Active AJAX based live dashboards

"Pushing Data to a browser"

A white paper looking into the technologies that allow the pushing of data to a web browser. This data is then used to shows a live dashboard of an automotive supply chain

Author: Jeffrey Porter (j.porter.1@warwick.ac.uk / jeff@porter.eu.com)
Date: 21 February 2007
Version: v 1.0b

Synopsis

During a program to track stillages in a supply chain via RFID tags, it was decided to create a live dashboard view of where they were located. At the start of the project there were several technologies that allowed the pushing of data to a webpage, but each had its drawbacks and none were in a state which the organisation considered stable enough for production.
This paper will cover the basics of each of these technologies, and then give an example of the final choice of AJAX (DOJO) and CometD.

Presumptions

This document presumes that the read has in-depth knowledge of Java (inc J2EE), JavaScript.
Some knowledge of Dojo/Ajax and an understanding of how these technologies work together.

TABLE OF CONTENTS

Acknowledgments

The author wishes to express sincere appreciation to Just van den Broecke for his prompt and detailed help in using his Pushlet's API., Chris Bucchere for his help with deploying the first release CometD and Alex Russell for his advice on AJAX and CometD.

Glossary

Ajax: Shorthand for Asynchronous JavaScript and XML, is a web development technique for creating interactive web applications. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server, thus preventing the entire web page being reloaded each time the user requests a change. This increases the web page's interactivity, speed, and usability.

WEB 2.0: A phrase coined in 2004, refers to a proposed second generation of Internet-based services. A combination of AJAX, CSS, XHTML, Web services.

Stillage: A large container, usually metal, that holds typically large components.

CometD: This is an API, AJAX based, that allows 'Low Latency Data for the Browser'. The letter D is for 'Daemon'.

XMLHttpRequest: is an API that can be used by JavaScript, JScript, VBScript and other web browser scripting languages to transfer and manipulate XML data to and from a web server using HTTP. XMLHttpRequest establishes an independent connection channel between a web page's Client-Side and Server-Side.

Stillage: A large metal container. Can contain good whilst in shipping.

Resources

The beta version of this document and the demo code will be available from the following locations

http://www.qenet.co.uk/warwick/whitepaper-pushTech.pdf

http://www.qenet.co.uk/warwick/dashsample.ear

http://www.qenet.co.uk/warwick/dashsampleSource.zip

These files are subject to removal at any time. Please contact author for latest versions.

Please note that I have not removed any lib files from the source or ear file. I've looked over the licenses for each and I believe that I am able to re-distribute them in their original untouched form. If I'm incorrect then can someone please inform me and I shall rectify the issue.

Lib files used:

- jetty-util-6.0.1.jar
- commons-logging.jar
- commons-lang-2.1.jar
- commons-collections.jar
- commons-codec-1.3.jar
- commons-httpclient-3.0.jar
- commons-logging.jar
- commons-logging-api.jar
- dojo 4.1
- rico 1.1

1. Introduction

It was the aim of the Knowledge Based Tools For Logistics And Wireless Tracking project to prove the benefit of new technologies in the supply chain in conjunction with the most appropriate wireless tracking technologies. The project was required to create a live view of the data being gathered and use new toolkits/technologies to achieve this. This paper covers the technology and expertise acquired in achieving this.

Over the past eighteen months, the face of internet websites has started to change. From simple static pages, towards more dynamic and involving web pages. Examples of these include Google's GMail , Google Maps and Flickr. These pages allow live interaction with the data presented. This is provided by technologies like AJAX, an integral part of what is becoming known as 'WEB 2.0'.

AJAX is technique for exchanging small amounts of data between a webpage and a server, without the need to reload the entire page each time. There are many implementations of AJAX; the one chosen for the KBTFLAWT project is DOJO.

2. Techniques for pushing data

There are several techniques for server side notification of data to a browser:

RMI/CORBA

The RMI/CORBA implementation, backed with an applet, registers back to the server as a client when a web page is loaded. The server can then use the stub object to the client allowing data to be sent back to the webpage. This can be complex and present issues with firewalls because the RMI can take place over a port other than 80 (standard HTTP port), and with browsers that are locked down on corporate networks.

