

# GLED – a ROOT based framework for distributed computing and dynamic visualization

M. Tadel\*, CERN, Switzerland

## Abstract

GLED is a ROOT-based OO research framework for fast prototyping of applications in distributed and multi-threaded environments with support for direct data interaction and dynamic visualization. This paper presents an overview of main features of GLED, including paradigms governing object habitat, object life-cycle, object aggregation, remote data synchronization and remote method invocation (OO version of RPC). Hierarchical server-client model is used to bind several computing nodes in a tree structure. Overview of GLED's support for multi-threaded execution of user code is made and OpenGL based 3D rendering infrastructure of GLED is briefly described. GLED is used by the ALICE collaboration for research on visualization technologies.

## INTRODUCTION

GLED is, in its core, a framework for management of object collections in a distributed, multi-threaded, multi-user environment. It allows for a fast development of class libraries for scientific computation & visualization, distributed applications for collaborative object management, protocols for intra/inter cluster communication and data transport as well as of advanced visualization systems for single and multi-user applications.

GLED is built on top of the ROOT data-analysis framework [1] and thus inherits its core features, including object serialization, versatile I/O infrastructure (files with inner directory structures, trees, rootd), networking, CINT – the C/C++ interpreter and a rich set of data analysis tools. GLED objects are derived from a common base `ZGlass`, which is in turn sub-classed from ROOT's `TObject`. GLED classes are instrumented with bindings for creation & execution of RPC requests, reference counting and auto-generated object-centric GUI. All the low-level code required for this instrumentation is generated automatically by the PROJECT7 parser & code-generator. It is an important part of GLED, similar in nature to `rootcint`, the dictionary and streamer generator of ROOT.

As a distributed run-time environment GLED offers object-collection management in a hierarchic server-client structure of nodes, management, routing & execution of RPC requests and provides cluster introspection & management interface. GLED can be dynamically extended via loading of plugin-modules called *libsets* (each libset is composed of core, GUI and OpenGL libraries). Develop-

ment of user-code is facilitated by a set of PERL scripts for creation of libset and class stubs.

The rest of the paper gives a brief overview of selected features of the GLED system; for a more thorough presentation see [3].

## OBJECT DATA-MODEL

Main features of the GLED data-model are determined by the distributed and multi-threaded nature of the object environment. To assure consistency of object collections across a cluster a strict policy is applied to object creation and destruction as well as to execution of methods that modify the object-data. Additionally, object access by multiple concurrent threads is regulated by a hierarchical mutex structure which allows per-object locking and separates frequent framework-related operations from the userland program flow. The situation here is less transparent as threads can perform operations with varying requirements for CPU processing and data-access. To avoid deadlocks and assure maximal possible parallelism different mechanisms and techniques must be used to accommodate different object-usage patterns (e.g. internal per-object request queues or replication of object-data into local storage).

Persistent data-elements of GLED classes should be either basic data-types or ROOT objects equipped with serialization functionality and dictionary. Constituent ROOT objects are owned by the GLED object which provides all necessary maintenance.

Special care is needed for referencing among GLED objects and there are two mechanisms provided to this end: *links* are used to reference a single object and implemented as smart-pointers; *lists* are collections of object references and they are GLED objects in themselves, derived from an abstract container base-class (`AList`). Internal storage mechanism is not imposed but the implementation must provide forward iterators. The accompanying infrastructure automatically performs reference counting (with optional back-references) and allows proper handling of object graphs during serialization, GUI building and 3D visualization.

### Method execution

Here we consider methods as member functions that either change the object data (including the trivial set-methods) or require consistent object state to perform their operations. As it must be possible to execute the exact same method with all the arguments on different nodes as well

---

\* matevz.tadel@cern.ch

as to delay its execution due to temporary unavailable object resource GLED introduces the concept of *method invocation request* (MIR). MIR is a serialized set of instructions that allow GLED run-time environment to execute a given method (sub-classed from ROOT's TMessage class). Bindings to create and to execute MIRs are auto-generated by PROJECT7. The programmer must export a method by specifying a corresponding pragma in the class declaration and thus exposes a well defined sub-set of the object's interface to the GLED system.

