Creating+a+New+Experiment+Driver

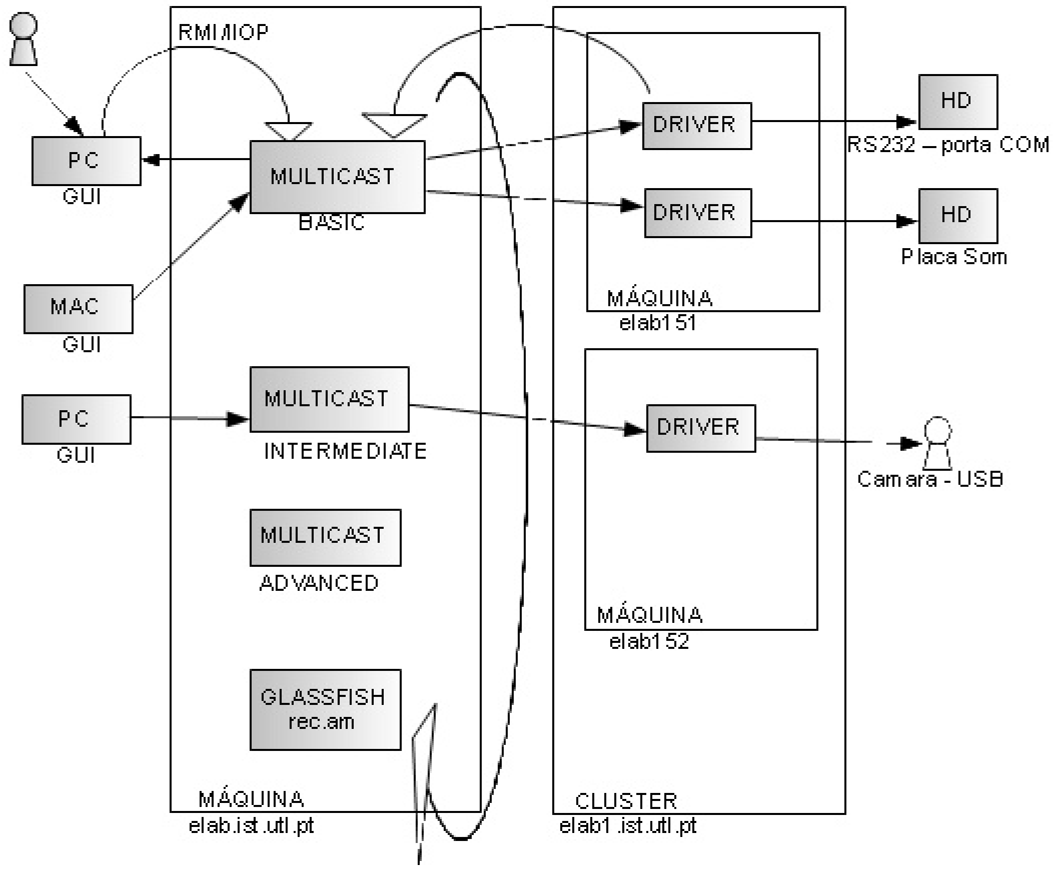
Version 1.1

Av. Duque D'Ávila 23 Sala 06A 1000-138 Lisboa Portugal
T: +351 213 590 623 F: +351 213 590 624
E-Mail: Info@linkare.com Web Site www.linkare.com

Creating a New Experiment Driver

Overview

The following picture presents a summary of the information flow associated to the ReC/e-lab project, as well as all participants in that flow. In the left side, there will be the experiment's GUI. In the middle, there is the multicast, which is the core of the system, serving as a mediator for the connections between clients and the hardware/apparatus. On the right-most side, we have the hardware itself and, on its left, there will be the drivers that allow the communication with the hardware.



Goal

This document complements the document **Creating a New Experiment GUI** by providing generic information about experiment drivers and aims to be a step by step tutorial of how to create a new experiment driver to comunicate with an hardware device or that generates virtual data.

Intended Audience

This document is targeted at software developers that need to create drivers for new experiments for elab.

Expected Background Knowledge

- Solid Java Programming Language.
- Knowledge of how to integrate java with hardware (hardware communication API's like jsr88-usb and JavaComm or RXTX, Sound API, JMF API).
- Data Acquisition Systems basic knowledge
- Some knowledge of the Scientific Methods in the field of Physics, Chemistry, Biology, Mathematics (and so forth, depending on the experiment), is quite valuable.

General Structure of a Driver

A driver can be as simple or as complex as the experiment it represents. At his most simplicity, the following files are required to build an experiment driver:

- A class that implements IDriver. This class is responsible to control the hardware and notify of hardware state changes.
- A class that implements SamplesSource. This class is responsible to receive the data from the hardware or generate virtual data.
- An HardwareInfo.xml, which specifies information about the experiment and driver inputs and outputs.
- A ServerMain class can also be specified (not mandatory) to run tests over the Driver.