Pushlets (http://www.pushlets.com/)

The use of Pushlets is move towards a more flexible and usable solution. The Pushlets API works to provide a solution that doesn't require an applet on the client side, and works via a long lived connection. This is achieved via a connection much like that used for streaming of multimedia, i.e. a quick time video. This connection, instead of being closed after the data is sent, remains open and receives data from the server.

Because this works via HTTP, all communication is done via a single connection to a servlet and should work behind any firewall, although there maybe some issues with proxy's caching data.

Another benefit to the Pushlets API is that is provides a mechanism to directly tie into an AJAX implementation . This provides much neater intergration, and saves the developer the time.

Pushlets do present some limitations though; the main one being the issue of scalability. As each client connects its takes up a socket, which in turn will hold on to at least one thread and, consequently, memory. If 100's of clients connect to this service the memory usage can therefore become a significant issue, also impacting processing potential.
Pushlets do not provide a mechanism for knowing what clients receive messages. This means that it can be difficult to tell which clients are still alive and the application can be left with messages not reaching clients and broadcasting messages to clients that have disconnected.
The final issue with Pushlets is that Proxies can buffer HTTP data, this isn't under the developers control so can prove difficult to resolve in a robust manner.

XMLHttpRequest

XMLHttpRequest is an API that can be used by JavaScript to transfer and manipulate XML data to and from a web server using HTTP, establishing an independent connection channel between a web page's Client-Side and Server-Side.

XMLHttpRequest is an important part of the Ajax web development technique, and it is used by many websites to implement responsive and dynamic web applications

XMLHTTPRequest is a lower-level method of client-server communication than RMI and Pushlets, but should not be considered a low level API.. It is, however, a very powerful method of communication.

XMLHttpRequest does have several drawbacks, the main one being that it doesn't allow you to request data from a different domain. E.g. if your site located on www.warwick.ac.uk can not make an XMLHttpRequest to www.iarc-online.com. You can, however, make the request to your domain and have your server retrieve the data from a different domain, then forward it back to the webpage.

CometD (http://www.cometd.com)

The cometD website defines "Cometd as a scalable HTTP-based event routing bus that uses a push technology pattern known as Comet. In essence, it provides a mechanism for pushing data to browsers via HTTP".

CometD does provide a better mechanism for pushing data to a client browser, but this does require some changes to the web server. The web server needs to implement Continuation mechanism for asynchronous servlets. The Continuation implementation allows the web server to hold open more than two simultaneous connections to a browser. This is required because a connection can be used for checking form data at the same time as requesting an image, whilst simultaneously sending new content. This makes the application react more quickly and in a user-friendly manner and wouldn't be possible on a 'normal' web server. Currently almost all web servers, Tomcat, Jetty, Apache etc, allow only two simultaneous connections per browser.

Currently Jetty 6xx comes with a Continuation implementation, as does Tomcat 6xx [*1]. In the KBTFLAWT project we are using JBoss 4.0.5 which, at the time of writing, only contains Tomcat5, which required the use of Jetty6 or put Jetty6's servlet engine into out project in order to implement cometD.

The table below highlights the performance benefits of CometD over alternative technologies, showing approximate numbers for thread usage and connections supported.

Table 1: Performance benefits of CometD

	Formula	Web 1.0	Web 2.0 + Comet	Web 2.0 + Comet + Continuations
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- 875
Stack memory	S=64*1024*T	32MB	694MB	57MB

3. Final Choices – System choices

As the KBTFLAWT project progressed, we used the RICO and DOJO toolkits for creating the 'look and feel' of the main dashboard page.

The push mechanism used has changed as more has been learnt about available functionality.

The first mechanism used was Pushlets, this was an effiecient method of achieving the desired functionality, but once testing started with large amounts of data (1KB maximum per client) and clients (200+ clients) it became apparent that the system would be unable to perform to requirements. This could be overcome by using higher performance hardware.

CometD was first made publicly available late November 2006 and, following some initial testing, we then integrated this into our project. Below are the steps taken and the lessons learnt.

4. CometD integration (DOJO, Jetty6, JBoss-4.0.5)

First, download Jetty, CometD and Dojo. There is a presumption that JBoss is already installed.

