Journal of Physics: Conference Series **119** (2008) 022004 doi:10.1088/1742-6596/119/2/022004

The ATLAS DAQ System Online Configurations Database Service Challenge

J. Almeida

Universidade de Lisboa, Edifício C8, Campo Grande, 1749-016 Lisboa, Portugal

M. Dobson, A. Kazarov[1], G. Lehmann Miotto, J.E. Sloper[2], I. Soloviev [1] and R. Torres[3]

CERN, Geneva 23, CH-1211, Switzerland
1 - on leave from Petersburg Nuclear Physics Institute, Russia
2 - on leave from University of Warwick, Norvey
3 - on leave from University Federal do Rio de Janeiro, Brazil

Igor.Soloviev@cern.ch

Abstract. This paper describes challenging requirements on the configuration service for the ATLAS experiment at CERN. It presents the status of the implementation and testing one year before the start of data taking, providing details of:

- the capabilities of underlying OKS object manager to store and to archive configuration descriptions, its user and programming interfaces;
- the organization of configuration descriptions for different types of data taking runs and combinations or participating sub-detectors;
- the scalable architecture to support simultaneous access to the service by thousands of processes during the online configuration stage of ATLAS;
- the experience with the usage of the configuration service during large scale tests, test beam, commissioning and technical runs.

The paper also presents pro and contra of the chosen object-oriented implementation comparing with solutions based on pure relational database technologies, and explains why after several years of usage we continue with our approach.

1. Introduction

The configurations database service described in the paper is a part of the ATLAS High-Level Trigger, Data Acquisition (DAQ) and Controls system [1]. It is used to provide the overall description of the DAQ system and partial description of the trigger and detectors software and hardware. Such descriptions cover the control-oriented configuration of all ATLAS processes running during data taking (including information such as: which parts of the ATLAS systems and detectors are participating in a given run, when and where processes shall be started, what run-time environment to be created for each of them, how to check status of running processes and to recover run-time errors, when and in what order to shut down running processes, etc.) and provide configuration parameters for many of them (overall DAQ data-flow configuration, online monitoring configuration, connectivity and parameters for various DAQ, trigger and detector modules, chips and channels).

The configurations database is used by many groups of developers from various systems and detectors. Each group is responsible to prepare their own part of the configuration and to write the code configuring their applications using common configuration service tools.

The configuration service is accessed simultaneously by thousands of processes during the boot and configuration phases of the run. As well, the configuration service is responsible to notify and to reconfigure processes, if any configuration changes happen during the run.

And finally, the configuration service archives used configurations, so later they can be browsed by experts and accessed by programs performing events processing.

The use cases and requirements listed above for the configuration service result into several challenges discussed in the next section.

2. Configuration service Challenges

2.1. Data Model

The database schema used by the configuration service for descriptions mentioned in the previous section includes about three hundreds classes. The classes are designed by different DAQ, trigger and detector groups. To speak on a common language across all of them, it is desirable to have a so-called core database schema describing common data types, which are extended by the groups. In the resulting schema all classes are interconnected via inheritance and relation links.

As an example of the schema extension, consider the core Application class, which contains several tens of generic properties to describe parameters of a process such as unique identity, command line parameters, various timeouts, actions to be taken in case of errors, environment variables, the application's binary file and host, initialization and shutdown dependencies, etc. Then each group takes the base Application class and extends it providing their specific properties. The important thing is that the instances of all these classes still can be considered as applications and be handled in the same style, e.g. by the code of the control software.

So, the first challenge requires the configuration service to support sophisticated data model able to describe complex extendable interconnected pieces of data.

2.2. Data Access Programming Interface

The configuration data are accessed by code written by developers from different groups. In many cases the code has to retrieve the same information and it is important that this is done in a consistent way. This becomes essential, if calculation of the configuration information requires several accesses to the database and the access to further parameters depend on the results of previous accesses. For example, to calculate some configuration parameters one has to execute a database query; then an algorithm is applied to the data returned by the query and the algorithm's result is used as parameters for the next query, etc. Or, the configuration information is a result of calculations on top of data returned by several independent queries. Ideally, such code should be implemented as a function available to any developer and the queries themselves should not be used by developers explicitly.