Still, the driver's programmer is free to create any additional classes and interfaces to better separate concerns, take advantage of Object Oriented capabilities, etc. The driver's programmer is free to create any packaging he wishes inside the source directory for the experiment, but he should not create any experiment specific classes outside of the source folder of that particular experiment.

Creating a new Experiment (The Rollpaper Experiment)

Step I - Create the Structure for a New Experiment

There is one single directory where elab keeps all its experiments. This directory (*experiments*) exists at elab's root directory.

The base structure should be created after you use the target:

ant -f build_new_experiment.xml create.experiment -Dexperiment.name=rollpaper

After executing this command, the *experiments/rollpaper/src/java/server* directory was created.

Let's now create two packages with the following files inside each of them:

- pt.utl.ist.elab.driver.rollpaper: experiment classes plus HardwareInfo.xml, when required
  - RollPaperDriver.java: Represents the hardware driver.
  - RollPaperDataProducer.java: Represents the producer of the data.
  - RollPaperSampleGenerator.java: Generates simulated data (since this sample experiment is a virtual experiment). It's not required by rec. It's just a helper class.
  - ServerMain.java: main class from which the hardware server for this experiment will be launched.
  - HardwareInfo.xml: contains the hardware configuration. It may not be necessary, depending of the implementation of the driver's getHardwareInfo() implementation.
- pt.utl.ist.elab.driver.rollpaper.resources: experiment resources for the driver
  - RollPaperBytesArray.txt: Used to temporarily store the byte array information of the image.
  - test.gif: Used in this experiment to simulate byteArray PhysicsVal types.

You are free to organize your packaging any way you want, but try to follow these standards. You may create subpackages freely, if you will.

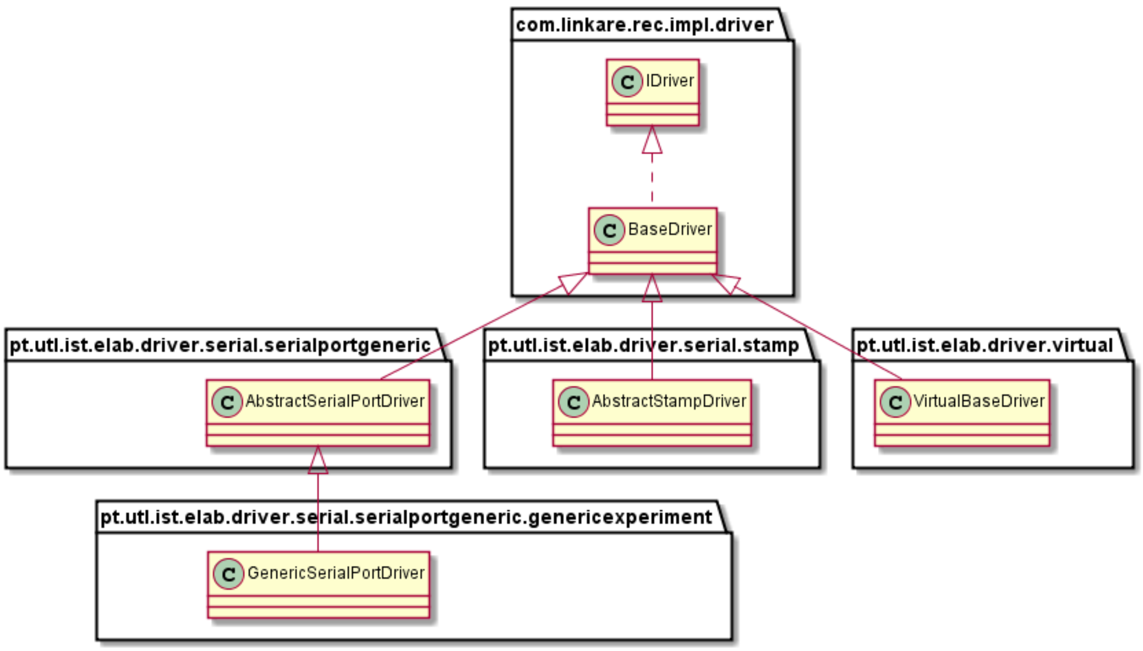
The files in the pt.utl.ist.elab.driver.rollpaper package will be explained in more detail in the following sections.

Step 2 - Create the Driver Class

Choosing the right parent class/interface

The driver class, which is responsible to control the hardware, must directly or indirectly implement com.linkare.rec.impl.driver.IDriver. The ReC framework provides an abstract class that already guarantee some common behaviour to all drivers that extend it. That's the BaseDriver class.

Elab also provides a set of abstract driver types. The most important is the GenericSerialPortDriver. The diagram below shows how these driver classes and interfaces relate to each other.


com.linkare.rec.impl.driver
IDriver
BaseDriver
pt.utl.ist.elab.driver.serial.serialportgeneric
AbstractSerialPortDriver
pt.utl.ist.elab.driver.serial.stamp
AbstractStampDriver
pt.utl.ist.elab.driver.virtual
VirtualBaseDriver
pt.utl.ist.elab.driver.serial.serialportgeneric.genericexperiment
GenericSerialPortDriver

When implementing a new driver, always remember to check if an abstraction for that driver type already exists that could make your job easier.

