Automatizing the Online Filter Test Management for a General-Purpose Particle Detector

Rodrigo Coura Torres[a,*], Andre Rabello dos Anjos[b], José Manoel de Seixas[a], Igor Soloviev[c]

[a] *Federal University of Rio de Janeiro, PO Box: 68504, 21941-972, Brazil*
[b] *University of Wisconsin, Madison, USA*
[c] *Petersburg Nuclear Physics Institute, Russia*

Abstract

This paper presents a software environment to automatically configure and run online triggering and dataflow farms for the ATLAS experiment at the Large Hadron Collider (LHC). It provides support for a broad set of users, with distinct knowledge about the online triggering system, ranging from casual testers to final system deployers. This level of automatization improves the overall ATLAS TDAQ work flow for software and hardware tests and speeds-up system modifications and deployment.

Keywords: Online triggering systems, automatic configuration, automatic execution, system integration, particle detectors.

1. Introduction

The Large Hadron Collider (LHC) [1], at CERN, is the world's newest and most powerful tool for particle physics research. It is designed to collide 14 TeV proton beams at an unprecedented luminosity of 10^{34} cm^{-2} s^{-1}. It can also collide heavy (Pb) ions with an energy of 2.8 TeV per nucleon at a peak luminosity of 10^{27} cm^{-2} s^{-1}. During nominal operation, LHC will provide collisions at a 40 MHz rate. To analyze the sub-products resulting from collisions, state-of-art detectors are placed around the LHC collision points.

Among LHC detectors, ATLAS (A Toroidal LHC Apparatus) [2] is the largest. ATLAS comprises multiple, fine-granularity sub-detectors for tracking, calorimetry and muon detection (see Fig. 1). Due to ATLAS high-resolution sub-detectors, $\sim$ 1.5 MB of information is expected per recorded event. Considering the machine clock, a data rate of $\sim$ 60 TBytes/sec will be produced, mostly consisting of known physics processes. To achieve a sizeable reduction in the recorded event rate, an online triggering system, responsible for retrieving only the most interesting physics channels, is put in place. This system is constructed using a three-level architecture [3]. The first-level trigger performs coarse granularity analysis using custom hardware. The next 2 levels (High Level Triggers - HLT) operate using full detector granularity and use complex data analysis techniques to further reduce the event rate [4].

The HLT software is developed using high-level programming languages and is executed using off-the-shelf workstations running Linux and interconnected through Gigabit Ethernet. Data are provided to HLT by specialized *Data Acquisition* (DAQ) modules known as Readout Systems (ROS). Both HLT and DAQ are referred to as *Trigger and Data Acquisition* (TDAQ). The TDAQ comprises thousands of devices, and has been designed to be highly configurable. To provide the run configuration of all TDAQ devices, a special database system was developed [5].

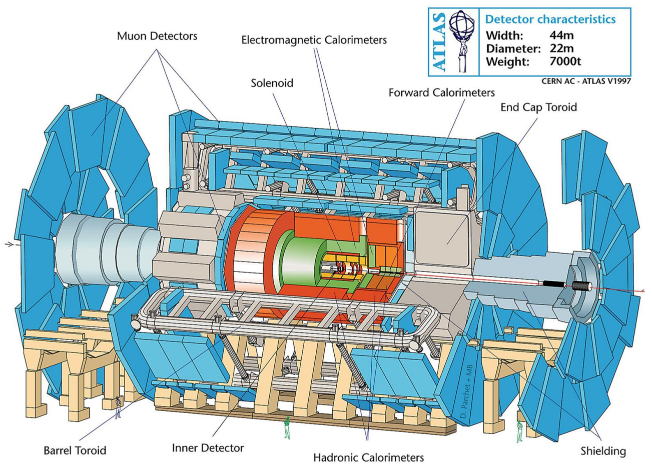


Figure 1: The ATLAS detector.

While there is a single production version of the TDAQ running at the ATLAS experiment, the TDAQ software itself can be run on many different testbeds for testing and evolution [6, 7]. Setting device and application parameters for a given TDAQ run is a complex task that requires very specialized knowledge. Misconfigurations may result in TDAQ operation errors, unnecessarily wasting project resources. Additionally, the TDAQ project involves more than 400 collaborators, with different testing interests. There are users who simply want to execute TDAQ with standard parameters, while developers might want to test special cases, requiring more specific adjustments.

To help all TDAQ users and final deployers, a scriptable environment was designed. Such environment was developed based

[*] Corresponding author

Email addresses: `torres@lps.ufrj.br` (Rodrigo Coura Torres), `Andre.dos.Anjos@cern.ch` (Andre Rabello dos Anjos), `seixas@lps.ufrj.br` (José Manoel de Seixas), `Igor.Soloviev@cern.ch` (Igor Soloviev)

on a multi-layer approach. Users with limited knowledge regarding TDAQ will feel more comfortable working at the highest layer, as it will demand only a few high-level parameters. On the other hand, experts might profit from the lower-level layers, since they allow the configuration of very specific parameters, for customized TDAQ operation.

Although the proposed environment automatizes the TDAQ configuration task, its execution still needs to be manually started, and running parameters (accept rate, number of processing cores running, etc) observed from the screen. This can be a tedious task, as tests of new software releases or hardware upgrades must be performed on a regular basis for different running setups. Therefore, an automated TDAQ execution environment was also developed. Such an environment is capable of automatically setting up TDAQ, as well as, for a given amount of time, collect monitoring information about its execution. At the end, post-processing analysis can take place, and results (graphics and histograms) be generated for further analysis.