As well, most of the developers, who need to read the configuration information stored in a database, are not database experts. It is thus desirable that the code to access the database data is completely hidden behind some high-level API of the programming language they are using. This allows users of the configuration service to effectively develop their application code instead of spending time on learning the low-level API of the database implementation and debugging various problems caused by the inefficient usage of the database or by database API version changes.

The mechanism to hide the low-level database implementation and to provide a configuration data access API using data types of a programming language is called the data access library (DAL). A DAL maps database data types on the types supported by the programming languages (e.g. relational table is mapped on C++, Java or Python class). A DAL also is responsible to instantiate database data as instances of the programming languages types (e.g. create for each row of the relational table corresponding C++ object) or to save back to the database the changes, if the instances have been

modified or new ones have been created. A DAL can also provide functions (called the DAL algorithms) to calculate data based on the result of several database queries, as it was explained in the previous paragraph.

A DAL can be written by hand or automatically generated from the database schema. The latter is required, if the database schema contains a large number of classes, or is simply changing often. The DAL generation is absolutely necessary to implement the ATLAS DAQ configuration service because of the high degree of complexity of the schema, its parallel development by several groups and the frequent changes during development cycles. For example, the configuration schema defined by the ATLAS DAQ–HLT (high level trigger) development release was changing at least once per week during last year.

Thus, the second challenge requires that the configuration service provides automatically generated DALs for programming languages used by the ATLAS software (C++, Java and Python).

2.3. Database Population

To prepare a configuration description for a data taking session one has to put into the database all the data describing the software releases, the online infrastructure, the data-flow, monitoring, trigger and detector applications, the hardware they use, trigger and detector specific parameters; furthermore all elements need to be organized into a controllable hierarchy. Even a minimal configuration for the online infrastructure test is composed of hundreds of configuration objects. It is expected, that the final configuration for the complete ATLAS will contain an order of several tens thousands of configuration objects. It is clear that the database describing such configurations cannot be filled manually. One has to have special tools to create, to compare or to copy configurations and to test their completeness and consistency.

At the same time, many configurations prepared for different types of runs (cosmic, physics, calibration, etc.) and for different sets of participating trigger and detector parts should co-exist.

To avoid duplication of information belonging to different co-existing configurations it is reasonable to share common configuration data. Note, the sharing of configuration data not only saves the space used by the database, but also improves maintenance of the data in case of changes, since modifications will have to be done only once. This approach is first of all applicable to the software releases descriptions, which are updated only when new releases or patches are installed. In a similar way, the description of all hardware elements needs to be updated only when there are corresponding changes in the physical world such as installation of new computers, changing cables between modules, network reconfiguration, etc.

Many configuration data can be and have to be generated automatically. For example, the description of a software release can be generated by analyzing it's installation area and cmt requirements files[1] used for the build. It contains all binary files with their supported platforms, the run-time environment they need (e.g. paths to shared libraries including used external packages, their specific environment variables, etc.). Another example is the generation of the hardware description. This can be done analyzing the ATLAS technical coordination installation databases containing information about installed hardware organized by ATLAS systems and detectors, or even by reading simple ASCII file containing list of hosts. Based on this information the configuration tool checks the state of the installed hardware, extracts missing parameters (e.g. network or module addresses) and saves the configuration description. Note, that the automatically generated data described in this paragraph can be shared between configuration descriptions for different runs.

The rest of the configuration data should be built using tools, which have a detailed knowledge about the architecture of the ATLAS DAQ, trigger and detector systems. Such tools can be used for different purposes, e.g. to prepare a database configuration testing DAQ read-out or event builder systems on test beds of the ATLAS experiment, to test high-level trigger algorithms with pre-loaded event data on a computer farm of an external institute, to test newly commissioned hardware at the

[1] CMT is a tool to build ATLAS software releases; the requirements file is an analogue of a make file

experiment site, or to describe the full ATLAS configuration. A generation tool will on one hand be used by experts, who need the capability of redefining any configuration parameters they would like to test; on the other hand it shall also be usable by non-experts, who only know the type of configuration they want to run: for those such a tool needs to be able to set default configuration parameters.

Summarizing, the third challenge requires the configuration service to provide a set of tools for automatic generation of configuration descriptions for the software releases, for various hardware installations and for the different data taking configurations.

2.4. Performance and Scalability

One of the goals of any ATLAS online service is to minimize as much as possible the experiment's downtime during the physics data taking period. The configuration service is used by thousands of applications during the boot and configuration stages of the data taking, and it has a significant influence on the overall time needed to setup a data taking session. By this reason it has to assure a scalable architecture to guarantee that this time does not depend too much on the number of clients of the configuration service.

