GLED – an Implementation of a Hierarchic Server–Client Model

Matevž Tadel*

September 11, 2003

Abstract

GLED is an implementation of a hierarchic server–client model written in `C++`, having an outside form of an object oriented framework. Master server exposes its object space to its first-level clients which in turn provide mirroring facilities to second-level clients and also export their own object spaces. Each client therefore plays a role of a client, a proxy and a server. Infrastructure for propagation and broadcasting of method invocation requests is provided and is general enough to allow for an efficient access control. Synchronization of object spaces is achieved via streaming of object graphs and a time-ordered delivery of method invocation requests. Objects can acquire their own threads to perform different tasks such as computation, data acquisition & analysis or dynamic visualization. The obtained results can be pushed to other computing nodes or storage devices. As threads are spawned and controlled by standard method invocation requests, schedulers & job control mechanisms spawning across a whole GLED cluster can easily be implemented. On viewer level, GLED features automatically generated GUI widgets and a sophisticated 3D rendering infrastructure.

1 Introduction

The basic ideas of GLED are stemming from recent developments in high energy physics (HEP) computing as well as from modern information technologies (`web` & `grid`) and concepts of cyberspace (interactive computer graphics, virtual reality, multi-user online games). During the last ten years the paradigm of object oriented programming was making its way into HEP, both by migration of experiments from `fortran` to `C++` and appearance of ROOT data analysis framework [1]. The methodology of a typical physical analysis didn't change much though: it still consists of preparing the input, running the batch job(s) and post-processing & visualization of output. While ROOT simplifies transfer of data from batch job processing to analysis framework, the problem of unification of all the stages of analysis remains only partially addressed and requires, for its efficient solution, high computing skills on the part of a user. This is not a serious problem for a large, mature analysis which takes several days to complete but in early stages of analysis preparation users typically run many small jobs and thus aspire for a fast response of computing infrastructure with the least possible involvement in technicalities.

With the appearance of relatively cheap super-computers in form of computing clusters and powerful desktop computers the time spent maintaining the working environment is contrasted out far more than ever before. Large computing farms with modern schedulers can easily handle jobs with minimal time requirements to provide an unhindered analysis & algorithm prototyping while desktop computers can store data in an order of ~100Mb (as well as obtain it via its network

*`Matevz.Tadel@ijs.si`

connection) and provide enough computing power for its processing and visualization. One of the basic aims of the GLED project is to address these new possibilities and integrate algorithm development, input preparation, job execution, data visualization and user-to-user communication in a common environment.

Another element of modern computing that has likewise grown is the size of datasets and databases being processed, thus putting a heavy load on all data transport layers, from networks to permanent storage devices. For WAN-s and inter-cluster communication the problem has been effectively addressed with the introduction of `grid` technologies, while LAN-s within large computing clusters with centralized storage systems are still posing a challenge as many improvements minimizing the data transfers can be envisioned. A hierarchic structure of computing nodes interwoven with data storage and communication nodes seems an obvious solution and it provided one of the models for hierarchic server–client structure of GLED. By virtue of the hierarchic structure, the inner computing nodes do not need a direct WAN access. This architecture might also be of prime interest for distributed multi-user interactive environments where hierarchic structure allows for balancing of WAN traffic and local CPU usage.

1.1 The main design features of Gled

The GLED project aims at building of an object oriented framework/toolkit by extending the notion of a C++ class. It provides the following extensions to the C++ class paradigm:

1. **Serialization of objects.** ROOT's infrastructure for serialization of objects provides a firm base for storage of objects and their transmission over the network.
2. **Serialization of method calls** allows for a transparent implementation of remote method calls with objects acting as the natural elements that generate/process the requests.
3. **Automatic generation of GUI classes**. The auto-generated widgets always send serialized method calls to the central message processing unit and are therefore decoupled from the actual objects they are representing. The GUI classes are a convenient way for viewing and editing of objects.
4. **Rendering classes** coupled with GLED's rendering infrastructure provide an alternative representation of objects. It can be used to build advanced visualization systems or as a flexible method for representing and interacting with large object collections.

