Real-time Device Monitoring Using AWS

1 Document History

Version	Date	Initials	Change Description
1.0	3/13/08	JZW	Initial entry
1.1	3/14/08	JZW	Continue initial input
1.2	3/14/08	JZW	Added headers and footers
1.3	3/14/08	JZW	Run spell checker
1.4	3/31/08	JZW	Additional redacting

2 Table of Contents

3 Introduction

This document describes a method for monitoring, in an on-going basis (in real-time), device variables, within a NET+OS-based AWS application. This document describes requirements in NET+OS, html and java that would be required to implement such an application.

3.1 Problem Solved

We have had a small number of customers that have asked whether our AWS implementation supports AJAX (asynchronous java and XML). This is using a set of existing technologies in a new way, that is being able to retrieve data "behind the scenes" of the web browser/server. I believe that "real" AJAX gives an application the ability, on a web page, to have certain fields update regularly while others update on a refresh. The technique described here does not solve that problem. Instead, what is described here is the ability to have a separate web page(s) that is continually monitoring and updating the contents of a page using a java applet.

Thus what this paper describes is a technique for creating a web page, attached to a java applet that is continually retrieving device data and updating this aforementioned page with this updated device data.

3.2 Audience

This paper is intended for software engineers with (at least a working) knowledge of the following technologies:

- java
- applets
- html
- NET+OS development
- NET+OS-based AWS application development
- C coding
- TCP/IP sockets development (NET+OS and java)

These technologies are all used in the development of applications demonstrating this real-time monitoring.

3.3 Assumptions

The technologies described in this document are applicable to NET+OS V6.x and V7.x. The V7.x applicability includes development under Digi ESP. For Digi ESP development, certain files may need to be moved around, but the basic technologies are common between command line interface (CLI) development and ESP (GUI) development.

In order to develop java applets you will need to download a java software development kit (SDK) from the java web site (java.sun.com). Applets can be developed either in an integrated development environment (IDE) such as netbeans or they can be developed freehand, in a text editor. That decision is up to the individual developer.

The techniques described here should be applicable to development in either the gnu or the Green Hills development environments.

3.4 Scope

This document presents an overview of what is required to develop a web page containing a java applet that monitors device data in real time. This paper is not an in depth description or tutorial of any of the following technologies:

- java application development
- java applet development
- java applet debugging
- html programming
- C programming
- sockets programming
- NET+OS development
- NET+OS-based AWS development
- AWS comment tags

The techniques described here are applicable to NET+OS (gnu and/or Green Hills) development only. These techniques do not describe methods applicable to the Digi Linux or the Digi .net development environments.

3.5 Theory of Operation

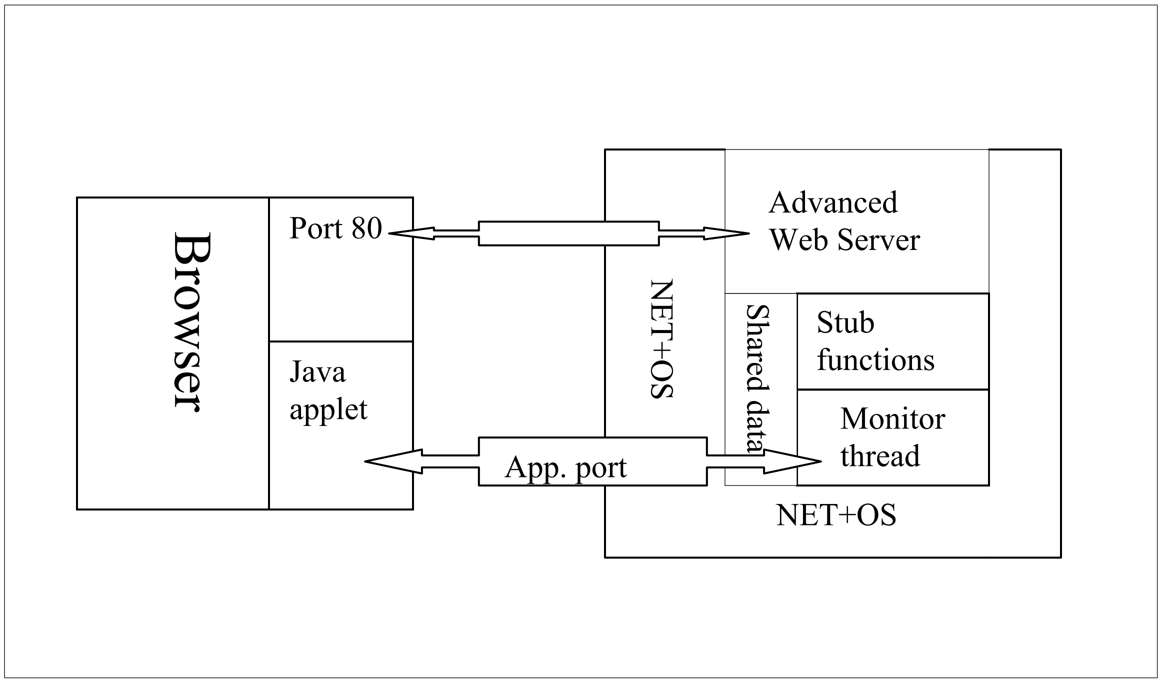
The application supplied “implements” a web based application for monitoring a furnace. I have given the furnace the following device data that need tracking:

- Temperature
- Pressure
- Voltage
- Current
- Fluid level

The software which this document describes is made up of 4 parts as follows:

- web (html) pages
- AWS stub functions
- A monitoring thread
- A java applet

A diagram of the system might look like the following:



On the left side of the diagram is a browser. This could be MS Internet Explorer, Mozilla firefox or Netscape Navigator. For that matter it could be any browser that supports java applets. The three that I mentioned are the “big three”. The block marked port 80 represents standard web pages that would be using the browser for communicating with the advanced web server. I call it port 80 because the standard port for communicating with web servers is port 80.

The block marked java applet is just that, a java applet served from the advanced web server to the browser. The applet was served to the browser over port 80. The applet itself, communicates with the NET+OS application using a port of your choice. That is why I have labeled it app (application) port. In the sample application supplied with this white paper, the applet and the monitor thread communicate over port 2001.

The u-shaped object represents the NET+OS operating system. The advanced web server, the stub functions and the monitor thread all use (or can use) NET+OS system calls and functions.

The java applet is shown communicating with a block called a monitor thread. This is a separate thread which is tasked with sending updated device data to the applet every time the applet asks for an update. In the application supplied, we have chosen to utilize UDP sockets (datagrams) for the communication between the monitor thread and the applet. This was done for simplicity. UDP is not a requirement. Every time the applet gets an update, it calls redraw to update the screen.

The block marked stub functions represents the AWS stub functions that you would write to supply device data to web pages. These would be no different from any web application you might have developed in the past.

The block marked shared data is important. In the supplied application, there is a web page that allows you to take an instantaneous “snapshot” of the device data. The applet allows you to continuously monitor device data. The data, in this case is a function that generates random numbers. Both the stub functions and the monitor thread call into it. In a “real” application, this random number provider function might be replaced by a function that calls into the device and extracts real world data (temperature, pressure, etc...). The point is that the data is available to both the monitor thread (and the applet) and the advanced web server, its stub functions and a web page.

The block marked the Advanced Web server, represents just that. What I’d like you to take away from this part of the diagram is that the web server is supplying updates to the “standard” web page(s) while the monitoring thread is supplying updates to the java applet.

If you surf to a device, running this application, you will see a “main” page, showing instantaneous (snap shot) data. This page has a link to a second page housing the applet. The applet sends a UDP datagram to the device, requesting an update. The monitoring thread, in the device, receives this datagram and puts together a simple structure containing the data (numbers separated by #s). The monitoring thread then sends this data back to the applet. The applet reads the data, using a string tokenizer, breaks the data into their constituent parts, converts them to integers and updates the page. There is a ¼ second delay and then the applet requests another update. If you are running the applet in a browser, you should see data being continually updated.

3.6 Conventions

There are no special conventions used in the text of this document.

4 Details

This section dives into the details of the attached application that demonstrates a real-time monitoring system. You might want to edit the files referred to in the fallowing text, allowing you to follow along with the text.

4.1 HTML code

4.1.1 Main page

The main web page is entitled furnaceMain.htm. It contains RpDisplayText comment tags for accessing device data in a conventional AWS way. The RpDisplayText comment tags refer to stub functions in file furnaceMain_v.c. For the purposes of this example application, these stub functions call function getRandNumber() which returns a random number. In a “real” application, these functions would call internal functions that might access device data associated with some physical device.

In addition, this page contains a link to a second web page which has the applet.

4.1.2 Applet page

The applet page initially displays an input window which requests the IP address of the device to be monitored. Given the security restrictions of applets, the only IP address that will work, is the IP address of the device that served the web pages to your browser. Once entered, the applet will begin exchanging UDP packets with the device and continually update the display.

4.2 Pbuilder utility usage

For this application, the applet code is included in the application. The .jar file can be placed in the file system, if you so desire. Please see the white paper explaining java applets in AWS applications in general for detailed information on including the .jar file in the NET+OS file system.

The two web pages and the jar file must be run through the PBuilder application, to be included in the application. You will find the .jar code included in the furnaceMain.c file, in the \pbuilder\html directory.

4.3 list.bat aka pbuiler.pbb contents

NET+OS version V6.X and V7.x using the GNU command line development process use file list.bat. NET+OS V7.X use pbuilder.pbb. These files are the input file to the PBuilder utility. The purpose of these files is to allow you to run the PBuilder utility once but process more than one file. It is important to remember that the first file listed in the list.bat/pbuilder.pbb file is considered the “main” page of the application by AWS (the page served when a “get /” is sent by the browser. The order of the other objects is not important, though we generally place all web pages before objects such as .jar (java archive) and .gif (image) files. This is convention more then a requirement.

