AMI

AMS Monitoring Interface

G. Alberti and P. Zuccon

INFN – Sezione di Perugia

v0.05

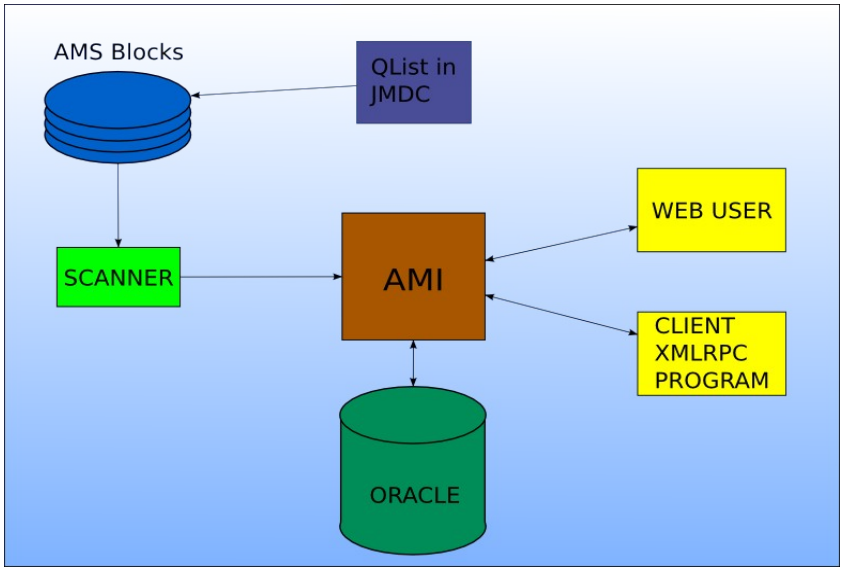
Overview

The AMS Monitoring Interface (AMI) is a set of tools aimed to archive, retrieve and display time based data collected within the so-called AMS Slow Control datastream.
Its main components are:

- AMI – The main framework
- Scanner – It scans data blocks and sends relevant data to the backend storage
- RDBMS (Oracle) – Backend to store data
- AMI Web Interface – Displays stored data on the web
- AMI xmlrpc interface – Gives online data availability for other purposes

The data path

Everything starts from the JMDC: using a set of QList commands running in



background, data from many subsystems like the TTCE, the Tracker, the PDS, etc. are collected and sent to the ground. After being received and deframed, data are organized into files containing AMS Blocks.
The problem we want to solve is how to store, retrieve and display these data in an efficient, scalable and extendable way.

Why AMI?

We could actually browse the data directly on the disk, writing a program that seeks and displays the data in many possible ways; however there are few issues that makes this approach not so convenient:

1. Looking for a particular data source or a particular time is not handy, as data are not ordered at all; to lookup something particular we should scan every time the whole data set or at least a large part of it (several GB for the time being, and it will grow)
2. This kind of browsing can be made only from a computer which has direct access to the disk containing the data
3. Every subsystem group should write its own software just to visualize a temperature or a current, and this makes the approach very unpractical
4. Every subsystem group would need concurrent access to a large part of the dataset, causing overhead and slow access if not correctly managed.

To overcome these problems we have chosen an approach based on a relational database accessed through a web application.
The use of a RDBMS addresses point 1 and 4, while the centralized web access addresses point 2 and 3; moreover, by exploiting the xmlrpc protocol, data can be easily accessed directly from any external analysis software.

The DataBase Stucture

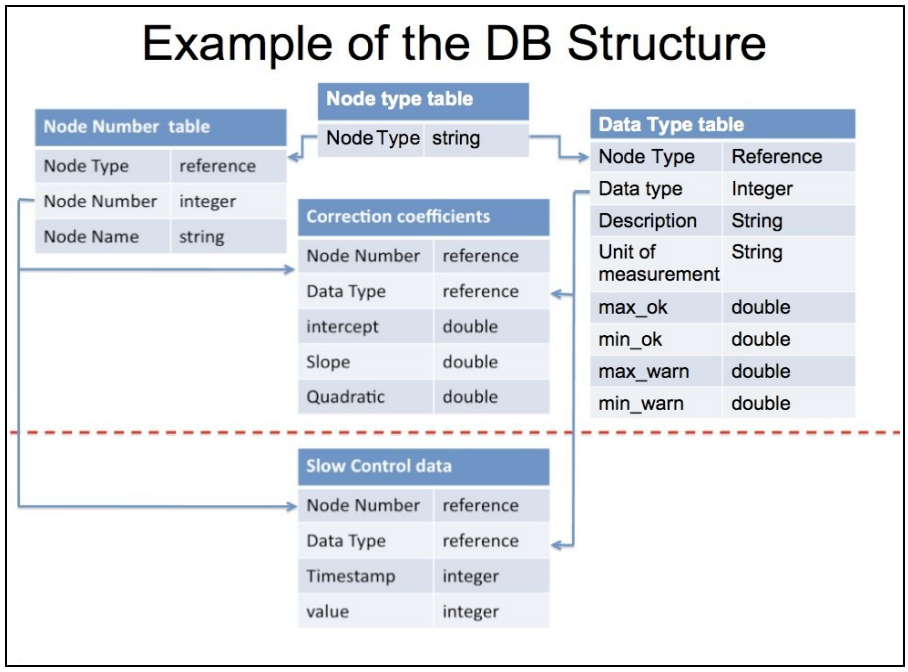
The AMS SlowControl data may be classified basing on the AMS unit (or node) which produces these data and on the command that must be sent to the node to retrieve a given set of data (datatype). The pair (node number, data type) uniquely identifies a set of data with a given data format (or at least this should be the design). The single information can be identified by the triplet node number, data type, cardinal number, where cardinal number is the position of the information in the dataset.
A single AMS SlowControl data part can be represented as the n-tuple:

(node_number, data_type, cardinal_number, timestamp, value)

A DB structure to hold this informations may be:

- A table containing node numbers
- A table containing data types, cardinal numbers with a reference to the node number table
- A table containing the pairs (timestamp, value) with references to the above tables.

In AMS many node numbers have the same functionality, for examples we have 8 Tracker Crates holding 16 JINF-T. All these 16 JINF -T have the same set of data types. It is then useful to introduce the concept of node_type as this will allow to eliminate duplicate entries in the data_type table.
The following scheme give the structure of the implemented database:



Here we internally join in one number data_type and cardinal number and we also introduce a table to store a set of conversion factors on a node_number, data_type basis as many data are produced as raw ADC values.
The tables above the dashed line are static, containing the core data structure; they do not grow with time, and they are updated when the DB is prepared to host a new node type or a new data type. The table below contains instead the real data coming from the SlowControl, and it grows with time. The values are supposed to be stored in an integer form, as they come from the electronics; when it is needed to plot the data, the correction coefficients will be used to obtain the real value. If, for example, we are reading a 12 bit ADC value of a temperature that goes linearly from -30 to 70°C, we will store the ADC raw value into the DB and this value will be automatically converted to a real value according to the equation

$$R = an^2 + bn + c$$

where R is the real value, a,b,c the correction coefficients and n the ADC integer raw value stored into the DB. In the example the values would be

$$a = 0$$
$$b = \frac{100}{4095}$$
$$c = -30$$

Data Collection: the QList

The JMDC collects data from AMS nodes at regular intervals. The data are replies to commands (data types) issued by the JMDC itself, and those commands are organized in a facility called "QList".
In a very easy approach, a Qlist entry is a set of commands associated with a node number; at regular intervals, the JMDC issues the commands to the appropriate node number and sends the replies to the ground within the LRDL and HRDL streams.
In order to monitor a particular node the first step is to add the appropriate command set to the QList, so the replies containing the data can be found on the AMS Blocks files on the ground.

Data on ground: the AMS Block format

After being received and deframed, data are organized into files containing AMS Blocks. The AMS Block format is rather simple: it basically contains informations about the "sender" of these data, the data themselves and their production time.
Each block can be split in 2 different parts, the header and the payload: the header contains the node number that owns the payload, the data type that is being replied, the unix time at which the command has been replied, status bits and few other less interesting data. The payload contains the raw data that are sent by the electronics; they may be very different, depending on the node number and the data type issued: they may contain data about temperatures, currents, voltages or other ADC values.
Standard commands and replies to standard commands are formatted as just described. A Qlist entry instead may contain a set of commands which are sent in sequence to each destination, and the replies to these commands are grouped within a container called envelope; an envelope is basically an AMS Block containing other AMS Blocks.
Here comes the first player of the AMI tools, the Scanner. It scans all the incoming blocks looking for interesting data, and when it finds them it decodes the blocks and put them into the DB through the AMI xmlrpc interface. AMI acts as a middleware between the end users (using either xmlrpc or web) and the DB.
The AMI-Scanner program is designed to cope with both types of replies, simple AMSBlock or envelope, opening the latter and processing the contained blocks if necessary.

How to monitor data with the AMI framework

The main goal kept in mind during the design of AMI was to obtain something easy to extend. Suppose we want to monitor a data type from a node "foo", for instance. The summary of the needed steps in order to make AMI handle it are the following:

- Prepare the DB structure to host "foo" data
  - Add the node_number to the DB (may be also the node_type)
  - Add the data_types to the DB with description, units, ...