Another important requirement is effective usage of the configuration service in case of small modifications of settings between data taking runs. In this case the affected clients should be able to read from the configuration service the changes since the previous run instead of re-reading the complete configuration: this is especially true for clients which are re-reading an unmodified configuration.

Most of the developers of code using the configuration service are not database experts. One cannot guarantee that they are using the service in an optimal way, e.g. their code is reading any configuration parameter only once. In addition, the configuration data can be read by several libraries developed by different groups and used within single process, which even more increases probability of non-optimal code. So, to achieve optimal performance the configuration service has to support caching of configuration data on the client's side and to read the data from the service only when this is really needed.

The forth challenge requires the configuration service to have a scalable architecture and performance in order to meet the demanding experiment's requirements. In case of small changes between runs the service has to support partial reconfiguration. As well, whenever possible, the service has to cache information on the level of client's process to prevent multiple requests to the configuration service for information, which was already read.

2.5. Archiving

Once the configuration data were used for data taking, they have to be safely archived. The archived data can be used later for processing of event data, to be browsed by experts or to be retrieved for preparing a new configuration. The configuration service has to guarantee, that once archived the data will not be removed or unintentionally corrupted, e.g. during a modification of the configuration for the next run. This requirement addresses the fifth configuration service challenge.

2.6. Easy Usage

The configuration service is part of the TDAQ releases. They are used not only at CERN, but also in many ATLAS collaborating institutes around the world. To be used, the configuration service has to be easily available there. In many cases the institutes cannot use the CERN-based remote configuration service because of various reasons (e.g. slow or unreliable network or even lack of internet connections for certain test-beds, various local software policies) and need to support the configuration databases themselves. At the save time one should not expect, that there will be knowledgeable database administrators or simply advanced database users at any site.

The last configuration service challenge is the requirement to be easily accessible or installable for all ATLAS collaborating institutes.

3. Configuration Service Implementation

The DAQ system and its predecessors (CERN R&D 13 group [2], ATLAS DAQ Prototype-1 Project [3]) had evaluated several shareware and commercial candidates for the configuration service, including persistent object managers, object and relational databases, but no system satisfying all the requirements was found [4]. Therefore it was decided to use as a prototype an existing object manager OKS [5] and make it capable of fulfilling the DAQ configuration service needs.

Initially, the OKS has used the Rogue Wave object persistency [6] storing objects in cross-platform binary files. At that time the Rogue Wave library was used as the base general purpose library within the DAQ system. The latter was then replaced by the C++ Standard library and instead of Rogue Wave persistency the OKS started to use human readable XML files (the possibility to easily browse, modify and distribute these files was one of the key points of its success). Then, to satisfy the DAQ needs the OKS was extended to implement remote access, to provide an abstract API layer using several OKS access implementations and to realize the OKS database archiving. These features are described in more details in the rest of this section.

3.1. The OKS Persistent Object Manager

The OKS is a set of tools to provide objects' persistency. It is based on the following object data model:

- the basic entity is an object with unique *object identifier*;
- objects with common properties and behaviour are described by a *class*, defining *attributes* (primitive types), *relationships* (links with objects) and *methods* to act on the object properties;
- classes support *inheritance* (multiple inheritance is allowed) with *polymorphism* (overloading of inherited object properties in a child class);
- *composite* objects (i.e. an object built from dependent child objects);
- *integrity* constraints (i.e. type and value restrictions on attributes and relationships).

The OKS classes and objects can be created and modified dynamically, put into a persistent storage and read back. The native OKS API for this is C++.

For effective data selection there is a query language. OKS allows active notification on data changes as well (i.e. call user callbacks when an object is created/deleted/modified).

The OKS supports several advanced functions to work with persistent schema and data: the schema evolution allows modifications to the schema as the user applications evolve, and the data migration permits data to be accessed by successive versions of a schema.

The main persistent storage for OKS is XML files. There are two types of them: the schema ones define classes and the data files store database objects. An XML file can include other files to build a complete database from well structured files and to share them between different OKS databases.