In our example, we'll extend directly the BaseDriver class from the ReC framework.

```
public class RollPaperDriver extends BaseDriver {
  public RollPaperDriver () {
  }
```

Note the definition of a no-args constructor, so newInstance() can be called by the ReC framework.

Implement getHardwareInfo()

This method is responsible to create an hardware configuration. A typical implementation for this method is:

```
public Object getHardwareInfo () {
final String baseHardwareInfoFile = "recresource://" + getClass (). getPackage (). getName ().
replaceAll ( "\\." , "/" ) + "/HardwareInfo.xml" ;
 String prop = Defaults . defaultIfEmpty (System . getProperty ( "HardwareInfo" ),
baseHardwareInfoFile );
if (prop . indexOf ( "://" ) == - 1 ) {
        prop = "file:///" + System . getProperty ( "user.dir" ) + "/" + prop ;
}
 java. net . URL url = null ;
try {
        url = ReCProtocols . getURL (prop );
} catch ( final java . net . MalformedURLException e ) {
        LoggerUtil.logThrowable ( "Unable to load resource: " + prop , e , Logger . getLogger
(ROLLPAPER_DRIVER_LOGGER ));
     try {
              url = new java . net . URL (baseHardwareInfoFile );
     } catch ( final java . net . MalformedURLException e2 ) {
              LoggerUtil.logThrowable ( "Unable to load resource: " + baseHardwareInfoFile , e2 , Logger .
getLogger (ROLLPAPER_DRIVER_LOGGER ));
     }
}
return url ;
}
```

This implementation requires that a file named HardwareInfo.xml exists in the same directory as the driver class (this file will be explained in a section below).

Although the most common implementation is having this file and return an URL to that file, that's not the only implementation possible for this method. This approach is usually better because it centers the configuration of the hardware on a single file. Since it's xml, it's easy to understand by people without some knowledge of programming, if they understand a minimum of the document structure (which is easier to learn than a programming language). Even to a programmer, the information in xml is more structured and it's easier to avoid mistakes.

The getHardwareInfo() method can return:

- An instance of the com.linkare.rec.data.metadata.HardwareInfo class. This implementation doesn't need that an HardwareInfo.xml exists.
- An URL to an HardwareInfo.xml file, which will be automatically converted into an instance of com.linkare.rec.data.metadata.HardwareInfo.
- A string representing the URL to an HardwareInfo.xml file.
- An InputStream to the HardwareInfo.xml file.

Implement getDriverUniqueID()

It identifies the driver. The implementation is very simple:

```
private static final String DRIVER_UNIQUE_ID = "ROLLPAPER_V1.0" ;
public String getDriverUniqueID () {
    return DRIVER_UNIQUE_ID ;
}
```

Implement init(HardwareInfo)

Does any initialization code necessary. In our example, we only need to notify the driver listener that the driver has been initiated, using a method already available on the BaseDriver.

```
public void init (HardwareInfo info ) {
       fireIDriverStateListenerDriverInited();
    //Other initialization code that may be necessary
}
```

Implement extraValidateConfig(HardwareAcquisitionConfig, HardwareInfo)

Before the driver configuration process, the Hardware configuration is validated according to the current HardwareInfo (check if mandatory values are present, if values are within certain ranges of values, etc).

Still, there are some validations that may not be possible to map in the HardwareInfo. There may exist context validations. As an example, let's imagine there are two configuration parameters:

param1 and param2. It is told in the HardwareInfo that param1 can have values between 1 and 3. But the experiment requires that param2 has values between 1 and 10 if param1 = 1, 11 to 43 if param1=2, and 4, 6, 12 to 15 and 42 if param1=3. This cannot be represented in HardwareInfo and requires more complex logic to check for these restrictions.

The extraValidateConfig is an extension point for the Driver to implement these complex rules that are not automatically validated by the framework.

In our cenario, no additional validation is necessary, so we'll leave the method empty.

Implement configure(HardwareAcquisitionConfig, HardwareInfo)

Prepares the driver to execute according to a given set of parameters. Sets all the parameters used as the hardware input and instantiates the data source that will collect the data from the hardware (or generate virtual samples if it's a virtual experiment). In the end, it notifies it's listener that it has reached the configured state.

```
public void configure (HardwareAcquisitionConfig config , HardwareInfo info ) throws
WrongConfigurationException , IncorrectStateException , TimedOutException {
this . config = config ;
this . info = info ;
final int tbs = ( int ) config . getSelectedFrequency (). getFrequency ();
final int nSamples = config . getTotalSamples ();
final float baseHeight1 = Float . parseFloat (config . getSelectedHardwareParameterValue (
"baseHeight1" ));
final float baseHeight2 = Float . parseFloat (config . getSelectedHardwareParameterValue (
"baseHeight2" ));
final float g = Float . parseFloat (config . getSelectedHardwareParameterValue ( "g" ));
final float outerRadius = Float . parseFloat (config . getSelectedHardwareParameterValue (
"outerRadius" ));
final float innerRadius = Float . parseFloat (config . getSelectedHardwareParameterValue (
"innerRadius" ));
 dataSource= new RollPaperDataProducer ( this , baseHeight1 , baseHeight2 , g , outerRadius ,
innerRadius , tbs , nSamples );
for ( int i = 0 ; i < config . getChannelsConfig (). length ; i ++) {
        configgetChannelsConfig (i ). setTotalSamples (config . getTotalSamples ());
}
 dataSource. setAcquisitionHeader (config );
 fireIDriverStateListenerDriverConfigured();
}
```

Note that it's not necessary to notificate for the configuring state, because the BaseDriver already does that step when configuring an experiment that extends from BaseDriver.

