COMPUTER SCIENCE MASTERS PROGRAM

Special Projects

XML Compression for Wireless Sensor Networks Applications

Class: CSS600A
Name: Steve Dame
Student ID: 314161
Date: March 17, 2011
Table Of Contents

1. Introduction ........................................................................................................................................... 3
2. Background Research ............................................................................................................................. 3
   2.1 XML General Background .................................................................................................................. 3
   2.2 WBXML General Background ............................................................................................................ 4
   2.3 Base64 Encoding ................................................................................................................................. 4
3. XML Sensor Data Format Example .......................................................................................................... 5
4. Conclusions ............................................................................................................................................... 11
REFERENCES ................................................................................................................................................ 13
APPENDIX A – CODE ................................................................................................................................... 14
1. Introduction

Wireless sensor networks are becoming a popular, low cost means to sense the world around us in new and meaningful ways. There is an emerging term known as *The Internet of Things* to describe the explosive trend of embedded sensor technologies in real-world objects, the challenges and opportunities this represents for information technology systems [1]. Our thesis or project will involve designing a custom low power wireless sensor (radio, processor hardware and sensor software) to capture data about the environment and transmit data to the cloud to be correlated, processed, trended, etc. A design goal, for this project, is to develop a compact method in which to transmit these environmental data, yet to maintain a machine and human readable structure to the data so that rules about the sensor data are transmitted in structured packets. This provides long term scaling and compatibility to keep track of generations of sensors that have been deployed, the type, units and accuracy of their data, and fault sensing about the structured data being transmitted around the Internet. These methods are especially critical when large numbers of sensors are deployed, as it is easy to lose track of versions and capabilities of different devices over time.

2. Background Research

2.1 XML General Background

XML (eXtensible Markup Language) is a part of the more general Standard Generalized Markup Language which was originally designed for the purpose of long term machine readability and sharing of documents. Long-term, as originally envisioned by IBM engineers Charles Goldfarb, Edward Mosher, and Raymond Lorie (“GML”) in the 1960s meant readable for several decades. XML represents a set of rules for encoding document for excellent machine readability and is the most popular way to transmit data around the Internet in practice today. [2] Several XML related technologies (parsers, validators and rule checkers) are mature and deployed as libraries or plugins to nearly every computer language in practice today from Java to C# as well as emerging languages such as Ruby.

A key strength of XML also comes from the schemas, structures and tools available to validate XML document transmissions end-to-end across the Internet. One of the more popular validation files is known as the Document Type Definition (DTD). DTDs use a formal syntax or set of constraints and rules that can be interpreted by rule checkers and validators to enforce quality on...
data packets received over the Internet. DTDs can be classified as external subsets, internal subsets, both internal and external subsets, or no subset at all. A Document Type Declaration associates a DTD with an XML document (or stream in our case). This declaration occurs just after the header in an XML document (or stream):

```xml
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0//EN" 
"http://www.w3.org/TR/xhtml1/DTD/xhtml1.dtd">
<root> . . .
</root>
```

The "-//W3C//DTD XHTML 1.0 //EN" refers to how all XHTML documents (XML documents) must conform to one of the SGML DTDs.

The URI "http://www.w3.org/TR/xhtml1/DTD/xhtml1.dtd" provides a link to a specific public DTD that is used to check the XML document contents that follow these two header lines (i.e. the XML data between <root> and </root>), where “root” can be any valid root tag defined in the DTD. An example of sensor data that might be structured in this way would

**2.2 WBXML General Background**

The focus of this study is to develop a wireless binary XML compression methodology and to explore different tradeoffs between generalized XML schema processing and high levels compression. Several different references were explored starting with the W3C definition [13]. However, it became increasingly difficult to align the WBXML standard as applied to our design objectives to a compressed XML schema for our sensor data. Also, the various open source code utilities from C, C++, Java, Python, Ruby, etc. all had shortcomings and very little documentation. Most of these utilities also either didn’t compile or had serious bugs. After several weeks of exploring WBXML, we decided to abandon it in favor of a custom protocol somewhat patterned from some of the concepts studied in the WBXML specifications.

