SLAC-PUB-11396
September 2005

Evolution of the Configuration Database Design*

A. Salnikov[a†]

[a]Stanford Linear Accelerator Center,
2575 Sand Hill Road, Menlo Park, CA 94025, USA

The *BABAR* experiment at SLAC successfully collects physics data since 1999. One of the major parts of its on-line system is the configuration database which provides other parts of the system with the configuration data necessary for data taking. Originally the configuration database was implemented in the Objectivity/DB ODBMS. Recently *BABAR* performed a successful migration of its event store from Objectivity/DB to ROOT and this prompted a complete phase-out of the Objectivity/DB in all other *BABAR* databases. It required the complete redesign of the configuration database to hide any implementation details and to support multiple storage technologies. In this paper we describe the process of the migration of the configuration database, its new design, implementation strategy and details.

1. INTRODUCTION

During the first years of the data taking the *BABAR* experiment [1] used the object database management system (ODBMS) Objectivity/DB [2] for most of its data storage needs. All main databases, which include event store, conditions, configuration, calibrations, and ambient databases, were implemented initially using this technology. Over the years of experience with these initial implementations some significant limitations had been discovered, which led *BABAR* to re-evaluation of the chosen technology. Experience with the alternative partial implementation of the event data store proved that ROOT [3] can be used as a replacement data storage technology [4]. In 2003 *BABAR* implemented a new event store based on two technologies – ROOT file I/O for event data, and relational databases for bookkeeping information [5].

After the successful re-implementation of the event store using an alternative technology, a new project was started for the migration of the remaining databases [6]. In this paper we describe the process of migration of the configuration database, its new design, and some implementation details.

2. CONFIGURATION DATABASE

The configuration database is an important part of the *BABAR* on-line system, and it has worked sufficiently well since the beginning of the *BABAR* data taking in 1999. Details of its initial design and implementation may be found in [7].

Currently, the Objectivity implementation of the configuration database holds about 60 MBytes of the configuration data. The data are represented by about 30 persistent classes. This makes it the most compact and simple database compared to other *BABAR* databases. It also makes it a perfect candidate for the pilot project within the whole migration effort.

The initial implementation of the configuration database contained several problems which had to be resolved during this migration. The main problem was that the database API exposed many details of the chosen implementation technology, such as persistent types, object handles, class names, etc. Additionally the database experienced a few evolutionary changes in the design during its lifetime which were not always consistent with the initial ideas.

The first phase of the database migration consisted in designing the new abstract interface to the configuration database that does not depend on any implementation technology and consis-

*SLAC-PUB-11396
†For *BABAR* Computing group

Presented at X International Workshop on Advanced Computing and Analysis Techniques in Physics Research, 5/22/2005-5/27/2005, Zeuthen, Germany

Work supported in part by Department of Energy contract DE-AC02-76SF00515

tently incorporates all changes in the design since the original implementation.

3. NEW DATABASE API

A new configuration database API was first prototyped in the Python high-level programming language [8]. The prototype allowed us to better understand and test all details of the new API. One particular implementation based on a relational database was built as a part of the prototype, and some important studies were performed with this implementation. The test showed that performance and scalability of the implementation based on relational database were the same or better than those of the initial implementation based on Objectivity/DB. The prototyping helped us to test quickly several different ideas for the implementation and optimize the structure of the SQL tables.

Because of the differences between the dynamically typed Python and statically typed C++, expressing the same API in C++ terms needed some additional work. One particular problem is passing non-polymorphic types through the abstract interfaces. The data classes which could be stored in the database are unrelated, they do not have a common base class. In C++ genericity for non-polymorphic (unrelated) types is achieved through the use of the template features. But abstract interfaces and template code do not mix well, there cannot be template virtual methods in C++. A solution for this problem was found through the use of the custom smart pointer class, which is analogous to a type-less pointer (pointer to void) but keeps additional run-time type information (RTTI). The smart pointer can be used with the virtual methods of the abstract API because it hides the type of the specific class used with this pointer. Database implementations use the RTTI stored inside the pointer to reconstruct exact type of the data class. Details of this solution are hidden from the client code, so the users only see an interface working with any arbitrary kind of user data.

The client code which uses configuration database services needed a migration to the new abstract API. This was probably the most time consuming task for the whole project, and it involved significant reorganization of the client packages to separate the code into the pure transient classes and specific persistent technology-dependent classes. This process introduced a number of the new packages into the system, but the net result was a better code organization with improved dependency control.

4. DATABASE IMPLEMENTATIONS

Because the new API is defined at the abstract level, it is possible to build and use any number of specific implementations. One obvious choice for the implementation was to use the existing Objectivity database. This implementation, which was built on the top of the old pre-migration database API, serves a number of purposes:

- as a proof of principle that the new API is sufficient and works as expected,
- as a default implementation to be used until we switch completely to the alternative implementations,
- as a source of the data from which alternative implementations will initialize their data.

This *bridge implementation* has been built and is currently in use in *BABAR* in places which have not switched yet to new implementations.

The choice of the implementation technology for alternative implementations depends on particular requirements for configuration data access from the client code. In *BABAR* there are two classes of clients with different requirements:

- production site needs reliable, fault-tolerant, concurrent read-write access to the data
- remote sites need zero-management, easy-to-use, fast, scalable read-only access to the data

For the second group of clients ROOT is an obvious choice because of its many attractive features, such as: persistent data definition using

C++-like syntax allows easy migration of the existing Objectivity schema. Lack of servers simplifies management at small installations like personal laptops. Larger installations with many concurrent clients could use xrootd server [9] for performance scalability. Finally, *BABAR* data distribution services already know how to distribute ROOT data files to external *BABAR* institutions.