MIR is composed from a header, streamed method arguments and an optional, arbitrary datagram. The header contains routing information that is relevant in the cluster context (the identities of issuing node and user and of the target node), fully quantified method that should be invoked and, of course, the identity of the target object. At execution time this part is processed by the GLED infrastructure. Following are serialized arguments with special care taken to properly serialize GLED-object pointers, as they have to be uniquely identified within a cluster. These are processed by the auto-generated `void ExecuteMir(ZMIR& mir)` method and the wrapped method is thence called with locally created copies of the arguments. A pointer to the MIR is stored in a thread-specific data segment and is therefore accessible to a programmer to be able to deduce execution context from within his code.

The last part of a MIR, the arbitrary datagram, can be used for data transport among nodes. There is no rule how to use this functionality, but the programmer is obliged to implement consistent creation and processing codes. ROOT objects are particularly well suited for being data-carriers in such situations.

### Access control

GLED's infrastructure for MIR processing also performs authorization checks during unfolding of the MIR header. Identity of the caller is compared against a set of *MIR filters* that are attached to the execution object. The provided infrastructure is general enough to allow construction of arbitrarily complex filtering schemes (including group identities and access lists). The authorization infrastructure is implemented as a set of GLED classes and thus it can be controlled by standard GLED mechanism (MIRs in a cluster context, auto-generated GUI). Details are described in [4].

## HIERARCHICAL SERVER-CLIENT MODEL

So far little has been said about the actual habitat of GLED objects, although its existence has been implied all along. The central role is played by a *saturn*: it manages network connections, performs MIR routing & execution and provides object registration facilities. One instance of saturn is present on each node in a GLED cluster and depending on its position in the node hierarchy it plays the

role of a master server (the top node, called *sun-absolute*), a proxy (intermediate nodes) or a client (leaf nodes). Each intermediate node can export additional objects to lower-lying nodes and thus plays the role of a proxy (in relation to his master) and that of a server to its own sub-tree (see Fig.1).

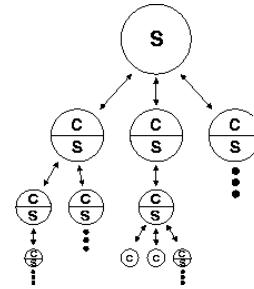


Figure 1: A scheme of node connectivity in a hierarchical server-client model. **S** (**C**) denotes a server (client) part of node's object space. The top-node is sun-absolute.

To be able to identify objects across the GLED cluster each object is assigned a unique identifier in a form of a positive integer. Each saturn in the hierarchy reserves a low-lying chunk of address space for its use and assigns it a *king* object to administer over it. Saturns of the next level can in turn, after connecting to their master and before accepting clients, reserve their own chunk of the object space which thus becomes their server space, visible to them and to their own sub-nodes. Thus, the only object space visible on all nodes of a given GLED cluster is that of sun-absolute. Additionally, a part of object address space remains unclaimed on each saturn: object stored there are visible local to the saturn itself and can be used to store data related to visualization, GUI structures or as temporary object store for computational algorithms. Cluster-wide layout of object address space is presented in Fig.2.

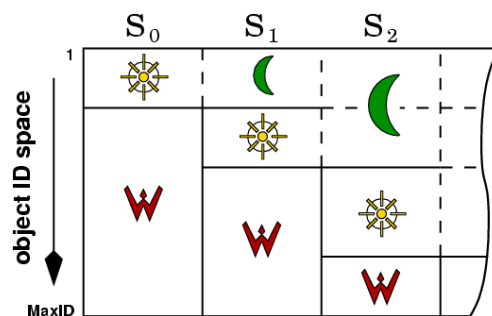


Figure 2: Progressive allocation of object-space while going downwards in the node hierarchy.  $S_0$  represents the master server and  $S_i$  the  $i$ -th level node. The sun, moon and fire symbols represent, respectively, server, proxy and local object spaces.

To allow partial exposure of server spaces to lower saturns each king further partitions its object space into smaller chunks and delegates *queens* to actually populate

it with objects and manage them. Queens perform object instantiation and deletion, perform serialization and deserialization and provide additional access control for all their objects. Queen-spaces are the atoms of replication: a client can choose if it needs to hold a copy of a given queen but after that it is required to hold the whole collection. Mirroring of queens is performed in an interplay between the involved saturns, the king and its mirror copy. Relation of dependence can be created from queen  $Q$  to  $Q'$ , provided that  $Q'$  is in the server or proxy space of the hosting saturn. This has two implications: first, the objects from  $Q$  can reference objects of  $Q'$  and second, queen  $Q'$  will be automatically mirrored prior to mirroring of  $Q$ .