Furthermore, GLED has a low-level support for object aggregation (lists and smart pointers) allowing an easy construction of arbitrarily complex object structures (graphs). The object graphs can be shared across a cluster with the hierarchic server-client paradigm assuring a consistent object state and proper method call propagation.

From architectural point of view GLED is designed as a modular application with a minimal kernel, providing base classes and hierarchic server-client code. User classes/code can be imported into a running kernel by dynamic loading of appropriate shared libraries.

1.2 Development goals

The development of the GLED framework focuses on elements that allow for fast prototyping and implementation of:

- **class libraries** for scientific computation and visualization,

- **distributed applications** for scientific computing and collaborative object-space access/management,
- **intra/inter cluster** communication and data-transport & addressing protocols and
- **advanced visualization systems** for single and multi-user applications.

Object & method call serialization, hierarchic server-client architecture and GUI & data presentation elements of GLED also make it an interesting platform for projects seeking voluntarily contributed CPU power for CPU intensive distributed computations (e.g. SETI@Home [2] and Folding@Home [3]). The wide availability of broadband network access for home PCs is turning them into powerful computational and data analysis engines, suitable also for tasks with high demands on network throughput. The hierarchic server-client model can be effectively used to minimize the network traffic, provided that the topology of a cluster matches that of a network.

While the main development goal of GLED is to provide a generic platform for scientific computing and prototyping of `grid` protocols, many of its features might be of interest to a wider public. In particular, we see two possibilities for extending GLED in this direction.

1. Data-base interfaces coupled with GUI and rendering support of GLED would allow for an easy implementation of advanced information systems.
2. The hierarchic server-client model, object & method serialization and rendering capabilities make GLED an excellent platform for implementation of virtual reality systems. In the extreme case GLED could even be used as a multi-player game engine.

Both of these extensions would further enhance GLED for its main purposes. It is our intention to attract open-source developers from external communities. Their contribution and wider interest for the project would also allow for an efficient dissemination of `grid` technologies to a wider public.

2 Overview and Terminology

The basic elements of GLED are objects, specializations of a class `ZGlass`, exported into a hierarchic server–client system. The common base class provides services for communication with the GLED system and, together with the PROJECT7 code-glue generator, enables its instances to be serialized in a context of an object graph as well as to send and execute *method invocation requests* (MIR). Compared to RPC [4], MIRs in GLED represent a distributed RPC implementation of a "call/no-wait" variety with internal rules that allow for propagation of MIRs through the hierarchy of a GLED cluster. ROOT's streaming mechanism guarantees interoperability of architectures and in this context corresponds to external data representation standard (XDR [5]) of RPC.

GLED objects are memory-resident and optimised for fast modification and immediate replication. Nodes in a GLED cluster share data and collectively modify it. If a given node desires access to some data it has to mirror it first and after that it receives and sends MIRs to keep the data in synchronization with the server. This is quite unlike LDAP [6] which is optimised for massive concurrent queries with slow modification and lazy replication. On the other hand, GLED objects can be imported from databases, either as static data, as query results or as dynamic data that can be changed and possibly reinserted into the database.

To be able to explain the details of functioning and habitat of GLED objects, an overview of the hierarchic server–client architecture and basic properties of the GLED system will first be made.

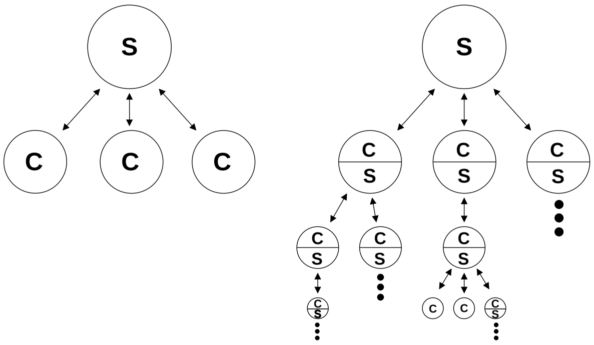

Figure 1: Comparison of standard (left) and hierarchic (right) server–client model. Circles represent computer nodes, while labels **S** and **C** stand for *server* and *client* respectively.