    - Add the conversion factors (may by updated at any time)
- Tell the Scanner how to decode the new data:
    - Follow the provided template to write a simple C/C++ function able to understand "foo" AMSBlocks payload

At this point when new data arrive they are stored in the DB and the AMI-web application generates on demand a plot for the new data. However by default it is possible to see only one quantity at the same time.
To have a better monitoring of the interesting data, that is displaying a set of informations within a web page, you must add an extension to the AMS-web application by writing a controller-view pair.
The controller is the data extraction part, the view is the actual display part.
Controller and view must be written in python, but in the AMS-package examples and functions for plotting are provided. Adding a simple controller-view pair is quite easy even if you are python primer.

Tutorial: adding node "foo" to AMI

Summarizing what happens in a more detailed view, when the data are on the disk, the Scanner process scans the data. For every found combination of data type and node number, it looks for a function capable of handling those data: if such a function is available in the Scanner library, the function is called. The function is supposed to decode the data and to store them into the DB. If instead the lookup fails, the Scanner continues processing the next AMS Blocks.
The DB has to be ready to host those data, as they are organized in a relational way. In order to add a new node number/data type then, the following steps are needed:

1. Get the subversion code at svn://ams-farm.pg.infn.it/AMI/scanner
2. Produce a list of the node numbers we want to monitor. An example template of the needed data is shown in Table 1.
3. Produce a list of data types associated with the node numbers above, along with their descriptions; for every ***ordered*** value in the reply of each data type we need also unit of measurement, range values (min and max ranges, min and max alert limits) and the correction coefficients. The ranges are not strictly needed. Order matters because the order defined will be used to fill the DB when arrays of data will be passed to the API. If, for example, one data type reads 2 temperatures T_a and T_b, after the decoding you shall create an array with those temperatures in the same order. An example template of the needed data is shown in Table 2 and Table 3.
4. Produce a code able to understand the data. Use the templates foo_decode.C, foo_decode.h as examples. Basically, this is the function which gets called when the node numbers and data types match yours; as parameters, among other stuff, you get:
    - a pointer to the raw block data (buf)
    - the buffer length (len)
    - the unix time at which the data have been produced (time)
    - the node number (nn)

The prototype of the function cannot be changed, while the name, obviously, must be.

5. Make the Scanner aware of the new function: in functions.h you should add the node types and data types you are interested in, following the commented example. Do not forget to #include your header file at the top of functions.h.
6. Edit the Makefile to make your code compile. There are comments inside the file about how to do that.
7. Compile the Scanner, using "make MODE=test". This enables a fake mode which just scans data and prints out what would be added to the database, but it does not touch the DB. This is useful to debug your code. To run it, use something like

   ```
   ./main -B <blocks directory> -f <blocks file>
   ```

   for example

   ```
   ./main -B /Data/BLOCKS/ISS1553/0088/ -f 001
   ```

For now, what you produce in step 2 and 3 has to be given to me[1] to be added to the DB; the other steps can be performed entirely on your own, but when you are happy with your code you should produce a patch and send me (or just send the whole code) so I can add your stuff to the real repository and to the running Scanner.
The next steps are optional, as AMI contains a simple template for every data type inserted into the DB structure, and its already able to display graph of them. However, it is strongly advised to build one or more dedicated web pages to display more informations in the same place.

Tutorial: writing a simple view for your data

Once the data are on the database, as previously stated, there is an automatic template system able to display every data type. However, it is strongly advised to build one or more dedicated web pages more suitable to monitor the subsystem we are interested in.
A web page is built dynamically by AMI using a page layout template, called **view**, and a kind of an engine called **controller** able to feed data into the view. AMI is designed to make the life easy for who needs just a simple table of graphs in one web page: there is a generic view ready to be used, very basic. If you need anything more complex, it shouldn't be hard to write another one using that as a template. In order to build such templates pages, one needs to extend the AMI internals: here it is shown the simplest case, as to build something more complex one should get much deeper into the framework code and understand more of it. I assume if you want to go that way you are smart enough to take a look at it by yourself; for the tutorial we will stick to the easy way.
So, as the view is ready, you just need to write a simple controller, in python language, similar to the one in the next page:

1 Drop an email to gabriele.alberti@pg.infn.it

```
def foo_data_types():
    # set the view to be used
    response.view = 'default/generic_view.html'
    # this is the node type name
    nt = 'Foo'
    # node names and node numbers we are interested in
    devices = [['Foo-A','0x10'],['Foo-B','0x11']]
    # process the time form
    try: side = request.args[0]
    # this is the default node number to show
    except IndexError: side = devices[0][1]
    try: leng = request.args[1]
    except IndexError: leng = '3_hours'
    form = __process_form(leng)
    # get data types i want to display
    dt1,sub1 =
      __get_data_types_by_description('Foo data type 1')[0]
    dt2,sub2 =
      __get_data_types_by_description('Foo data type 2')[0]
    # generate png with a table layout 1 row 2 columns
    P = [[
      __prepare_graph(nt,dt1,sub1,side,leng,w='300',h='100'),
      __prepare_graph(nt,dt2,sub2,side,leng,w='300',h='100')
    ]]
    # pass everything to the view
    return dict( t='FOO Data types', pngtable=p, form=form,
        d=devices, showing={'side':side,'l':leng} )
```

having marked in light blue the parts that need to be changed. In this example, after the initial part to prepare framework variables, we get the data types wanted by their descriptions, in this case just 2 data types; then, we prepare 2 images, one for each data type, of width 300 and height 100 each.
P is an array of array elements; every array is a raw in the table, in this case we have just one array of 2 images, so we get a table 2 columns 1 row.
The resulting view should be added to the framework by me.

AMI xmlrpc interface API

In this section it is summarized the AMI xmlrpc API. This is just a reference, and it can be used by who wants to get the raw data from the database and use them for their own stuff. The server URL to be used is

`http://ams.cern.ch:8081/AMI/show/call/xmlrpc`

Functions:

1) `get_node_types()`
   - Arguments: none
   - Return value: an array of strings, containing the node types stored into the DB.
2) `get_node_numbers(<string>)`

- Arguments: the node type as returned by 1)
- Return value: an array of 2 elements arrays. Every sub-array are 2 strings describing:
  - node number name
  - node number in hex

`3) get_node_numbers_by_node_name(<string>)`

- Arguments: the node name as returned by 1)
- Return value: an array of 2 elements arrays. Every sub-array are 2 strings describing:
  - node number name
  - node number in hex

`4) get_data_types(<string>)`

- Arguments: the node type as returned by 1)
- Return value: an array of 8 elements arrays. Every sub-array are 8 strings describing:
  - data type in hex
  - data type description
  - unit of measurement
  - maximum value
  - minimum value
  - maximum alert value
  - minimum alert value
  - subtype

`5) get_data_types_by_description(<string>)`

- Arguments: the data type description as returned by 4)
- Return value: an array of 2 elements array. Every sub-array are 2 strings describing:
  - data type in hex
  - subtype

`6) get_data_types_by_description_and_node_type(<string>,<string>)`

- Arguments: the data type description as returned by 4) and the node name as returned by 2)
- Return value: an array of 2 elements array. Every sub-array are 2 strings describing:
  - data type in hex
  - subtype

`7) get_vals(<string>,<string>,<string>,<string>,<string>)`

- Arguments:
  - data type in hex
  - subtype
  - node number in hex
  - start time, in unix time (seconds since epoch)
  - stop time, in unix time (seconds since epoch)
- Return value: an array of 2 elements array. Every sub-array are 2 strings describing:

    - unix time of the following datum
    - the raw datum as stored into the DB

8) `get_daq_coeff(<string>,<string>,<string>)`

- Arguments:
    - data type in hex
    - subtype
    - node number in hex
- Return value: an array of 3 elements. Every element is a float describing:
    - the *a* correction coefficient
    - the *b* correction coefficient
    - the *c* correction coefficient

9) `get_real_vals(<string>,<string>,<string>,<string>,<string>)`

- Arguments:
    - data type in hex
    - subtype
    - node number in hex
    - start time, in unix time (seconds since epoch)
    - stop time, in unix time (seconds since epoch)
- Return value: an array of 2 elements array. Every sub-array are 2 strings describing:
    - unix time of the following datum
    - the real datum converted with the coefficients

Templates tables

Note: the following tables are made with fictitious data; however, they try to be coherent with themselves. The tables can be supplied to me in xls, ods or csv formats.

Node Number	Node Name	Node Type
0x1c	USCM-CCEB-A	CCEB
0x1d	USCM-CCEB-B	CCEB
0x6c	TTCE-A	TTCE
0x6d	TTCE-B	TTCE

Table 1: Example of Node Numbers table

NT	DT	Description	Unit	Min OK	Max OK	Min W	Max W	Subtype
CCEB	0x1c	CC1 Temp	K	275	294	270	295	0
CCEB	0x1c	CC2 Temp	K	275	292	270	293	1
CCEB	0x1c	Bx	T	0,7	0,9	0,68	0,93	2
CCEB	0x1b	By	T	0,12	0,2	0,1	0,25	1

Table 2: Example of Data Type (DT) table

NNum	DT	Subtype	a	b	c
0x1c	0x1c	0	0	1,34	2,34
0x1d	0x1c	0	0	3,56	3
0x1c	0x1c	1	0	1	0
0x1d	0x1c	1	0	1,2	-6

Table 3: Example of Coefficients table