High Level Interface to Conditions Data at Belle II

M Ritter\textsuperscript{1}, T Kuhr\textsuperscript{1}, M Starič\textsuperscript{2} for the Belle II Software Group

\textsuperscript{1} Ludwig-Maximilians University Munich, Excellence Cluster Universe, Boltzmannstr. 2, 85748 Garching, Germany
\textsuperscript{2} Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia
E-mail: Martin.Ritter@lmu.de, Thomas.Kuhr@lmu.de, Marko.Staric@ijs.si

Abstract. The Belle II experiment at the SuperKEKB $e^+e^-$ accelerator is preparing for taking first collision data next year. For the success of the experiment it is essential to have information about varying conditions available in the simulation, reconstruction, and analysis code.

The interface to the conditions data in the client code was designed to make the life for developers as easy as possible. Two classes, one for single objects and one for arrays of objects, provide a type-safe access. Their interface resembles that of the classes for the access to event-level data with which the developers are already familiar. Changes of the referred conditions objects are usually transparent to the client code, but they can be checked for and functions or methods can be registered that are called back whenever a conditions data object is updated. The framework behind the interface fetches objects from the back-end database only when needed and caches them while they are valid. It can transparently handle validity ranges that are shorter than the finest granularity for the validity of payloads in the database. Besides an access to the central database the framework supports local conditions data storage which can be used as fallback solution or to overwrite values in the central database with custom ones.

1. Introduction

The Belle II detector \cite{1} and the SuperKEKB accelerator are currently under construction at the KEK laboratory in Tsukuba, Japan. The aim of this next generation B factory experiment is to collect 50 times more data than its predecessor Belle \cite{2} and to use this data to search for new physics in a variety of $B$ meson, charm hadron, or $\tau$ lepton decays with unprecedented precision. This requires detailed information on varying calibration and detector conditions to be available when analyzing the data.

2. Software Framework

The Belle II software Framework (basf2) \cite{3} is a C++/Python framework to process the events recorded by the Belle II detector. Events are grouped in so-called runs which mark a data-taking period with stable operating conditions. Recorded events are processed one by one using a sequential set of algorithms called “modules.” As all events are independent from each other we can benefit from current architectures with multiple cores by processing different events in parallel. To share the large amounts of static memory and simplify the implementation of the single modules we parallelize using the \texttt{fork} system call which enables copy-on-write sharing of memory pages between processes.
The Belle II Conditions Database is designed as a representational state transfer (REST) service. Communication is performed by standard HTTP using XML and JSON data. The database manages conditions data on run granularity. Binary objects called payloads which are defined for a certain interval of validity (IoV) are grouped into so-called global tags. The conditions database is by design agnostic to the contents of the payloads and only identifies them by name and revision number. The integrity of all payloads is verified using a checksum of the full content.

The choice of a standardized REST api makes the client implementation independent of the actual database implementation details and allows for a simple and flexible implementation of clients in different programming languages.

One requirement for the conditions client interface is that it needs to also work inside the data acquisition network which does not have a network connection to the outside. To facilitate this the client can also use a local, file-based backend which reads the list of payloads from an index file and the payload contents from files in a given directory. Command line tools written in Python allow to easily download all payloads and IoVs defined in a global tag to be used locally or upload a locally prepared folder to be available for all users.

3. Access of Conditions Objects

The software framework assumes that payload contents are serialized ROOT [4] objects. User access to conditions objects is provided using two interface classes, one for single objects called `DBObjPtr` and one class for arrays of objects called `DBArray`. These class references payload objects, so called `DBEntry` objects in a global store, the `DBStore`. Multiple instances of the interface class all point to the same object. Access to the conditions objects is available in C++ and in Python where the class interface has been designed to be as close as possible to the already existing interface for event level data. Users familiar with the event level storage should have little problems accessing conditions data.

The interface classes always point to correct payload object for the current run, updates are transparent to the user. If the user needs to be aware when the object changes he can either manually check for changes or register a callback function to be notified on change. Figure 1 visualizes the relations between all the entities.