The remaining of this paper is organized as follows. Sec. 2 details the trigger and data acquisition modules. Sec. 3 describes the configuration database service used to provide configuration parameters to the TDAQ infrastructure. In Sec. 4, the developed environment for configuring the TDAQ infrastructure is shown. In sequence, Sec. 5 shows the environment created for automatically operate the TDAQ system. Conclusions are derived in Sec. 6.

2. Online Triggering System

As already mentioned, ATLAS requires an online triggering system for coping with the huge stream of data generated by LHC collisions. Fig. 2 shows such triggering system in details.

The first level (L1) has a latency time of only ~ 2.5 μs per event. To achieve the required processing speed, this level operates only on calorimeter and fast muon detection information. The L1 system will reduce the event rate to ~ 75 kHz. Also, this level is responsible for selecting the detector regions effectively excited by a given selected event. These regions are called *Regions of Interest* (RoI) and will be used by the second level (L2) [8].

The L2 will receive the RoI information and will validate the first-level decision using full detector information with fine-grained granularity. This level makes use of complex algorithms, which are implemented in high-level programming language (C++). For providing an average processing time of 40 *ms* per event, a set of ~ 500 personal computers (PC) is used for this level. At L2 output, the event rate will be reduced to ~ 2 kHz [3].

The third and last level, also known as *Event Filter* (EF), will make the final decision, using all the information available for a given event. Very complex algorithms are used within this level, requiring a set of ~ 1,900 PCs for providing an average processing time of 4 sec per event. Data from events approved by this last level (~ 200 Hz) are stored in data servers, for further offline analysis [9].

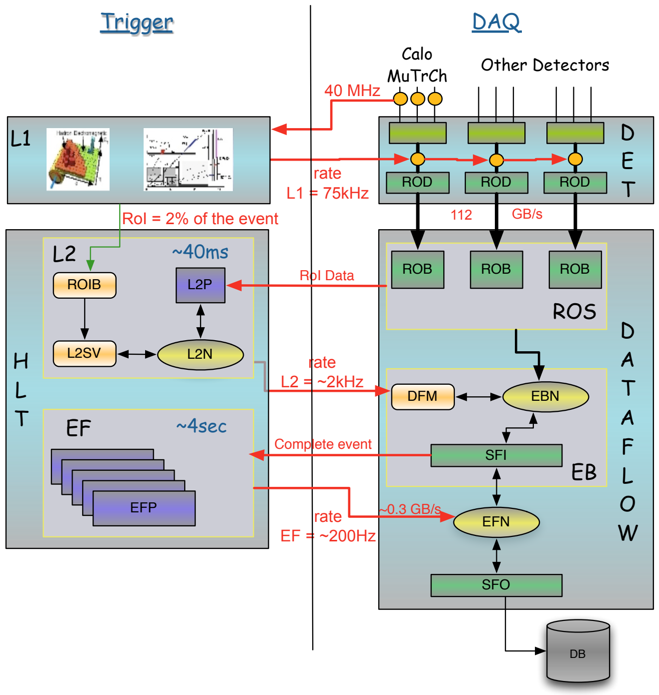


Figure 2: The online trigger system.

After L1 execution, approved events are sent from the detectors, via the *Read-out Drivers* (ROD), to the *Read-out Buffers* (ROB), within the *Read-out System* (ROS), to be available for HLT analysis. Also, L1 sends to the *RoI Builder* (RoIB) the information about all RoI found from a given collision. The RoIB selects a *L2 Supervisor* (L2SV) and sends the RoI information to it. The L2SV selects an available *L2 Processing Unit* (L2PU), running on a *L2 Processor* (L2P), for processing the incoming RoI. The selected L2PU executes the L2 processing chain on the incoming event, requesting RoI data (cell energy, tracking information, etc) to the ROS as needed. At the end, the L2 result is sent back to the L2SV, which then propagates the result to the *Data Flow Manager* (DFM). If an event did not pass L2, the DFM sends a message to the ROS requesting its deletion. Otherwise, the *Event Building* (EB) is started with the event being passed to a *Subfarm Input* (SFI) node, which will retrieve the full event information from the ROS and assemble it as a single formatted data structure, propagating, upon request, the built event to one of the *Event Filter Processors* (EFP) for the final filtering step. The events approved by this level are finally propagated to the *Subfarm Output* (SFO), which is responsible for storing the approved events in data servers for further offline analysis.

TDAQ data traffic is handled by Gigabit Ethernet connections between the TDAQ modules. Four networks are available for improved bandwidth management. The control network is responsible for the configuration, control and monitoring information traffic. For event data, the L2N, EBN and EFN handle, respectively, the L2, Event Builder and Event Filter data traffic.

Table 1 shows the amount of processors needed for each

Table 1: Number of needed processors for each TDAQ module.

Module	N^o Processors
ROS	~ 150
RoIB	1
L2SV	12
L2PU	~ 500
DFM	12
SFI	~ 100
EFP	~ 1,900
SFO	~ 5
Online	20
Monitoring	32
File Servers	~ 80

TDAQ module [3], in order to sustain the envisaged data throughput, considering machines with 4 processing cores, with a 3 GHz clock each.

The TDAQ was built using, as much as possible, commercially available parts and technology [10] for simplified installation and part replacement. All nodes are server PC types, and the connection between them is implemented using Gigabit Ethernet technology. The running applications were developed using objected oriented software engineering, and implemented in C++.