The *ROOT read-only implementation* was built to satisfy the requirements of read-only clients. ROOT-persistent classes are almost exact copies of Objectivity DDL files, except for obvious differences in the data description languages. Due to the low volume of the data in the configuration database it is possible to keep the whole database in one ROOT file, which avoids complex file-allocation management. Both metadata and object data are stored in the ROOT tree structures, making the data model analogous to the relational model. Lack of full indexing support in ROOT, especially for strings, required introduction of few additional data structures to make data access sufficiently fast. Overall, thanks to the small size and simple structure of the configuration database, building the ROOT implementation was straightforward.

The requirements of the first group of clients could be satisfied only by the true database implementation with full support of ACID properties (Atomicity, Consistency, Isolation, and Durability). Considering that we are seeking to replace an object-oriented database, the only remaining viable option is a relational database. The relational database has to be augmented with some object-relational mapping mechanism, which is not trivial despite many existing free or commercial products.

It would be beneficial if such mechanism could reuse existing persistent schema from the different implementation avoiding duplication of efforts for devising a new schema. Because the Objectivity schema will be phased out, the only remaining schema is the ROOT schema. In ROOT it is possible to do object data serialization in the platform-independent format using the specialized buffer classes. Serialized data can be extracted from the buffer, optionally compressed, and stored as byte-strings. The reverse process allows complete reconstruction of the object data from the byte-strings stored externally. In the case of the relational database the byte-strings can be stored as a binary large object (BLOB) as a part of a relational table. This works well for the self-contained object without external references, but configuration data objects all fall into this category.

The *read-write implementation* was built with this serialization mechanism using the MySQL database [10] as a storage of the object metadata and the BLOBs with object data. MySQL provides all required database properties needed to implement reliable and dependable storage of the configuration data. Additionally it provides features such as data replication, load balancing, etc., which simplify data management tasks.

5. BUILDING APPLICATIONS

As a result of this migration process there are now three different implementations of the configuration database, any of which can be used by the client applications. The decision about which particular implementation should be used depends on where the application runs, and this decision should be delayed until run time so that the same application could run anywhere. This means that depending on which site a given application is going to run it must have access to the implementations supported by that site.

Not every implementation can be linked directly into the client applications. For example, MySQL is an "optional" software in *BABAR*, which means that not all sites are required to install it. If the MySQL implementation was linked statically or dynamically into the application, it would not be able to run at the remote sites where MySQL was not installed because of the missing shared libraries. The solution for this problem is to load optional implementations or their parts dynamically at run time and only if a specific implementation is requested by the client.

The dynamic loading of shared libraries is rather standard and straightforward across UNIX platforms supported by *BABAR*. But it requires special care to avoid multiply defined symbols in a mixed environment where parts of the implemen-

tation could appear in both static and dynamic libraries. As a result, in *BABAR* we are loading dynamically only the shared MySQL client library (`libmysqlclient.so`.) The rest of the MySQL implementation is linked statically in all applications that need configuration database services. The complete ROOT and Objectivity implementations are also linked into the applications, but the Objectivity implementation will be removed once the migration is complete. In this way any application can work with ROOT or Objectivity implementations anywhere because both ROOT and Objectivity are required currently for every site, and also with MySQL implementation where it was installed.

6. CONCLUSION

It is worth emphasizing again how important it is to have clear abstract interfaces in case of complex and evolving software systems. If *BABAR* had good abstract interfaces for configuration database from the beginning, the migration process from one technology to alternatives would be less painful. The new design described in this paper introduced abstract interfaces to the *BABAR* configuration database. This already has significant benefits for *BABAR* – the client code is free now from the unnecessary implementation details, and for the particular site it is possible to choose the best suitable implementation technology.

Early prototyping in a high-level language played crucial role in the design of the new API and allowed us to get quick answers to some questions. Implementation of this new API in C++ would have benefited from the features missing currently in C++, such as reflection and introspection.

While at the time of this writing *BABAR* is still using the bridge Objectivity implementation across all sites, work is in progress to distribute the data for alternative implementations and to start using the ROOT implementation at remote sites and the MySQL implementation in production.

Overall, this migration effort represents a significant experience which will certainly be used in the migration of the remaining *BABAR* databases.

7. AKNOWLEDGMENTS

This work is supported by the U.S. Department of Energy under contract number DE-AC02-76SF00515.

REFERENCES

1. B. Aubert *et al.*, The *BABAR* Detector, NIM A479:1-116, 2002.
2. Objectivity/DB object database management system, www.objectivity.com.
3. ROOT – An Object-Oriented Data Analysis Framework, root.cern.ch.
4. T.J. Adye *et al.*, KANGA(ROO): Handling the micro-DST of the *BABAR* Experiment with ROOT, Comput.Phys.Commun. 150:197-214, 2003.
5. M. Steinke *et al.*, How to build an event store – the new Kanga Event Store for *BABAR*, In proc. CHEP 2004, September 2004, Interlaken, Switzerland.
6. R. Bartoldus *et al.*, The Future of Non-eventstore Databases in *BABAR*, BABAR-NOTE-0576, Nov. 2003.
7. R. Bartoldus *et al.*, Configuration Database for *BABAR* Online, Proc. CHEP03, March 2003, La Jolla, USA.
8. Python programming language, www.python.org
9. A. Hanushevsky *et al.*, The Next Generation Root File Server, In proc. CHEP 2004, September 2004, Interlaken, Switzerland.
10. MySQL database management system, www.mysql.com.