INTERPRETATION OF THE DAEMON PUBLISHING DEMO

Elliott McCrory, 2 July 2008

TABLE OF CONTENTS

INTRODUCTION

The Daemon Infrastructure in the LHC control system is based on the tools that are being used throughout the control system. For developing daemons and doing publishing, it is suggested that the daemons be developed as Spring[1] beans. This is a new concept for Fermilab/LAFS developers, and (likely) for accelerator physicists in general. But this framework, while complicated to understand in its entirety, is easy to use. Marek Misiowiec has created an example for a publishing daemon, and the "business" is entirely contained in one class. This class is easy to extend and to modify. Marek's example is available from CVS as `lsa/lsa-demo-daemon`.

First, I will describe in full detail how this example works. Then I will describe how I modified the existing example, ignoring a lot of the Spring details, to create a non-trivial daemon.

This document is unnecessary if you want to create your daemon without understanding the underlying Spring-based pieces. In this case, you can probably get away with reading only the last section, A Realistic Example of a Publishing Class.

[1] See http://en.wikipedia.org/wiki/Spring_framework for an explanation of the Spring Framework, and http://www.springsource.com for the main commercial homepage.

DAEMON EXAMPLE USING SPRING

There are two ways to run the publishing in this package, through `ConcentrationStarter`[2] or through `ParameterPublisherExample`. The former is how Marek does publishing in the BLM concentrators, and the latter is a simplified example of publishing one parameter without "concentration."

For our use for Instrumentation Daemons, I believe that `ParameterPublisherExample` is a better, simpler way to do the job at hand, so I will focus on this.

RUNNING THIS EXAMPLE

1. Check out the code from CVS and configure the project in the normal way.
2. Run `ParameterPublisherExample.java` as a Java application in the background. This publishes a new value every five (5) seconds
3. Run `ParameterMonitor.java` as a Java application. This subscribes to the data published by `ParameterPublisherExample` and shows some output.

KEY ELEMENTS OF THIS EXAMPLE

JMS PUBLISHING

Data are published using JMS, but using a JAPC/LSA wrapper. This wrapper ensures consistent use of JMS across applications, and all JMS publishing should be done this way. In CERN AB/CO, a topic must be created for your publisher. Marek chose to use an existing topic, "LHC_BLM", for this demo. You can see this specified in two places:

1. Primarily, this topic is specified in the file `concentration.parameters`, as the property `demo.concentrated.parameter`.
2. In the file `ConcentrationStarter.java`: `result.setPropertyName()`. This occurrence only specifies the default value of `demo.concentrated.parameter` if it is not otherwise set.

For the Fermilab daemon processes, it will be necessary to create a new publishing topic for each new daemon.

CLASSES IN THE PACKAGE

- **ConcentrationAdapter**. This class implements the two JAPC interfaces for receiving parameters. When a new parameter comes in, this class sorts out which one is the one we are interested in and returns it (as an `AcquiredParameterValue`). This class can be ignored.
    - `Implements cern.japc.group.ParameterGroupValueReceivedAdapter, cern.japc.spi.adaptation.AcquiredParameterValueAdapter`

[2] The example that Marek developed is based on a project he is working on, the BLM concentrators. This explains the use of the word "concentrator" throughout the example. I believe that this choice of wording has nothing to do with the actual functionality of the example.