This example was created from the scenario of wanting to watch the movement of stillages around a supply loop. To keep the design of this simple we have a class check a directory for new data, store that data, and wake up the StillagePublisher to push data to the webpage. The StillagePublisher will then push the data to the webpage.

The parts used in this example are a webpage with changing graphics, a HTTPServlet ("StillagePublisher") that will publish data out to listening webpages, a SessionBean ("DirectoryPoller") that will check for new data and wake up the "StillagePublisher".

JBoss

No additional steps other than downloading and installing with default settings are required.

Jetty

Jetty can be obtained via the Jetty website (http://dist.codehaus.org/jetty/)

Two sets of files are required, the util jar from jetty's main release, and the src files of jetty/cometD. These files are both contained in the zip 'jetty-6.0.1.zip'

File	Destination
\lib\jetty-util-6.0.1.zip	WAR\WEB-INF\lib\
\extras\cometd\src\main\java*.*	Place the '.java' files into your build environment and create a 'jetty.jar' file. This file should then be placed into WAR\WEB-INF\lib\

Fig 1. Files located in the WAR

Dojo

Dojo is available via the main Dojo website (http://dojotoolkit.org/). In this example we are using version 0.41 (http://download.dojotoolkit.org/release-0.4.1/dojo-0.4.1-ajax.zip).

Dojo does not require any jar files to be on the path for compilation, but does require the js (JavaScript) libraries to be present in the deployed WAR file.

Fig.2 Dojo; file location in the WAR layout.

Rico

Rico is being used in this example to change the colour and size of objects on the page.

JSP

The code in the JSP breaks down into two sections. The JavaScript at start and the html below.

This is the code that makes up the JavaScript section

```
<script language="JavaScript" type="text/javascript">
  var djConfig = {
  isDebug: true
}
</script>
<script type="text/javascript" src="/js/prototype.js"></script>
<script type="text/javascript" src="/js/rico.js"></script>
<script type="text/javascript" src="/js/dojo.js"></script>
<script type="text/javascript" src="/js/sample.js"></script>

<script language="JavaScript" type="text/javascript">
  dojo.require("dojo.widget.ComboBox");
  dojo.require("dojo.io.cometd");

  cometd.init({}, "cometd");
  cometd.subscribe("/hello/world", false, "publishHandler");
</script>

<script language="JavaScript" type="text/javascript">
 publishHandler = function(msg) {
   if (msg.data.typeping == "PING") {
      xxxEffectPing(msg);
   }
   if (msg.data.typeloops == "LOOPS") {
      xxxEffectLoops(msg);
   }
}
</script>
```

The djConfig needs to be defined before the external js code is referenced, otherwise you will get an error when loading the page.

The above code connect to the cometd servlet with the topic of '/hello/world' and the method name to call when data is returned ('publishHandler')

The publishHandler method checks the type of message it receives and update each part of the page depending on what the message contains. If only a ping message is sent, then only the ping data on the webpage is updated.

The second part of the jsp page is the HTML code.

```
<table width="500px" border="0" cellspacing="0" cellpadding="0">
  <tr><td>
    <table width="100%" border="0" cellspacing="0" cellpadding="0">
      <tr>
      <td><span type="text" id="loop1LoopName" name="loop1LoopName" class="medium"></span></td>

       <td><div  id="loop1Bar1"  class="loopBar  roundNormal"><a  id="loop1Bar1Message"  style="color:#5b5b5b"
>Bar1</a></div></td>

       <td><div  id="loop1Bar2"  class="loopBar  roundNormal"><a  id="loop1Bar2Message"  style="color:#5b5b5b"
>Bar2</a></div></td>

       <td><div  id="loop1Bar3"  class="loopBar  roundNormal"><a  id="loop1Bar3Message"  style="color:#5b5b5b"
>Bar3</a></div></td>

      </tr>
    </table>
</td></tr>

<tr> <td>Each bar contains the percent used to work out the colour & the width of that bar.</td> </tr>
<tr> <td><span type="text" id="ping" name="ping" class="medium"></span></td> </tr>

</table>
```

The HTML code is fairly simple, it contains 3 ‘div’ objects which will be changed into the bars. A ‘span’ object that will be updated with the ping data.