2.1 Towards a Hierarchic Server–Client Model

The basic idea of a hierarchic server-client model lies in extension of a simple server–client communication graph to a more general tree structure (Fig. 1). The most notable difference, beyond the appearance of multiple levels, is that each client becomes also a server for its clients and a proxy in respect to its server (as well as for all higher servers on the way to the master server). Separation of server spaces of clients belonging to one server is in fact the feature that makes the hierarchical model interesting: it allows nodes to share data selectively, making it visible only to the nodes that need access to it. The topology of a tree is arbitrary and can change dynamically. In practice it is dictated by the structure of the problem being addressed and should provide optimal functioning of the nodes while minimizing data transfers among different levels of the cluster hierarchy.

Another point to notice from the above figure is that a master server and pure-client nodes are just degenerate cases of the general node behaviour. This presents a drastic departure from the standard server-client model. To emphasize this difference and to avoid confusion, the term *Saturn* was chosen to signify an operational node in a GLED cluster. The naming is based on the analogy with planetary system and propagation of light: light emitted by the planet Saturn is partially the reflected light from the Sun and the rest is produced by the processes proceeding under the veil of the planet's atmosphere. The master server is called either a *zeroth-level Saturn* or *Sun Absolute*. Additionally, symbol $\mathcal{S}$ is used for any Saturn, $\mathcal{S}_0$ for the Sun Absolute and $\mathcal{S}_i$, $i > 0$ for an *i-th level* Saturn.

Since objects need to be uniquely identified across different nodes, they are numbered with positive integers spawning an *object ID space*. Intervals from the available set are claimed by Saturns in succession as they get attached into the system. Fig. 2 shows an example of object space partitioning for a chain of three Saturns. The linear partitioning with pre-reservation of desired ID-space interval minimizes efforts associated with routing of MIRs without the loss of generality.

Objects in the server space (or *sun-space*) of $\mathcal{S}_0$ can be seen on all GLED nodes. MIRs directed at them from lower Saturns (or *moons*) are always routed up to $\mathcal{S}_0$ where the requests are checked and, if they are accepted, executed and broadcasted to $\mathcal{S}_1$ Saturns. They in turn execute the MIR and broadcast it further down the hierarchy. For a $\mathcal{S}_i$ Saturn, the object spaces of all higher Saturns are effectively merged into a common ID-space partition, called its *moon-space*, providing the proxy service. This furthers the light analogy: the light emitted by a Saturn on behalf of its higher Saturns is the reflected light, just as we see the Moon in the reflected light of our Sun. A Saturn has no executive power over its moon-space although operations (threads) issuing from it can perform tasks on behalf of other Saturns.

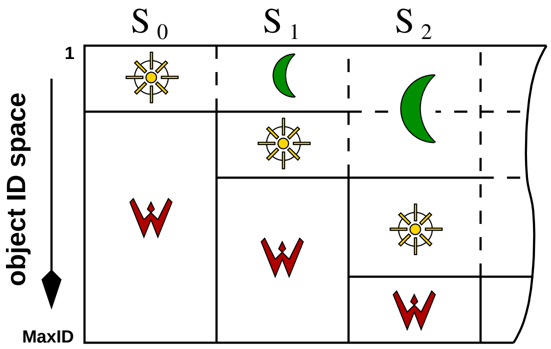

Figure 2: A scheme of object ID space partitioning. For each Saturn the sun symbol represents its server space, the moon symbol its proxy-client space and the fire symbol shows the local space of the Saturn.

The same holds for each Saturn: it is the master of its sun-space and all MIRs affecting it must pass through its execution control prior to their execution and broadcasting.

Sun-space of each Saturn is further subdivided to allow for its partial exposure to lower Saturns. Migration of objects between different object-spaces is also possible. This features are discussed in section 4.

Another important feature of such partitioning is the natural arising of a *local object space* (or *fire-space*) on each Saturn (Fig. 2). It contains objects pertaining to local computation and viewing. Fire-space extends over the object-space following the Saturn's reserved sun-space. As the names suggest, objects belonging to fire-space are entirely local (light emitted by a simple fire can not reach other planets). Since global and local objects share the same ID-space, local objects can easily reference the global ones to modify their own behaviour. Also, a user is provided with the same interface in both cases.