3. Configuration Database Service

The TDAQ description given in Sec. 2 makes clear the complexity of its nature. Besides, TDAQ was designed to be highly configurable, aiming at supporting a broad range of running purposes. As a result, in order to run a given TDAQ setup, one must provide the running configuration parameters for each TDAQ module. These configuration values (queue sizes, execution nodes, communication protocol, etc) must be supplied, during TDAQ setup, for thousands of different applications. Therefore, there is a clear claim for a database service which will be responsible for storing the configuration parameters, as well as providing them for all TDAQ applications.

The *Object Kernel Support* (OKS) [5] was created aiming at providing the configuration services for TDAQ. It is used to provide the overall description of the DAQ system and partial description of the trigger and detector software and hardware. This description covers either high-level control configuration (which parts of TDAQ will be enabled in a given run, when a given process should be started and on which nodes, monitoring configuration, network setup, etc) as well as module specific parameters (queue sizes, timeout values, etc). Due to the large number of modules involved, each group working on a given module (L2, EF, etc) is responsible for preparing their own configuration description using the configuration service tools available. During TDAQ execution, the configuration parameters are accessed by thousands of modules during TDAQ boot and setup. Also, OKS is responsible for notifying the modules about any change performed to the configuration database during TDAQ execution phase. Finally, OKS is also capable of archiving the configuration database used during TDAQ execution for future reference and analysis. Stringent requirements regarding data model, data access programming interface, performance and scalability, among others, make the use of commercially available relational databases almost impossible [11].

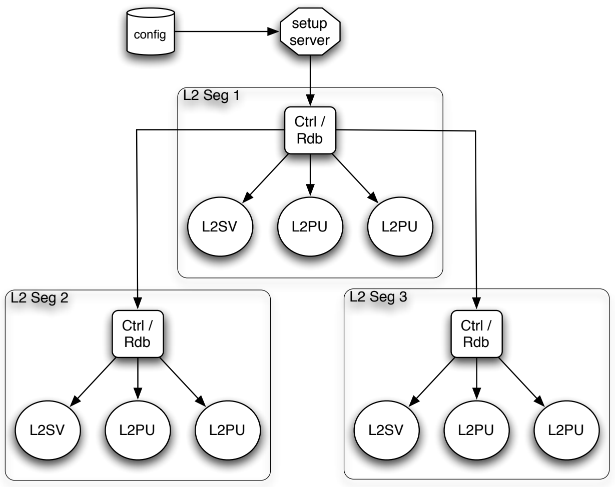


Figure 3: OKS hierarchical control strategy.

Data modeling in OKS is implemented as a class, in a similar way as found in object oriented programming [12]. For the OKS case, each class is described (inheritance, attributes, etc) in a schema file, which is used when a configuration database must be created or accessed. Additionally, OKS provides access libraries for creating or accessing a configuration database for a given TDAQ run. This is accomplished by instantiating the classes defined within the schema file and setting their attributes and relationships. The final configuration database is read during TDAQ configuration phase by control applications that will startup the TDAQ applications on their respective nodes, as described in the configuration database, and provide the configuration parameters upon application request.

For ensured scalability, TDAQ applications can be logically grouped into segments. Each segment contains a control application, responsible for managing the applications within its segment, as well as a *Remote Database Server* (RDB), which is used to access the configuration database from different clients, as well as caching the results of OKS queries. Multiple segments can be created and hierarchically organized, in order to reduce the work load for both control and RDB applications.

Fig. 3 shows an example of such hierarchical approach for a multi-level L2 segment. The control application running in the first segment manages only the L2 applications within its segment, as well as the control applications within the children segments. Also, each RDB server is seen only by the applications in its segment, as well as the RDB servers in the child segments. Therefore, by increasing the number of segments in a tree-like manner, the number of TDAQ applications can grow without creating configuration / start up bottleneck issues.

For configuration database creation and access, OKS provides API in C++ and Java, so that the database persistency implementation, and its physical location (local or remote) are kept hidden from the final user. Nevertheless, the configuration of each application, as well as their organization into segments, are still under the user responsibility, which constantly represents a tedious and delicate task, requiring expert knowledge, not always available.

4. Automatic Database Configuration

When a TDAQ execution is envisaged, several considerations must be analyzed prior to the generation of the configuration file, for instance:

- Which TDAQ sections (L2, EB, EF) to enable? If EF is desired, for instance, EB must be present as well.
- Which modules per section should be used? If data are coming from the detectors, then the RoIB must be enabled. If Monte Carlo simulation is being used, the RoIB may be disabled.
- How many applications of each module should be used? It is an inconsistency to have more L2SV than L2PU, or more than one DFM, etc.
- How should each application be setup? The number of working threads must be correlated to the number of available cores in the node being used for a given application. The timeout value for a ROS response after a L2PU data request must be smaller than the L2PU decision timeout for the L2SV.
- How the segment organization should be made? If each segment has few applications, control applications will remain idle mostly of the time, wasting resources. On the other hand, an excessive large segment implies in control and monitoring application overloads, reducing the overall system efficiency.

From the above items, one can infer how complex the setup of a given TDAQ execution can be. Very technical details should be analyzed, and the required knowledge for ensuring that all modules will operate in harmony is impractical for many TDAQ collaborators with little knowledge about the system implementation. Thousands of modules must be configured, and time may be wasted in the attempt of making all modules correctly set for combined operation.

At the beginning of the TDAQ development, configuration databases were created by hand, since a configuration database persists as a text file, with a syntax very similar to XML [13]. While this solution might be suitable for small configurations, it rapidly shows to be impractical for larger TDAQ configurations, due to the great amount of applications to be configured. Also, the required expert knowledge made this approach impossible for novice users.