**2.3 Base64 Encoding**

One problem with the transmission of purely binary streams through the Internet is that many serialization and object transmission protocols work at the lowest common denominator ASCII character sets or more commonly UTF-8 [6] presently. To ensure that a binary packet has the best possibility of passing between different hosts, languages and networks a binary-to-text en-
coding scheme know as Base64 [7] utilizes a radix-64 method of characters to map binary data to these characters prior to transmission.

3. XML Sensor Data Format Example

In an example of our sensor data, we have the following XML structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensor_data>
  <sensor_id capability="THLPBR" version="0.12">X88</sensor_id>
  <time>1300261583</time>
  <temperature unit="celsius">4.7</temperature>
  <humidity>82.8</humidity>
  <light sensor="cds">89.6</light>
  <pressure unit="millibar">1012</pressure>
  <batt>4.73</batt>
  <rssi>-61</rssi>
</sensor_data>
```

Figure 3.1 - XML Formatted Sensor Data

The structure of this XML data packet is detailed below in table 3.1. In this quasi-WBXML optimized schema, there is a Binary XML Content structure that follows along with some of the data types and token structure of the WBXML. Specifically we adopt the data types such as:

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_int8</td>
<td>Unsigned byte value</td>
</tr>
<tr>
<td>Byte</td>
<td>Signed byte</td>
</tr>
<tr>
<td>u_int16</td>
<td>Unsigned integer value (always preceded by a lower case ‘u’)</td>
</tr>
<tr>
<td>u_int32</td>
<td>Unsigned (always preceded by a Upper case ‘U’)</td>
</tr>
<tr>
<td>int16</td>
<td>Signed 16 bit integer value (always preceded by a lower case 'x')</td>
</tr>
<tr>
<td>int32</td>
<td>Signed 32 bit integer value (always preceded by a Upper case \’X\’)</td>
</tr>
</tbody>
</table>

Table 3.1 - Modified WBXML Data Types

Next we examine the inherent schema of this sensor data in order to derive a set of encodings for the main tags and attributes.
A specific character table is built from the structure of the element names and their attributes as defined in the sensor schema of figure 3.2. The characters are simply allocated in order from the English alphabet. A constraint on our simple sensor packets is that there may not be any more than 26 elements. This corresponds to the number of capitalized characters in the English alphabet. Although this seems like an unreasonable long term constraint, we are striving for ultimate compression factors and therefore this trade-off can be tolerated for most of the typical environmental sensors with numerous ways to extend this approach should the need arise. Each element shall have a 0x80 (MSB set) shall be added to the character to signify that it is an element of the XML structure. Each element shall bracket additional attributes and data within the element structure (with the MSB set as well).

<table>
<thead>
<tr>
<th>Encode Char</th>
<th>Definition (Tag)</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>“sensor_data”</td>
<td></td>
</tr>
</tbody>
</table>
