor Networks. nesC is built as an extension to C with components "wired" together to run applications on TinyOS [1].

TinyOS TinyOS is an embedded operating system, written in nesC programming language, as a set of cooperating tasks and processes. It is designed to incorporate rapid innovation as well as to operate within the severe memory constraints inherent in sensor networks [1].

Multihop Routing Multihop routing is necessary when using wireless radios where each mote can only communicate with a small subset of the whole network. Messages are passed between multiple nodes until they reach their destination.

Python Python is an interpreted programming language created by Guido van Rossum in 1990. Python is fully dynamically typed and uses automatic memory management; it is thus similar to Perl, Ruby, Scheme, Smalltalk, and Tcl [1].

Some benefits of Python are:

1. Portability
2. Numerous preexisting modules
3. Rapid development time
4. A simple, yet powerful, syntax

Twisted Twisted is an event-driven networking framework written in Python and licensed under the MIT license. Twisted projects support TCP, UDP, SSL/TLS, multicast, Unix domain sockets, a large number of protocols (including HTTP, NNTP, IMAP, SSH, IRC, FTP, and others), and much more. Twisted is based on the event-driven programming paradigm, which means that users of Twisted write short callbacks which are called by the framework [1].

CherryPy CherryPy is a framework for object-oriented web development using the Python programming language. It is designed for rapid web application development by wrapping the HTTP protocol but stays at a low level [1].

Cheetah Cheetah is a template engine that uses the Python programming language. It can be used standalone or combined with other tools and frameworks. It is often used for server-side scripting and dynamic web content by generating HTML, but can also be used to generate source code [1].

RSS RSS is a family of web feed formats, which are specified in XML and used for Web syndication. RSS is used by (among other things) news websites, weblogs and podcasting [1].

SSS Simple Sensor Syndication (SSS), is the name we will be using for our project

1.2 Overall Description

1.2.1 User Characteristics

The users will be educated scientists who already know how to program. They will have a good knowledge of wireless sensor networks and how to define events for the motes. The users do not have to be experts on nesC or Python, but a basic knowledge of the languages would be helpful. However, the web interface should make it easy to define new events without such expertise.

1.2.2 Product Overview

Because the project has three different applications within it, it is best to describe the sections separately. The applications will be for the sensors, the server, and the web interface. Figure 1.1 shows the connection between these three components.

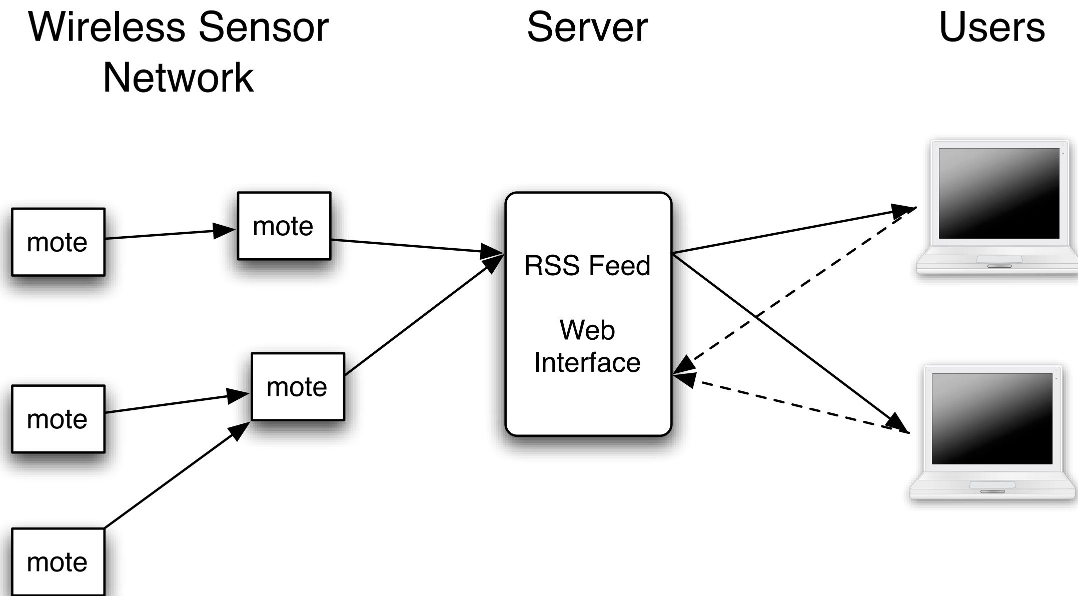

Figure 1.1: Connection and flow of the SSS application.