Another persistent storage for OKS databases are relational databases. OKS uses the LCG CORAL [7] library that allows transparent usage of several relational database back-ends such as Oracle, MySQL and SQLite. This type of storage is oriented for archiving purposes: to check-in OKS files, to browse archives, and to check-out archived files. The archiving supports incremental versioning to store only differences between a complete base version of a database and its last version. As an example the modification of an object's attribute will be stored in the database as a new row of relational table and not as a complete copy of object. This feature drastically reduces space used by OKS archives.

OKS provides distant access to the databases via remote database servers (RDB), developed on top of CORBA [8]. The RDB server is used to access a single database from different clients running on computers without a common file system. It also solves scalability problems, when access is required by a huge number of clients, by caching the results of OKS queries.

OKS provides two graphical applications to work with the OKS schema and data. The schema editor is UML-like graphical editor to browse and to modify the database schema, see left part of Fig. 1 as example. The data editor is a graphical editor to browse and to modify the database data. It allows

customizing appearance of the user-defined views presenting relations between objects. An example is shown on right part of the Fig. 1. Alternatively the data can be accessed in a tabular format. The data editor has a graphical query constructor. Once a query is constructed, it can be saved and used for future data retrieval using the editor or OKS API.

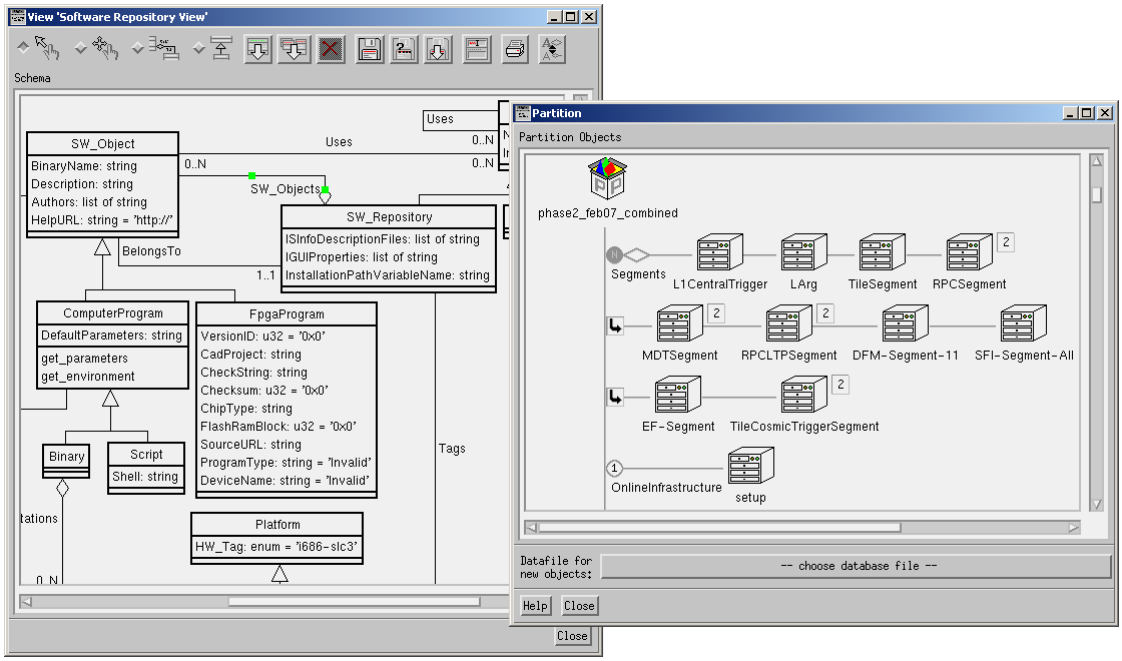

Fig. 1. Examples of the OKS Schema Editor (left side) and of the OKS Data Editor (right side) views.