Implement start(HardwareInfo)

This method is called when the driver's client asks for the experiment to start.

Usually, the driver should inform its listener that the experiment has reached the starting state. In the meanwhile, it asks the data producer to start acquiring data. After this step, the driver has actually started, so it notifies its listener that it has reached the started state.

```
public IDataSource start (HardwareInfo info ) throws IncorrectStateException {
      fireIDriverStateListenerDriverStarting();
      dataSourcestartProduction ();
      fireIDriverStateListenerDriverStarted();
   return dataSource ;
}
```

The data source will be explained later.

Implement startOutput(HardwareInfo, IDataSource)

It makes it possible to start the experiment, but using a data source other than the default for this driver. We won't be needing this (and most experiments don't need it also), so we'll just return null.

```
public IDataSource startOutput (HardwareInfo info , IDataSource source ) throws
IncorrectStateException {
   return null ;
}
```

Implement stop(HardwareInfo)

Called when the driver's client decides to stop the experiment.

It usually uses a similar logic to the start method. It first notifies the listener that it has reached the stopping state, then it asks the data source to stop producing data, and finally it notifies the listener that it has reached the stopped state.

```
public void stop (HardwareInfo info ) throws IncorrectStateException {
      fireIDriverStateListenerDriverStoping();
      dataSourcestopNow ();
      fireIDriverStateListenerDriverStoped();
}
```

Implement reset(HardwareInfo)

Used to reset the hardware. A reset may be done when

In the rollpaper example, no hardware reset is needed, so let's leave the method blank.

Add a hook for data producer callbacks when it stops feeding data

It's not required by the ReC framework, but it's desirable that the driver is notified after the datasource stops producing data. This is only necessary for virtual experiments.

```
public void stopVirtualHardware () {
      fireIDriverStateListenerDriverStoping();
      fireIDriverStateListenerDriverStoped();
}
```

Step 3 - Configuring HardwareInfo

As mencioned earlier, the Hardware configuration can be represented as an xml file or as an object instanciated from HardwareInfo class. The most commons approach is using the xml file, since it's easier to read and maintain. We used this approach on our rollpaper example.

Header and Description

```
<?xml version="1.0" encoding="UTF-8"?>
<HardwareInfo id= "ROLL_DOWN_TOILET_PAPER_V1.0" familiarName= "Rollpaper" name=
"Rollpaper" version= "01" manufacturer= "Elab - UTL" driverVersion= "v1" >
<description>
        <![CDATA[
             Rollpaper.
             Drop two rolls of toilet paper of the same size. One is completely free to fall and the other hung
by a tip.
   Of course, the free roll reaches the ground first because the acceleration of the first roll is less
stuck.
   The goal of the experiment is to predict the initial heights that the rollers must have in order to
reach
   the ground simultaneously. In the process, you have to use concepts of acceleration and momentum
of inertia.

             ]]>
```

```
</description>
```

The HardwareInfo is the root element of the HardwareInfo.xml document. It contains an experiment identifier and several informative attributes. You can also add a description for the experience. This description is not currently used by the elab interface. It should also be a key to a resource bundle entry, to allow internationalization.

The id attribute must match exactly to the location attribute of the apparatus element in the ReCFaceConfig.xml for the experiment. The string is case-sensitive. If they are different in any way, the experiment will appear in the interface, but will not be available, since the framework won't be able to map the experiment in the client with an experiment registered in the multicast controller.

Experiment Parameters

An experiment parameter can be defined with the following xml block:

```
<parameter name= "height1" type= "ContinuousValue" value= "10.0" >
<selection-list>
  <value order= "1" >10.0 </value>
  <value order= "2" >100.0 </value>
  <value order= "3" >1E-10 </value>
</selection-list>
</parameter>
```

The attributes name and value are self explanatory.

The attribute type can have the following values:

- OnOffValue: acts like a boolean, with 0 and 1 as possible values.
- ContinuousValue: any decimal value. The values are Scientific notation is allowed.
- SelectionListValue: a fixed set of values.
- BlackBoxValue: any type that doesn't fit in the first three types.

The selection-list element child elements define the range of selection.