Sensor Application

The application for the sensors will be written in nesC. The SSS component will have functions that can be used by other applications. It must be easy to incorporate and it cannot affect the other applications. It will be used to send back data each time it is called or for a given number of times.

The other applications we will write for the motes will be used for demos. These may include sensing significant changes in light (when a light is turned on or off), using IR sensors to detect motion, and checking the light reading from an overhead projector. The applications must show off the functionality of the SSS component. It also must be interesting and possibly useful to scientists that would attend the demo.

Server Application

The application on the server will be used to collect and analyze the data coming in from the motes. It must publish events as an RSS feed for users to subscribe to. The user must be able to write event detectors that will specify when events occur, and how the events are published in the RSS feed. The application should be easy to use and make it so anybody can "walk" into the wireless sensor network and receive RSS feeds. It also must be general enough to allow for many different kinds of events. Helper methods will be defined (such as computing the average of the data from a set of motes) so that it is easier for scientists to define new event detectors.

Web Interface

The web interface will be used by scientists to define events and subscribe to RSS feeds. It will make it easier for the user to interact with the server application. It must look nice and be easy to use. It also must allow enough freedom so that scientists can define a wide variety of events. We will assume that the user knows how to write Python and nesC code that is compatible with the motes, SSS component and server application. Python and nesC code or outlines will be provided to assist the user when writing events.

1.2.3 Constraints

Because the motes are running on batteries, our component and demos try to conserve battery power as much as possible. One suggestion from the client was to use a component called Drain because it allows for multihop routing (defined in section 1.1.3). Our component is also small in order to conserve the limited memory of the motes.

Chapter 2

Design

2.1 Event-Oriented Design

The overall design of our project is event-oriented. This means that rather than describing what happens all the time in our applications, we only describe what happens when an event occurs. In order to explain how this works, we can relate what is happening to what happens with mail. The only time somebody goes to the post office is when they have something to send. There is no reason for them to go if they have nothing to send. Likewise, the mailman only comes to your house if he has a letter for you. In our project, there is no reason to do anything when there is not an event to handle. Figure 2.1 shows how the two situations relate.

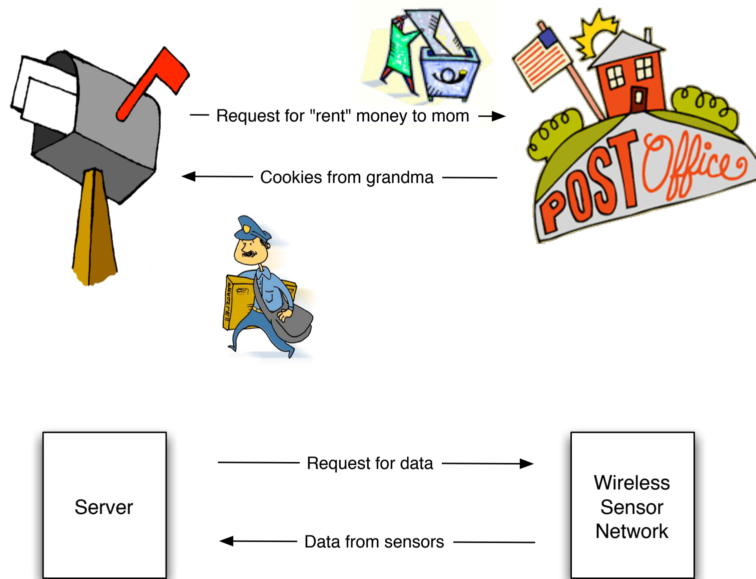

Figure 2.1: Mailman-SSS analogy

There are several benefits with an event-driven system:

- TinyOS, Twisted and RSS feeds are all event driven.
- We only need to define what happens when an event occurs.
- Allows easy definition of requirements.
- Conserves battery power, which is extremely important in WSNs.

2.2 Sensor Side Design

The application that will run on the sensors was written in nesC [3]. The SSS component has functions that can be used by other applications. It will be used to send back data each time an event is detected or a request from the server is received.