The OKS provides utilities to print out the contents of a database, to compare schema and data files, to merge databases, to insert and to extract the OKS databases from the archive and to list its contents. The OKS archive Web interface is shown on Fig. 2:

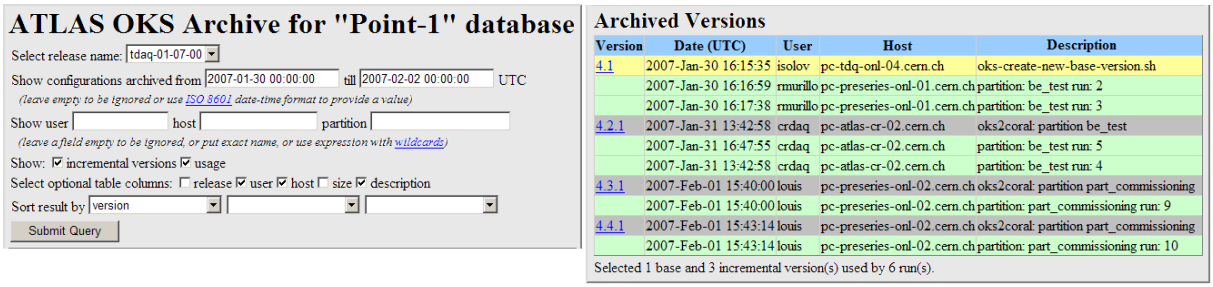

Fig. 2. Example of Web interface to OKS archive

One can select the archived data by release name, by time intervals, and by user, host and partition masks. The result can be presented with different level of details and sorting policies. The archived data can be compared and extracted from the archive as OKS databases.

3.2. Configuration Service Programming Interface Layers

The code of ATLAS programs does not use the OKS or RDB APIs directly. Instead there are two layers of configuration service API to avoid any dependencies on the database implementations.

The first config layer provides an abstract interface to work with databases and to access configuration objects. This interface exists for C++, Java and Python programming languages. The implementations of this interface are available as plug-ins for OKS XML files, OKS relational archives and for the RDB server. This layer is used to work with an arbitrary database schema for

reading its description, loading or creating new databases, querying them to get configuration objects, accessing their properties and subscribing on database changes notification. The config layer is normally used by the ATLAS control and infrastructure applications which are working with user classes that are not known at compilation time.

The DAL layer uses the above abstract config one to map database schema on C++, Java and Python classes and to instantiate the database data as appropriate objects of such classes. When necessary, user-defined methods can be inserted into DAL classes to implement algorithms on top of the config objects. The DAL is automatically generated from the OKS schema for the above mentioned programming languages by the *genconfig* tool, which is a part of the configuration service. The DALs are used by all ATLAS applications which need to get the configuration description.

Fig. 3 presents the relations between databases interfaces and users of the configuration service (the Partition Maker tool is used to generate configuration descriptions and it will be described in one of following subsections).

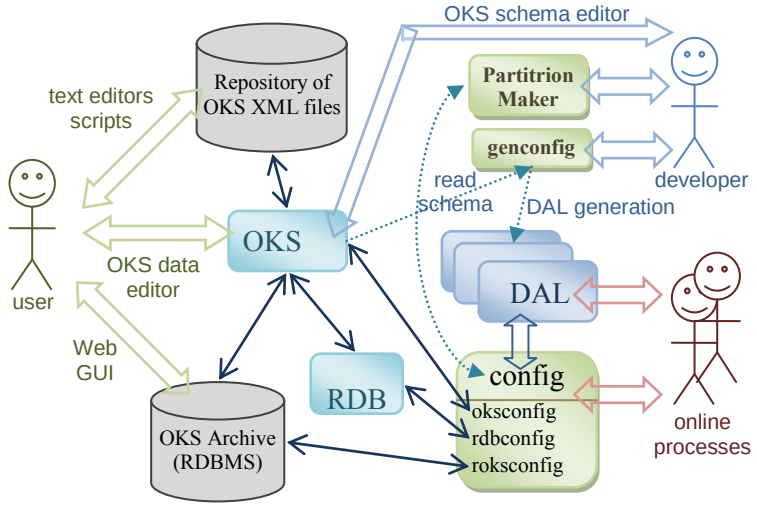

Fig. 3. Configuration service interfaces and users

As it was already mentioned, the user's code does not depend on the database implementation; in the most frequent case of DAL usage it deals with high-level generated classes only. An example of such code is shown below, on Fig. 4. The generated classes and their methods are shown in bold.

```
  // (1) initialize configuration object
Configuration db("oksconfig:daq/partitions/test.xml");
  // (2) find partition executing a DAL algorithm
const Partition * p = daq::core::get_partition(db, "test");
  // (3) read partition's default host
const Computer * h = p->get_DefaultHost();
  // (4) print out hostname
std::cout << "Default host: " << h->UID() << "\n";
  // (5) get pointer on new host
const Computer * h2 = db.get<Computer>("pc-tdq-onl-02");
  // (6) set new partition's default host
p->set_DefaultHost(h2);
  // (7) commit database changes
db.commit();
```