The first automatized approach was based on templates. In this case, the database syntax of each object was described in

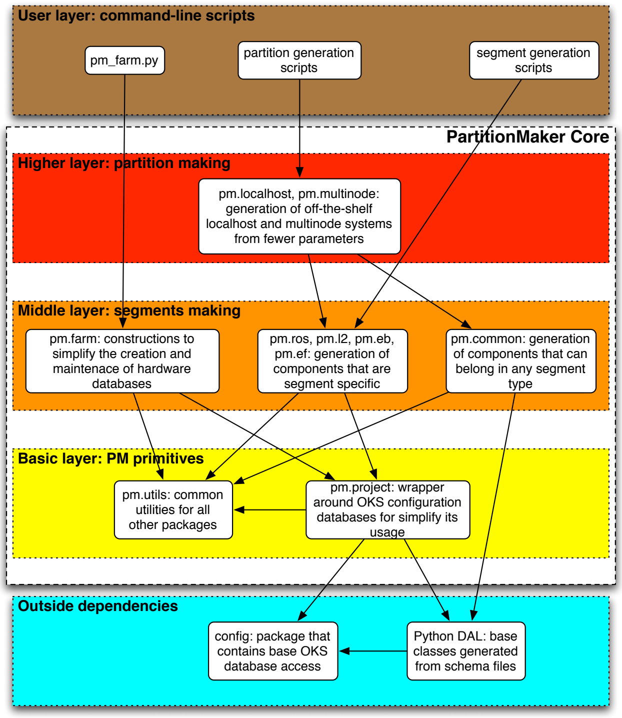

User layer: command-line scripts
pm_farm.py
partition generation scripts
segment generation scripts
PartitionMaker Core
Higher layer: partition making
pm.localhost, pm.multinode: generation of off-the-shelf localhost and multinode systems from fewer parameters
Middle layer: segments making
pm.farm: constructions to simplify the creation and maintenace of hardware databases
pm.ros, pm.l2, pm.eb, pm.ef: generation of components that are segment specific
pm.common: generation of components that can belong in any segment type
Basic layer: PM primitives
pm.utils: common utilities for all other packages
pm.project: wrapper around OKS configuration databases for simplify its usage
Outside dependencies
config: package that contains base OKS database access
Python DAL: base classes generated from schema files

Figure 4: *PartitionMaker* block diagram.

a template file. The most important parameters contained tags that could be replaced by its desired configuration value. The replacement was made by a script designed for that task, using a simple "find-and-replace" approach. As a result, the user was forced to set all the configurable parameters, and any non-configurable attribute should be set manually after the script execution. Although this approach was able to simplify the configuration task, it had many disadvantages. For instance, the number of parameters to set, although greatly reduced, was still very large. Also, since the template was previously generated, any change to the database schema file could make the template useless until the template developer had implemented the necessary changes. Finally, this approach had no consistency check, resulting that only expert users were able to successfully use them.

Later on, a software layer was built on top of this script, in order to automatize the template tags setting by processing some high-level parameters provided by the user (number of nodes per application, number of segments, etc). This second version resulted in a manageable number of parameters to be set by a greater number of users, but it was still cumbersome to use, and it lacked a consistency check, as well as dynamic adaptation to cope with schema file changes.

Having the past experience in mind, the *PartitionMaker* was developed. Its block diagram is presented in Fig. 4. The *PartitionMaker* philosophy is to embed expert knowledge in such a way that even novice TDAQ users can generate flawless configuration database files. Also, the *PartitionMaker* dynamically connects to OKS, which makes any schema change immediately available for *PartitionMaker*.

When first initialized, *PartitionMaker* reads the schema files and dynamically generates a Python [14] representation of all classes defined within the schema file. This is accomplished by using C++ / Python bindings [15], so that *PartitionMaker* can profit from resources already available in OKS, and OKS implementation issues are kept hidden from the configuration environment.

After initialization, users access the *PartitionMaker* functionalities to achieve their configuration goals. In order to support a broad range of users (essential for large high-energy physics collaborations), *PartitionMaker* is organized in a three layer structure.

The *PM primitives* layer is responsible for basic *PartitionMaker* functionalities. It contains an interface responsible for abstracting OKS access for users, being used for loading and writing configuration files. It also provides the rules for representing a schema defined class as a Python class. Finally, this layer also provides functionalities for searching and modifying modules within a configuration database.

The *Segment* layer takes care of grouping modules in a meaningful way for a given TDAQ run, such as hardware clusters and segments. The segment modules are responsible for producing TDAQ segments (L2, EB, etc). This layer takes the proper care in order to ensure that the modules within a given segment will be correctly configured for proper combined operation.

The *Partition* layer is responsible for combining the resources from the lower layers, in order to provide a fully operational configuration database based on just few high-level parameters. This layer ensures that the segments and modules generated by the lower layers will operate in harmony, not jeopardizing TDAQ execution due to configuration mismatches.

On top of all these layers, there is the *User* layer. It is composed by a set of command-line scripts envisaged for quickly providing configuration databases based on default values. Also, this layer allows users to develop their own, custom made configuration scripts, helping them achieving their specific configuration goals.