2.2.1 Components

Components can be used by other nesC applications by being wired in and fulfilling the interface requirements. Wiring components together means relating parts of one program to parts of another. A component's interface will describe what must be defined by the user of the interface and what can be used by the other application. See Figure 2.2.

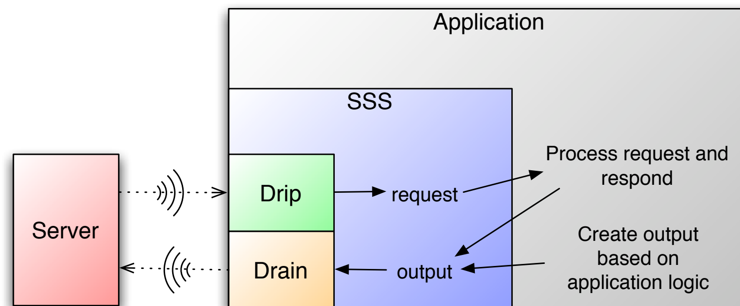

Figure 2.2: Incorporation of nesC components and how they communicate with each other and the server

2.2.2 Packets

To optimize our component, we created our own packet types. We needed two different kinds of packets: one to send commands from the server to the motes, and one to send data from the motes to the server. The command packets will contain the data the server needs the motes to send back. The data packets will contain a field describing what kind of data is being sent back and a field holding the data.

There are several reasons we created our own packets:

- Allows us to define what information will be sent.
- Design changes that require packet changes will not create too much extra work.
- Allows us to define data types within the packet. The data types will describe which sensor the data is coming from.
- Saves batteries because the radio is only transmitting as much information as is required by our component.

2.2.3 Tasking the Network

In order to send packets out to the network, we use a nesC component called Drip. Drip was written by the creators of TinyOS and is used to send packets out through the network. Each mote listens for Drip packets at all times, which works well in our event-oriented network. Once a Drip packet is received, the data must be stored on the mote in case it needs to be retransmitted. However, no matter how many times a mote hears the same Drip packet, it will only analyze the data once.

Drip makes our work much simpler. It has only two events (methods) that we must define to use it: request and rebroadcast. In addition, it takes care of the new packet detection and rebroadcasting, so we do not need to do any additional work to ensure that these features work. Figure 2.3 shows how Drip packets make it to every mote within the network.

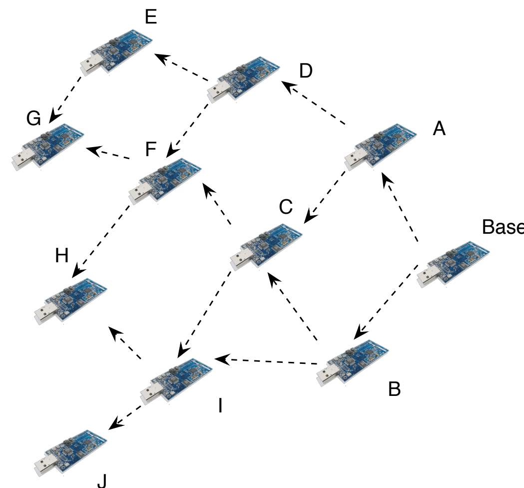

Figure 2.3: Drip packets being sent and rebroadcasted through the network

2.2.4 Building a Routing Tree

In order to get messages back to the base station, we use another component called Drain. It first builds a routing tree to the base station for the motes to send messages through, then it packages up messages and sends them through the network according to the tree.

To build the tree, a packet is sent out from the base station requesting that a routing tree be built. Any mote that can hear the first packet is one hop away, meaning that its packet must travel to and get resent by only one other mote. These motes then rebroadcast the tree building packet to anybody who can hear. Motes that are one hop away ignore the second broadcast of this packet since they are already in the tree, but any new motes that hear the packet know they are two hops away. The process continues until there are no longer motes left to add to the tree. Figure 2.4 illustrates how a tree is built.

