Contents lists available at ScienceDirect

Fusion Engineering and Design

journal homepage: www.elsevier.com/locate/fusengdes

Hierarchical dynamic containers for fusion data

David Fridrich[*], Jakub Urban

Institute of Plasma Physics of the CAS, Za Slovankou 1782/3, 182 00 Prague 8 Libeň, Czech Republic

ARTICLE INFO

Keywords:
Data exchange
High performance computing
Integrated modelling
Data processing
Open-source

ABSTRACT

Scientific data can be naturally organized into hierarchical tree structures. One can find several examples of such a hierarchy, for example XML or JSON for general purpose and specifically HDF5 for scientific data. Data types in the EU-IM and IMAS frameworks are also organized into trees. Similarly, experimental data from fusion experiments have tree structures. We propose a new tree-structured data communication platform oriented mainly for integrated modelling and experimental data processing. This effort is part of our "Data Access and Provenance tools for the exploitation of EUROfusion experiments project" EUROfusion Engineering Grant.

Our Hierarchical Dynamic Containers (HDC) library provides a similar functionality for in-core, shared memory communication as HDF5 does for files. Thus, the library enables fast yet easily manageable data exchange within multiple numerical codes (typical examples are integrated modelling workflow actors). HDC enables a zero data copy mechanism even across programming languages and processes. The tree structure is built around key-value storages, managed by a plug-in system. Thus users can choose from a variety of existing solutions of the underlying storage, including private, shared or distributed memory, persistent files or databases.

The core library is written in C++11, highly leveraging the Boost library, and provides bindings for C, FORTRAN and Python (other languages are foreseen). The intent is to provide a clean, simple and universal API, that can be also used for unified access of files or databases. HDC is planned to be used in IMAS and for COMPASS, JET, MAST-U and TCV databases.

1. Introduction

In past two decades, computer hardware has evolved dramatically, CPU performance and memory capacity increased by orders of magnitude. However, the size of synthetic and experimental data has also grown significantly, so nowadays in order to utilize all available computational power, the demands on code developers and the code itself have grown rapidly. For effective work, large amount of scientific data have to be decoupled into smaller tasks distributed among multiple processors or even computational cluster nodes. In such cases, the effectiveness of data exchange becomes one of the main concerns.

However, having a rapid communication library would not suffice. Especially for integrated modelling, the data exchange has to be stable, easily available and fault tolerant. To be really helpful, the library has to be easy to use, which does not mean only to have a simple Application Programming Interface (API), but also to offer commonly used paradigms for data organization. For fusion data, the most natural way is organizing data into hierarchical structures or trees. One can find several examples of tree-like data organization. A number of file formats build on tree structures, e.g. XML, JSON and especially HDF5 [1], which has become de facto standard for scientific data (de-)serialization to files. Specifically within the fusion community, we should mention EUROfusion Integrated Modelling (EU-IM) [2], Integrated Modelling & Analysis Suite (IMAS) [3] projects aiming for integrated modelling and scientific data handling systems like COMPASS DataBase [4] or MDSplus [5,6], where data are also organized into trees. Simple example of such a hierarchical data structure can be seen in Fig. 1, which shows a schema of a simplified magnetics IMAS IDS (interface data structure).

Further, the data communication library should be flexible and runtime oriented in order to prevent repeated recompilation of large blocks of code due to minor changes in the data structure. The whole solution should be easily extensible, scalable and should provide a simple installation procedure. Last but not least, the data communication platform should provide bindings to all programming languages commonly used within the scientific community and should support (de-)serialization from/to common data file formats and databases.

The primary use case we target on is fast in-core data exchange for integrated modelling and data analysis, including EU-IM and IMAS applications. On top of that, an appropriate design can naturally enable to unify in-core communication with data access. Since data are typically loaded from or stored to files or databases, we can reduce the

[*] Corresponding author.

https://doi.org/10.1016/j.fusengdes.2018.02.053
Received 22 June 2017; Received in revised form 15 February 2018; Accepted 15 February 2018

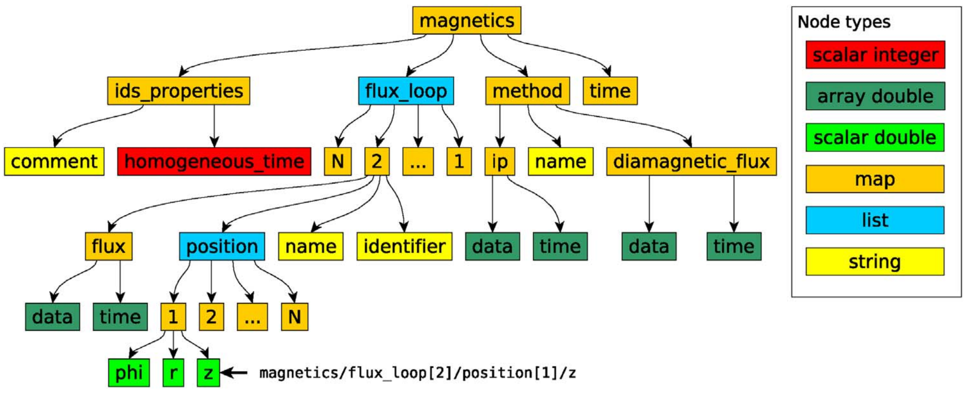

Fig. 1. Diagram showing part of the *magnetics* IDS object structure. For sake of simplicity nodes representing errors were omitted.

number of tools needed for a whole scientific code/workflow. The library has to be convenient and productive even for very basic use cases, such as working with data within a simple, single-developer project, while still well scalable to large projects. This effort is part of our Data Access and Provenance tools for the exploitation of EUROfusion experiments project EUROfusion Engineering Grant.