Kings and queens are implemented as GLED classes, together with helper classes `SaturnInfo` and `EyeInfo` that represent a node and a user respectively. Thus one can say that hierarchical object-space & cluster management is implemented in GLED itself, with a little help from the saturn. MIRs are used extensively throughout these classes and one can easily derive from the queen class to implement custom object management and addressing.

## MULTI-THREADING

GLED as an application relies heavily on usage of threads. Saturn itself uses them for accepting of new network connections, routing, execution and forwarding of MIRs as well as to provide a delayed MIR execution service (similar in spirit to the `at` UNIX command). Additionally, ROOT's main event-loop with CINT console runs in one thread and GLED's GUI (using the FLTK toolkit) in another. This simplifies the structure of the event loop as independent tasks can be managed at their natural location and not centrally.

Users have two options to execute their code in separate threads: the first is simply to request execution of a MIR in a detached thread, the second is to use the `Eventor` GLED class. The latter one is suitable for tasks that need to be performed in a loop.

### *Detached-thread MIR execution*

Any method can be flagged for detached execution by specifying an appropriate pragma at its declaration. MIR will get the detached execution flag set at creation time and the saturn will automatically spawn off a dedicated thread when executing such a MIR. At the end of the execution the thread will terminate. This mode of operation should be chosen for tasks whose running time is either long or indeterminate due to reliance on availability of certain resource. In contrast to ordinary MIR execution, here the object is not locked on entry. Thus the programmer is required to properly invoke object-locks when appropriate.

### *Eventor-Operator tree*

The `Eventor` base-class is a top-level control for user thread execution providing interface to start/stop and sus-

pend/resume a thread. `Eventor` itself provides event-loop control functionality and exposes virtual functions to be called before/after the run and each event. The code that will be actually executed is determined by objects of class `Operator` that are inserted into eventor-list. Operators are lists themselves, so trees of arbitrary depth can be constructed. The user-code is put into the virtual, exception throwing `void Operator::Operate(Eventor*)` function. Each operator provides control over descent into its own operator sub-tree and thus it is easy to turn sub-algorithms on and off by using GLED's auto-generated GUI. Sub-classes usually introduce further data-members that help users properly steer the algorithm chain.

Eventors can be run in two modes: on all nodes of the cluster or only on a specified node. The latter case is interesting when event processing takes a long time and produces a manageable amount of resulting data that needs to be broadcasted back to the participating part of the cluster. This is a question of balancing CPU versus bandwidth usage.

`Eventor` can also be instructed to enter the operator-tree traversal at regular time intervals. System-time or fictitious internal time can be used as a time source. Current time is stored in the eventor itself and is made accessible to operators during the traversal. Special operator sub-class `TimeMaker` can be used to additionally transmute the incoming time by a specified formula or to limit it into a specific range by clamping it or by looping it into a triangular, saw-tooth or sinusoidal waveforms. This is important if one uses the operators to perform dynamic animation.

## 3D RENDERING INFRASTRUCTURE

GLED uses OpenGL as its default renderer. While the infrastructure is general enough to allow different renderers only a minimal implementation for POV-Ray rendering is being considered. Parallel class structure is used to allow programmers to implement actual rendering functionality. E.g. class `Sphere` implies usage class `Sphere_GL_Rnr` whenever it needs to be rendered with GL. The fact the rendering code is hand-written gives a lot of flexibility to the programmer and allows him to write optimal code with minimal data-transfer overhead. On the other hand, one can also use the set of the available graphics classes to construct his visualization.

The render driver traverses the object hierarchy, automatically instantiates renderers and calls lower-level functions in the proper sequence, as dictated by the object state. Links to other objects can be used to invoke rendering-engine state changes (like lighting, texturing, alpha-blending) during the traversal. This is handy as several objects can share the same set of rendering attributes.