2.2 The Main Aspects of Gled

GLED is built upon ROOT [1] and thus inherits its versatile I/O infrastructure (access of local and remote files with inner directory structure) and a rich set of data storage, analysis and presentation functions. CINT [7], a C/C++ interpreter, is also a part of ROOT's heritage and presently plays the role of a scripting language.[1]

As mentioned before, GLED-enabled classes stem from the common base `ZGlass`. Following the analogy with light any GLED class can be referred to as a *glass* (material, not a vessel) and any instance as a *lens*. Infrastructure enabling lenses to be fully operational is rather elaborate and could as such pose a heavy load on a programmer. To bypass this obstacle, definitions of glasses are preprocessed by two parsers/code generators:

- ROOTCINT (part of the ROOT system) produces a dictionary, serialization method and interpreter bindings;
- PROJECT7 (part of GLED) generates accessor methods, methods for streaming and processing of MIRs, connectivity introspection methods as well as GUI components for a given glass.

[1]Currently CINT is not thread safe and therefore limits GLED's performance to a single scripting environment per Saturn. Usage of a loosely coupled scripting language (e.g. `perl6`) would allow several concurrent scripting sessions.

Both produce code that is in part included back into the class definition. Therefore some of the class semantics is implicit in data and function member declarations.

MIRs are always enclosed in a *context*, specifying the target lens, the identity of a caller and all lens-type arguments. This allows for authorization and dependency checking prior to complete unfolding of the request and provides a transparent way for creation of requests originating from a GUI.

To simplify the extension of the GLED system, build process and component deployment, glasses and their supporting structures are grouped into *library sets* which are dynamically loaded into a running system.

Viewing part of a library set is separated from its main part, following the general rule that the objects themselves should not know how and when they are being observed. Each Saturn can accept local viewers, or *Eyes*. They can be individually authenticated to allow different access privileges. Eyes instantiate lens-views, provide MIR sending facility, receive change notification messages and distribute them to targeted viewing fragments. Viewing classes have a read-only access to glass data and send MIRs to their Saturn to actualize the desired changes or to initiate actions.

3 The Data Model

In GLED the basic data element is an object of class derived from the common base class `ZGlass`, i.e. a lens. In any OO framework the functionality of a base class determines its programming paradigm: it imposes certain structure on programs by providing a means for static data structuring and models of algorithm invocation. In dynamic systems data can influence functioning of algorithms and algorithms can change data as well as its structuring, therefore making a distinct separation impossible in certain cases. GLED is rather permissive in this respect and does not enforce strict control over object access and method execution. Rather it provides infrastructure for object locking (each lens can be individually locked), setting & accessing of data and MIR processing and execution. This mechanisms must be used by the programmer to insure object graph consistency.

3.1 Data Members and Lens Aggregation Methods

The basic model for a glass is that of one automaton in a multi-cellular state-machine with dynamic topology. Data members of a glass represent its internal state. If they are declared as *exported* they are synchronized among GLED nodes that share an instance of this class. Non-exported members hold data that is local to each Saturn and they are not included in the data-stream created during object serialization. They can also serve for data that can be computed locally or as a storage for intermediate results and structures needed for visualization.

Data members can either be of simple types or they have to be descendants of ROOT's base class `TObject` which provides streaming facilities.[2] Data members that can be explicitly set by value must provide the interface in a form of an appropriate *set* method. Unless those methods have additional side effects they can be generated automatically by the PROJECT7 generator. Further, set methods must also be available in their streamed versions so that the elementary changes can propagate across the hierarchy of Saturns. They can always be generated automatically as they only produce a MIR containing the streamed value for the data member in question. This way the problem of setting the values of data members is directly mapped on the problem of method execution. Execution and propagation of MIRs is the subject of next subsection.

[2]ROOT provides a rich set of classes for data storage, among others also matrix and vector classes. Creation of custom classes is trivial and is supported by the GLED build process.

A special treatment is provided, on all levels of GLED, for an exported pointer to a *glass*, called a *link*. Links are the first option for aggregation of glasses and they extend a notion of object state into the realm of topological connectivity.