Fig. 4. Simplified example of C++ code using generated DAL

Line 1 shows how to create a configuration service object. The argument of its constructor defines the implementation that will be used and its parameters. In this example the OKS XML file will be

used. If one wants to use RDB or relational archive, just this argument needs to be changed; there is no need to recompile or re-link the program. Line 2 shows how to use an algorithm. The get_partition() one is written by hand. Line 3 shows how to access configuration object via reference using the generated method for the relation "DefaultHost" defined in the database schema. Line 4 demonstrates how to get the unique identity of an object. Line 5 shows how to get from configuration service an object of certain class knowing its identity. The line 6 sets new value for the relation. The last line saves changes to the persistent storage, i.e. in our example into the XML file.

3.3. Configuration Schema

The structure of any configuration object is defined by the database schema. The DAQ defines a common database schema agreed with trigger and detector groups, which extend it to introduce properties of their specific configuration objects. The common schema contains an order of hundred classes and defines several domains:

- The *software releases* including programs, libraries, supported platforms, external software packages they use and the variables they need.
- The *hardware* including racks, computers, crates, modules, interfaces, cables, links, networks and their composition into hardware systems.
- The *control* including applications (i.e. a process to be started under certain conditions; it points to a computer and program and defines their parameters), resources (a hardware object or an application which can be temporary disabled in the scope of given configuration), resource sets (group of closely coupled resources) and segments (an individually controlled container object, that is composed of resources, applications and nested segments).
- The *configuration* including partition object (contains parameters and segments for given run). The partition object is used as the root of the tree of other objects, describing the configuration. Via segments, their nested segments, resources, applications and their relations with parameters the partition object implicitly references any another object participating in the configuration description. This allows finding any piece of configuration data by simple navigation between objects instead of execution of queries. The latter fact allows pre-loading of a configuration description into the client's cache and avoiding later requests from such clients to the configuration service: that helps a lot with performance and scalability issues.

The data-flow and monitoring groups extend above schema and define another hundred classes to provide the overall description of their configuration parameters. The trigger and several detectors also extend the schema to introduce classes describing their specific configuration parameters. The total number of configuration classes used for M4 detector commissioning run was about three and half hundred.

3.4. Organisation of Database Repositories

The OKS XML files are stored as database repositories. To make a repository visible for configuration service it is enough to add it to the colon-separated path variable. Then an OKS file can be referenced relative to the value of this variable.

Each repository is versioned, when the database schema is modified. Usually this happens together with the installation of new software releases.

A repository contains folders assigned to different groups of users. A folder has a predefined structure for schema extension, software, hardware, segments and stand-alone partitions. Each group has write access permissions for its folder and is responsible to prepare segments, which can be inserted into combined ATLAS partitions.

There are several repositories across CERN sites. One of them is dedicated for development and is available on the afs file system. Its latest version corresponds to the last release (rebuilt each night). The software and hardware descriptions are automatically regenerated together with the release build. Another one is the ATLAS production repository. It is available at ATLAS experiment site and contains configurations for the detector technical and commissioning runs.

3.5. Generation of Configurations

The Partition Maker tool has been developed to generate descriptions for different tests and participating systems. It contains embedded ATLAS DAQ system and trigger expert knowledge in order to guide a user through the configuration process and to minimize a risk of configuration errors.

The tool has three levels architecture. Each level uses resources from lower levels. The lowest layer is responsible for representing trigger or DAQ components as classes and ensures correct configuring of them. The middle level creates segments composed of objects built from previous level and guarantees their correct interaction within each segment. The highest level links together segments and certifies that all together they result a meaningful configuration.

The Partition Maker allows the user to work in any of the three levels. Novice users can work in the highest level only, in order to quickly generate a configuration with minimal input information. The users with more experience can use resources from lower levels to prepare specific configurations.

The Partition Maker has been written on the Python programming language. This allows easy development of scripts using it and integration of user configuration knowledge.

