AMI: AMS Monitoring Interface

AMI: AMS Monitoring Interface

Gabriele Alberti[1,2], **Paolo Zuccon**[1]
[1]INFN Sez. Perugia, via Pascoli, 06100 Perugia, Italy
[2]DMI, Università degli Studi di Perugia, via Pascoli, 06100 Perugia, Italy
E-mail: `gabriele.alberti@pg.infn.it, paolo.zuccon@pg.infn.it`

Abstract. The problem of an easy access to the AMS-02 Slow Control data, is considered and resolved by the design, the implementation and the deployment of a web application with an RDBMS backend. This application, built on top the *Web2py* framework, allows for an easy browsing of the slow control data even from remote sites.

1. Introduction on AMS-02

The Alpha Magnetic Spectrometer -02[1] is a particle physics experiment devoted to the precise measurement of the cosmic ray fluxes from a low Earth orbit; it will be installed on the International Space Station and it is supposed to take data for more than ten years.

The principal physics goals of AMS-02 are:

- the search for primordial anti-matter by the observation of anti-nuclei with $Z \geq 2$, the observation of anti-helium nuclei would imply the existence of anti-matter domains, the observation of anti-carbon nuclei would imply the existence of anti-matter stars.
- the search for Dark Matter signatures by observing peculiar spectral features on the rare components of the cosmic rays fluxes as the positrons and the anti-protons.
- the precise measurement of the nuclei fluxes up to Iron in order to constrain the cosmic rays production and propagation models.

To achieve its goal AMS is equipped with a permanent magnet proving a field of $\sim 0.15T$ confined within a cylindrical volume and with a negligible magnetic moment. A silicon tracker, composed by nine layer of high resolution detectors ($10\mu m$ in the bending plane) placed within and outside the magnet bore, provides an accurate measurement of the particle trajectory and hence of its rigidity.

A set of scintillator layers placed above and below the magnet bore determines the direction of the particle along its trajectory and measures the Time of Flight (TOF).

A Ring Image Cherenkov (RICH) detector allows for a precise velocity measurement, in particular for higher charge nuclei and allows for the isotope separation.

A Transition Raditation Detector (TRD) on top and an Electromagnetic Calorimeter (ECAL) on the bottom allow for an efficient proton electron separation and for an accurate electron energy measurement.

By means of the energy released in the silicon and in the plastic scintillators and from the number of photons emitted as Cherenkov radiation, a redundant measurement of the particle charge up to Iron is done.

The AMS subdetectors and the other AMS subsystems devoted to guarantee the correct working in the hostile space environment are producing a large amount of health and status data, such as temperatures, currents, voltages, pressures and status words. The temperature sensors alone are more than 300 and counting also the other measured parameters we reach a number of the order 2000.

A tool designed to manage and display these data is the subject of this work.

2. AMS Monitoring

In AMS each subdetector or subsystem has various sensors and probes designed to monitor the health and the status of the system, as temperature, voltage or pressure.

Most of these quantities need to be constantly monitored to be sure the system is working properly and to detect a possible hardware failure early.

In a normal ground environment the common approach would be to write one or more monitor programs polling the hardware at the desired frequency and displaying the acquired data.

This is not the case for a space experiment as AMS where the commanding or uplink channel is not always available or may suffer of a very limited bandwidth. Furthermore the uplink and the downlink channels may be available in different periods.

The very limited uplink bandwidth is carefully managed by NASA and usually the "commanding windows" have to be used to instruct AMS on how to perform during the next uplink unavailability.

In AMS-02, the on-board main computer (J Main Digital Computer) polls all the subsystems regularly about their status and sends the replies to the ground in the so-called *housekeeping data stream* whenever the downlink is available.

2.1. Data organization

AMS commands and data are expressed in a format called AMSBlock. The anatomy of an AMSBlock can be described as a header and a payload, for what concerns the replies the header contains few words: the packet length, the address of the replying subsystem (node number), the code of the command which originated the reply (data type), a time stamp and some handshaking control bits. The payloads instead contains the data in a format that depends on the node number and on the datatype.

The *housekeeping data stream* is made by a sequence of AMSBlocks, each coming from a different part of the system and containing informations about that part of the system.

After a quite complicate path through the NASA interfaces and protocols, the AMSBlocks are eventually written to a disk accessible by the AMS-02 users, to be processed by the monitor programs and, possibly, the offline software.

On disk the AMSBlock are organized on files containing all the AMSBlocks that arrived to ground within one minute. When 1-minute file is ready is then sent from NASA to the AMS control room (Payload Operations Control Center); there it is unpacked and made available to the monitoring programs. As a consequence the most fresh data to be used as real time monitor data are at least one minute late with respect to the moment the measure occurred.