The second aggregation method is provided by a *glass* `ZList`. A list does not own its elements, it simply references existing lenses. The list interface is suitable for organization of ordered collections with frequent additions and removals of elements. It is flexible enough and can easily be extended over the same basic interface.[3]

Using both aggregation methods arbitrarily complex object graphs can be constructed. Also, these graphs need to be synchronized over the GLED cluster, often in a form of sub-graphs connected to an already existing object system. Therefore a versatile interface is provided for marking of sub-graphs, their streaming and rebuilding. Structures encapsulating a sub-graph are called *comets* to be reminiscent of celestial objects that traverse a planetary system or even leave it altogether. Comets can be stored into files or sent to another GLED node for implantation in one of its object spaces. Advanced deep-copying can also be achieved using the comet mechanism.

The base class includes reference counting facilities, mandatory for streaming and management of multiply connected graphs. Methods for modification of links and lists take care of that automatically.

3.2 Data Manipulation & Method Execution

A fair number of methods of any class usually provides a means for changing the static data (either in an object itself or in an object that is accessible from it). In GLED, special care must be paid to methods that change the *exported* data of a glass. Such methods must be executed on all Saturns holding a copy of the lens in question in order to retain consistency of object-spaces. This effectively means that the method call needs to be serialized and delivered to all nodes that must execute it.

Serialization operators for basic types are provided by ROOT and streamers for user-defined classes (as well as for glasses) are automatically generated by ROOTCINT. PROJECT7, in turn, generates wrappers for creation and execution of MIRs. Its operation is guided by instructions encapsulated in comments following the declarations of methods. This provides a natural extension of `C++` semantics without affecting its syntax. ROOT in itself already uses such comments for control of streaming.

3.2.1 Set methods

The *set* methods for exported glass members fall directly into this category. To minimize the amount of typed code these methods as well as methods for generation of their MIR-producing counterparts can be generated automatically. Examples of instructions for PROJECT7 parser are shown in Fig. 3, top part. E.g. a name of a lens is stored in a member `mName` and the `Xport{GS}` declaration (`Xport` ∼ *eXport*) in the comment field following it induces the generation of methods `GetName()`, `SetName()` and MIR-producing `S_SetName()`. Links as rather special members have to be declared as such (e.g. `Link{}` for member `ZNode::mParent`). The generated code is shown in Fig. 4.

3.2.2 Ordinary methods

For other methods only their MIR counterpart needs to be generated (declaration `Xport{E}`; `E` ∼ *Exec*). MIRs contain a method context intended to hold glass-type arguments for the call. A good example of methods that use this facility are list modification methods (e.g. Fig. 3: `ZList`). Declaration

[3] Implementation of `ZList` is based on STL `list` container and can be used with STL algorithms.

```
class ZGlass : public TObject {
  ...
  TString    mName;      //  Xport{GS} 7 Textor()
  TString    mTitle;     //  Xport{GS} 7 Textor()
  Bool_t     bActive;    //  Xport{GS} 7 BoolOut()
  Short_t    mRefCount;  //! Xport{G}  7 ValOut(-width=>4)
  ...
};

class ZList : public ZGlass {
  ...
  virtual void Add(ZGlass* g);                       // Xport{E} Ctx{1}
  virtual void AddBefore(ZGlass* g, ZGlass* before); // Xport{E} Ctx{2}
};

class ZNode : public ZList {
  ...
  ZNode*  mParent;                 // Xport{GS} Link{} Structural parent
  ...
  Int_t MoveLF(Int_t axis, Real_t amount);           // Xport{E}
  ...
};
```

Figure 3: Code snippets from GLED base classes. `C++` comments following the declarations contain instructions for further handling by the GLED system. The `//!` syntax prevents streaming of that member and thus declares it un-exported (this syntax is inherited from ROOT). Comments of the form `Instruction{args}` are instructions for the PROJECT7 parser/code generator and fully specify which methods will be auto generated for the data member. Constructs following `7` in comments fields are constructors for `perl` classes that generate code for GUI widgets.