| B           | “sensor_id”      | c – “capability”  
v – “version” |
| C           | “time”           |            |
| D           | “temperature”    | u – “unit” = [F | C]  
F – Fahrenheit  
C – Celsius |
| E           | “humidity”       |            |
| F           | “light”          | s – “sensor” (type) [C]  
C - cds |
| G           | “pressure”       | u – “unit” [m|b|p|i|k|h]  
m – millibar  
b – bar  
p – psi  
i - In of Hg  
k – kPa  
h - hPa |
| H           | “batt”           | u – “unit” = [V | C]  
V – Volts  
C – Current |
| I           | “rssi”           | u – “unit” = [m | d]  
m – mW  
d - dBm |

Table 3.2 - Modified WBXML Character Encoding for the Sensor Schema
Table 3.3 below shows the compression method used to encode the entire 320 byte packet into a 61 byte modified WBXML encoded binary stream. This stream is then encoded using base64.

<table>
<thead>
<tr>
<th>XML ELEMENT STRUCTURE</th>
<th>RAW BYTE COUNT</th>
<th>BINARY OPTIMIZED ENCODING</th>
<th>COMPRRESSED BYTE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;?xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
<td>38</td>
<td>81</td>
<td>1</td>
</tr>
<tr>
<td>&lt;sensor_data&gt;</td>
<td>13+38=51</td>
<td>[&quot;A&quot; + 0x80]</td>
<td>1+1=2</td>
</tr>
<tr>
<td>&lt;sensor_id capability=&quot;THLPBR&quot; version=&quot;0.12&quot;&gt;X88&lt;/sensor_id&gt;</td>
<td>60+51=111</td>
<td>[0x80+*B'] + 'c' + &quot;THLPBR&quot; + 'v' + 0xC + &quot;X88&quot;</td>
<td>13+2=15</td>
</tr>
<tr>
<td>&lt;time&gt;1300261583&lt;/time&gt;</td>
<td>23+111=134</td>
<td>&quot;C&quot; + 0x80 + &quot;1300261583&quot;</td>
<td>11+15=26</td>
</tr>
<tr>
<td>&lt;temperature unit=&quot;celsius&quot;&gt;4.7&lt;/temperature&gt;</td>
<td>44+134 =178</td>
<td>[&quot;D&quot;+0x80] + &quot;uc&quot; + &quot;4.7&quot;</td>
<td>6+26=32</td>
</tr>
<tr>
<td>&lt;humidity&gt;82.8&lt;/humidity&gt;</td>
<td>178+25=203</td>
<td>[&quot;E&quot;+0x80] + &quot;82.8&quot;</td>
<td>5+32=37</td>
</tr>
<tr>
<td>&lt;light sensor=&quot;cds&quot;&gt;89.6&lt;/light&gt;</td>
<td>31+203=234</td>
<td>[&quot;F&quot;+0x80] + &quot;sc&quot; + &quot;89.6&quot;</td>
<td>7+37=44</td>
</tr>
<tr>
<td>&lt;pressure unit=&quot;millibar&quot;&gt;1012&lt;/pressure&gt;</td>
<td>40+234=274</td>
<td>[&quot;G&quot;+0x80] + &quot;um&quot; + &quot;1012&quot;</td>
<td>7+44=51</td>
</tr>
<tr>
<td>&lt;batt&gt;4.73&lt;/batt&gt;</td>
<td>17+273=290</td>
<td>[&quot;H&quot;+0x80] + &quot;4.73&quot;</td>
<td>5+51=56</td>
</tr>
<tr>
<td>&lt;rssi&gt;-61&lt;/rssi&gt;</td>
<td>16+290=306</td>
<td>[&quot;I&quot;+0x80] + &quot;-61&quot;</td>
<td>4+56=60</td>
</tr>
<tr>
<td>&lt;/sensor_data&gt;</td>
<td>14+306=320</td>
<td>[&quot;A&quot; + 0x80]</td>
<td>1+59=61</td>
</tr>
<tr>
<td><strong>TOTALS</strong></td>
<td><strong>320</strong></td>
<td><strong>61</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 - Raw Sensor XML vs. Compressed binary

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
81 C1 C2 63 54 48 4c 50 42 52 76 0C 58 38 38 C3
31 33 30 30 32 36 31 35 38 33 44 75 63 34 2e 37
45 38 32 2e 38 46 73 63 38 39 2e 36 47 75 6d 31
30 31 32 48 34 2e 37 33 49 2d 36 31 41
```