In order to get the data from a particular subsystem the most straightforward approach is to scan the whole *housekeeping data stream* as it is made available on disk, extract the required data and display them.

The system in facts doesn't guarantee any ordering on the packets within the stream both because they are produced by asynchronous requests in the JMDC and because the communication channel does not guarantee that packets are delivered in sequence.

This approach however is very resource consuming as independent programs need to scan the same stream and if an user needs, for instance, a particular temperature of a subdetector in a determined time window, he has to seek through (almost) the whole stream.

The idea to develop the AMS Monitoring Interface (AMI) comes from the need to find a more practical approach to solve this problem.

3. AMI: AMS Monitoring Interface

Keeping in mind how the AMS data flow works and considering that every AMSBlock contains unique time based informations, the idea to use an RDBMS appeared as the most promising.

A well designed DB structure allows to seek through the whole dataset with a simple query, letting the DB handle all the searching for desired information. Having a RDBMS as backend allows to implement a lot of valuable features, such as a graphic output for the various data, long term studies of various parameters, or correlation among several measured data. The latter in particular needs random access to the data, operation that would be hard with the one minute file approach.

Another request we want to fulfill is to make the AMS Slow Control data available to as much users as possible. The first yet most effective development has been a web application to show those data on a browser.
This approach has many advantages, particularly:

- It allows for multiple access with a single application
- There is no need to run a specific program on the client
- A web server with an RDBMS backend has been proven to allow very large scalability
- There is a huge amount of examples and a proven possibility to implement advanced features

The AMI conceptual design is a web application as frontend to an RDBMS combined with a plotting facility

3.1. The data base structure

The first step in the implementation, has been the design of the data base structure. We have chosen a solution that exploits the structure of the AMS-02 subsystems.

From the Slow Control data point of view AMS-02 can be seen as a set of objects (nodes) producing various data. As a consequence of its modular and redundant design, AMS-02 presents sets of nodes of the same kind. These nodes share the same *node_type* code, but each one has its own unique *node_number.*

Nodes with the same *node_type*, share the same set of commands (*data_type*) and replies. The reply to a single command (*data_type*) may contain multiple quantities, as for example the temperatures measured in a set of different points. To properly treat these feature we introduced the *extended_data_type* implemented as a 32 bit integer. The 16 least significance bits contains the *data_type* and the upper 16 most significant bits identify the single datum within the reply. The pair *node_type, extended_data_type* represent then unambiguously a data kind.

The structure of the database is easily made: the main table contains the list of *node_type*s, then there is a table of the *extended_data_type*s with the reference to a given *node_type.* A separate table lists the *node_number*s with a reference to the corresponding *node_type.*

For each *extended_data_type* we store: a string describing the quantity, a string with the measurement unit and the warning and the alarm limits.

The actual data are stored on a separate table containing: the timestamp, the datum, a reference to the *node_number* and a reference to the *extended_data_type.*

The provided data are usually expressed as ADC values, a conversion function is then needed to convert the ADC number to meaningful values in a given unit. We then introduced a separate table, with reference to the *node_number* and to the *extended_data_type*, that contains the coefficients of a polynomial correction function.

The structure of the data base is depicted in figure 1. The database primary keys are not shown, for each table a sequential row counter is used as primary key.