`Ctx`$\{n\}$ instructs PROJECT7 to store the first n arguments into the context part of the MIR while retaining the same function call signature.

3.2.3 MIRs and their Routing

A MIR is just a buffer containing a streamed context and identification of the method to be executed followed by its arguments (and possibly by some custom data). A MIR contains all the information needed to execute the method and can be created at any point in the code that has access to the lens. Then the MIR must be passed to local Saturn either via a socket or directly from the calling thread. Each Saturn provides sophisticated routing services and an execution enviroment to assure that MIRs are properly processed (routed and executed) by the Saturn hierarchy.

There are two types of MIRs, handled differently by the Saturn's routing code and serving different purposes.

Flares Flares cause a synchronized change of a lens on all Saturns possessing it. Imagine a lens in a sun-space of Saturn $\mathcal{S}_i$. Flares targeted at the lens must originate from $\mathcal{S}_i$ or from any of its moons. If a flare originates from a moon it is first routed, level by level, up to the $\mathcal{S}_i$, where the context check is performed. Then the MIR is passed to $\mathcal{S}_{i+1}$ Saturns and executed on the $\mathcal{S}_i$ Saturn. The $\mathcal{S}_{i+1}$ Saturns, upon receiving a flare from above, pass it on to $\mathcal{S}_{i+2}$ Saturns and execute it themselves with no further checking: the context check is only done on the top Saturn. *Flares* are shot into the system and then their light is scattered among the Saturns.

Beams Beams invoke a method in a lens on a single Saturn, without passing the request to lower Saturns. A beam is routed via common parent of the sender and the receiver. At the destination the checks are performed and the MIR is executed. Methods invoked by beams often start a new thread or serve as a source of new beams or flares. *Beams* cut through the Saturn structure without effecting any other hierarchy node than the receiver.

The type of MIR can be set after a MIR has been created so a common interface can be used for production of both kinds of MIRs.

3.2.4 Execution

Execution of a MIR proceeds on three levels:

1. first the appropriate *library set* handler is selected and a lens is locked for the execution,
2. the library set handler (auto-generated) determines the class that will provide the actual execution method,
3. the `E_Exec` method (auto-generated, e.g. Fig. 4) of a glass determines the method, de-serializes the arguments and makes the actual method call.

During the method execution a lens has access to MIR data, including the message buffer. This information can be used to determine the type of MIR that invoked the call. Further, the message can be followed by any amount of custom data which is thus made available to the lens-method processing the MIR.

3.2.5 Updating of lens-views

At this point it is appropriate to explain the functioning of a view-update mechanism. Its cycle is initiated from a lens itself by calling its *stamping* method (e.g. Fig. 4, in method `ZGlass::SetName()`). This in turn instructs Saturn to notify its Eyes that the object has changed. Each Eye then performs an update of corresponding views.[4] Viewer updates are, as demonstrated, largely arbitrary. Further, they can be blocked for any lens or a group of lenses belonging to a common chunk of object-space. The auto-generated *set* methods adhere to stamping and relieve the programmer of this duty.

Fig. 5 shows a schematic representation of a flare execution on a Saturn.

3.3 MIR Processing Performance Measurements

To benchmark the performance of the MIR processing mechanisms several simple tests have been performed. The goal was to determine the performance of the infrastructure itself and none of the tests does anything useful: they simply invoke the relevant mechanisms of GLED.

The benchmarking code was implemented within the GLED framework itself. The tests used thread execution environment of GLED (described in more detail in Sec. 6) The execution performance of an empty thread loop is shown in *"Thread Loops"* column of Tab. 1.

