XML Sensor Gateway for Wireless Sensor Networks Applications

<table>
<thead>
<tr>
<th>Class:</th>
<th>CSS600A (2\textsuperscript{nd} Phase)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>Steve Dame</td>
</tr>
<tr>
<td>Student ID:</td>
<td>314161</td>
</tr>
<tr>
<td>Date:</td>
<td>June 7, 2011</td>
</tr>
</tbody>
</table>
# Table Of Contents

1. Introduction .................................................................................................................. 3  
2. Background .................................................................................................................. 3  
3. Components of the Software System ........................................................................... 4  
4. Conclusions .................................................................................................................. 6  
REFERENCES .................................................................................................................. 8  
APPENDIX A – CODE ..................................................................................................... 9
1. Introduction

This is the second special project related to my eventual master’s project in wireless sensor networks. The over-arching objective of this wireless sensor networks system is to be able to aggregate data from tens of thousands to millions of sensors into a very large database. Thus each building block on the path to this large robust system needs to be developed. The scope of this portion of the project is to build an embedded device that will be capable of receiving data payloads over the air via a digital radio serial port, compressing and sending these packets through the Valhalla Wireless backhaul remote radio devices, receiving the raw output from the Valhalla Wireless base station via UDP packets, filtering these packets for XML (after decompression).

2. Background

In phase one of this special projects class, we explored XML compression and wrote some code to compress XML data into a much smaller payload than the verbose XML. Although it was a desired goal of this project phase to implement compression/decompression also, the infrastructure transmission and embedded firmware to create the sensor payload was more challenging and time consuming. So, we elected to defer the compression/decompression to the next phase of activities. What was more important was to set up the infrastructure and perform some end-to-end tests to demonstrate the robustness of the system.

Figure 2.1 shows the flow of sensor data through the various hardware components of the system. The system is a multi-tier wireless sensor network, with a long range back-haul radio system and Linux, Ruby on Rails server backend. The low power AgBee radios provide a diverse quantity of low cost and low power sensors which are aggregated together through an AgBee Concentrator embedded radio and system on a chip interface. This concentrator is connected to a 900MHz back-haul remote radio via a hardwired serial port running at 115Kbaud.

On the left, the sensing is performed in the AgBee radio/sensor devices, converted to digital packets and sent through the 2.4GHz network, bridged through the AgBee-Concentrator to connect through the 900Mhz backhaul network from Valhalla Wireless. The data packets are then routed through a server which the 900MHz base station is connected to via Ethernet. UDP packets are sent to this Linux server (hercules.uwb.edu) and then they are passed to the CSS Virtual Machine (cssvm01.uwb.edu). CSSVM01 handles injecting the sensor data into the MySQL data-
base and the Ruby on Rails webserver for accessing and manipulating sensor data that has been previously stored in the database.

3. Components of the Software System

Figure 3.1 shows the flow of sensor data through the various software components. The XML_Sensor device is actually a collection of several programs which are compiled and embedded in a Cypress PSoC3 host processor (programmable System on a chip). This processor will eventually interface to the 2.4GHz radio devices as the key gateway for wireless concentration. At this stage of the project, those radios are not connected to the PSoC host processor. So, XML
Data packets are just sent from the Gateway device at this time. These XML sensor packets contain real temperature data coming from a real thermistor device on the gateway device. There are two gateways, each having a unique sensor identification (sensor_id) so that they can be distinguished at the server end of the system.

The system is currently operating smoothly from end to end with real XML packets flowing every 10 seconds from each sensor gateway device. Data is captured at the Hercules server using the Java listener program written by Valhalla Wireless. This listener program then just writes each line of data it receives to a local file called xmldata.txt. The data from xmldata.txt is then easy to pick up and pass to the STDIN of the XML transmission program by performing a $ tail -f xmldata.txt.