With this multi-layer approach, users with very distinct knowledge about the TDAQ infrastructure are able to achieve their configuration needs. Novice users can work using only available command-line scripts, which will require only a few number of high-level parameters, such as the number of applications of each type (ROS, L2PU, etc). The other low-level parameters are derived from the ones input by the user, so that, at the end, a standard, flawless, partition is returned. On the other side, expert users can profit from custom generation scripts using the lower layers to achieve very specific configuration goals, by setting any desirable parameter, at a cost of a better understanding of the impact of each parameter to the TDAQ infrastructure operation. Users can even work on multiple layers, for instance, using the higher layer to create a standard partition, and then, using the lower layer resources, edit the returned partition structure to accomplish their configuration needs.

Fig. 5 shows an example of the generation flow of a partition envisaged for a L2 run containing 8 ROS, 1 L2SV and 20 L2PU. In this case, the user provides, via a command-line script, only

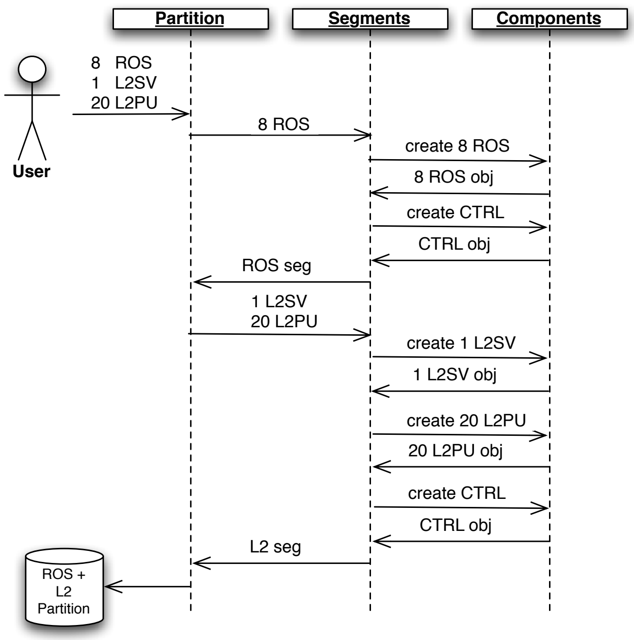


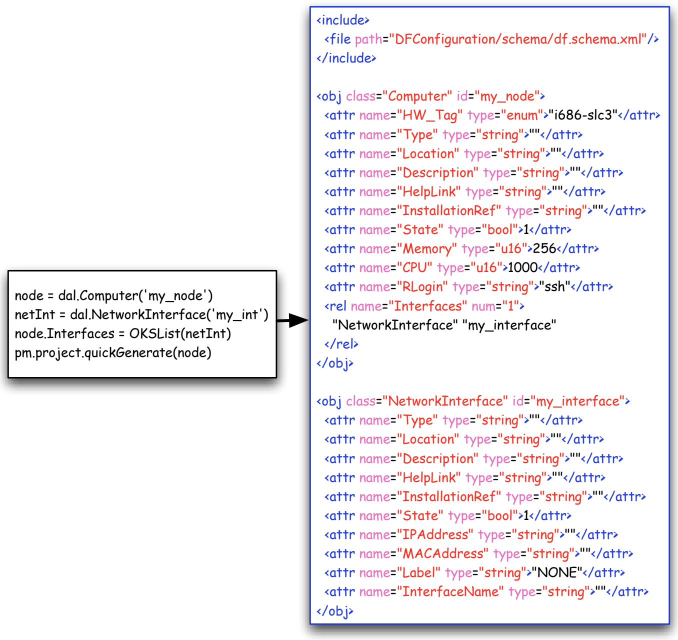
Figure 5: *PartitionMaker* generation flow example.

the high-level parameters (the number of applications of each type) to the *Partition* layer. Next, the *Partition* layer starts the configuration database creation process by splitting the task into the creation of a ROS segment and a L2 segment. It first calls the *Segment* layer, passing, at this time, the ROS segment information. The *Segment* layer then, requests to the *PM Primitives* layer the creation of the components needed for the ROS segment. Once the segment is created, it is returned to the *Partition* layer, which moves on to the L2 segment creation, in a similar way as performed to the ROS segment. At the end, the final configuration file is returned to the user, ending the configuration creation process.

The *PartitionMaker* represents the configuration structure via Python classes. Therefore, prior to exporting this structure as a configuration file, the user is capable of editing it, aiming at generating a specific configuration design. Finally, the *PartitionMaker* allows previously generated configuration files (even built without using *PartitionMaker*) to be loaded and edited using its embedded functionalities.

Another important task is the acquisition of hardware infrastructure information. For this, a special module was developed for retrieving information about the hardware envisaged for TDAQ operation. This information comprises CPU clock information, amount of available memory, network interfaces available, etc.. Later, the gathered information is used to generate a hardware database for usage when the partition file is created. For time optimization sake, this modules uses multiple threads [16] for accessing the available hardware, so that it takes, for instance, only a few tenths of seconds to acquire the information of many hundreds of hardware nodes.