Methods invoked in a context of a Saturn can optionally emit viewer update requests or *stamps* (see Sec. 3.2.5). As stamping presents an execution penalty (serialization of the stamp, broadcasting to viewers and processing of the stamp by viewers) the two modes have been tested separately. In

[4] Presently the update is sub-optimal in a sense that all members of a lens declared in a given class are refreshed. There are special notifications for *link* and *list* changes as well as the possibility of user-specified notifications.

```
// Get/Set methods for ZGlass::mName
const Text_t* ZGlass::GetName() const {return mName.Data();}
void ZGlass::SetName(const Text_t* s) {
  mExecMutex.Lock(); mName = s; mExecMutex.Unlock();
  Stamp(LibID(), ClassID());
}

// Method to create MIR for calling ZGlass::SetName()
ZContext* ZGlass::S_SetName(const Text_t* s) {
  ZContext* _ctx = new ZContext(mSaturnID);
  *_ctx->Message << (LID_t)1 << (CID_t)1 << (MID_t)101;
  _ctx->Message->WriteArray(s, strlen(s)+1);
  return _ctx;
}
// and its execution part.
Int_t ZGlass::E_Exec(TBuffer* buf) {
  MID_t methId; *buf >> methId;
  switch(methId) { ...
  case 101: {
    Text_t* s; s=0; buf->ReadArray(s);
    SetName(s); delete s; return 0;
  } ...
}

// Method to create a MIR for calling ZList::AddBefore()
ZContext* ZList::S_AddBefore(ZGlass* g, ZGlass* before) {
  ZContext* _ctx = new ZContext(mSaturnID,
    (g ? g->GetSaturnID() : 0),
    (before ? before->GetSaturnID() : 0));
  *_ctx->Message << (LID_t)1 << (CID_t)2 << (MID_t)1001;
  return _ctx;
}
// and its execution part.
Int_t ZList::E_Exec(TBuffer* buf) { ...
  case 1001: {
    ZGlass* g = dynamic_cast<ZGlass*>(mCtx->Beta);
    ZGlass* before = dynamic_cast<ZGlass*>(mCtx->Gamma);
    AddBefore(g, before); return 0;
  } ...
}
```

Figure 4: Snippets of code produced by the Project7 code generator.

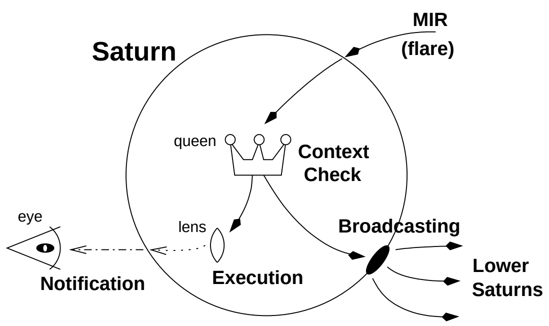

Figure 5: Execution of a *flare* on a Saturn. In case of a *beam* the broadcasting of MIR is not performed.

Tab. 1 they are represented by column groups (a) *"Method call"* (thread calls an empty method in a lens) and (b) *"Method call & Stamp"* (the method also performs stamping). Furthermore, as streaming and emitting of the stamp does not occur unless viewers are connected to a Saturn, we have made two sets of measurements on each machine: one with pure server and another with a single connected viewer. For pure servers there is a minimal performance penalty associated with stamping. In case (b) it is interesting to note the performance benefits of a two-processor machine where server and viewer threads can run in parallel.

For both methods types three different modes of method invocation have been tested:

1. **Direct:** A direct method call. For empty method calls the cost is equvalent to the cost of a function call, while for methods with stamping the stamping procedure is also performed (note the drastic performance drop when a viewer is connected). These measurements also represent reference values for MIR-based invocation schemes described below.

2. **MIR:** MIR is created within the main thread and posted directly to the Saturn for processing and execution. All MIR processing proceeds within the same thread.

3. **Beamed MIR:** MIR is created on a client machine and sent to the server in *beam* mode. Saturn's thread listening to client sockets receives the MIR, unpacks it and processes it as in the above case.

   It is important to note that the limiting factor was the processing speed of the server and not the available network bandwidth.