The allows the writing program to continue to write to this file, while the reader picks up the last few lines of code that have been stored to xmldata.txt.

```xml
<?xml version="1.0" encoding="UTF-8"?><sensor_data><sensor_id capability="T" version="0.12">agcm-c600</sensor_id><time>1307238560</time><temperature unit="Fahrenheit" type="omega">76.4</temperature></sensor_data>
```

Figure 3.2 - XML Packet Originating at AgBee Sensor Device

```text
Processing for agcm-c600...

Sensor_id:agcm-c600 Version:0.12 Capability:T
Unix Time:1307238560
Temperature Type:omega Unit:Fahrenheit Value:76.4
```

Figure 3.3 - XML Packet Transmitted to CSSVM01 and XML Parsed (ready for database)
Figure 3.3 shows the final output from the transmitted data across Distributed Ruby (dRB) to the CSSVM01 server.

**4. Conclusions**

Initially a lot of work went into creating the embedded source code and operation on the PSoC chip, converting real resistance into temperature using the Steinhart-Hart equation and coefficients specified by the Omega corporation. After a few difficulties in getting all of these pieces to run, the PSoC devices were deployed to the lab where they currently pump out sensor data every 10 seconds. This data makes it successfully across the network to the UDP port on the VikingXE base station radio.

Since the data is variable and has extra status information, etc. that is NOT XML, then an XML filter was needed. This turned out to be a simple REGEX comparison. It was expeditious to break up the listener.java, the XML_dRB_Client.rb, the XML_dRB_Server.rb and send data to each of these entities. This now works robustly and we are all set to write actual data to the MySQL database using the Active Record. The data is simply piped from one process to another as follows:

```
$ nohup java listener >xmldata.txt &  # puts this processor in the background
# this reads the past few lines of code automatically and passes them to
# the dRB Client program for transmission to the parent.
$ tail -f xmldata.txt | XML_dRB_Client.rb
```

Of course the server needs to be started prior to setting up the client calls.
```
./XML_dRB_Server.rb
```

I had hoped to create a set of models in a Ruby on Rails database and populate it with real data from the network. This is a straightforward process of creating scaffolding to create the different modes and then using the “Active Record gem” to process the raw sensor data into
the database. Therefore, this will be the main subject of our phase 3 of this special projects activity and will likely be done within the summer timeframe so that a full website of sensor data is available for viewing.
REFERENCES


APPENDIX A – CODE

```ruby
#!/usr/bin/env ruby
#---------------------------------------------------------------
#  Program: XML_dRB_Client.rb
#  Description: Streams a Filtered XML Sensor Packet to the XML_dRB_Server
#               using a distributed queue protocol such as provided by "drb"
#  Inputs: $stdin
#  Author: Steve Dame (sdame@uw.edu)
#  Version: 0.1
#---------------------------------------------------------------
require 'drb'
require 'base64'
require 'rexml/document'

# Get a unique name for this sensor
URI = "cssvm01.uwec.edu"

DRb.start_service
drb_string = "druby://#{URI}:61676"
queue = DRBOBJ ect.new_with_uri(drb_string)

$stdin.each do |l|
  # filter for only XML records
  if l =~ /<\?xml/
    myXML = l.chomp.to_s
    # now, use XPath to extract only the sensor_id
    doc = REXML::Document.new(myXML)
    sensor_id_element = REXML::XPath.match(doc, '//sensor_id')
    # extract the sensor id value and use that as the name
    name = sensor_id_element.text
    myStrBase64 = Base64.encode64(myXML); queue.enq('request' => ['Report', 'Process'][1], 'from' => name, 'sensor' => myStrBase64)
  end
end
```

A.1 - XML Filter and dRB transmitter to the web server or data base backend.
require 'drb'
require 'base64'
require 'rexml/document'

SSAFE = 1          # Minimum acceptable paranoia level when sharing code!