Java

There are three main java class in this example,

- The servlet that publishes the data to the client web pages.
- The session bean that discovers the new data to publish.
- The publisher that is called by the session bean that links into the Servlet.

The use of this final publisher class is a little controversial and was created because I could not find a better way to link a session bean and a servlet. There is very little documentation on CometD at the time of creating this guide and my project which is why this approach was used.

The servlet that publish the data to the webpage has two main parts. The ‘doGet’ method that publishes the message, and the method that constructs the message to be published.

This is the first part that publishes the message.

```
protected void doGet(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse) throws ServletException, IOException {
    Bayeux b = (Bayeux)getServletContext().getAttribute(CometdServlet.ORG_MORTBAY_BAYEUX);
    Channel c = b.getChannel("/hello/world");
    if (c != null) {
      Map message = new HashMap();
      publishTime(message, ""); // ping message.
      publishLoopData(message); // bar size etc.
      c.publish(message, b.newClient());
    }
}
```

The second part of code is the method that creates the message to publish.

```
private void publishLoopData(Map message) {
  try {
   // define this type of event for the JS code
   message.put("typeloops", "LOOPS");
   String loopname  = "loop1";
   // get random width
   int seedSize = 0;
   //30 is used to keep the bar to a min size else it could be 1px wide
   while ((seedSize <=30 ) || (seedSize >= 300)) {
      seedSize = random.nextInt(300);
   }
   // generate data for 3 bars.
   int sizeX1 = seedSize;
   int sizeX2 = seedSize / 2;
   int sizeX3 = seedSize * 2;

   message.put("loops",  "1");
   message.put("loop1", loopname );
   message.put(loopname + "BarCount", "3"); // 3 bars in the loop
   message.put(loopname + "Bar1PctToShw",
   message.put(loopname + "Bar1WidthToShow", "" + sizeX1 );;
   message.put(loopname + "Bar2PctToShw",
   message.put(loopname + "Bar2WidthToShow","" + sizeX2 );;
   message.put(loopname + "Bar3PctToShw",
   message.put(loopname + "Bar3WidthToShow","" + sizeX3 );;

   System.out.println(sizeX1 + " :  "  + sizeX2 + "  :  " + sizeX3 );
  } catch (Exception e) {
   publishTime(message, e.getMessage());
  }
}
```

This message is constructed using a HashMap. The above code constructs the data for 3 bars, each with a name that can be referenced via the JavaScript code.

The second main class is the session bean, the “DirectoryPoller”. This was created as a simple way to push data into the system, a class that runs on a Thread that checks a directory for any new XML files at set intervals. In this example though the class has been cut right down to call the publishing code since the Servlet described above creates its own random data.

The only code of interest in this cut down class is the run method that makes the call to the Publisher class. This code shows a call to publish all data being made.

```
public synchronized void run() {
            Publisher.getInstance().publishAlerts(Publisher.METHOD_ALL);
}
```

The final third class is the Publisher class. This singleton class uses HTTPClient, an Apache project, to make a call into the Servlet that publishes data. This is probably not the best way to approach this, but I can’t see a way of having a class that does that the StillagePublisher class does, i.e. get hold of the servlet context allowing a message to be published.

The two main method of this class are the setup method, this setups up HTTPClient to behave like an IE browser. You can also define login credentials for your website if you have security defined.

This is the setup method.

```
private void setup() {
  // reduce loggin output
  String loggingBase = "org.apache.commons.logging.";
  System.setProperty(loggingBase + "Log", "org.apache.commons.logging.impl.SimpleLog");
  System.setProperty(loggingBase + "simplelog.showdatetime", "false");
  System.setProperty(loggingBase + "simplelog.log.httpclient.wire.header", "off");
  System.setProperty(loggingBase + "simplelog.log.org.apache.commons.httpclient", "off");

  client = new HttpClient();
  // set browser settings
  client.getParams().setCookiePolicy(CookiePolicy.BROWSER_COMPATIBILITY);
  client.getParams().setParameter(HttpMethodParams.USER_AGENT, "User-Agent:Mozilla/4.0
(compatible; MSIE 6.0; Windows NT 5.1; SV1)");
  client.getParams().setVersion(HttpVersion.HTTP_1_1);
  client.getParams().setHttpElementCharset("US-ASCII");
  client.getParams().setContentCharset("ISO-8859-1");
  // IE is single header only
  client.getParams().setBooleanParameter(HttpMethodParams.SINGLE_COOKIE_HEADER, true);
}
```