Table 3.4 - Compressed binary representation of the XML Data

**QUJjVEhMUEJSdiBYODhDMTMwMDI2MTU4M0R1YzQuN0U4Mi44RnNjODkuNkd1**

**bTEwMTJjINC43M0ktNjFB**

Table 3.5 - Base64 ASCII representation of the Compressed XML Data

We have created Ruby programs below which can be used to simulate various sensor data packets and sent across a network between a client (or multiple client sensors) and a sensor server. This client server code was tested on both a MacOSX platform as well as the UW-1-320 ma-
machines running on different machines to show how the data is transported through the Remote Method Invocation (RMI) facility provided by the rubygem ‘drb’.
```ruby
#!/usr/bin/env ruby
require File.join(File.dirname(__FILE__), 'lib/SensorRender')

# Program: sensor_rmi_client.rb
# Description: Builds a test XML sensor packet and sends it to the server
# using a distributed queue protocol such as provided by "db" S
#  Inputs: SENSOR_NAME - string id
#   URI - Server IP address (e.g. localhost | uw1-328-20 | 10.6.6.1)
#  Author: Steve Dame (sdame@uw.edu)
#  Version: 0.2

require 'db'
require 'base64'

# Get a unique name for this sensor
NAME = ARGV.shift or raise "Usage: #{File.basename($0)} SENSOR_NAME URI"
URI = ARGV.shift or raise "Usage: #{File.basename($0)} SENSOR_NAME URI"
puts "SENSOR NAME: #{NAME}" + " URI: #{URI}"

DRb.start_service
    db_string = "druby://(URI):61676"
    queue = DRbObject.new_with_uri("druby://(URI):61676")
    queue = DRbObject.new_with_uri(db_string)

    1000.times do |x|
        puts "#{x}"
        mysensor = SensorXML.new
        root = mysensor.addRootElement("sensor_data")
        mysensor.addSubElement_Value(root, "sensor_id", '#{NAME}")
        mysensor.addAttribute("capability", "THLPER")
        mysensor.addAttribute("version", "0.12")
        mysensor.addSubElement_Value(root, "time", Time.now.to_i.to_s)
        mysensor.addSubElement_Value(root, "temperature", "%.1f" % (50*rand)).to_s
        mysensor.addAttribute("unit", "celsius")
        mysensor.addSubElement_Value(root, "humidity", "%.1f" % (50 + 50*rand)).to_s
        mysensor.addSubElement_Value(root, "light", "%.1f" % (100*rand)).to_s
        mysensor.addAttribute("sensor", "cds")
        mysensor.addSubElement_Value(root, "pressure", "%.2f" % (1010 + 10*rand)).to_s
        mysensor.addAttribute("unit", "millibar")
        mysensor.addSubElement_Value(root, "batt", "%.2f" % (3 + 2*rand)).to_s
        mysensor.addSubElement_Value(root, "rest", "%.0f" % (-30-70*rand)).to_s
#
        myst = mysensor.renderXML
        mystBase64 = Base64.encode64(myst)
        queue.enq('request' => ['Report', 'Process'])[1], 'from' => NAME, 'sensor' => mystBase64)
        sleep 1 # simulating network delays
    end
```

Figure 3.2 - Client for sending XML Sensor Data (sensor_rmi_client.rb)
```ruby
#!/usr/bin/env ruby
require File.join(File.dirname(__FILE__), 'lib/SensorRender')
# This is a server to receive a distributed queue of sensor data
# Inputs: URI - IP address of this server plus a port number
#----------------------------------------------------------------------
require 'drb'
$SAFE = 1  # Minimum acceptable paranoia level when sharing code!

URI = "localhost"
# URI = "uu1-320-20"
#ARGV.shift or raise "Usage: #{File.basename($0)} URI"
puts "Server RMI URI=#{URI}"
DRb.start_service
#drb_arg = "druby://localhost:61676"
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
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']}..."
  end
  mysensor = SensorXML.new
  mysensor = job['sensor']
  puts "-----------------------------
  puts "#{job['sensor']}"
  puts "-----------------------------
  sleep 2  # Simulate real work
  puts 'Processing complete.'
end
end
```

Figure 3.3 - Server for receiving XML Sensor Data (sensor_rmi_server.rb)
4. Conclusions