For a Continuous Value, the order of the value element represents:

1. Minimum Value
2. Maximum Value
3. Step (the value that each unit increment adds to the previous value)

For other types, the order represents the valid values for the parameter. Here are some examples of parameters for the other types, for reference (not part of the rollpaper experiment):

```
<parameter name= "MovieOnOff" type= "OnOffValue" value= "0" >
   <selection-list>
      <value order= "1" >0 </value>
      <value order= "2" >1 </value>
   </selection-list>
</parameter>
<parameter name= "Material" type= "SelectionListValue" value= "Chumbo (isolante)" >
   <selection-list>
      <value order= "1" >Madeira [10mm] </value>
      <value order= "2" >Corticite [10mm] </value>
      <value order= "3" >Tijolo [10mm] </value>
      <value order= "4" >Cobre [0.2mm] </value>
      <value order= "5" >Cobre [0.4mm] </value>
      <value order= "6" >Cobre [0.8mm] </value>
      <value order= "7" >Cobre [1.6mm] </value>
      <value order= "8" >Cobre [3.2mm] </value>
      <value order= "9" >Janela de controlo (Ar) </value>
      <value order= "10" >Chumbo (isolante) </value>
   </selection-list>
</parameter>
```

```
<parameter name= "graph2" type= "BlackBoxValue" value= "t.| r |" >
</parameter>
```

To complete the example, we have 4 more parameters defined in the rollpaper experiment, all of the same type:

```
<parameter name= "height2" type= "ContinuousValue" value= "10.0" >
<selection-list>
  <value order= "1" >10.0 </value>
  <value order= "2" >100.0 </value>
  <value order= "3" >1E-10 </value>
</selection-list>
</parameter>
<parameter name= "g" type= "ContinuousValue" value= "9.8" >
<selection-list>
  <value order= "1" >0.2 </value>
  <value order= "2" >12.0 </value>
  <value order= "3" >5E-11 </value>
</selection-list>
</parameter>
<parameter name= "outerRadius" type= "ContinuousValue" value= "10.0" >
<selection-list>
  <value order= "1" >7.0 </value>
  <value order= "2" >12.0 </value>
  <value order= "3" >1E-10 </value>
</selection-list>
</parameter>
<parameter name= "innerRadius" type= "ContinuousValue" value= "4.0" >
<selection-list>
  <value order= "1" >0.5 </value>
  <value order= "2" >7.0 </value>
  <value order= "3" >1E-10 </value>
</selection-list>
</parameter>
```

Frequency Scale

The frequency by which the samples are collected. If necessary, the user may set the frequency in the interface.

This element can be defined as below

```
<frequency-scale label= "Sampling Interval" >
<min-frequency><Frequency type= "SamplingIntervalType" multiplier= "none" >10 </Frequency></
min-frequency>
<max-frequency><Frequency type= "SamplingIntervalType" multiplier= "none" >50 </
Frequency></max-frequency>
<step-frequency><Frequency type= "SamplingIntervalType" multiplier= "none" >1 </Frequency></
step-frequency>
</frequency-scale>
```

The min-frequency and max-frequency define the internal in which the frequency may vary. The step-frequency defines the frequency variation between each value possible to select.

The Frequency type may have the following values:

- SamplingIntervalType: total number of samples.
- FrequencyType: number of samples in a time unit, given in a multiple of Hz.

The multiplier defines a potence of 10^(3*x) that will be multiplied by the value of the Frequency element. This avoids using very big numbers, by making all the calculations with small values in a certain unit and multiply the result by the multiplier's potence value. The multiplier can have the following values:

- fento (x = -5)
- pico (x = -4)
- nano (x = -3)
- micro (x = -2)
- milli (x = -1)
- none (x = 0)
- kilo (x = 1)
- mega (x = 2)
- giga (x = 3)
- tera (x = 4)

You can also define the default frequency for the experiment:

```
<Frequency type= "SamplingIntervalType" multiplier= "none" >15 </Frequency>
```

Sampling Scale

Defines the number of samples that will be obtained in an experiment's execution. The element names follow the same logic as in the frequency-scale.

```
<sampling-scale>
<min-samples>10 </min-samples>
<max-samples>250 </max-samples>
<step-samples>10 </step-samples>
</sampling-scale>
```

Channel Info

The ChannelInfo element is used to define one channel, or output, of the experiment. One basic sample of a channel configuration is:

```
<ChannelInfo name= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.0.name"
independent= "false" direction= "CHANNEL_INPUT" >
<scale label= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.0.label"
   physicsunitname= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.0.physicsunitname"
   physicsunitsymbol= "m" multiplier= "none" selected= "true" >
  <min><PhysicsVal type= "float" >10 </PhysicsVal></min>
  <max><PhysicsVal type= "float" >100 </PhysicsVal></max>
  <step><PhysicsVal type= "float" >1 </PhysicsVal></step>
  <errordefault><PhysicsVal type= "float" >1E-5 </PhysicsVal></errordefault>
</scale>
</ChannelInfo>
```

The ChannelInfo name represents the name of the channel. It's a property file defined in the experiment's messages.properties in the resources package of the client (see Creating an Experiment GUI). The direction can be CHANNEL_INPUT or CHANNEL_OUTPUT.