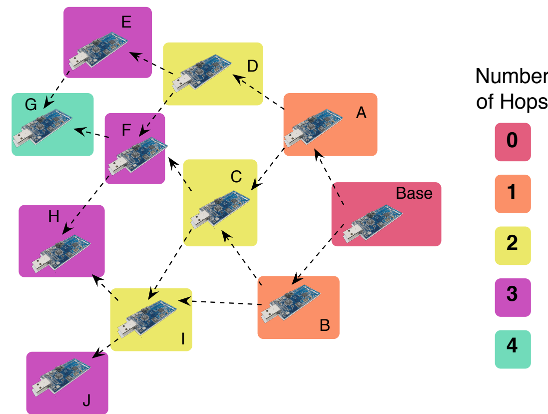

Figure 2.4: Routing tree being build

2.2.5 Sending Data to the Server

After building the routing tree, Drain then takes in another packet to be sent back to the base station. The packet, which will be a SSS packet in our component, is wrapped in a Drain packet that has some extra information about the data, such as where it came from. Drain takes care of moving the packet through the tree and delivering it to the base station. Figure 2.5 shows how the packets are routed to the base according to the tree.

Like Drip, Drain only needs to have a few events defined. These events are confirmations of commands in the Drain component. Our SSS component can call one Drain command to get a block of memory to store the SSS packet in, and Drain will signal an event in our SSS component when it has finished. There is another command in Drain to send off the packet, with another signal coming from Drain saying the command is complete. This is better illustrated with Figure 2.6.

2.3 Server Side Design

The server application was broken into a few smaller subsystems (shown in Figure 2.7). First, a *networking subsystem* was constructed to communicate with the sensors. This subsystem stores the readings from the sensors in *data regions*. Concurrently, the *event subsystem* will periodically run the event detectors on these data regions to determine when events have occurred. When an

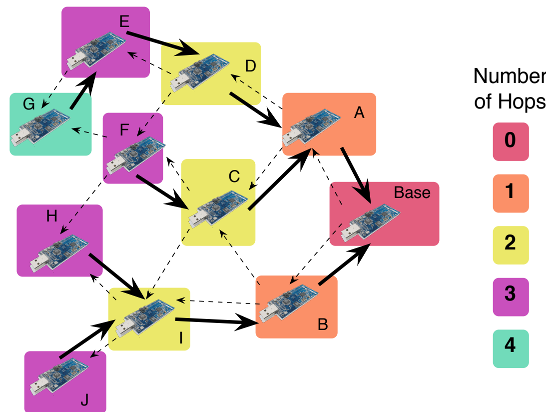

Figure 2.5: Drain packets going to the base mote using multihop routing

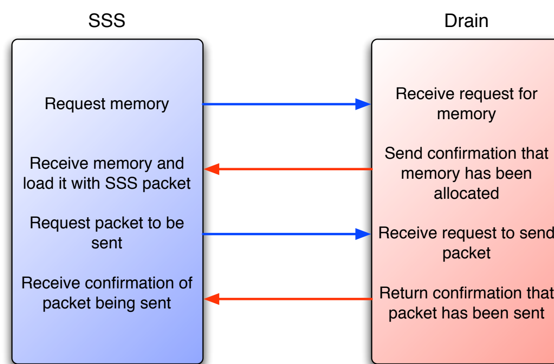

Figure 2.6: SSS interface with Drain

event occurs, information about it will be sent to the *RSS publisher*, which will add an item to any applicable RSS feeds.

2.3.1 Networking Subsystem

The server application will communicate to the sensors through a mote programmed with the *TOSBase* application. This program, provided with the TinyOS distribution, acts as a bridge between the sensor network and a computer. Motes programmed with *TOSBase* will forward any radio packets they hear to their USB port, as well as transmit any data sent to them over the USB port.

Because the USB ports of these motes will be connected to hardware devices (we will use TMote Connects), the data must be sent over IP to any computers that want to communicate with the wireless sensors. The TinyOS group solved this problem by developing a serial forwarder protocol, along with servers and clients written in Java. Because we would like to program our application server in Python, we need to implement a client for this protocol.