Since the entire environment was implemented as a set of Python modules, users can take advantage of the many re-



```
node = dal.Computer('my_node')
netInt = dal.NetworkInterface('my_int')
node.Interfaces = OKSList(netInt)
pm.project.quickGenerate(node)
```

```
<include>
  <file path="DFConfiguration/schema/df.schema.xml"/>
</include>

<obj class="Computer" id="my_node">
  <attr name="HW_Tag" type="enum">"i686-slc3"</attr>
  <attr name="Type" type="string">""</attr>
  <attr name="Location" type="string">""</attr>
  <attr name="Description" type="string">""</attr>
  <attr name="HelpLink" type="string">""</attr>
  <attr name="InstallationRef" type="string">""</attr>
  <attr name="State" type="bool">1</attr>
  <attr name="Memory" type="u16">256</attr>
  <attr name="CPU" type="u16">1000</attr>
  <attr name="RLogin" type="string">"ssh"</attr>
  <rel name="Interfaces" num="1">
   "NetworkInterface" "my_interface"
  </rel>
</obj>

<obj class="NetworkInterface" id="my_interface">
  <attr name="Type" type="string">""</attr>
  <attr name="Location" type="string">""</attr>
  <attr name="Description" type="string">""</attr>
  <attr name="HelpLink" type="string">""</attr>
  <attr name="InstallationRef" type="string">""</attr>
  <attr name="State" type="bool">1</attr>
  <attr name="IPAddress" type="string">""</attr>
  <attr name="MACAddress" type="string">""</attr>
  <attr name="Label" type="string">"NONE"</attr>
  <attr name="InterfaceName" type="string">""</attr>
</obj>
```

Figure 6: Partition creation example.

sources available within this technology, such as an integrated development environment (IDE) with a high-level syntax language for configuration database prototyping. An example of such IDE is displayed in Fig. 6. In this case, the configuration information for a hardware node is manually created. The user first creates a computer node object called "my_node". Since the node needs a network interface object as well, an interface named "my_interface" is also generated and linked to the computer node created. At the end of the process, the configuration structure is converted to a OKS configuration file (right side of the figure).

Finally, the easiness of usage of the developed environment allows plugins to be quickly developed for very particular applications, not covered in the default implementation. In addition, as mentioned earlier, configuration generation scripts can be written, allowing the automatic creation of configuration database files, which is a great step forward for running the TDAQ infrastructure in an automated way.

The approach used in the *PartitionMaker* has resulted in a clear productivity gain for TDAQ developers. Before, expert knowledge was frequently required in order to configure the TDAQ infrastructure correctly. TDAQ execution problems due to misconfiguration were common at that time. With the *PartitionMaker*, potential configuration problems could be identified prior to TDAQ execution, avoiding TDAQ operation abortion. Since *PartitionMaker* embeds expert knowledge, collaborators became more confident during the configuration phase. This results, as well, in better working time optimization for TDAQ experts, since their need for generating configuration database files was greatly reduced, letting them focus on other TDAQ priorities.

5. Automatic Testing of the ATLAS Trigger System

As most detector systems, ATLAS makes use of a Finite State Machine (FSM) to drive its components into running state [17]. Control is led by specialized applications that form a structured tree of control. Commands are passed via a graphical user interface to the run control root and broadcasted down to the several nodes of the system in an orderly manner. Here are the different relevant state transitions that one must drive the system through until run data can be collected:

1. Boot: at this point, the configuration database file is read, and the run control tree started. Each controller starts up any controller placed hierarchically under it within the segment tree. Then, each controller will start up any application within its segment, as well as all the configured RDB servers. All the applications are started up on the computer node designed for them in the configuration file.
2. Configure: during this phase, the applications will retrieve their dynamic configuration parameters from their RDB servers, which contain a cached copy of the original database configuration file. The TDAQ infrastructure is ready to begin its operation.
3. Run: this is the phase where data would actually be read from ATLAS detectors, during normal operations, or from preloaded files if the TDAQ execution is envisaged for testing purposes. During this step, monitoring applications sample all TDAQ resources (memory consumption, L1 accept rate, CPU utilization, etc). They also generate online histogramming with timing information, general physics quantities, etc. Finally, log files are created on the application nodes with any useful information for posterior analysis (e.g. warning and error messages issued). This running state is sustained until further user intervention or problems occur.

When stopping the system, the reverse steps must be taken until it is found again in its primary uninitialized state.

While manually issuing the commands above is affordable for nominal operations, it is quite tedious to perform all theses steps manually when several testing cases must be covered, either for release validation or for tests performed automatically during the night (overnight verifications), or just to test new acquired hardware [7]. To fulfill this gap, the *Runner* environment was developed. Its main goal is to provide a programmatic way of running the TDAQ infrastructure, as well as collecting monitoring information from the monitoring applications and providing, if desired, a summarized report of the overall TDAQ execution.

A general example of the *Runner* execution flow is shown in Fig. 7. The running script defines the configuration database and parameters for collecting monitoring information like frequencies and time length. At first, the Runner will scan the configuration parameters inside the database and find the addresses of the monitoring nodes where information during the run is collected. It then requests the TDAQ infrastructure to be executed for a time long enough so that a number of monitoring samples can be obtained.