URI = "csvn01"
puts "Server RMI URI #{URI}"
DRb.start_service

drb_arg = "druby://#{URI}:61676"

def run_queue(url)
  puts url
  queue = Queue.new  # Containing the jobs to be processed

  # Start up DRb with URI and object to share
  DRb.start_service(url, queue)
  puts "Listening for connection..."
  while job = queue.deq
    yield job
  end
end

#---
run_queue(drb_arg) do |job|
  case job['request']
  when 'Report'
    puts "Reporting for #{job['from']}... Done."
  when 'Process'
    puts "Processing for #{job['from']}..."
    myXMLBase64 = "#{job['sensor']}"
    puts myXMLBase64
    puts "---------------------------Base64---------------------------------------------"
    puts myXML = Base64.decode64(myXMLBase64)
    puts myXML
    puts "-------------------------------- Database Values -----------------------------"
    puts "--------------------------- Database Values -----------------------------"
    # parse the XML data to extract attributes and values
    doc = REXML::Document.new(myXML)
    root = doc.root

    # Parse Sensor ID information
    sv = root.elements["sensor_id"]["version"]
    puts "Sensor_id:" + sv
    ts = root.elements["sensor_id"]["capability"]
    puts "Capabilities:" + ts
    # Parse Time Information
    tm = root.elements["time"]["text"]
    puts "Unix Time:" + tm
    # Parse Temperature information
    t_unit = root.elements["temperature"]["unit"]
    t_type = root.elements["temperature"]["type"]
    t_value = root.elements["temperature"]["text"]
    puts "Temperature:" + t_type + " Unit:" + t_unit + " Value:" + t_value
  end
end

A.2  - XML dRB Server which collects dRB Client data
Several pages of AgBee firmware are available upon request. There are too many functions and chip level structures than to be able to document within a WORD document. The essence of the firmware is that it builds manual XML packets in either compressed or uncompressed code as shown in this snippet. These various strings are preformatted and “cat” with other strings and values to form the final XML packet.

```
// Uncompressed XML --> "xml_
const BYTE xml_header_length = 38;
CHAR_ARRAY xml_header[] = "<?xml version="1.0" encoding=""UTF-8""?>";
CHAR_ARRAY xml_sensor_root_begin[] = "<sensor_data>";
CHAR_ARRAY xml_sensor_root_end[] = "</sensor_data>";
CHAR_ARRAY xml_sensor_id_begin[] = "<sensor_id ";
CHAR_ARRAY xml_sensor_id_cap[] = "capability="";
CHAR_ARRAY xml_sensor_id_ver[] = "version="";
CHAR_ARRAY xml_sensor_id_end[] = "</sensor_id>";
CHAR_ARRAY xml_time_begin[] = "<time>";
CHAR_ARRAY xml_time_end[] = "</time>";
CHAR_ARRAY xml_temp_begin[] = "<temperature ";
CHAR_ARRAY xml_temp_unit[] = "unit=""celsius"">";
CHAR_ARRAY xml_temp_end[] = "</temperature>";
CHAR_ARRAY xml_camera_begin[] = "<camera ";
CHAR_ARRAY xml_camera_type[] = "type=""color ccd"" ";
CHAR_ARRAY xml_camera_xres[] = "xres=""640"" ";
CHAR_ARRAY xml_camera_yres[] = "yres=""480"" ";
CHAR_ARRAY xml_camera_compression[] = "compression=""JPEG"" ";
CHAR_ARRAY xml_camera_end[] = "</camera>";
CHAR_ARRAY xml_humid_begin[] = "<humidity>";
CHAR_ARRAY xml_humid_end[] = "</humidity>";
CHAR_ARRAY xml_light_begin[] = "<light ";
CHAR_ARRAY xml_light_type[] = "sensor=""CDS"">";
CHAR_ARRAY xml_light_end[] = "</light>";
CHAR_ARRAY xml_pressure_begin[] = "<pressure ";
CHAR_ARRAY xml_pressure_unit[] = "unit=""millibar"">";
CHAR_ARRAY xml_pressure_end[] = "</pressure>";
CHAR_ARRAY xml_battery_begin[] = "<battery>";
CHAR_ARRAY xml_battery_end[] = "</battery>";
CHAR_ARRAY xml_rssi_begin[] = "<rssi>";
CHAR_ARRAY xml_rssi_end[] = "</rssi>";
```

A.3 - XML builder strings for the PSoC processor