A channel must define a scale. The label matches the value showed in the data tables on the client table display. The same for the physicsunitsymbol. The label and the physicsunitname should also be in a properties file, to allow internationalization. The multiplier was explained above, and has the same possible values of frequency scale.

A scale includes four different PhysicsVal:

- min: the minimum value that goes thru the channel.
- max: the maximum value that goes thru the channel.
- step: the difference in value between two consecutive possible values in the channel.
- errordefault: the error of the channel.

The type of the PhysicsVal can have the following values:

- boolean (with any of the pairs on/off, yes/no, true/false, 1/0 being valid)
- byte
- short

- int
- long
- float
- double
- byteArray (requires that another attribute named mimetype is given. Also need that an href or a file attribute is present)

Here's the rest of the channel configuration for the rollpaper example:

```
<ChannelInfo name= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.1.name"
independent= "false" direction= "CHANNEL_INPUT" >
<scale label= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.1.label"
  physicsunitname= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.1.physicsunitname"
  physicsunitsymbol= "m" multiplier= "none" selected= "true" >
 <min><PhysicsVal type= "float" >10 </PhysicsVal></min>
 <max><PhysicsVal type= "float" >100 </PhysicsVal></max>
 <step><PhysicsVal type= "float" >1 </PhysicsVal></step>
 <errordefault><PhysicsVal type= "float" >1E-5 </PhysicsVal></errordefault>
</scale>
</ChannelInfo>
<ChannelInfo name= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.2.name"
independent= "false" direction= "CHANNEL_INPUT" >
<scale label= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.2.label"
  physicsunitname= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.2.physicsunitname"
  physicsunitsymbol= "m" multiplier= "none" selected= "true" >
 <min><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></min>
 <max><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></max>
 <step><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></step>
 <errordefault><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></errordefault>
</scale>
</ChannelInfo>
<ChannelInfo name= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.3.name"
independent= "false" direction= "CHANNEL_INPUT" >
<scale label= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.3.label"
  physicsunitname= "rollpaper$rec.exp.rollpaper.hardwareinfo.channel.3.physicsunitname"
  physicsunitsymbol= "m" multiplier= "none" selected= "true" >
 <min><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></min>
 <max><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></max>
 <step><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></step>
 <errordefault><PhysicsVal type= "byteArray"
   href= "recresource:///pt/utl/ist/elab/driver/rollpaper/resources/RollpaperByteArray.txt"
   mimetype= "image/jpeg" /></errordefault>
</scale>
</ChannelInfo>
```

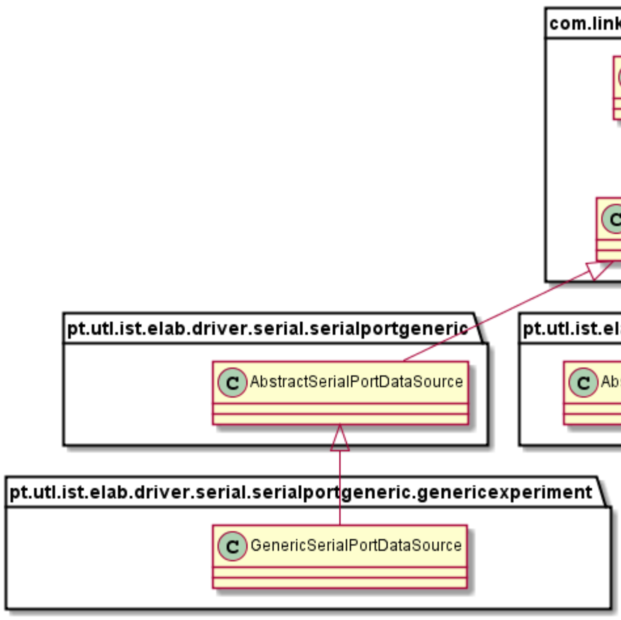
Please, be aware that the used xml parser doesn't handle well spaces and line breaks, so try to keep the formatting of the scale and frequency-scale child elements as shown in the examples. Not doing this might cause parsing errors hard to find.

Step 4 - Create the DataSource Class

The datasource class is responsible to collect the data sent by the hardware and store it temporarily, until a certain number of samples was collected or until a certain amount of time has passed. When one of these conditions is met, the samples currently stored in the data source are dispatched to the multicast controller, which will in turn deliver the samples to any interested client. In that moment, the data source clears its memory of the samples that were just sent and keeps collecting samples from the hardware. In the case of a virtual experiment, like our example, the data source is also responsible to generate "simulated" data.

