Transparently managing time varying conditions and detector data on ATLAS

C. Leggett, LBNL, Berkeley, CA 94720, USA

Abstract

It is essential to provide users transparent access to time varying data, such as detector misalignments, calibration parameters and the like. This data should be automatically updated, without user intervention, whenever it changes. Furthermore, the user should be able to be notified whenever a particular datum is updated, so as to perform actions such as re-caching of compound results, or performing computationally intensive task only when necessary. The user should only have to select a particular calibration scheme or time interval, without having to worry about explicitly updating data on an event by event basis. In order to minimize database activity, it is important that the system only manage the parameters that are actively used in a particular job, making updates only on demand. For certain situations however, such as testbeam environments, pre-caching of data is essential, so the system must also be able to pre-load all relevant data at the start of a run, and avoid further updates to the data.

We present the scheme for managing time varying data and their associated intervals of validity, as used in the Athena framework on ATLAS, which features automatic updating of conditions data occurring invisibly to the user; automatic and explicit registration of objects of interest; callback function hierarchies; and abstract conditions database interfaces.

REQUIREMENTS

Online and offline requirements demand considerably different operation modes for conditions database access. In the online environment, calibration constants and other time dependent data are usually preloaded at the beginning of a run, and remain fixed for the duration of the run, in order to minimize lengthy database access. This type of access pattern is simple to model, and makes the loading of constants fairly easy. In the offline world, users access data in a much more random manner, selecting events from many different runs, which may use very different calibrations. The user should not have to concern himself with making sure that the time dependent data is up to date, rather this should be handled automatically by the framework.

In certain situation, the user does need to know when time dependent data has changed, such as when a secondary value uses the time dependent data in a computationally expensive function, and wants to cache the function result. The function must be recomputed when the data changes. This can be accomplished with the use of callback functions that are triggered at the appropriate time. Furthermore, dependency trees are needed, as one function's result may depend on that of another.

Conditions databases can contain thousands if not millions of time ordered data. It would be grossly inefficient to monitor all of them for changes. Instead, the user must notify the framework which are the items of interest. This is accomplished via a registration procedure. Furthermore, even when time intervals are monitored, the actual data need not be loaded from persistent form unless it is actually used. This can be done using proxies that use evaluation upon demand.

A separate service, called the Interval of Validity Service (IOVSvc), is used by the framework to manage all aspects of the time dependent data registration, loading, and validity checks.

DATA REGISTRATION

There are two distinct methods of making the IOVSvc aware of which time dependent data it should manage:

- registering a DataHandle `handle` and its corresponding database identifier `dbKey`:

```
regHandle(const DataHandle& handle,
          const string& dbKey);
```

- registering a DataHandle and an associated callback function against a database identifier:

```
regFcn(StatusCode (T::*fcn)(IOVSVC_CALLBACK_ARGS),
       const T* fcnObj, const DataHandle& handle,
       const string& dbKey, bool trigger=false);
```

  Here, the callback function is identified by its signature, and a pointer to it. The final optional boolean argument, if set `true`, will cause the function to be immediately triggered as soon as it is registered. This is to account for those rare cases when the registration is performed during the middle of a run, instead of at initialization time.

During the registration procedure, the data store associates a proxy with the DataHandle or callback function, by which means its access to the persistent store is controlled. Proxies are used as the basis, as they are unique, and easily identifiable.

Instead of registering a DataHandle against a database key, a DataHandle can be registered against a previously registered DataHandle. Likewise, a callback function can be registered against a previously registered callback function, allowing a chain of functions to be built up:

```
regFcn(StatusCode (T::*fcn1)(IOVSVC_CALLBACK_ARGS),
       const T* obj1,
       StatusCode (T::*fcn2)(IOVSVC_CALLBACK_ARGS),
       const T* obj2, bool trigger=false);
```

Furthermore, a callback function can be registered against multiple DataHandles or other callback functions, such that the dependency tree grows in the form of an acyclic graph (see Fig.1). When the callback functions are triggered, the graph is walked from top to bottom, with functions on each level of the graph being called in the proper order.

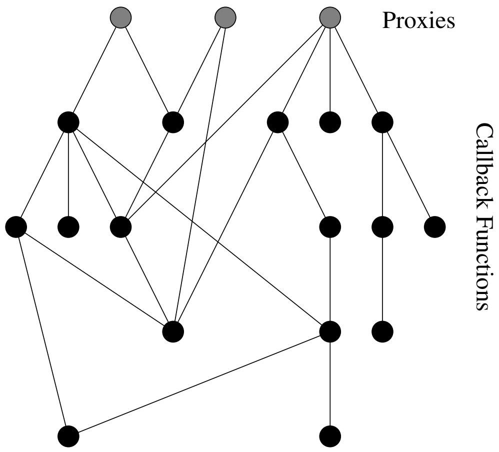

Figure 1: A hierarchical function callback graph.