- **ConcentrationStarter**. This is the main class for handling the concentrator-based publishing. This class can be ignored.
    - Implements `java.lang.Runnable`
- **ParameterHolder**. Used in the two publishing example. This becomes the Spring bean that handles the publishing of our custom parameter.
- **ParameterMonitor**. This class subscribes to the parameter that is being published. It allows you to see that the parameter is actually being published properly.
- **ParameterPublisher**. Becomes the Spring bean that handles the publishing in the simplified publishing example.
- **ParameterPublisherExample**. This is the main class for handling the simplified (non-concentrator) publishing. This class is listed in Appendix 1, PARAMETERPUBLISHEREXAMPLE.JAVA.
    - Implements `java.lang.Runnable`
- **PropertyPlaceholderConfigurerExt**. This is a utility class that extends the Spring class `PropertyPlaceholderConfigurer` so that the configuration parameters are read in early in the start-up cycle, and they can be used in the Spring XML file.
    - Extends `org.springframework.beans.factory.config.PropertyPlaceholderConfigurer`
- **ServiceLocator**. This class is used by both publishing examples to return the beans that do the required operations (services).
- **BCTReader.java.** A class copied from an SPS expert that subscribes to the SPS beam current transformer for a particular SPS user. This class is used by …
- **ExtendedParameterPublisher.java.** This class utilizes `BCTReader` to get the beam intensity from the SPS for a specific user cycle. Then it calculates three new numbers based on this reading and the elapsed time between readings. See the next section for discussion of these last two classes.

The classes `ParameterHolder`, `ParameterPublisher` and `ServiceLocator` are necessary to make the Spring system work properly, but can probably be used without modification by our daemons. If this is true, we should put these classes into a separate CMS package and simply use the JAR file from this package in each of the daemons.

SPRING COMPONENTS

Much of the work in binding together the objects in this demo is specified in the XML file, `publishing-spring-beans.xml`. In this file, four (4) beans are created:

1. **`placeholderPublishing`**. This bean is created so that the Spring framework starts properly on your beans. In particular, it causes the properties files to be loaded so that the rest of the Spring beans can utilize the values defined there. For example, a property called `property.one` would be accessed in subsequent beans with the syntax `${property.one}`. The attributes of this bean are:
    - Class: `cern.lsa.demo.daemons.concentration.PropertyPlaceholderConfigurerExt`. This is a helper class in this package that extends a Spring class, `PropertyPlaceholderConfigurer`, that deals with the configuration of the Spring environment.
    - Location of the property file: `concentration.properties`

    - The property `systemPropertiesModeName` is set to the value `SYSTEM_PROPERTIES_MODE_OVERRIDE`; this means that properties that are already set in the system environment take precedence over the properties set in any other way (for example, from the properties file).
2. **`DemoJmsPublishService`**. This bean configures the JMS Publishing class by setting the property, `jmsTopicPrefix`, to the desired value, `${demo.jms.topic.prefix}` (which is, in this example, set to `CERN.LHC.BLM` in the file `concentration.properties`).
    - Class: `cern.japc.ext.remote.jms.JmsPublishServiceImpl`
3. **`DemoParameterPublisher`**. Creates the bean for doing the publishing
    - Class: `cern.lsa.demo.daemons.concentration.ParameterPublisher`
    - Uses Spring to "inject" objects into this class:
        - The publishing object, `DemoJmsPublishService`, using the setter method, `setJmsPublisher()`.
        - It also specifies the parameter name to be published, `${demo.concentrated.parameter}`, through the setter `setPublishedParameter()`.
4. **`ParameterPublisherExample`**.
    - Class: `cern.lsa.demo.daemons.concentration.ParameterHolder`
    - Injects the previous bean for publishing into the main class.

MAIN CLASS DISSECTED

The main publishing class, `ParameterPublisherExample.java`, is the place where all the publishing of the daemon's data takes place. This simple example published one `java.lang.String` value every five seconds.

Java.lang.Runnable.run()

Marek has chosen for `ParameterPublisherExample` to implement `java.lang.Runnable`. This means that all of the meaningful, run-time code is contained in the method `run()`. Here is the code for that method (the entire listing is in Appendix 1).

```
public void run() {
     publisher =
        ServiceLocator.<ParameterPublisher> getSingleton("DemoParameterPublisher");

     System.out.println("Concentration started!");

     while (true) {
         try {
             // sleep a bit
             Thread.sleep(5000);
         } catch (InterruptedException ie) {
             ie.printStackTrace();
         }
         try {
            System.out.println(
                 "publishing dummy value " + (System.currentTimeMillis()/1000));
            publisher.publish(createDummyParameterValue());
         } catch (Exception e) {
             e.printStackTrace();
         }
      }
```

```
    }
```

The first statement gets the publishing object, created previously by the Spring environment and (which is `ServiceLocator.java`).