3.6. Scalable Online Access at ATLAS Point-1

The programs using configuration service are running on rack-mounted computers, separated by ATLAS systems and detectors. So, the programs belonging to a rack require similar configuration data. We are running one RDB server on the rack's local file server (LFS). Such RDB server reads master copy of configuration data from central file server and provides access to them for all processes running inside the rack as shown on Fig. 5:

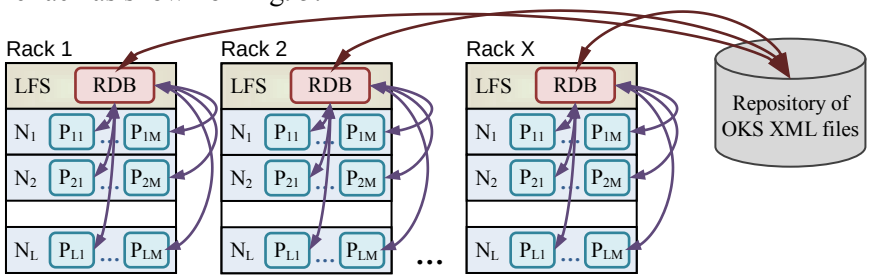

Fig. 5. Schema of the RDB-based scalable access on Point-1

The maximum number of clients per one RDB is ~ 240 (30 HLT computers per rack, 8 processing tasks per computer). To achieve better performance, the number of clients per RDB server can be reduced by running more RDB servers per rack or using dedicated racks for online services.

3.7. Offline Access

At start of a new data taking run the configuration data are automatically archived into relational database. A record referencing archived configuration is added to the ATLAS conditions database. Later one can to access configuration data using programming interfaces or to browse and restore them using graphical user interfaces.

3.8. Experience with OKS Usage

The OKS-based configuration service has been used since the very beginning by the ATLAS online software. It has been exploited in combined test beams [9] [10], large scale tests [11], recent technical and commissioning runs. The service was actively used to prepare the configuration descriptions of the test runs and to provide access to them for the online processes. It always met functionality and performance requests of the users.

4. The Current Object-Oriented Approach: Advantages and Drawbacks

In the past we have tested many candidates for the implementation of the configuration service for ATLAS. The object database management systems seemed to be the best suited choice, since they

have the data model and programming interfaces satisfying our needs and they supported data versioning that is useful for data archiving. However we have failed to a find scalable and reliable implementation. This was the main reason, why we have started our own persistent object manager prototype project ten years ago. Since then we are keeping an eye on technologies used by other groups and experiments to implement configuration databases. Most frequently such solutions are based on the usage of relational database management systems. This can be explained by several reasons including their prevalence and maturity in the database world, the availability of commercial and freeware systems, a suitable data model for certain types of configuration data. Another important booster for this choice is that relational database specialists are very requested on the employment market also in non-science sectors: this makes their study appealing for young scientists without an already well defined career path. Below we summarize why the usage of relational databases does not give to us benefits comparing with existing solution.

In the beginning we listed several challenging requirements to the configuration service. A pure relational technology does not address them and cannot be used without tools developed on top of it.

To populate databases one has to use special utilities. It is unrealistic to put all data by hand for both, the existing OKS-based implementation and the relational one. The prototyping of such utilities is a bit simpler with current implementation, since XML files can be produced by simple scripts, easily taken into account by configuration service tools, editable and removable. In case of relational databases such prototyping requires more complicated actions dealing with relational storage especially if tables are shared with other developers.

The installation and support of relational databases requires a qualified support. Many users from external institutes installing trigger and DAQ releases, whenever possible, are trying to avoid programs using relational databases.

The classical relational data model does not support inheritance, which is one of the key aspects in the data model used by the configuration service. One cannot create a new table and mark it as derived from an already existing table, so that existing queries would take this new table into account. As well, the relational data model does not support arrays. To implement them one has to create extra tables or encode them as BLOBs, and then provide special functions to calculate values of a "multi-value" column. Without a layer built on top of the relational data model it cannot satisfy our requirements.

The relational databases do not provide a DAL. Instead a developer using relational databases has to learn the SQL language and the database API allowing using it. A DAL for a relational database can only be designed, when the relational data model extension discussed above is known. In our case this means development of own tools or usage of third-party ones, which do not satisfy all our needs (e.g. user-defined algorithms, mapping on all used programming languages).

The advanced relational database management systems provide scalable solutions. However even an Oracle farm composed of several servers cannot handle in an acceptable time simultaneous requests coming from tens of thousands clients. To achieve the required performance one has to develop tools similar to our RDB servers, which serves a set of relational database clients and caches the results of SQL queries for them [12]. Such tools access the relational database server only when a query coming from a client was never performed.

