Utility software for ATLAS TDAQ

Summer student project report

Argyris Zardilis
az325@cam.ac.uk

September 6, 2013

The main bulk of the work undertaken was for two different applications so the report is structured accordingly into two standalone sections.

1 Information Service Aggregator

1.1 Introduction

ATLAS Trigger and Data Acquisition (TDAQ) online software consists of thousands of application instances operating on physics data from the moment they leave the detector to the point they are written permanently on disk. These applications publish monitoring information about their status using the TDAQ online Information Service (IS). In the most common use-case experts operating the system, either for testing or during physics data-taking, want to get a broad view of the system on a higher level than the single application level. To this end the Aggregator application gathers and merges subsets of the monitoring information of the entire system and publishes these aggregates using the IS itself. That way the aggregates can be inspected by system experts to gain a more general view of the system, its performance and behaviour.

1.2 Application

The IS consists of a number of servers running on the TDAQ infrastructure. The clients of these IS servers are information providers and receivers. Information providers are usually the TDAQ applications that publish histograms and state variables. Examples of state variables are the number of events received by a trigger application or the CPU usage of an application. These state variables are grouped into structures called IS objects which are the basic unit of communication between IS servers, providers and receivers. IS objects have unique names and a set of fields specified formally in schema files. Providers register the objects when they start running and update them at regular intervals which vary across applications. Receivers on the other hand subscribe to the IS servers to get updates of specific IS objects. A histogram gatherer already exists so the present application only covers numeric state variables contained in IS objects.

The machines in the computer farm, where the set of applications run, are grouped into racks of 3040. Conventionally there is one IS server per rack and all the applications

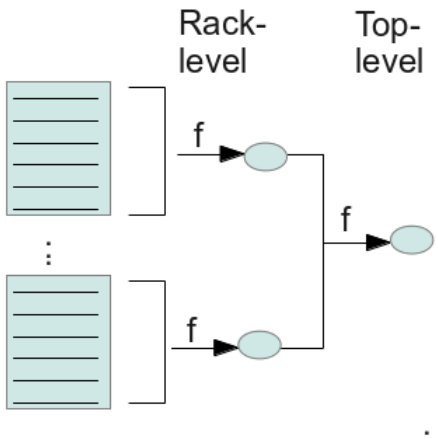

Figure 1: Two-layer aggregation

running in that rack publish their monitoring information to that server. So for n number of racks there are the corresponding n IS servers. It is of interest to see an aggregation for each rack but also for the entire system (all racks). Therefore the aggregation performed is hierarchical; there exists one aggregator application per rack and a top level aggregator application that aggregates the aggregate IS objects the rack level aggregators publish (Figure 1).

The aggregation, whether rack-level or top-level however, follows the same steps. It starts by gathering the required information from a specific input IS server and at the same time filtering them using specific criteria. Then the gathered IS objects are merged using a function f to a single IS object which is then published to the output IS server (Figure 2) . It should be noted here that aggregation is robust to changes in the IS object schema, a feature which is particularly useful during testing where the design of the system and consequently the IS object schema are not stable.

The rack-level and top-level aggregator application exist as different scripts but because they share the same general structure they use the same underlying software objects to perform the common tasks as outlined in the previous paragraph, namely gathering/filtering, merging and publication. Then they only differ on the criteria they use to select the IS objects to gather and the merging functions used. Both are command-line scripts taking as arguments the filtering criteria, name of input IS server for the gathering step and the name of the output IS server for the publishing step. They also accept the gathering interval in seconds as a command line parameter. The criteria used for selection can be the type of the IS object and/or a regular expression for the name of the object. As merging functions the rack-level aggregator uses sum, standard deviation and average while the top-level one sum, sum of standard deviations and weighted average. For fairness, in the top-level calculations the weight of each input is taken into account as well. The weight is the number of applications aggregated over by the rack-level application to get a specific IS object. These numbers are all published by the rack-level applications.