We could not find much documentation on the specifics of the serial forwarder protocol, but since TinyOS is open source we were able to look at the source code to try to decipher it. Com-

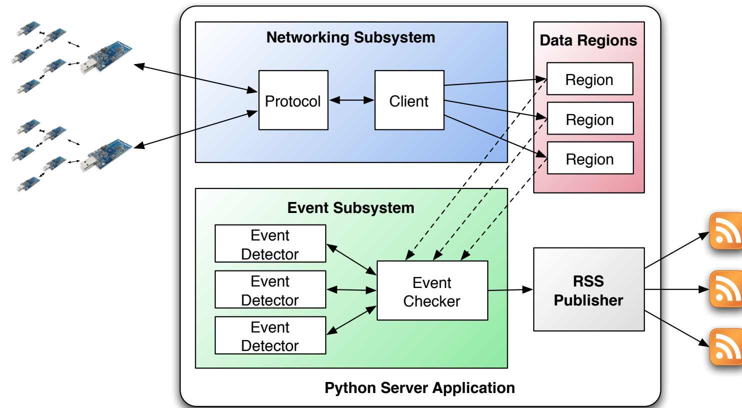

Figure 2.7: Subsystems and logic flow of SSS server application

bining this with information collected with a packet sniffer, we were are able to understand the protocol (details shown in Figure 2.8).

To implement the networking subsystem in Python, we used the Twisted Framework [5] [6]. Twisted allowed us to implement the above protocol with ease, and connect to as many TOSBase motes as we want to without adding any complexity to our program.

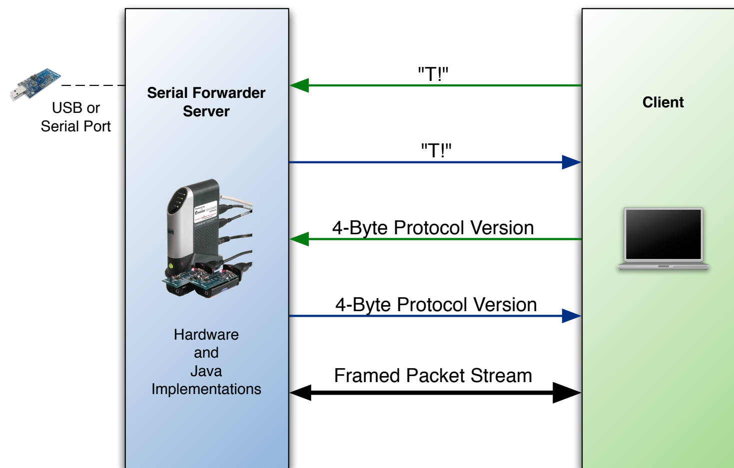

Figure 2.8: Reverse engineered Serial Forwarder protocol

2.3.2 Data Regions

Because we have a large amount of data coming in from the sensors, we need an intelligent way of storing it so that the event detectors can grab what they need. Each data region will be a function of the data that comes in the system. For example, a data region could be defined as mote id < 3

and > 7, which means it will hold the data from motes 4, 5 and 6. Another example is getting only specific types of data, such as humidity or temperature data.

To do this, the data regions are implemented as maps. Every data element that arrives in the system will have a unique ID assigned to it. Initially, this ID will be generated from the data type and source address, but this can be changed in the future. Whenever new data with the same UID as a previous item of data arrives, it will overwrite the old data. This will be adequate for the type of event detectors that we plan to write, but the regions could be changed in the future to store previous data elements if necessary for more complicated detectors.

These data regions have many helper methods defined so that it is easy for event detector functions to parse out only the data they are interested in.

2.3.3 Event Subsystem

Event detectors were implemented as simple classes with an `event` method defined which takes in a data region as its lone parameter. The event detector will analyze the data in the region, and return an Event object describing any events that have occurred in the region.

To allow concurrent event detectors in the system, there is an EventChecker class which will schedule periodic calls to these event detectors. This scheduling will be done with the `loopingCall` function provided by the Twisted framework, which allows a function to be called every n seconds.

2.3.4 RSS Publisher

When events are detected by event detectors, a relevant entry in the RSS feed must be created. To do this, an RSS publisher class was created. It will accept the Event objects returned by the event detectors, and convert them into a format that can be put into the feeds. We used a Python module called PyRSS2Gen [7], which abstracts the creation of RSS feeds into simple objects and methods.

2.3.5 Web Interface

The web interface will be a key part of the deliverables, because it allows the user to interact with the system. We would like the back-end (server) of the system to be as loosely coupled to the front-end (user interface) as possible; therefore, we implemented them as separate applications with a standard interface between them. For this interface, we decided to use XML-RPC for several reasons [9]. First of all, it is a very simple RPC implementation, which means that there are implementations for many different languages. Second, the Twisted framework has support for creating XML-RPC servers and clients. XML-RPC is a method for performing remote procedure calls. This means that an application can run functions and retrieve the results on a different application (possibly written in a different language, and running on a different computer). Using XML-RPC will allow the web interface to request data and make changes to the server.