This example publishes new values once every five seconds, so it sleeps for 5000 milliseconds.

The method `publisher.publish()` takes as it's argument a `cern.japc.AcquiredParameterValue` object, which is created by the utility routine `createDummyParameterValue()`.

Thus, in the `run()` method, there are only two lines that would need to be changed—how you determine when to publish values and how you create the `AcquiredParameterValue` to publish. For the foreseen LAFS daemons, the timing will be determined by the reception of data, so there will not be a need to `sleep()`.

Method createDummyParameterValue()

There is a lot of flexibility built into JAPC parameters. Consequently, it takes several steps to create a parameter from scratch. Here is the method `createDummyParameterValue()`, adding line numbers for clarity:

```
1  private AcquiredParameterValue createDummyParameterValue() {
2  Map<String, SimpleParameterValue> valuesToReturn =
                   new HashMap<String, SimpleParameterValue>(1);
3  valuesToReturn.put(
               "crateNames",
               ParameterValueFactory.newParameterValue(new String[]{"dummy"}));
4  MapParameterValue resultValue =
               ParameterValueFactory.newParameterValue(valuesToReturn);
5  AcquiredParameterValueImpl result = new AcquiredParameterValueImpl();
6  result.setHeader(
               new ValueHeaderImpl(System.currentTimeMillis()*1000, // Acq stamp
                                   0,     // cycleTimestamp
                                   ParameterValueFactory.newSelector(null)));
7      result.setParameterName(
               System.getProperty("demo.concentrated.parameter", "LHC_BLM/Demo"));
8      result.setValue(resultValue);
9      return result;
   }
```

Line 1 and 2: The method returns an `AcquiredParameterValue`, which holds a `Map<String, SimpleParameterValue>`. The key of this `Map` is interpreted as the name of the property for the virtual device you are returning.

Line 3: The key/property is “crateNames”, and the value is a parameter that contains a `String []` value of length 1, which is the word, “dummy”.

Line 4: An object of type `cern.japc.MapParameterValue` is created based on the `Map` created in lines 2 and 3.

Line 5: A new `cern.japc.AcquiredParameterValue` instance is created, using the “`Impl`” class. (This pattern is seen quite frequently in the code produced by AB/CO/AP.) This will be the return value.

Line 6: The header for the result is created. The time stamp is taken from the system clock; there is not cycle stamp and there is no timing selector.

Line 7: The name of the parameter is set, taken from the name specified in the configuration file. This step utilizes the flexibility of the Spring system, but it may be argued that your daemon will know, a priori, the name of its parameter(s), so going to the configuration file might not be necessary.

Line 8: The value of our virtual parameter is set!

Line 9: The parameter is returned, ready to be published.

A REALISTIC EXAMPLE OF A PUBLISHING CLASS

This example has been modified to create a somewhat more realistic publishing example. This new example is contained in the class `ExtendedPublisherExample.java`, as described above. This class gets the beam intensity from the SPS for a specific user cycle. Then it calculates three new properties based on this current reading and the elapsed time between readings:

1. "totalCharge": Total beam intensity in the SPS over the last 5 minutes;
2. "averagePower": Average beam power over the last five minutes;
3. "peakPower": A peak power value, which is the measured intensity over an interval of 0.1 seconds[3].