![Figure 1. Relationships for the Conditions Database Interface. The user only interacts with the `DBObjPtr` and `DBArray` classes, everything else is handled transparently and can be configured independently.](image)

The objects in the global store are updated at the beginning of each run if needed. To allow for conditions which change inside of one run the client supports a so-called intra run
granularity: Payload objects which inherit from an abstract base class **IntraRunDependency** can contain multiple objects for different parts of the run. The correct one will be selected transparently depending on certain criteria like event number or timestamp.

4. Creation of Condition Data
To create payload objects we also provide two interface classes to simplify the procedure of preparing the objects, serializing them and committing them to the database. Users can just instantiate one of the creation classes, add objects to them and commit them to the configured database with a user supplied IoV. This includes support for intra run dependency. The possibility for a local file based database allows for easy preparation and validation of payloads as is needed during the calibration of the detector. Once all payloads are aggregated locally they can be easily tested and then uploaded to the central database using a simple command line tool.

5. Storage backends
Payloads in the central database are represented by logical filename, revision number and content checksum. While the central server provides a way to download all payloads directly via HTTP this is not the only possible distribution system. As the payloads are usually a large number of small files it might be beneficial to investigate alternative ways to distribute them. The database client can easily accommodate different storage implementations for the payloads by defining a cascade of storage backends to look for the actual payload content. After downloading the metadata on all needed payloads it will check all configured backends one by one for the needed files. If no storage backend contains the required payload it will automatically try to directly download the payload from the central server.

Currently only file based storage, for example shared filesystems or Cern-VM FS [5] are implemented. However it would be trivial to extend this to other technologies, for example key-value based storage systems like MICA [6]. Also hybrid systems where multiple payload objects are consolidated into larger archives similar to git packfiles [7] could be considered as well. This allows for a highly flexible distribution of the needed payload files which can be adapted as the content of the conditions database evolves.

The fallback solution to simply download missing payloads over http makes it trivial to use

![Figure 2. Program flow for downloading payloads from the central server if it cannot be found by any other storage backend.](image-url)
the software without the need for explicit payload storage configuration. Care has to be taken as many processes might try to download the same conditions files which can cause high load on the server and race conditions in the local filesystem with different processes trying to create the same file. By default the client will try to download missing payloads into a common cache directory. POSIX file locking is used to avoid multiple downloads to the same file. If obtaining an lock fails for any reason or if the checksum does not match the expected value the payload will be downloaded into a process specific temporary file. Figure 2 shows the program flow for this download procedure.

6. Summary
Access to conditions data is a crucial part of the data processing. We have implemented a high level interface to the Belle II Conditions Database using a REST api.

Access for the users is kept as simple as possible by requiring them just to instantiate one interface class to gain access to the correct data for any given event.

Distribution of the actual payload contents is highly flexible and can be expanded or configured easily. Any missing payloads will automatically be downloaded via HTTP so no special setup is required.

References
[1] Abe T et al. 2010 Belle II Technical Design Report Preprint arXiv:1011.0352
[2] Abashian A et al. 2000 The Belle Detector Nucl. Instrum. Meth. A 479 117
[3] A Moll 2011 The Software Framework of the Belle II Experiment J. Phys.: Conf. Series 331 032024
[4] Brun R and Rademakers F 1997 ROOT: An object oriented data analysis framework, Nucl. Instrum. Meth. A 389 81.
[5] P Buncic, C Aguado Sanchez, J Blomer, L Franco, A Harutyunian, P Mato and Y Yao 2010 CernVM a virtual software appliance for LHC applications J. Phys.: Conf. Series 219 042003
[6] H Lim, D Han, D Andersen and M Kaminsky 2014 11thUSENIX Symposium on Networked Systems Design and Implementation (NSDI 14) (Seattle: USENIX Association) pp 429–444
[7] https://www.git-scm.com