The web interface was developed using CherryPy [10]. We used CherryPy so that we can still make use of tools such as Twisted, and also not have to waste any time learning a new language. The connection of the web interface and back-end are shown in Figure 2.9

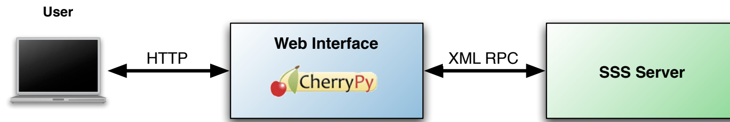

Figure 2.9: Connection of web interface and SSS back-end

2.4 SenSys Demo

For the SenSys 2006 conference [11], we would like to prepare an interactive demo to show off the usefulness of an interactive sensor network. To do this, we will have a number of sensors set up in the conference room (hopefully about ten) collecting various types of data, such as light, humidity, and possibly infrared light. We will then connect a base station mote (a sensor that acts as a bridge between the wireless radios and the server) to a laptop, and run our server application on it. We would then like to set up a few laptops to act as users of the sensor network, which will be connected to the web-interface of the server.

The main focus of our demo will be to demonstrate the interactivity promoted by our proposal. Conference attendees will be able to walk up to our client laptop, create new event event detectors and watch the resulting RSS feed. We hope that by showing how easy it is, we can win some support.

As an added note, we would also like to allow conference attendees to use their own laptops to connect to our server. This should be fairly easy to do, since the interface to our application will be primarily web-based. The only issue would be if the conference room did not have wireless access available. This problem could be solved in two ways. First, our server could host an ad hoc network which attendees could connect to. Second, we could bring a network switch to the conference, and allow attendees to physically connect to it.

Chapter 3

Results

3.1 Overview

The project had several different subsystems, each of which had mandatory and optional requirements. Although we were not able to complete all of the optional requirements, we were able to complete all of the mandatory ones.

3.1.1 Sensor Side

We completed all of the sensor requirements. We made a nesC component that can be easily wired into any other application. Besides wiring our component in the configuration file, the application also includes an event called “Sss.request” and can call “Sss.output” to send packets back to the server. We tested it with several already existing and newly made applications, and successfully integrated our component without negatively impacting the original applications.

In order to make a demo, we created our own nesC applications. Some of them are SenseSssDemo, BlinkSss, and ButtonSss. BlinkSss toggles the red LED and sends packets back after a certain number of blinks. ButtonSss sends out a packet every time the button on the mote is pushed. SenseSssDemo sends out a packet every time the data from any sensor changes more than the threshold. It also sends out the current reading when it receives a request for that data.

3.1.2 Server Side

On the server side, we completed nearly all of our requirements. The networking subsystem for communicating with the motes was completed and works as expected. Because the protocol was partially reverse engineered, some work may still need to be done to make the sure the protocol is implemented correctly. Twisted allows us to connect to multiple serial forwarders. The building of the Drain routing tree is not done automatically by our program, and therefore must be done manually by the user (which involves the execution of a simple Java program).

The data regions subsystem was completed with a few layers of abstraction so that changes can easily be made in future. Data is stored in the regions keyed by a “unique ID” generated from the

data. Currently, the data is considered unique by its source and data type, but this can easily be changed later by manipulating the unique ID generation function.

The event checking subsystem was implemented successfully, with the support of dynamically importing new event detectors added by the user, and simple instantiation of detectors and mapping them to regions. The only feature that did not get completed was the individual manipulation of the scheduling of these event detectors, (as they currently all run at the same frequency).

The RSS publisher subsystem was implemented using a Python module to abstract the generation of the XML so that newer versions of RSS can be easily supported in the future. Multiple feeds can be set up by the user, and can be configured to output to the folder of the web server.

3.1.3 Web Interface

An XML-RPC interface to the server was successfully created. This interface allows other programs to get information about the system, as well as make changes to it. We created a web application using CherryPy which allows access to the RSS feeds, and the configuration of the server. The web interface fulfilled its requirements, and allows manipulation of nearly every aspect of the server.