Both scripts and the common code base providing the common functionality are written in Python and use the existing IS Python bindings to get access to the Information

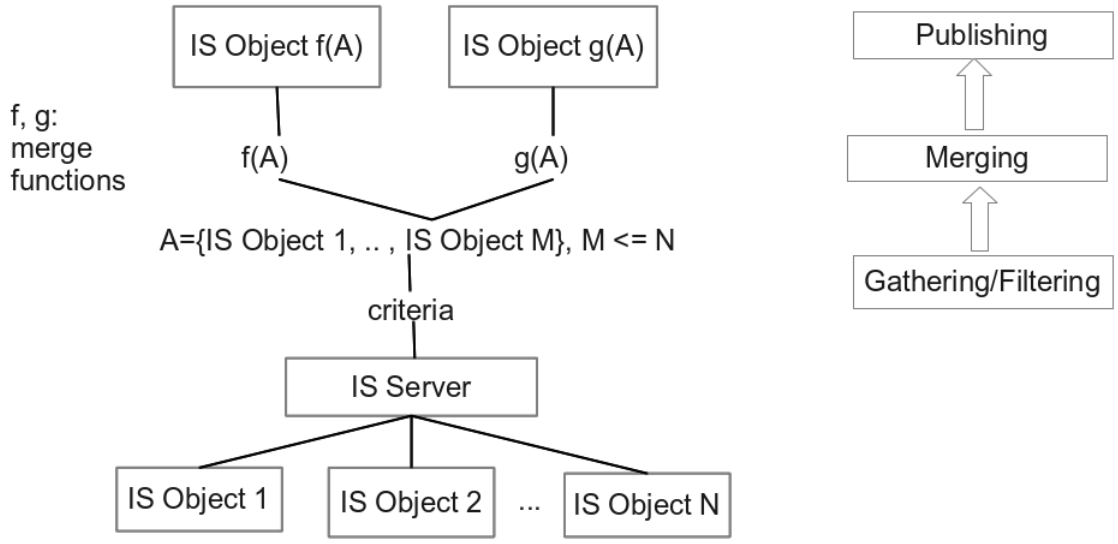

Figure 2: General aggregator structure

Service infrastructure (read IS objects, write to IS servers etc.)

2 Partition configurations

2.1 Introduction

The configuration for ATLAS TDAQ applications resides in an object-oriented database which contains information such as what applications should be included in the system, where they should run, how they should interact etc. Manually creating these databases is a tedious, error-prone and time consuming process so in general automatic tools are used to automate all or part of the process. These automatic applications are naturally highly coupled to the specific system design and database class schema. During this period of the Long Shutdown 1 (LS 1) and using the experience gathered during Run 1 the TDAQ system is undergoing a redesign making it more flexible and scalable and also improving its performance to meet the specifications of Run 2. Due to these changes, the application used so far, *PartitionMaker*, can no longer be used. For this purpose an application was developed for the automatic creation of partition configurations following the new design of the system.

2.2 Application

Every hardware or software object in the system is represented by a database object which contains a number of attributes specifying its behaviour. In this object-oriented world objects can be members of other objects thus forming more complex structures representing complex relations and interactions between software and physical objects in