DISSECTING THIS PUBLISHING CLASS

Starting from Marek's example, the `sleep(5000)`was removed (along with the `try/catch`) and a subscription to an SPS beam-current transformer (BCT) was created. The data from the SPS are managed in the class, `BCTReader.java`.[4]

When the subscription to the SPS BCT is created, "`this`" is added as a `ParameterValueListener` to `BCTReader`. In the `valueReceived()`method, some boiler-plate interpretation occurs, but the meat of the calculations is in the method `calculatePowers()`. This method returns success or failure, which then triggers the creation of the `AcquiredParameterValue` from the method `createParameterValue()`. The three scalar properties are returned in this parameter.

Note that you can create any type and number of properties in your virtual device. The receiver of this parameter may either hard-code the property names or it can use the introspection of `AcquiredParameterValue` to figure out what has been published. The monitoring class in this example, `ParameterMonitor`, hard-codes what to expect.

OTHER MODIFICATIONS

[3] This power calculation is probably not exactly right, in particular, it only receives data for one SPS user cycle. But (I hope) it shows how data-driven calculations work.
[4] This class was written by Anthony Rey (AB/OP)—I made only a couple of minor changes to it

The only other change was to the file `concentration.parameters`—the value of `demo.concentrated.parameter` was changed to "`LHC_BLM/Demo2`" so as not to conflict with the example Marek wrote.

SAMPLE OUTPUT

Here is a sample of the output of the publisher/daemon, `ExtendedParameterPublisher.java`:

```
8 Intensity: 1.4591200256347656E12, time=Wed Jul 02 14:28:02 CEST 2008
Time interval: 287.999 seconds. Total charge=1.0340719909667969E13
    Ave power=2588.707110278679 Watts,
    peak power=1051996.123268398 Watts.
Publishing LHC_BLM/Demo2
8 Intensity: 1.5859999656677246E11, time=Wed Jul 02 14:28:50 CEST 2008
Time interval: 288.001 seconds. Total charge=9.008479909896852E12
    Ave power=2255.177033435853 Watts,
    peak power=114347.40021886963 Watts.
Publishing LHC_BLM/Demo2
8 Intensity: 1.5225599670410156E12, time=Wed Jul 02 14:29:38 CEST 2008
Time interval: 287.999 seconds. Total charge=9.151219930648805E12
    Ave power=2290.9263870541467 Watts,
    peak power=1097735.0421011483 Watts.
Publishing LHC_BLM/Demo2
8 Intensity: 1.6652999877929688E12, time=Wed Jul 02 14:30:26 CEST 2008
Time interval: 288.0 seconds. Total charge=9.341539907455445E12
    Ave power=2338.5631777649714 Watts,
    peak power=1200647.7194876298 Watts.
Publishing LHC_BLM/Demo2
8 Intensity: 1.2053599548339844E12, time=Wed Jul 02 14:31:14 CEST 2008
Time interval: 288.0 seconds. Total charge=9.008479909896852E12
    Ave power=2255.1848639116633 Watts,
    peak power=869040.2279118099 Watts.
Publishing LHC_BLM/Demo2
8 Intensity: 1.3163800048828125E12, time=Wed Jul 02 14:32:02 CEST 2008
Time interval: 288.0 seconds. Total charge=8.88159987449646E12
    Ave power=2223.421687634478 Watts,
    peak power=949083.4458819163 Watts.
Publishing LHC_BLM/Demo2
```

And here is a sample of the output of the program (`ParameterMonitor.java`) that is monitoring this virtual parameter:

```
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001682]
    Average Power: 2588.707110278679
       Peak Power: 1051996.123268398
     Total Charge: 1.0340719909667969E13
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001730]
    Average Power: 2255.177033435853
       Peak Power: 114347.40021886963
     Total Charge: 9.008479909896852E12
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001778]
    Average Power: 2290.9263870541467
```

```
          Peak Power: 1097735.0421011483
        Total Charge: 9.151219930648805E12
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001826]
       Average Power: 2338.5631777649714
          Peak Power: 1200647.7194876298
        Total Charge: 9.341539907455445E12
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001874]
       Average Power: 2255.1848639116633
          Peak Power: 869040.2279118099
        Total Charge: 9.008479909896852E12
++ got value for LHC_BLM/Demo2 Cycle=[0] AcqStamp=[1215001922]
       Average Power: 2223.421687634478
          Peak Power: 949083.4458819163
        Total Charge: 8.88159987449646E12
```