The publish method is show below. This creates a post method that is sent to the stillagePublisher servlet.

```
public void publishAlerts(int method) {
try {
  NameValuePair request  = new NameValuePair("request"   , "");
  PostMethod postMethod6 = null;
  if (method == METHOD_ALL) {
    postMethod6 = new PostMethod("http://127.0.0.1/stillagePublisher?method=all");
  } else {
    postMethod6 = new PostMethod("http://127.0.0.1/stillagePublisher?method=time");
    postMethod6.setRequestBody(new NameValuePair[] { request });
    postMethod6.setDoAuthentication(false);
    client.executeMethod(postMethod6);
    postMethod6.releaseConnection();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Build

To link together the Jetty servlet engine and the cometD engine we need to add mappings to them both. In the web.xml you need to add the following:

```
<servlet-mapping>
  <servlet-name>cometd</servlet-name>
  <url-pattern>/cometd/*</url-pattern>
</servlet-mapping>


<servlet>
  <servlet-name>cometd</servlet-name>
  <servlet-class>org.mortbay.cometd.CometdServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
```

All that's left to do is build the WAR file to give a working example of cometD with Dojo.

Source Code

The entire source code and eclipse project from this demo are available via the links provided at the start of the document. Alternatively email myself the author and I will try to help.

The Results

Below are two simple screen shots showing the output of the demo application.

Fig 3: example view

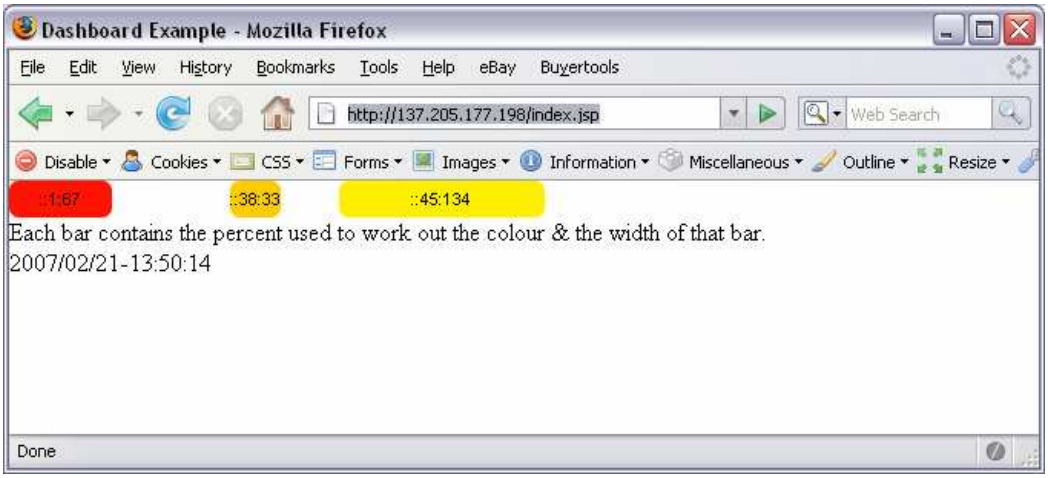


Fig 4: example view

The next stage for us on the project was to integrate this push technology into a dashboard view. The figure below shows the live dashboard, the names of those involved have been blanked out.

The dashboard shows:

- Two live loops and the positions of the stillages in each of them
- View of the events that have happened recently
- View of the alerts that have happened in the system. I.e. good arriving at the wrong location or goods arriving at a location late

Using an XMLHTTPRequest, any item in the event/alert list can be clicked on to display popup information obtained from the server.

The two screen shots below show the evolution of the prootype.