Although initial research led us to believe that WBXML would be a fruitful method to provide universal XML encoding and decoding, many hours were spent trying to find working utilities on the web as a starting point to developing our own compression scheme. Finally, all indicators were pointing to a possible deprecated technology and one that we shouldn’t spend any more time investing in trying to either make existing tools work, or developing our own. Also, within a few days of the end of the quarter, I became aware of a new XML compression technology [5] that was just accepted by the W3C consortium called Efficient XML Interchange (EXI). Future research will include a more thorough examination of EXI and search for utilities in the market. There appears to be a solid enterprise support for EXI, albeit with proprietary tools at this point from “Agile Delta”. In the interim a limited but highly efficient XML compression scheme has been created that is tailored to the type and structure of the relatively small XML packets coming from our wireless sensor devices. We created this compression technique in C because it is likely to need to also run on embedded C coded sensor devices, so it would nice to make it available at both the C level and the Ruby level. Also, the speed of execution of the compression would be nice to optimize utilizing a compiled language such as C. Finally, the manipulation of raw strings and binary numbers is a bit easier to code in C as a stream processor.

Also our original goal was to be able to populate a database with output from the XML that was transported between different machines. Although we were able to create a system using Rails which allowed us to create database entries from a web browser, we were not able to spend any time connecting that rails environment to the lower level ruby transport client/server code.
This will be subject also of some future work and investigation as well as integrating an actual embedded platform to receive data from low power wireless nodes and transmit that data to a database to be populated after parsing the XML data structure that has been base64 decoded, uncompressed back into its original XML form.
REFERENCES


http://doi.acm.org.offcampus.lib.washington.edu/10.1145/800209.806456


require File.join(File.dirname(__FILE__), 'lib/SensorRender' )
#==============================================================================
#     Program: sensor.rmi_client.rb
# Description: Builds a test XML sensor packet and sends it to the server
# using a distributed queue protocol such as provided by "drb"
#      Inputs: SENSOR_NAME - string id
#           URI - server IP address (e.g. localhost | uw1-320-20 | 10.0.0.1)
#      Author: Steve Dame (sdame@uw.edu)
#     Version: 0.2
#==============================================================================
require 'drb'
require 'base64'

# Get a unique name for this sensor
NAME = ARGV.shift or raise "Usage:  #{File.basename($0)} SENSOR_NAME URI"
URI = ARGV.shift or raise "Usage:  #{File.basename($0)} SENSOR_NAME URI"
puts "SENSOR NAME: #{NAME}" + "URL: #{URI}"

DRb.start_service

# Testing queue = DRbObject.new_with_uri("druby:#{URI}:61676")
queue = DRbObject.new_with_uri("druby://#{URI}:61676")

100.times do |x|
Puts "I:#{x}"

mysensor = SensorXML.new
root = mysensor.addRootElement("sensor_data")

mysensor.addSubElement_Value( root, "sensor_id", "#{NAME}" )
mysensor.addAttribute( "capability", "THLPBR" )
mysensor.addAttribute( "version", "0.12" )

mysensor.addSubElement_Value( root, "time", Time.now.to_i.to_s )
mysensor.addSubElement_Value(root, "temperature", ("%.1f" % (50*rand)).to_s )
mysensor.addAttribute("unit", "celsius" )

mysensor.addSubElement_Value(root, "humidity", ("%.1f" % (50 + 50*rand)).to_s )

mysensor.addSubElement_Value(root, "light", ("%.1f" % (100+rand)).to_s )
mysensor.addAttribute("sensor", "cds" )

mysensor.addSubElement_Value(root, "pressure", ("%.0f" % (1010 + 10*rand)).to_s )
mysensor.addAttribute("unit", "millibar" )

mysensor.addSubElement_Value(root, "batt", ("%.2f" % (3 + 2*rand)).to_s )

mysensor.addSubElement_Value(root, "rssi", ("%.0f" % (-30-70+rand)).to_s )
#
mysensor.writeSensorXML_ToFile($stdout)