On the client side the results of SQL queries can also be cached. However when the same data or parts of them are retrieved by different SQL queries, such caching mechanism cannot help. This is different from the DAL and config objects created using our configuration service, where the effectiveness of the cache mechanism does not depend on the relations used to access the object.

And finally, the safe archiving of configuration data is one of the most difficult problems arising for users of the relational databases. Since configuration data are changed quite often (e.g. pieces of ATLAS systems and detectors may be enabled / disabled before each new run), it is not enough just to modify existing relational data. In such case their history of usage will be erased. Thus, the relational data used for a run have to be archived and some versioning mechanism has to be provided. Using a true relational model such mechanism cannot be easily implemented, since any single change of data results in the necessity to version all data having relations to it; in turn all data having relations on data

versioned during the previous step have to be versioned also, etc. The requirement to archive data complicates the relational tables storing configuration data and especially the tools modifying them. Differently, the OKS archiving provides incremental versioning and any single modification of OKS data results into a single record in the archive (i.e. one new row in OKS relational table describing value of object's attribute or relationship). This is possible due to fact that OKS relational tables used for archiving store values of OKS entities like class, object, and value of attribute or relationship. One should not expect to find in the OKS archive a table with name "Module" or "Application". Their descriptions will be stored into several OKS tables allowing to deal with arbitrary database schemas and their evolutions, and to implement space-preserving archiving.

5. Conclusions

In this paper we have shown how the present implementation of the configuration service for the ATLAS experiment is capable of meeting all the challenging requirements posed to it.

A comparative analysis with other approaches such as the usage of relational databases has been performed showing that, despite the power and maturity of relational databases, it would not be possible to use them directly without implementing several layers on top of them for the purpose of the experiment's configuration. Especially the requirement of providing data access libraries in an environment of still frequently changing database schema and the need for efficient archiving of different configuration versions remain very hard to meet.

Therefore we consider that the choice of basing the configuration service on a homemade persistent object manager, OKS, is still justified and is, at present, the best way to satisfy all requirements put forward by the ATLAS experiment.

References

[1] ATLAS Collaboration, "ATLAS High-Level Trigger, Data Acquisition and Controls Technical Design Report", - 2003-022, June 2003
[2] G. Ambrosini et al., "The RD13 Data Acquisition System", proceedings of the CHEP conference, San Francisco, California, USA, 1994
[3] G.Ambrosin, et al., "The ATLAS DAQ and Event Filter Prototype "-1" Project", Computer Physics Communications 110, 1998, pp.95-102
[4] Maria Skiadelli, "Object Oriented database system evaluation for the DAQ system", diploma thesis, Patras Polytechnic School, Greece, March 1994
[5] R.Jones, L. Mapelli, Yu.Ryabov, I.Soloviev, "The OKS Persistent In-memory Object Manager", IEEE Transactions on Nuclear Science, vol 45, No 4, Part 1, August, 1998, pp. 1958 – 1964
[6] Rogue Wave Software, Inc., "Tools.h++ Foundation Class Library for C++ programming", March 1996, for more information see URL http://www.roguewave.com
[7] CORAL - COmmon Relational Abstraction Layer, a package of CERN LHC Computing Grid Project, see http://pool.cern.ch/coral/
[8] I. Alexandrov et al., "Experience with CORBA communication middleware in the ATLAS DAQ", proceedings of the CHEP conference, Interlaken, Switzerland , 2004
[9] I. Alexandrov et al. , "Online Software for the ATLAS Test Beam Data Acquisition System", proceedings of the 13th IEEE - NPSS Real Time Conference, Montreal, Canada , 2003
[10] S. Gadomski et al., "Deployment and use of the ATLAS DAQ in the Combined Test Beam", proceedings of the 14th IEEE - NPSS Real Time Conference, Stockholm, Sweden , 2005
[11] D. Burckhart-Chromek et al., "Testing on a Large Scale: running the ATLAS Data Acquisition and High Level Trigger Software on 700 PC Nodes", proceedings of the CHEP conference, Mumbai, India , 2006
[12] L. Lueking et al., "FroNtier : High Performance Database Access Using Standard Web Components in a Scalable Multi-tier Architecture", proceedings of the CHEP conference, Interlaken, Switzerland , 2004