Choosing the right parent class/interface

Much like the driver, it is represented by a single interface: com.linkare.rec.impl.driver.IDataSource. The ReC framework also provides some base implementation in the form of the abstract class com.linkare.rec.impl.driver.BaseDataSource. Elab provides some abstract classes with more specific behaviour that can be extended by a data source implementation.



com.linkare.rec.impl.driver
IDataSource
BaseDataSource
pt.utl.ist.elab.driver.serial.serialportgeneric
AbstractSerialPortDataSource
pt.utl.ist.elab.driver.serial.stamp
AbstractStampDataSource
pt.utl.ist.elab.driver.virtual
VirtualBaseDataSource
pt.utl.ist.elab.driver.serial.serialportgeneric.genericexperiment
GenericSerialPortDataSource

In the rollpaper example, we'll extend from com.linkare.rec.impl.driver.BaseDataSource.

```
public class RollPaperDataSource extends BaseDataSource {
    private final int NUM_CHANNELS = 4 ;
    private int tbs ;
    private final int nSamples ;
    private RollPaperDriver driver = null ;
    private RollpaperHardwareSimulator samplesGenerator ;
    private boolean stopped = false ;
    public RollPaperDataSource (RollPaperDriver rollPaperDriver , float baseHeight1 , float
baseHeight2 , float g , float totalRadius , float innerRadius , int tbs ,
     int nSamples ) {
        this . driver = rollPaperDriver ;
        this . nSamples = nSamples ;
        this . tbs = tbs ;
        samplesGenerator = new RollpaperHardwareSimulator (NUM_CHANNELS , baseHeight1 ,
baseHeight2 , g , totalRadius , innerRadius , getAcquisitionHeader ());
    }
```

Please, note that the constructor receives all the parameters and creates an object of the type RollpaperHardwareSimulator. This is just an auxiliary class for this experiment. It acts as the physical hardware, from where the data source will receive the data samples. We'll see the implementation of this class ahead in this tutorial.

Implement the producer thread

The samples must be received and processed in a thread other than the hardware server's main thread, which means we need to start our own thread. Let's begin by creating our thread class, in this case, as an inner class:

```
private class ProducerThread extends Thread {
    private int currentSample = 0 ;
    @Override
    public void run () {
        try {
            Thread . sleep ( 1000 );
            PhysicsValue [] value ;
            int counter = 0 ;
            while (currentSample < nSamples && !stopped ) {
                if (counter % tbs == 0 ) {
                    value = samplesGenerator . getNewSample ( 1 / tbs * currentSample );
                    addDataRow ( value );
                    Thread . sleep ( tbs );
                    currentSample ++;
                }
                counter ++;
            }
            join ( 100 );
            endProduction ();
            driver . stopVirtualHardware ();
        } catch ( final InterruptedException ie ) {
            LoggerUtil . logThrowable ( "Error in Thread: " , ie , Logger . getLogger (RollPaperDriver .
ROLLPAPER_DRIVER_LOGGER ));
        }
    }
}
private void endProduction () {
    stopped = true ;
    setDataSourceEnded ();
}
```

As you see, the producer thread delegates the samples production in the RollpaperHardwareSimulator. The Thread is only responsible to control the flow of the samples production. As you see, after each iteration, the Thread sleeps for the frequency time, to simulate the parametrized frequency.

At the end of the production of all the samples, it notifies it's listener that the production ended and notifies the driver that the virtual hardware terminated generating samples.

The method endProduction is an auxiliary method used to notify the data source listener that the data feed has reached the end (the maximum number of samples was received).

Implement the method to start producing samples

As you may remember, in the driver's start method, the startProduction method of the datasource was called. The method name is not mandatory (it is not defined in any interface), but it is necessary to create in the datasource some method to start producing data.

```
public void startProduction () {
      stopped= false ;
    new ProducerThread (). start ();
}
```

Implement the method to stop producing samples

This method is called by the driver when the stop method of the driver is called.

```
public void stopNow () {
      stopped= true ;
      setDataSourceStoped();
}
```

Implement the auxiliary class RollpaperHardwareSimulator

As mencioned earlier, this class simulates the physical hardware and creates the data that the datasource will receive and store.

This is the class declaration, attributes and constructor:

```
public class RollpaperHardwareSimulator {
    private final int numChannels ;
    private final double baseHeight1 ;
    private final double baseHeight2 ;
    private final double gravityAcceleration ;
    private final double totalRadius ;
    private final double innerRadius ;
    private final HardwareAcquisitionConfig hardwareConfig ;
    private double rollAcceleration ;
    public RollpaperHardwareSimulator ( int numChannels , double baseHeight1 , double baseHeight2 ,
double g , double totalRadius , double innerRadius ,
        HardwareAcquisitionConfig hardwareConfig){
        this . numChannels = numChannels ;
        this . baseHeight1 = baseHeight1 ;
        this . baseHeight2 = baseHeight2 ;
        this . gravityAcceleration = g ;
        this . totalRadius = totalRadius ;
        this . innerRadius = innerRadius ;
        this . hardwareConfig = hardwareConfig ;
        this . rollAcceleration = getRollAcceleration (). doubleValue ();
    }
```