myStr = mysensor.getSensorXML.to_s
myStrBase64 = Base64.encode64(myStr);
queue.enq('request' => ['Report', 'Process'][1], 'from' => NAME, 'sensor' => myStrBase64)
sleep 1 # simulating network delays
end
#!/usr/bin/env ruby
require File.join(File.dirname(__FILE__), 'lib/SensorRender')
#==============================================================================
#     Program: sensor_rmi_server.rb
# Description: provides a server to receive a distributed queue of sensor data
#              This program receives uncompressed XML data that has been
#              converted to base64 and then decodes the base64 encoding to
#              reveal the original XML sensor data packets
#      Inputs: URI - IP address of this server plus a port number
#      Author: Steve Dame (sdame@uw.edu)
# Version: 0.2
#==============================================================================
require 'drb'
$SAFE = 1  # Minimum acceptable paranoia level when sharing code:
URI = "localhost"
# URI = "uw1-320-20"
#URI = ARGV.shift or raise 'Usage:  #{File.basename($0)} URI'
puts "Server RMI URI #{URI}"
DRb.start_service
#drb_arg = "druby://localhost:61676"
drb_arg = "druby://#{URI}:61676"
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']}"
    myXML = Base64.decode64(myXMLBase64);
    puts "=====================================================================
    sleep 2    # Simulate real work
end  
end
end
#---
#!/usr/bin/ruby -w

require "rexml/document"
include REXML
#
# Class: SensorXML
#
# Description:
# # Inputs:
# # Returns:
#
class SensorXML

  def initialize
    xml_prolog = %{<xml version="1.0">}
    doc_type = %{<!DOCTYPE wml PUBLIC "-//AGCOMM//DTD WML 1.1//EN" "http://www.agcomm.net/DTD/agbee_1.xml"} # Commented out
    root_element = %{'sensor_data'}
    # @xml_doc = REXML::Document.new xml_prolog + doc_type
    # @curtime = Time.now.to_s
    # @cur_element = @xml_doc
    # @root_element = nil
  end

  def writeSensorXML_ToFile(filename)
    # @xml_doc.write ($stdout, 0)
    puts @xml_doc
  end

  def getSensorXML
    return @xml_doc
  end

  def addRootElement(element)
    @root_element = @cur_element.add_element(element)
    return @root_element
  end

  def addElement(element)
    @cur_element = @cur_element.add_element(element)
    return @cur_element
  end

  def addSubElement(parent, element)
    @cur_element = parent.add_element(element)
    return @cur_element
  end

  def addElement_Value(element, value)
    @cur_element = @cur_element.add_element(element)
    value = @cur_element.add_text(value)
    return @cur_element
  end

end

#==============================================
# Method: writeSensorXML_ToFile
# Description: Write XML tree formatted sensor data to a filename
# Inputs: filename - file to store data
# Returns:
#
def writeSensorXML_ToFile(filename)
  @xml_doc.write ($stdout, 0)
  puts @xml_doc
end

#==============================================
# Method: getSensorXML
# Description:
# Inputs:
# Returns:
#
def getSensorXML
  return @xml_doc
end

#==============================================
# Method: addRootElement
# Description:
# Inputs:
# Returns:
#
def addRootElement(element)
  @root_element = @cur_element.add_element(element)
  return @root_element
end

#==============================================
# Method: addElement
# Description:
# Inputs:
# Returns:
#
def addElement(element)
  @cur_element = @cur_element.add_element(element)
  return @cur_element
end

#==============================================
# Method: addSubElement
# Description:
# Inputs:
# Returns:
#
def addSubElement(parent, element)
  @cur_element = parent.add_element(element)
  return @cur_element
end

#==============================================
# Method: addElement_Value
# Description:
# Inputs:
# Returns:
#
def addElement_Value(element, value)
  @cur_element = @cur_element.add_element(element)
  value = @cur_element.add_text(value)
  return @cur_element
end

Stephen G. Dame sdame@u.washington.edu
def addSubElement_Value(parent, element, value)
    cur_element = parent.add_element(element)
    value = cur_element.add_text(value)
    return cur_element
end

def addValue(value)
    value = cur_element.add_text(value)
    return cur_element
end

def addAttribute(key, value)
    value = cur_element.add_attribute(key, value)
    return cur_element
end

def readSensorXML_FromFile(filename)
end

def compressSensorXML(compression_schema)
end

def xxx
end

Stephen G. Dame sdame@u.washington.edu