COMMENTS AND CONCLUSIONS

PERSONAL COMMENTS

The Spring framework has been officially adopted by AB/CO for their application programs. For example, most of the Fixed Displays are implemented in this environment. It is arguable that the “simplicity” of this framework is not really very simple since every Java programmer needs to become a Spring Bean programmer and an XML programmer. The folks in AB/CO/AP who have gone through this learning curve really like Spring. And my initial introduction to it is not entirely awful—the simplicity of the resulting Java code is fabulous. But I am not fully convinced that Spring, overall, is a simplification.

Having said that, Spring is the environment that AB/CO/AP has written the Daemon Infrastructure example(s) in. I hope that this document shows you that it is possible to ignore great chunks of the Spring stuff to focus on your daemon.

CONCLUSIONS

To write your daemon to publish data, start with this package and modify the class `ExtendedParameterPublisher.java`. We may be able to encapsulate some of the Spring management to further simplify this process.

APPENDIX 1, ParameterPublisherExample.java

```
package cern.lsa.demo.daemons.concentration;
import java.util.HashMap;
import java.util.Map;

import cern.japc.AcquiredParameterValue;
import cern.japc.MapParameterValue;
import cern.japc.SimpleParameterValue;
import cern.japc.factory.ParameterValueFactory;
import cern.japc.spi.AcquiredParameterValueImpl;
import cern.japc.spi.ValueHeaderImpl;

/**
 * This class starts the LHC BLM concentrator server. Command "exit" is used to stop
the server.
 *
 * @version $Revision: 1.1 $, $Date: 2008/06/30 15:56:30 $, $Author: emccrory $
 *
 * to run needs : -DserverId=LhcBlmServer
 */
public class ParameterPublisherExample implements Runnable {

    private static Thread serverThread;

    private ParameterPublisher publisher;

    public void run() {
        this.publisher = ServiceLocator.<ParameterPublisher>
getSingleton("DemoParameterPublisher");

        System.out.println("Concentration started!");

        while (true) {
            try {
                // sleep a bit
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
            try { // and then prepare a dummy value to publish it using custom virtual
parameter publisher:
                System.out.println("publishing dummy value " +
(System.currentTimeMillis()/1000));
                this.publisher.publish(createDummyParameterValue());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }


    private AcquiredParameterValue createDummyParameterValue() {
        Map<String, SimpleParameterValue> valuesToReturn = new HashMap<String,
SimpleParameterValue>(1);
        valuesToReturn.put("crateNames",
                           ParameterValueFactory.newParameterValue(new
String[]{"dummy"}));
        MapParameterValue resultValue =
ParameterValueFactory.newParameterValue(valuesToReturn);
        AcquiredParameterValueImpl result = new AcquiredParameterValueImpl();
        result.setHeader(new ValueHeaderImpl(System.currentTimeMillis() *
1000,      // acqTimestamp
                                             0,    // cycleTimestamp
                                             ParameterValueFactory.newSelector(null)))
;
        result.setParameterName(System.getProperty("demo.concentrated.parameter",
"LHC_BLM/Demo"));
        result.setValue(resultValue);
        return result;
    }
    public static void main(String[] args) {
        ParameterPublisherExample ppe = new ParameterPublisherExample();
        serverThread = new Thread(ppe);
        serverThread.start();
    }


    /**
     * @return the publisher
     */
    public ParameterPublisher getPublisher() {
      return publisher;
    }


    /**
     * @param publisher the publisher to set
     */
    public void setPublisher(ParameterPublisher publisher) {
      this.publisher = publisher;
    }
}
```