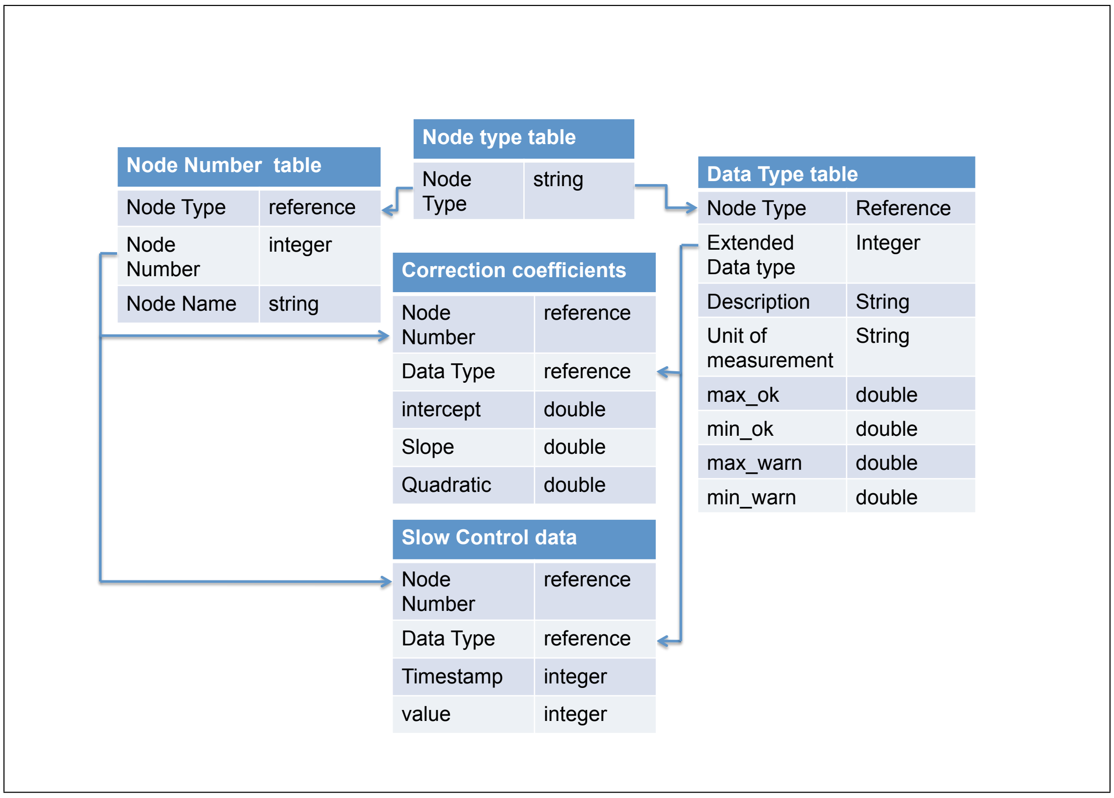

Figure 1. *Simplified DB layout. Primary keys (not shown) are sequential index counters on each table.*

3.2. Tools: Python, Web2py, rrdtool

Choosing the tools to design our web application required a careful evaluation; many different toolkits are available and most of them have been proven to be really flexible and performing.

The first requirement has been to have Python as programming language. This choice is motivated mostly by the features of this language as the lightweight, the fast prototyping and the large amount of available libraries. Our previous experience in programing in Python was also a crucial factor since it drastically reduced the time of development. We also required for the framework to be available as free software.

Among many others, we considered: *Turbogears*, *Django*, *Zope*, *Cherrypy*, and *Web2py*. A the end our choice has been *Web2py*[1].
Web2py is a simple tool written by a single developer, and it has several advantages:

- It is very small, as its footprint on disk is $\sim$ 10 MB,
- It uses the MVC (Model View Controller) paradigm,
- It comes with a powerful webserver usable in production, making very easy the deployment (it's still possible to use a fully featured Apache HTTP server though)
- It has a simple administrative interface to access the database
- It has an integrated authentication library
- It provides an interface to implement web services (JSON, XMLRPC) writing only few lines of code

[1] http://www.web2py.com

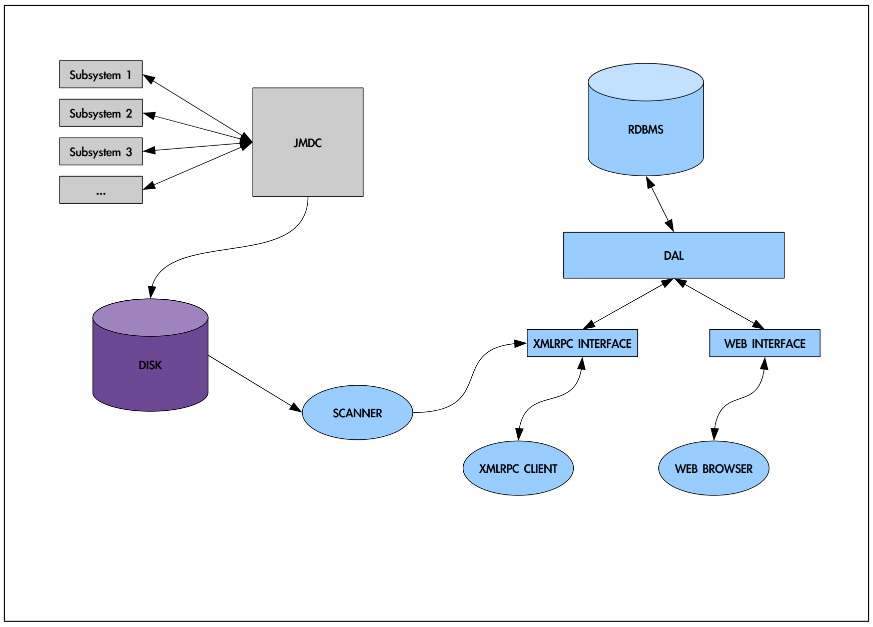

Figure 2. *AMI architecture*

- Even if its community is rather small it seems to be very reactive to help solving problems and issues

After choosing the framework, we had to choose a tool to easily generate plots. As our primary need is to plot quantities over time, we have chosen *rrdtool*[2], a well known tool of proven robustness, that is also free and open source.