LAZY DELEGATION BY PROXIES

Accessing data in the Athena framework is accomplished via proxies managed by StoreGate [1], the ATLAS data store. Once the framework has been made aware of which proxies are associated with time dependent data via the registration procedure, it will wait until the data is used before accessing persistent stores. This is accomplished via the overloading of the dereferencing operator. A DataHandle that is registered with the IOVSvc behaves slightly differently than a normal DataHandle - instead of being reset at the end of each event, the IOVSvc manages its storage lifetime policy.

The first time the DataHandle is dereference, the IOVSvc uses the current time information to read both the appropriate validity information, and a reference to the actual location of the data from a conditions database. The proxy associated with the DataHandle is configured to use the correct location of the persistent data, and the validity information is loaded into the IOVSvc. The proxy then reads in the data from its persistent form, and caches it in the DataHandle. If any callback function is associated with the DataHandle, it is subsequently triggered.

At every subsequent DataHandle access, so long as the DataHandle is in a valid state, the cached data is returned, with no further access to the persistent store.

FUNCTION CALLBACKS

One of the requirements that the database group had given us, was that the callback functions should not be required to inherit from a specific callback base class. This made the handling of the callback functions much more challenging, as in C++ there is no easy way to identify functions in a generic manner. We were permitted to specify the function arguments, which was defined in a macro for easy use by users to IOVSVC_CALLBACK_ARGS. The ultimate solution was to use Boost::bind and C++ trickery. In order to uniquely identify a callback function, we used the address of the function, but it is difficult to extract this information. C++ does not provide any way to copy the address into a variable, but it will allow you to print it out, so the following scheme can be used:

```
template<typename T>
void CallBackID::set(StatusCode (T::*updF)(IOVSVC_CALLBACK_ARGS),
                     const T *obj) {
  p_obj = (void*) obj;

  char *str = new char[8];
  sprintf(str, "%x", updF);
  m_offset = strtol(str,NULL,16);

  // make allowances for various versions of gcc
  #if (__GNUC__ < 3)
     m_offset /= 0x10000;
  #endif

  delete str;

  std::ostringstream ost;
  ost << std::hex << "[0x" << (int)p_obj << "]+" << m_offset;

  m_objName = System::typeinfoName(typeid(*obj));
  m_name = m_objName + ost.str();

}
```

The variable `m_name` contains information such as `LArCell[0x944328]`, which is sufficient to uniquely identify any function, and is parseable by humans to facilitate debugging.

The macro `IOVSVC_CALLBACK_ARGS` expands into a list of strings, which provides the keys of the conditions database entries which have changed.

INTERVAL MANAGEMENT

The IOVSvc stores validity information in multiple sorted `std::multisets`, one set for each time boundary (start of validity range / end of range), and time format (run/event, timestamp, etc). These sets are ordered by increasing or decreasing time, depending on whether the set corresponds to the start of the interval, or the end. All validity information is associated with individual data proxies, which are unique.

At each time boundary, which is user selected at initialization, the multisets are scanned for validity, comparing them with the current time. Both ends of the validity interval are checked. Invalid entries are removed from the sets, and the scanning stops as soon as the first valid entry is found. The sorting of the sets ensures that all subsequent entries are valid. Entries that were marked as invalid are reset, causing the associated data proxy to be marked invalid and reset, so that subsequent accesses to the corresponding DataHandle will trigger the reloading of the validity intervals, and access to the persistent store.

The IOVSvc can also manage the processing of events that are temporally out of order.

JOB CONFIGURATION

During initialization, users can select when the validity ranges should be checked. Currently, the options are to check the validity information at the beginning of each event, the beginning of each run, or at the beginning of the job. Since the selection is controlled by Incidents fired by the framework, it is easy to expand this behavior to any required time interval, such as once per hour, every ten events, or every three runs.

In the online environment, it is often desired that database access should be minimized, and calibration constants be loaded once during initialization and never changed. The IOVSvc also supports this mode of operation, and permits both validity ranges and the associated conditions data to be preloaded at the start of the job. This behavior can be tuned so that the constants are reloaded at the start of a new run if so desired.

Currently, the IOVSvc is seeing use in the online and offline environments, and has proved itself in the recent Combined Test Beam run for the ATLAS experiment, during both data taking, and for offline analysis.

ACKNOWLEDGMENTS

This work was supported in part by the Office of Science, High Energy Physics, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

REFERENCES

[1] P. Calafiura, C. G. Leggett, D. R. Quarrie, H. Ma and S. Rajagopalan, eConf **C0303241** (2003) MOJT008 [arXiv:cs.se/0306089].