In this case, the constructor only sets the configuration for the "virtual hardware".

```
public PhysicsValue [] getNewSample ( double time ) {
      PhysicsValue[] value = null ;
    double heightHeld = getCurrentHeight (baseHeight1 , time , rollAcceleration ). doubleValue ();
    double heightFreefall = getCurrentHeight (baseHeight2 , time , gravityAcceleration ). doubleValue ();
    byte [] snapshotHeld = getSnapshot ();
    byte [] snapshotFreeFall = getSnapshot ();
    if (snapshotHeld == null && snapshotFreeFall == null ) {
          value= new PhysicsValue [numChannels - 2 ];
    } else if (snapshotFreeFall == null ) {
          value= new PhysicsValue [numChannels - 1 ];
    } else {
          value= new PhysicsValue [numChannels ];
    }
      value[ 0 ] = new PhysicsValue (PhysicsValFactory . fromDouble (heightHeld ), hardwareConfig .
getChannelsConfig ( 0 ). getSelectedScale (). getDefaultErrorValue (),
          hardwareConfig.getChannelsConfig ( 0 ). getSelectedScale (). getMultiplier ());
      value[1 ] = new PhysicsValue (PhysicsValFactory . fromDouble (heightFreefall ),
            hardwareConfig.getChannelsConfig ( 1 ). getSelectedScale (). getDefaultErrorValue (),
hardwareConfig . getChannelsConfig ( 1 )
                . getSelectedScale ()
                . getMultiplier ());
    if (value . length > 1 ) {
            value[2 ] = snapshotHeld != null ? new PhysicsValue (PhysicsValFactory . fromByteArray
(snapshotHeld , "image/gif" ),
                        hardwareConfig.getChannelsConfig ( 2 ). getSelectedScale (). getDefaultErrorValue (),
                        hardwareConfig.getChannelsConfig ( 2 ). getSelectedScale (). getMultiplier ()) : null ;
    }
    if (value . length > 2 ) {
            value[3 ] = new PhysicsValue (PhysicsValFactory . fromByteArray (snapshotFreeFall , "image/
gif" ), hardwareConfig . getChannelsConfig ( 3 ). getSelectedScale ()
                   . getDefaultErrorValue (),
        hardwareConfig.getChannelsConfig ( 3 ). getSelectedScale (). getMultiplier ());
    }
    return value ;
}
```

Returns one single sample of data, with a PhysicsVal for each channel.

Please note that if the last channels have null values, those columns will not be sent. ReC supports not sending the last columns if null, to avoid sending these values, which are expensive in corba. To the current date, there was no performance problems that justify not sending the null values when they are present, so this functionality was lightly tested. Using this approach might lead you to face some problems with the multicast controller when deserializing a sample with less columns than the number of expected channels.

So, if your experiment's performance is low and you need to cut somewhere, not sending null values in the last channels is a good place to start.

To complete the rollpaper example:

```
private byte [] getSnapshot () {
    if (Math . random () < 0.3 ) {
        return null ;
    }
    try {
            InputStream is = this . getClass (). getClassLoader (). getResourceAsStream ( "pt/utl/ist/elab/
driver/rollpaper/resources/test.gif" );
        byte [] buffer = new byte [ 1024 ];
            ByteArrayOutputStream baos = new ByteArrayOutputStream ();
        while (is . read (buffer ) > 0 ) {
                baos.write (buffer );
        }
        return baos . toByteArray ();
    } catch (IOException e ) {
        return null ;
    }
}
private BigDecimal getCurrentHeight ( double baseHeight , double time , double acceleration ) {
    return new BigDecimal (baseHeight ). subtract (getDistance (time , acceleration ));
}
private BigDecimal getDistance ( double time , double acceleration ) {
    return new BigDecimal (acceleration ). multiply ( new BigDecimal (time ). pow ( 2 )). divide ( new
BigDecimal ( 2 ));
}
private BigDecimal getRollAcceleration () {
      BigDecimal numerator = new BigDecimal ( 2 ). multiply ( new BigDecimal (gravityAcceleration )).
multiply ( new BigDecimal (totalRadius ). pow ( 2 ));
      BigDecimal denominator = new BigDecimal ( 3 ). multiply ( new BigDecimal (totalRadius ). pow ( 2
)). add ( new BigDecimal (innerRadius ). pow ( 2 ));
```

```
    return numerator . divide (denominator );
}
```

Step 5 - Creating the ServerMain Class

This is the executable java Main class which is invoked in the hardware server's start scripts. The most common implementation of a ServerMain class is:

```
public class ServerMain {
    private static String RollPaper_HARDWARE_LOGGER = "RollPaper.Logger" ;
    static {
        final Logger l = LogManager . getLogManager (). getLogger (ServerMain .
RollPaper_HARDWARE_LOGGER );
        if (l == null ) {
LogManager . getLogManager (). addLogger (Logger . getLogger (ServerMain .
RollPaper_HARDWARE_LOGGER ));
        }
    }
    public static void main ( final String [] args ) {
        try {
                ORBBean getORBBean ();
            new BaseHardware ( new RollPaperDriver ());
                Thread currentThread (). join ();
        } catch ( final Exception e ) {
                ORBBean getORBBean (). killORB ();
                LoggerUtil logThrowable ( "Error on Main..." , e , Logger . getLogger (ServerMain .
RollPaper_HARDWARE_LOGGER ));
        }
    }
}
```

Step 6 - Integrating with the Driver's Client