3.3. Architectural overview

AMI is composed of different parts, as shown in Figure 2:

- RDBMS: The data base backend
- DAL: The *Web2py* Database Abstraction Layer that holds the database structure description
- AMI web interface: It allows to distribute the plots of the quantities to be monitored
- AMI XMLRPC interface: It allows for remote programs to interact with the database
- The Scanner: The utility that scans the *housekeeping data stream* and fills the database via the XMLRPC interface

The RDBMS backend is managed by the web2py's DAL, the Database Abstraction Layer. It offers an API capable of offering database access programmatically, with no need to write SQL code. With it, we have a homogeneous way to access the data throughout the application, and we don't need to take care of connecting or authenticating to the database. As already told, web2py uses the MVC paradigm: the data are represented virtually by a Model, that is a set of class-like interfaces which describe how the database should be organized and offers an access to it (DAL). The View gets the data from the Model and renders them in a suitable way for interaction, in our case a webpage containing forms and plots. The Controller is the layer in charge of taking input from the user, requesting or modifying data in the Model and preparing a response to be rendered by the View.

[2] http://www.mrtg.org/rrdtool/

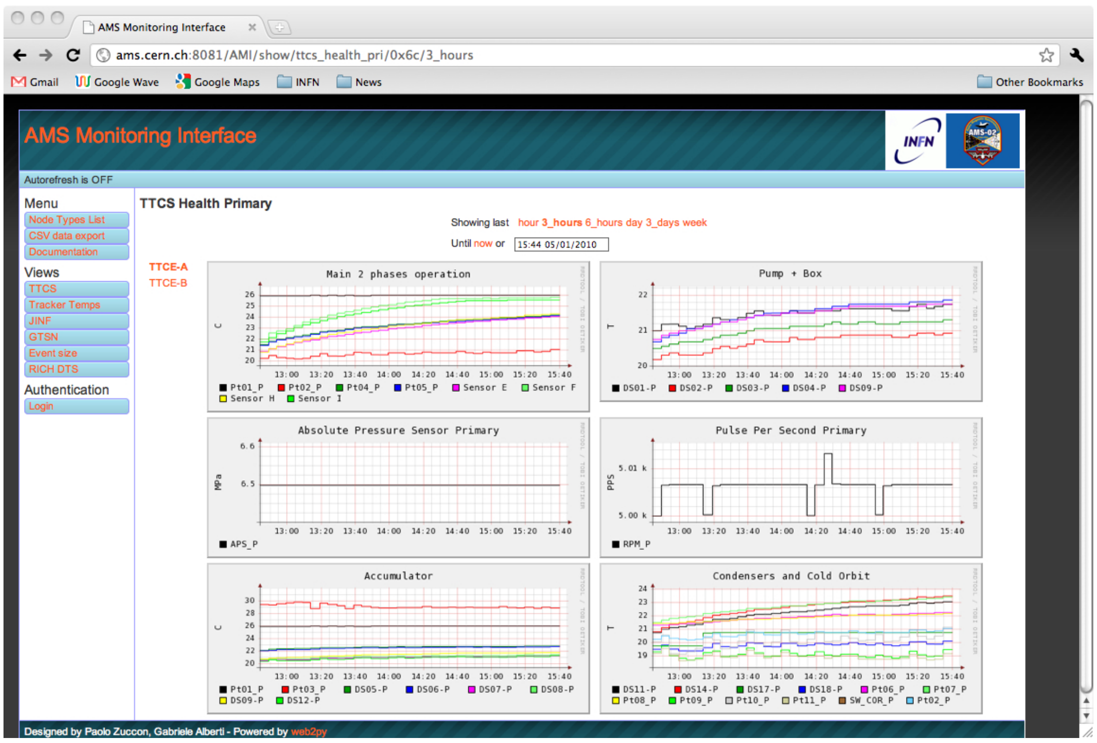

Figure 3. *Example of a "custom view"*

3.3.1. Web interface The main goal to achieve when we have decided to develop AMI was an easy way to browse the data: a web interface. Using the web allows to have the data available worldwide and it allows the delivery on an ever growing quantity of devices, either computers, smartphones or, recently, tablets.