As object properties can be changed either from the GLED GUI or from user-threads, it is easy to construct scenes with dynamic visualization. The time-generation infrastructure described in the previous section can be used both for real-time animation (using system-time) and for

key-frame animation (using internal time with fixed steps) needed for encoding of movies. The transition between the two can be easily made by modifying two parameters in the `Eventor` class.

GLED also supports rendering into an overlay: objects rendered there receive all standard keyboard, focus and mouse events. This allows interactivity to be achieved on the 3D viewer level and makes it very easy to set-up full-screen interfaces for demonstration purposes.

## USAGE OF GLED

GLED is used as an advanced visualization engine in the ALICE collaboration, both to explore different modes of event-data visualization and to prepare dynamic visualizations and animations of high-energy collisions and distributed processing of data on the grid (see [5]).

While there has been no real-life usage of GLED for distributed computing a set of simple applications and stress-tests have been implemented, including basic cluster monitoring (n-tupling load-avgs, mem/cpu usage), execution of local commands wrapped in Gled threads, a set of numerical demos (ODE integration, stochastic minimization) and data-transfer benchmarking.

### *ALICE visualization prototype*

With all the infrastructure of GLED it was relatively easy to prototype large parts of ALICE Event Visualization Environment and to test different visualization modes for simulated and reconstructed data as well as to prepare advanced rendering techniques for display of raw-data. In particular it proved of great value to have automatically generated GUI as we were able to add additional visualization parameters into the application and test them interactively in matter of minutes. This helped us evaluate the performance of commodity hardware for the challenging task of visualizing lead-lead events with more than 60.000 primary tracks and event-data size of the order of 80MB for raw-data and 1.5GB for full simulation/reconstruction pass. As GLED is a ROOT based framework it is straightforward to port the developed code into the more restricted environment of the AliROOT framework (undergoing work).

GLED is still being used for production of outreach movies of simulated lead-lead events and for preparation of high-resolution images for color plates in ALICE publications.

### *Visualization of ALICE@grid*

GLED was successfully used for dynamic visualization of ALICE distributed computing infrastructure at the SuperComputing '04 and '05 conferences. We prepared basic elements for visualization of world map, sites, data dispersal, connections between sites and processes, data-exchange and job progress. In 2004 a PROOF master and a ROOT based AliEN client were running from withing GLED and the main focus was on showing the progress of

a distributed analysis using AliEN for performing file location queries and maintaining PROOF slave presence at the available sites while PROOF was doing the real analysis. In 2005 all the information from AliEN and from the jobs themselves was stored in a MonALISA repository and retrieved by GLED via a JAVA client running as a separate process. This time the emphasis was on showing the large scale grid infrastructure with many jobs available for real-time monitoring or replay during the demonstrations. We also used the overlay-based OpenGL GUI that allowed the visitors a direct interaction with the demo.

## CONCLUSION

While GLED is not a mature product it offers a viable alternative for research projects that need to instrument their code for distributed operation or with instant GUI and dynamic visualization. The procedure for this is relatively simple: one creates a new libset, plugs in the project code (either directly or as an external library) and wraps the API and data-structures in a set of GLED classes. Additionally, one might find an unexpected benefit in data-analysis facilities of the ROOT framework.

Further development is focused on generalization of object-graph serialization procedures allowing natural handling of references to external object-sources and on removing the restriction of tree-like topology of GLED clusters.

## ACKNOWLEDGEMENTS

The author would like to thank ALICE computing group for accepting GLED and for encouraging its further development. GLED would never exist without ROOT: many thanks to all involved, and in particular to Fons Rademakers for his support.

## REFERENCES

- [1] R. Brun and F. Rademakers, *ROOT – An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also <http://root.cern.ch/>.
- [2] M. Goto, *C++ Interpreter - CINT*, CQ publishing, ISBN4-789-3085-3.
- [3] M. Tadel, *GLED – an Implementation of a Hierarchic Server-Client Model*, Applied parallel and distributed computing (ed: Y.Pan, L.T.Yang), Vol.16, Nova Science Publishers, 2005. <http://www.gled.org/docs/gled/>
- [4] M. Tadel, *Authentication & Authorization Infrastructure of GLED*, <http://www.gled.org/docs/auth/>.
- [5] <http://www-f9.ijs.si/~matevz/gled-movies/>.
- [6] The GLED project homepage: <http://www.gled.org/>.