Future improvements to the web interface would include security features such as user accounts with limited privileges and a more friendly user interface. Due to the fact that this field session focused on developing a good demo for the SenSys conference, these features can be implemented later.

3.1.4 SenSys Demo

We have developed a few demonstrations of the Simple Sensor Syndication project to use at SenSys 2006. The first demo we have designed involves setting up a projector to shine on six sensors attached to the wall. The projector will be set up to show a flash video of a burglar robbing a house. Each of the six motes will be set up in rooms inside the house, and their light sensors will be positioned so that when the burglar enters the room, he will move over it. The motes will be programed to detect this change in light over the sensor, and report an event to the server. Event detectors will be set up on the server to detect these events and output them to an RSS feed. This demo has been completed and works as expected.

Another demo we have planned is attaching motes to stuffed animals, and tracking their position in a room. To do this, each sensor will be programmed to send a beacon at a specified interval. Motes will be placed on the ceiling of the room to receive these beacons, calculate the received signal strength and send this information back to the server. The server will take these signal strengths, and try to determine the location of the stuffed animal (probably by assuming it is closest to the mote that received the highest signal strength). We are currently testing this demo to make sure the signal strength information will be enough to determine the location.

3.2 Lessons Learned

We had several problems while implementing and testing our design.

3.2.1 Testing Drain with Low Radio Power

While testing Drain, we wanted to turn down the radio so we could test within the office. However, while trying to test, we could not get all of the packets back to the server. We thought the motes farther away from the base mote would use multihop routing to get packets to the server.

After a while, we finally figured out that because we did not turn down the radio on the base mote, the Drain tree was set up incorrectly. Because all of the motes could hear the initial packet sent out by the base mote, they thought they were one hop away. But when they tried to send packets back, the radio was too weak to send all packet back to the base mote. In order to fix the problem, we had to turn down the radio on the base mote so the tree was built correctly. The radio on the base mote must be weaker than the application mote with the weakest radio power.

3.2.2 Web Interface

At first, the web interface seemed like a relatively simple part of this project. After working on the web interface for a while, we realized that we did not have all of the functions in the server side application that we needed. There were many editing, adding and deleting functions that we had to go back and add while we were trying to create the web interface.

Another problem we had was having all of our HTML and logic in one CherryPy document. The document became very long and hard to read and edit. We decided to use templating with Cheetah to try to reduce this clutter. This meant we had to redo much of our web interface to transfer the HTML to templates.

3.2.3 No Software is Bug-Free

We had several strange problems with our applications that seemed hard to diagnose and fix. We spent a lot of time trying to fix some bugs which ended up not being our fault. Because some of the technology is relatively new, it has not been tested enough to get it bug-free. After searching the internet for our problems, we found solutions for them.

We also had problems with Cheetah and CherryPy. When we used them together, the server would restart after every page request. This lead to long delays in response and problems with pages not being available or correctly loaded. At first, we thought we were doing something wrong with our code, but after a quick online search, we realized it was a bug in CherryPy. We found a patch that had been posted only a week earlier that fixed the problem.

Bibliography

[1] *Wikipedia, The Free Encyclopedia* Accessed on June 22, 2006. *http://www.wikipedia.org/*

[2] TinyOS *An open-source operating system designed for wireless embedded sensor networks.* http://www.tinyos.net/

[3] nesC *A programming language for deeply networked systems.* http://nescc.sourceforge.net/

[4] Python *An interpreted, interactive, object-oriented, extensible programming language.* http://www.python.org/

[5] Twisted *An event-driven networking framework.* http://www.twistedmatrix.com/

[6] Fettig, Abe *Twisted Network Programming Essentials.* O'Reilly Media, Inc.; 2005

[7] PyRSS2Gen *A Python module to generate RSS feeds.* http://www.dalkescientific.com/Python/PyRSS2Gen.html

[8] RSS: Really Simple Syndication *RSS 2.0 specification.* http://blogs.law.harvard.edu/tech/rss

[9] XML-RPC *Simple cross-platform distributed computing, based on the standards of the Internet.* http://www.xmlrpc.com/

[10] CherryPy *A pythonic, object-oriented web development framework.* http://www.cherrypy.org/

[11] SenSys 2006 *The 4th ACM conference on embedded networked sensor systems.* http://sensys.acm.org/2006/