From the web interface it is possible to browse, by name, the *extended_data_types* stored in the database for a given node. It is then possible to select a time window to be displayed, either up to the current time or up to a timestamp selected by the user. A custom python code within *Web2py* extracts the requested data from the database, uses rrdtool to generate a plot of the data as a png picture file and makes it available to the client.

In this way is possible to generate on request a plot for every data type stored in AMI.

Another possibility is to plot sets of data types that need to be shown in one page to have a feeling of what is going on. Those are the dedicated views which gather together more data types in one or more plots and show them in the same page. Using the DAL interface we have developed an internal set of functions to access the data very easily, and to code a "custom view" it is just matter of few lines of code. An example of such view is shown in Figure 3.

3.3.2. XMLRPC interface Web2py allows very easily to build other kind of views, among them particularly interesting for us is the XMLRPC interface. Through this interface it is possible to call from an external application (XMLRPC Client) several pieces of code (procedures) defined and running in the server. In our case, it is possible to insert new data into the RDBMS or request data already stored.

3.3.3. The Scanner The JMDC periodically collects the data from all the subdetectors and sends them to the ground. Once they reach our ground recording system, the data are written to the disk as one minute files containing AMSBlocks, ready to be analyzed. The first step needed in order to make AMI work is to scan these files looking for the data we want to be stored in the database. The scanner is a small, easy to extend C++ application able to parse the data, look up for a function capable to decode them and use the XMLRPC interface to fill the database. A python script takes care of launching the application as new files become available.

4. Deployement and optimizations

The full AMI chain of the scanner, database and web application has been deployed on various environments.

The development environment has been a single machine using SQLite as a backend.

The first production deployment that held the data of a single subsystem, had a single machine running the Web2py server and a MySQL instance.

The current production deployment, holding the data of many AMS-02 subsystem, runs on a high availability system with Oracle 11g as backend.

During the development several problems have been found and solved. The first problem arose from the choice to have a single table to store all the pairs of timestamp and value. As the table grew the database became slower and slower, to give an example the views displaying one day of data taking were taking up to 60 seconds to be produced and displayed. The problem has been solved by creating a timestamp-value table for each *extended_data_type* so the search has been broken into finding the right table and searching the wanted timestamp through it.

By side effect, this produced a very large number of tables that was also an issue since *Web2py* by default checks for the existence of all the tables at every connection. We however modified our application to customize the default *Web2py* behavior and have it to avoid unnecessary checks.

We also noted that concurrent access of multiple users was also degrading the performance of our web application. Given that, in our case, most of the user are interested to look at the most recent data, typically we have many users performing the same queries. To make the page loading faster, we then implemented a caching system for the plots. Before this optimization, at every connection AMI built a new plot, that is a png image. This involves fetching the data from DB, organizing them, creating the rrdtool database, and finally creating the png image file. Having a lot of clients asking for the same data (the latest) was unpractical. If the plot for the data requested already exists, the cached copy is served to the client. This has been a significant improvement on AMI speed, with the loading time to display a 6 hour data window passing from an average of 2.5 s to 0.7 s. Furthermore given the typical pattern of access corresponding to users looking at the last few hours of data with an update period of 3 min, the first user asking for new data will miss the cache while all the others will hit it. So we have an missed cache rate of $\sim$ 3 min.

We have also to consider that, at the moment of the implementation of the caching system, the typical number of concurrent users was 4, while in the latest production environment this number has grown to 15. We expect a increase of this number as the experiment will be deployed to the International Space Station.

AMI is currently used to monitor several subsystems of AMS-02, it allows for experts that built that subsystem to remotely monitor it.

5. Improvements and future plans

Several options are considered to improve the AMI performances and features:

- Give alternatives to rrdtool for the plotting feature. We are considering tools, such as matplotlib[3], that could also produce different plots like correlations or 3D representations.

[3] http://matplotlib.sourceforge.net/

- Introduce a custom view editor that allows the users to build and organize their own views directly from a web interface.
- Try to use NoSQL as backend instead of a traditional RDBMS. The use of NoSQL could give better performance as the number of stored variable increases and would allow also for an easier multi site deployment. However this step is very optional, and we would probably need to go deep into *Web2py* to see if it is viable.

6. Conclusions
We designed, implemented and deployed a web application that allows for an easy retrieval and display of the AMS-02 slow control data. The application is routinely used to monitor the health and status of various AMS-02 subsystems and allows for remote experts to quickly diagnose the source of possible anomalies.

References
[1] Zuccon P 2008 *NIM* A **596** 74