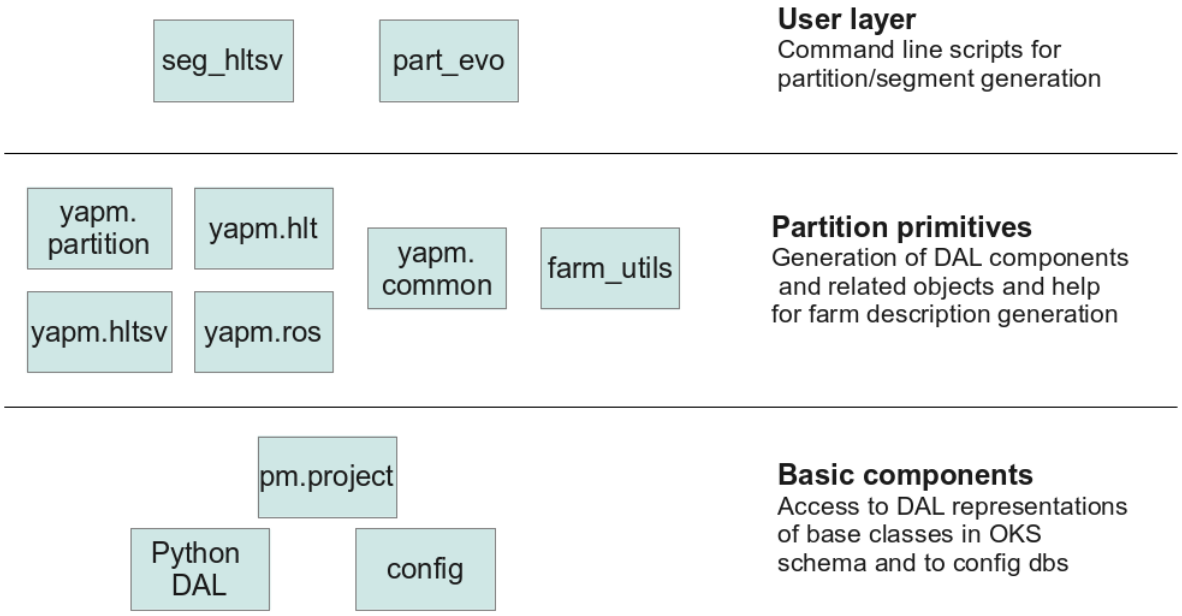

Figure 3: Structure of partition making tool

the real system. There are also database objects that do not directly correspond to real parts of the system, either software or hardware, but are rather abstract entities that are used for grouping related objects together. This makes the system more modular and helps with coordination and control. These abstract groupings are called Segments. Segments are themselves objects which can contain other Segments thus overlaying an abstract tree-like control structure on top of the real physical-software objects and interactions. Another abstract notion is that of a Partition which acts as a top-level Segment which contains all the Segments that are not parts of other Segments (roots of tree) .

The application consists of a number of Python scripts arranged in 2 layers. At the user level there are a couple of command line scripts that the user of the tool can call giving a number of parameters to specify the design of the system the resulting database is going to describe. One of them creates a full-blown partition while the other one creates one of the top-level segments which is directly underneath the Partition in the tree structure. These user-level scripts in turn use *yapm* to create the standard objects needed. All the functionality for creating objects, setting their attributes and relations is packaged into *yapm*. *yapm* in turn uses some existing lower-level libraries to get programmatic access to the database and its functions (add objects, update objects etc.). This is summarised in Figure 3 . The creation in both the Segment and Partition cases follows a bottom-up approach starting from from the smaller basic segments that do not contain other composite objects and then proceeds to the top-level segments and then to the partition object itself (Figure 4).

The tool is able to take a small number of parameters that determine the database that will be created and in turn how the system will look like. A Python dictionary describes where the application instances should run and consequently the number of instances that will be in the system. It also takes other various parameters like network addresses, file-system paths to write log files. Another useful feature is the ability to

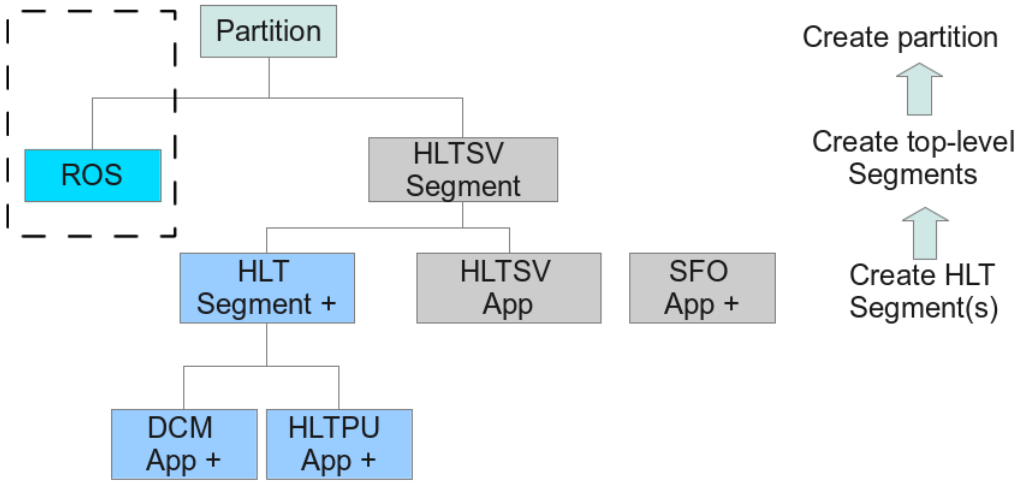

Figure 4: Generation flow

create non-standard Segments or Partitions by providing a modification function that the application will use to post-process the created database.

The application started as a temporary solution for use in the Technical Runs but has since grew to a more complete tool. The experience gained during its development with the discovery of new use-cases coming with the new design can be used to inform the design of the update of the official tool, *PartitionMaker*.

3 Conclusion

Both the applications were used successfully in the testing periods during this Summer. The IS information Aggregator was particularly useful during testing because it gave in a glance a quick overview of system behaviour. The aggregated values it produced were used for taking measurements of the performance of the system with the new design. The partition configuration tool also proved quite useful in the testing environment where configuration of different flavours were needed to test different parts of the system. In the limited time available for tests on the production system it saved valuable time. These applications can be used in future testing periods and the experience from their design can inform the design of the final tools that will go into production for Run 2.