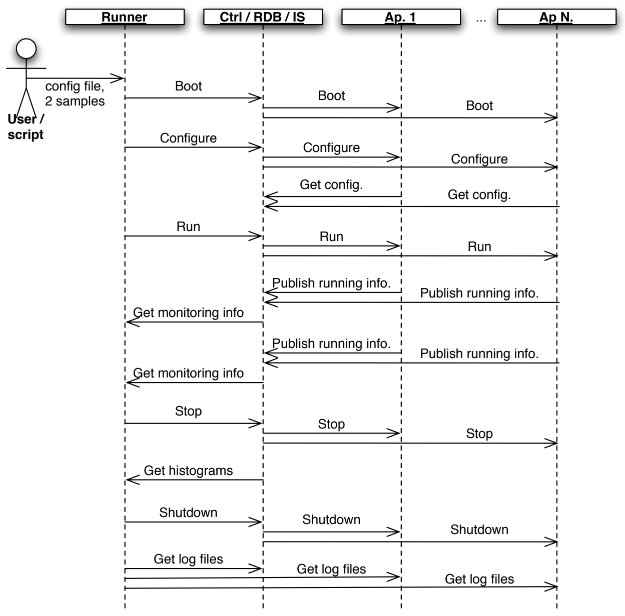


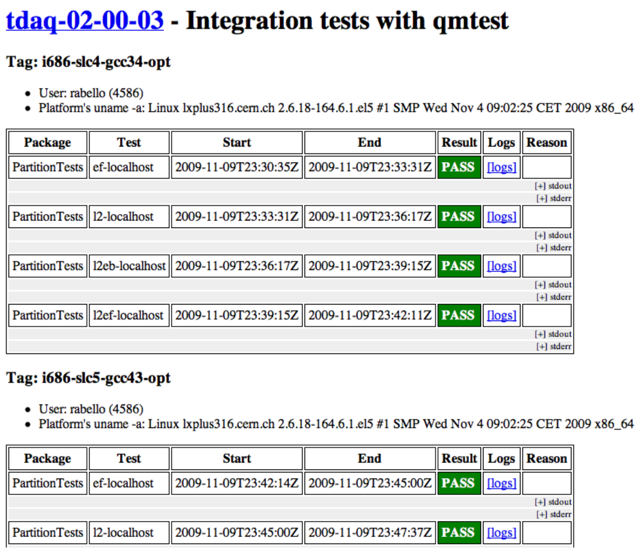
Figure 7: Runner execution flow example.

Once the required monitoring samples are collected, the *Runner* issues the stop command to the controller, which propagates the command to all TDAQ applications. In the sequence, the online histograms are retrieved from the IS server, and the TDAQ infrastructure is requested to be shutdown and reset into its initial state. As the last step, the *Runner* connects to each of the nodes used in the run to collect their log files. All monitoring and log information is recorded into files in a user specified directory.

The *Runner* is presented as a set of Python modules that interact with currently provided DAQ tools, forming a binding between a scriptable language and the ATLAS data acquisition system. The same development environment and syntax language available for *PartitionMaker* are available for the *Runner*. Plugins and running scripts can be easily developed for this environment. By combining both environments with an analysis framework that supports Python, like ROOT [18], the configuration, execution and analysis of the data obtained during TDAQ execution can be fully automated.

5.1. DAQ and HLT Release Validation

The ATLAS DAQ and HLT subsystems maintain a series of daily release builds that accumulate minor software fixes (production branches) and/or major modifications (development branches). Once software has been validated in offline testbeds, it may be commissioned at the ATLAS Control Room for production deployment. Integration tests are a necessary step for software validation. We have implemented integration tests based on the *Runner* for simple DAQ and HLT structures, which are executed every night, after candidate releases are built. These tests allow the community to quickly react to release problems and cure pathologies as soon as they appear


tdaq-02-00-03 - Integration tests with qmtest

Tag: i686-slc4-gcc34-opt

- User: rabello (4586)
- Platform's uname -a: Linux lxplus316.cern.ch 2.6.18-164.6.1.el5 #1 SMP Wed Nov 4 09:02:25 CET 2009 x86_64

Package	Test	Start	End	Result	Logs	Reason
PartitionTests	ef-localhost	2009-11-09T23:30:35Z	2009-11-09T23:33:31Z	PASS	[logs]	
						[+] stdout
						[+] stderr
PartitionTests	l2-localhost	2009-11-09T23:33:31Z	2009-11-09T23:36:17Z	PASS	[logs]	
						[+] stdout
						[+] stderr
PartitionTests	l2eb-localhost	2009-11-09T23:36:17Z	2009-11-09T23:39:15Z	PASS	[logs]	
						[+] stdout
						[+] stderr
PartitionTests	l2ef-localhost	2009-11-09T23:39:15Z	2009-11-09T23:42:11Z	PASS	[logs]	
						[+] stdout
						[+] stderr

Tag: i686-slc5-gcc43-opt

- User: rabello (4586)
- Platform's uname -a: Linux lxplus316.cern.ch 2.6.18-164.6.1.el5 #1 SMP Wed Nov 4 09:02:25 CET 2009 x86_64

Package	Test	Start	End	Result	Logs	Reason
PartitionTests	ef-localhost	2009-11-09T23:42:14Z	2009-11-09T23:45:00Z	PASS	[logs]	
						[+] stdout
						[+] stderr
PartitionTests	l2-localhost	2009-11-09T23:45:00Z	2009-11-09T23:47:37Z	PASS	[logs]	

Figure 8: Integration tests for release validation.

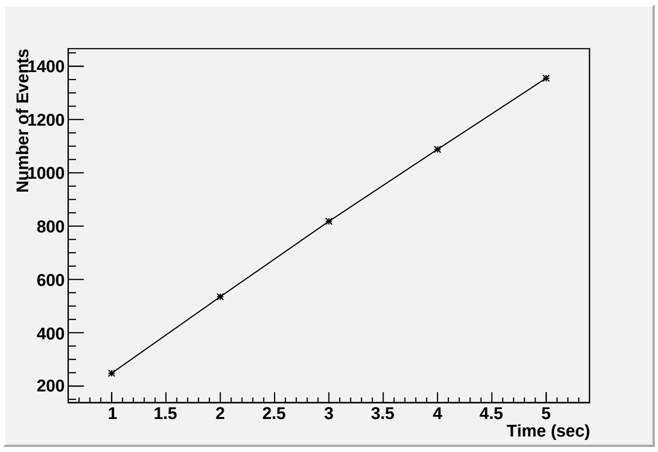


Figure 9: Number of L2 output events for a localhost test.

in the software. An example output of the test suit is shown in Fig. 8. It allows users to identify tests that have failed and access log files and statistics gathered by the *Runner*.