4.4 monitorThread() and getRandNumber() functions

On the NET+OS side of the application, the heavy lifting is accomplished by two functions, namely monitorThread() and getRandNumber(). These are located in the root.c file. getRandNumber() is extern(ed) in file furnaceMain_v.c giving the stub functions access to the random numbers

MonitorThread() is a thread started in applicationStart(). It waits to receive a packet from the applet. When received, it gets updated device data (by calling getRandNumber), places this data into a buffer and sends the data back to the host who sent the update request. This is done in an endless loop.

getRandNumber() combines output from the NET+OS random number generator with time data to create a number. getRandNumber() takes the modulo of this number. The result of the modulo operation is returned as the current random number.

4.5 Stub functions

In file furnaceMain_v.c are located the stub functions. There is one stub function for each field on the main (instantaneous) web page. Each stub function calls getRandNumber()

(externed back to the root.c file) to access the updated device data. The data is returned, through the advanced web server to the browser over (by default) port 80.

4.6 Java applet

To quote from the book “Learning Java” by Niemeyer & Knudsen, “An applet is a part of a web page, just like an image or a hyperlink. It “owns” some rectangular area of the user’s screen. It can draw whatever it wants and respond to key board and mouse events in that area. When the web browser loads a page that contains a java applet, it knows how to load the classes and the applet and run them”.

4.6.1 Operation

After being loaded, the applet associated with this application, asks the user for the IP address of the device to be monitored. You must enter the IP address of the device that served your web page. The applet creates a thread that exchanges UDP datagrams with this device. The applet sends an update request message in a datagram, to the device. The device packages updated information into a datagram and returns a datagram to the applet that made the request. When the applet receives the update, it updates the fields of the main object (a furnace). It then calls for the screen to be repainted (update the page being viewed by the user). The applet then waits 250 milliseconds and repeats the process. this is an endless loop.

4.6.2 Caveats

There are some severe restrictions applied to the operations of an applet. If you are not familiar with them, we‘d advise either surfing the web or getting a book on the subject. We talk about a few of these restrictions below.

4.6.2.1 Security restrictions

4.6.2.1.1 Files uploaded

When testing the applet before adding the sockets code (having the update code locally develop random numbers and repaint using them, we found that the application could not read the additional class files off the local file system. We found that it was easier to archive the while application into a .jar file and then let the browser sort out the classes later. Thus this application is based on using the jar utility to archive the set of class files and having the web page refer to both the archive and the primary class.

4.6.2.1.2 Outgoing sockets

Applets are only allowed to open sockets back to the web server that served the web page and applet data to the browser. We believe this restriction is in place to keep applets with nefarious intentions from being uploaded and then going out and causing all sorts of trouble.

5 Conclusion

This paper has demonstrated and explained a method for real-time monitoring of device data, using java applets in an AWS application. This paper has shown this capability to be a straight forward process that should be accessible to any developers that need this capability. The reader should keep in mind that there are strict security restrictions (highlighted above) and that the developer must be familiar with them, before beginning the task of developing this type of application.

6 Glossary of Terms

.jar file – A .jar file is a java archive file created using the jar utility. It is analogous to the tar utility, which has been available for many years with the UNIX operating system. The resultant archive file can hold a number of java class files (java compiled files). The browser is capable of extracting the required class files form the archive when an applet is being executed.

AWS – Advanced Web Server. An embedded web server that is shipped as part of a NET+OS development kit. It is available for both GNU and Green Hills development kits.

AWS comment tags – Comment tags that are added to an html page. The comment tags are processed by the PBuilder utility, producing the requisite C code, allowing web pages to be built into a NET+OS-based AWS application.

Browser – A program, using on a PC or UNIX system for accessing content from a web server. Examples are MS Internet Explorer, Mozilla FireFox and Netscape Navigator.

Command Line Interface (CLI) – a method of accessing commands on a PC or a UNIX system, that involves textual input to a terminal as opposed to using a mouse and using point and click method. This might be referred to DOS shell mode or UNIX shell mode.

Datagram – The basic unit of data movement, across the internet, when the UDP/IP protocol is employed.

GUI – Graphical User Interface – a method of running programs, on a PC or a UNIX system, that employs a mouse and the point and clock method.

HTML code – Hypertext Markup Language, the language in which web pages are traditionally written.

IDE – Integrated Development Environment – A GUI package facilitating the development of software applications. Netbeans and Digi ESP are examples of GUIs.

Java – A programming language, developed by sun Microsystems, that provides an object oriented environment for software development.

Java applet – A software component, written in the Java language, which allows java software content to be included in a web-based application.

NET+OS – An embedded real-time operating system and development environment, developed and distributed by Digi International

PBuilder utility – The utility, included in the AWS component of Digi's NET+OS, that converts HTML pages into C code for inclusion in a NET+OS-based AWS application.

Stub functions – Callback subroutines, included in a NET+OS-based AWS application. The AWS calls into the applications stub functions, giving AWS access to the application's device data (either for reading or writing).

Tokenizer – A function capable of breaking a string into tokens (pieces) using a particular character as the separation point between tokens (pieces). For example if we had the string "See/Jane/Run". The three tokens are the words "See", "Jane" and "run". The separation points are the "/" characters. The tokenizer is capable of returning the three tokens to a program, given the initial string.

Example Code