Fig 5: Live dashboard view – Version 1 (Basic integration of the main dashboard features)

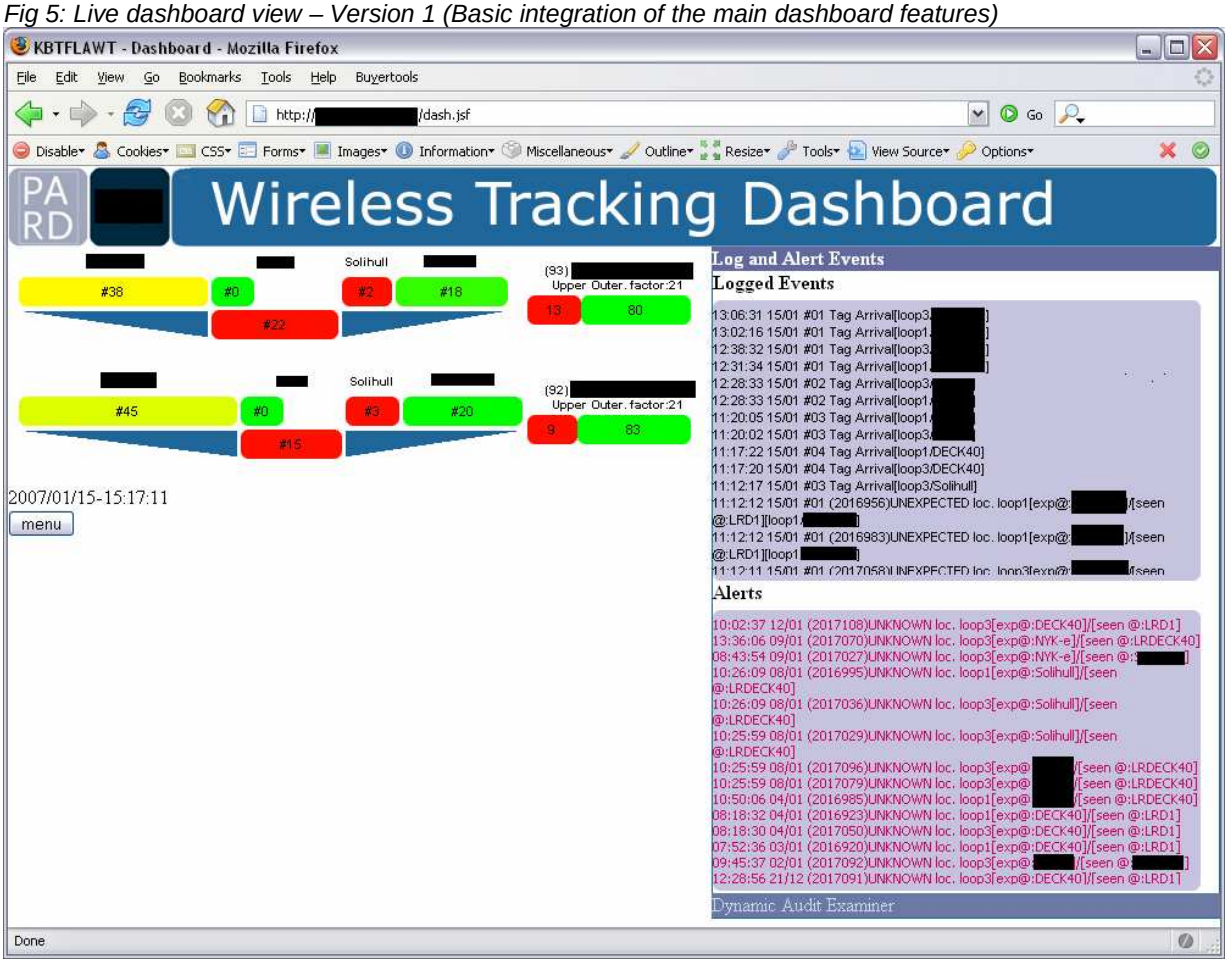
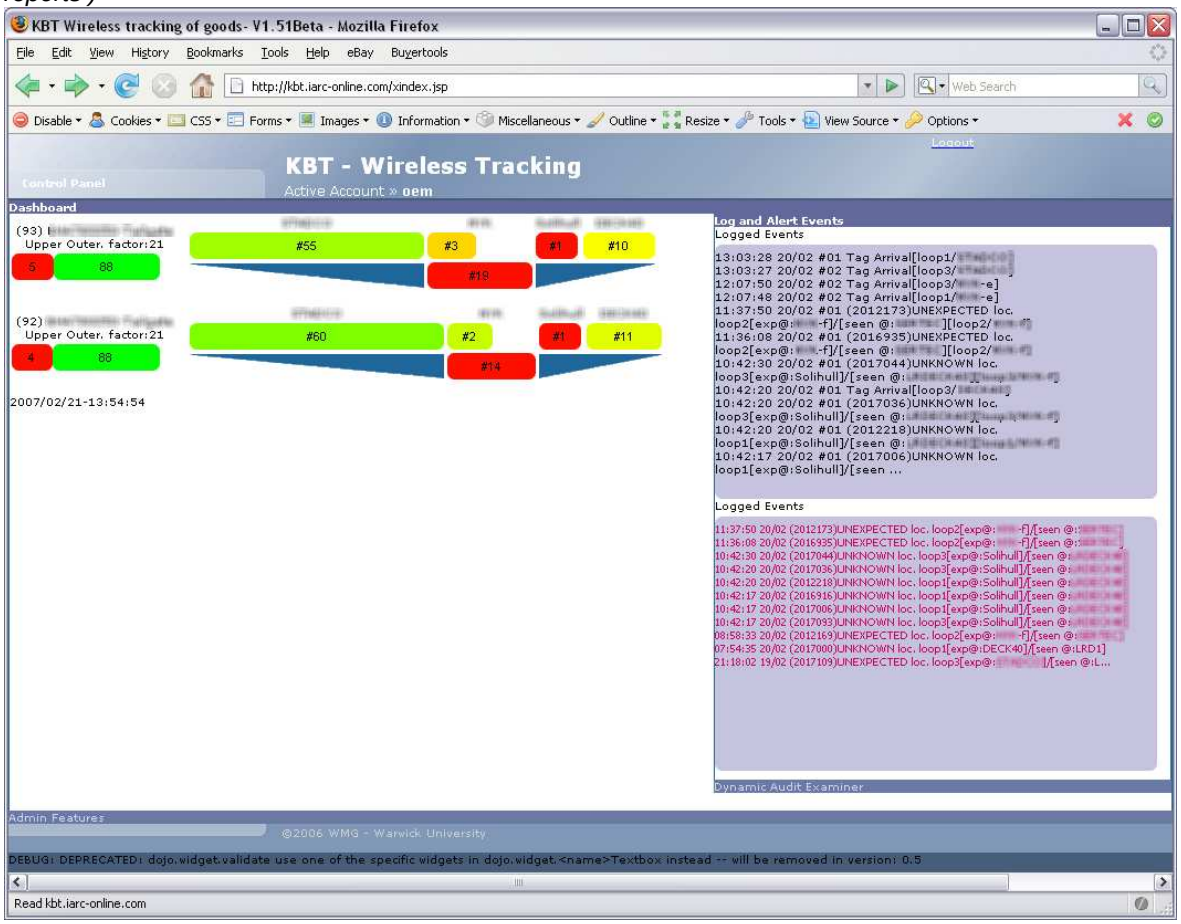


Fig 5: Live dashboard view – Version 2 (Further integration including concertinaed admin page and reports)



Bibliography

[1] http://engrm.com/blogometer/2005/11/17/pushlets-in-java/

[2] http://www.pushlets.com/

[3] http://blogs.webtide.com/gregw/2006/08/03/1154583360000.html

[4] http://www.cometd.com/

[5] http://www.mail-archive.com/users@tomcat.apache.org/msg19324.html

[6] http://dist.codehaus.org/jetty/

[7] http://docs.codehaus.org/display/JETTY/Continuations

[8] http://blogs.webtide.com/gregw/2006/08/03/1154583360000.html

[9] http://docs.codehaus.org/display/JETTY/Continuations

[10] http://ajaxian.com/archives/cometd-brining-coment-to-the-masses

[11] http://www.eclipse.org/atf/downloads/index_0.1-20060609-1400.php

[12] http://www.mail-archive.com/users@tomcat.apache.org/msg19324.html

[13] http://www.javaworld.com/jw-03-2000/jw-03-pushlet.html?page=2

[14] Wikipedia: WEB 2.0 def: http://en.wikipedia.org/wiki/Web_2