In most cases, the successful sequence of boot, execution for a short period of time and shutdown of the TDAQ software is enough for validating software changes in nightly tests. However, some users might want to look more into the details of a particular run. As another example, it is presented, in Fig. 9, a plot generated after the automatic execution of the ATLAS trigger system for a period of 5 seconds. The plot presents the number of events approved by one L2PU in a localhost[1] test. The test shows a linear evolution of the number of processed events (constant L2 output rate), which is considered the expected behavior for this particular test.

[1]Localhost execution is when all TDAQ modules enabled for a given run are running on a single computer.

6. Conclusions

The TDAQ infrastructure, by being highly configurable, may require specialized knowledge for its setup and deployment. A flexible configuration database service (OKS) was put in place to cope with its stringent conditions.

The *PartitionMaker* is a configuration environment bound into Python, based on OKS. Because of its multi-layer architecture, users with different levels of expertise can quickly achieve complex configuration schemes, supported by the expert knowledge built-in. Functionalities from the multiple layers in the *PartitionMaker* can be combined to generate not only standard configurations, where most parameters are automatically derived, but also very specific configurations, where parameters must be set according to the envisaged TDAQ operation goal (e.g. testing new software releases).

The *Runner* environment was developed for automatic execution of any TDAQ configuration and to retrieve monitoring information and log files distributed among several application nodes. It can be called directly by the user, as well as executed from a script file, resulting in a powerful mean of running automatic software and hardware validation tests.

Both *PartitionMaker* and the *Runner* make use of Python's development environment and syntax language. This makes the creation of plug-ins to extend their functionalities a simpler task. Testing scripts can be written describing the TDAQ configuration using the *PartitionMaker* and then, automatically executed using the *Runner*. The very same program can make use of Python analysis framework for further analysis of the monitoring data retrieved, generating an analysis report. In this way, the operation of TDAQ tests can be fully automatized. This system was recently deployed for validating newly acquired HLT racks and infrastructure with success.

Nowadays, such automated environments are being widely used by TDAQ collaboration, even by novice TDAQ users. Nightly validation tests, technical runs, and large scale tests [6] are fruitfully profiting from such automatization facilities.

Acknowledgements

The authors would like to thank CAPES, CNPq, FAPERJ, FINEP (Brazil), CERN and European Union for their financial support. Also, we would like to thank the TDAQ collaboration of ATLAS for discussions concerning this work.

[1] L. Evans, P. Bryant, LHC machine, Journal of Instrumentation (2008 JINST 3 S08001).

[2] The ATLAS Collaboration, The ATLAS experiment at the CERN large hadron collider, Journal of Instrumentation (2008 JINST 3 S08003).

[3] I. Riu, et al., Integration of the trigger and data acquisition systems in ATLAS, IEEE Transactions on Nuclear Science 55 (1) (2008) 106–112.

[4] K. Kordas, et al., The atlas data acquisition and trigger: concept, design and status, Nuclear Physics B - Proceedings Supplements 172 (2007) 178 – 182. doi:DOI: 10.1016/j.nuclphysbps.2007.08.004.

[5] R. Jones, L. Mapelli, Y. Ryabov, I. Soloviev, The OKS persistent in-memory object manager, IEEE Transactions on Nuclear Science 45 (4) (1998) 1958–1964.

[6] D. Burckhart-Chromek, et al., Testing on a large scale: Running the ATLAS data acquisition and high level trigger software on 700 PC nodes, in: Proceedings of the Computing in High Energy and Nuclear Physics (CHEP), Mumbai, India, 2006, pp. 60–65.

[7] G. Unel, et al., Studies with the ATLAS trigger and data acquisition pre-series setup, in: Proceedings of the Computing in High Energy and Nuclear Physics (CHEP), Mumbai, India, 2006, pp. 922–926.

[8] G. Aielli, et al., Status of the atlas level-1 central trigger and muon barrel trigger and first results from cosmic-ray data, in: Real-Time Conference, 2007 15th IEEE-NPSS, Batavia, USA, 2007, pp. 1–6. doi:10.1109/RTC.2007.4382845.

[9] M. Abolins, et al., Integration of the trigger and data acquisition systems in ATLAS, Journal of Physics: Conference Series 119 (2) (2008) 1–10.

[10] A. dos Anjos, et al., The second level trigger of the ATLAS experiment at CERN's LHC, IEEE Transactions on Nuclear Science 51 (3) (2004) 909–914.

[11] J. A. Simões, et al., The ATLAS DAQ system online configurations database service challenge, Journal of Physics: Conference Series 119 (022004) (2008) 1–11.

[12] J. Keogh, M. Giannini, OOP Demystified, McGraw-Hill, 2004.

[13] E. R. Harold, W. S. Means, XML in a Nutshell, 3rd Edition, O'Reilly, 2004.

[14] The python programming language [online] (March 2009).

[15] C++/python interfacing [online] (March 2008).

[16] A. S. Tanenbaum, Modern Operating Systems, 2nd Edition, Prentice Hall, 2001.

[17] I. Alexandrov, et al., Online software for the atlas test beam data acquisition system, IEEE Transactions on Nuclear Science 51 (3) (2004) 578–584. doi:10.1109/TNS.2004.828793.

[18] R. Brun, F. Rademakers, ROOT - an object oriented data analysis framework, Nuclear Instruments and Methods in Physics Research 389 (1-2) (1997) 81–86.