Finally, we stress that the main intent of our effort is neither to create a new data standard nor a new persistent storage technology. We rather propose to establish a well specified abstraction for existing data models and, at the same time, a software library that will effectively work with this abstraction, i.e. with all the mentioned standards and formats. More specifically, available tools will be reused and only wrapped around in order to provide a single unifying API where appropriate.

2. Design

Based on the design goals and requirements described in the previous chapter and also on the most key design aspects of already existing successful systems like MDSplus or IDAM, we have implemented a data communication library named HDC—Hierarchical Dynamic Containers. This library is described in details in the following text.

The library itself consists of two layers both built on well-known abstractions. The tree structure is used for the frontend, because this concept of data organization is well established in scientific community. For the data storage layer (backend), we use the key-value store (hash map) abstraction. Multiple existing solutions for storing key-value pairs can be found, each of them providing a unique set of features. There are very fast hash maps, intended to be used within private or shared memory, or slower systems offering communication among multiple computers (i.e. via network) or data persistence. Moreover, the APIs of key-value stores are concise and consist of small number of functions or methods, which enables to quickly change the backend using a plugin system. Hence, one can always select the proper backend for a given purpose. The HDC library can be easily extended to support more storage backends using such a plugin system. The extension of supported data storages is straightforward: most data storage plugins would fit into a hundred lines of C++ code. In the current state, the used storage plugin can be configured from a JSON configuration file, command-line options or directly from the code in run-time. Support for using different plugins for different parts of data is envisaged.

Currently, the HDC library supports two key-value storages: `std::unordered_map` for rapid data exchange within the private memory and *Yahoo MDBM* for shared memory, inter-process communication. In run-time support for several other backends, allowing operation in distributed memory systems like computer grids or cloud computing platforms, will be added in the near future.

The HDC library features a sufficient, yet not exhausting set of supported data types similar to Python, i.e. list, (ordered) dict, empty node and a subset of NumPy [7] data types. One can use scalars and arrays of commonly used numerical types (signed and unsigned integer, float, double, Boolean) and strings. In the tree hierarchy, these types represent leaf nodes. Parent nodes can be either an ordered map or a list. The type system also includes an empty (null) node type and reference to data with external ownership. This is fundamentally an identical type system as in Conduit [8].

Note that flexible, cross-language function interfaces can be built using HDC. This would be similar to Python, which uses `tuple` and `dict` types to work with variable number of arguments or keyword arguments. This could, for example, complement systems like `f2py` or simplify the `fc2k` machinery in IMAS.

Another key aspect of the design is the orientation on runtime usage. In contrast with UAL [9], which uses complex, pre-compiled C structures or FORTRAN derived types for data organization into three hierarchy, HDC adopts the approach used in scientific databases or HDF5 files, where the data are referenced by string uniform resource identifiers (URIs). In the HDC case, this string will be most often the path within the hierarchy, like in HDF5 files or POSIX file systems. For example, the path of the data for *z*-component of the first array member of position of second `flux_loop` in `magnetics` structure illustrated in Fig. 1 can be written in the following format: `magnetics/flux_loop[2]/position[1]/z`.

For users, our approach provides an additional flexibility, development agility and significantly reduced need for bookkeeping of interdependencies of source codes (APIs) and binary objects (ABIs). For HPC developers and system administrators, the approach significantly reduces the development and maintenance time and the overhead needed for any change in the data model because there is no necessity to recompile the library and, subsequently, the users' codes. The compliance with given data specifications, such as IDS or CPO types, can be still maintained and even enforced if necessary by implementing an appropriate validation plugin. Note that run-time validations offer more flexibility over compilation time, static type checking, because the validation system has access to the actual data.

Using the string for data referencing also enables easy extension of functionality, including support for multiple different storages at a time or accessing data from files. For example, for a direct HDF5 file data access, one could specify the protocol and the file path (this is not yet implemented in HDC), e.g.: "`file+hdf5:///path/to/data/file.h5:group/dataset`".

Although the presented library is especially designed for fast runtime data exchange, it should also provide a way to make the data persistent. Currently, JSON and HDF5 (de-)serialization options are available. The complete data persistence is currently possible for MDBM plugin because the MDBM database can be saved into a file, which, together with its path and the key of the root node, makes the data completely persistent. This is, however, not well suited for long-term storage as the stored data are system/version dependent.