Machine	Thread Loops	Method call			Method call & Stamp		
		Direct	MIR	Beam	Direct	MIR	Beam
AMD -2600 Dual	245.6 ± 1.4	244.9 ± 1.2	78.6 ± 0.4	54.6 ± 0.3	234.5 ± 1.3	75.1 ± 0.5	52.9 ± 0.3
as above + viewer	243.0 ± 1.3	242.6 ± 0.8	79.2 ± 0.4	51.4 ± 0.5	87.6 ± 0.3	28.4 ± 0.3	22.4 ± 0.2
AMD -2600 Single	237.9 ± 0.5	237.3 ± 0.6	76.4 ± 0.2	53.1 ± 0.1	226.8 ± 0.4	72.8 ± 0.1	51.4 ± 0.4
as above + viewer	235.1 ± 2.1	234.6 ± 1.2	78.2 ± 1.6	47.3 ± 0.3	64.4 ± 0.1	21.8 ± 1.1	17.9 ± 0.1
P3 -1000 Dual	123.8 ± 0.3	123.0 ± 0.3	35.8 ± 0.1	27.2 ± 0.1	111.6 ± 0.4	34.6 ± 0.1	26.6 ± 0.1
as above + viewer	123.9 ± 0.2	123.2 ± 0.2	35.6 ± 0.1	26.4 ± 0.0	54.5 ± 0.4	18.1 ± 0.2	15.0 ± 0.2

Table 1: Results of execution performance measurements for different machine configurations expressed in thousands of executions per second. AMD stands for *AMD Athlon* and P3 for *Pentium III* processor. See Sec. 3.3.

The presented results show the execution penalties for different elements of the GLED infrastructure. It is interesting to observe that the measured performance spawns over one order of magnitude. In real-life uses of GLED the very basic mechanisms would be used for data transport and only a few specialized objects would emit viewer update requests. By allowing a programmer to use an arbitrary combination of framework elements, GLED can serve to implement a wide range of advanced RPC techniques with optional continuous updating of connected viewers. But as we have demonstrated that comes at a cost.

4 Management of Saturn's Object-Spaces

A sun-space and a fire-space of each Saturn are both managed (or *ruled*) by its *king* lens. A king has purely administrative duties: it holds its reserved object-space and delegates *queens* to actually rule upon their entrusted ID-space chunks (*queen-spaces*). Queens can be serialized to, or resurrected from, comets (see Sec. 3.1) and they are the smallest object-space elements that can be mirrored by lower Saturns.

Upon connection, a new Saturn only receives serialized kings and queens without the actual contents of their object spaces. The mirroring of each queen-space is performed separately and can be dynamically controlled. Request for mirroring of a queen on a given Saturn (this request is also a MIR) can originate from any of the Saturns that know about the existence of a given queen. A queen can be marked as *mandatory*: in this case all new Saturns are requested to pull it into its moon-space.

Further, queens provide lens instantiation facilities: a lens can be created and attached into the existing lens-graph by sending the queen a single MIR. Lenses can be created as default objects (by sending library set and class IDs of desired glass) or from an existing serialized lens.

Another very important role of queens is *blessing* of MIRs directed at their subjects. A Saturn performs rudimentary context checking to provide error trapping, while queens can offer extensive dependency and access control.[5] By sub-classing the `ZQueen` class, a programmer can gain a complete control over creation of lenses and their access in respective queen-spaces.

Kings and queens are *lenses* and they are placed in the first available address of the ID object-space they are ruling. Each *ruler* also manages itself and can be accessed using the standard GLED mechanisms (MIRs, GUI control).

A queen can depend on other queens in the same sun-space or in the moon-space of a given Saturn. When a queen $\mathcal{Q}$ depends on a queen $\mathcal{Q}'$, subjects of $\mathcal{Q}$ can reference subjects of $\mathcal{Q}'$ by either aggregation method (link to them or hold them as list members). This means that $\mathcal{Q}'$ must be mirrored by a Saturn prior to mirroring of $\mathcal{Q}$. Otherwise the reconstruction of $\mathcal{Q}$'s queen-space would result in unresolved references.

Queens can migrate to other kings, either to higher or lower levels in Saturn hierarchy. This is facilitated by the ability of queens to create comets. Saving them to disk, for permanent storage or swapping purposes, is thus also possible.

Queen Types: Ordinary queens are created with no subjects and later instantiate them into their queen-spaces. When a lower level Saturn requests mirroring of a queen-space, it has to be serialized and sent to the moon. This is reasonable for dynamic data that is constantly changing. But often one needs a static library of lenses that can be referenced from other dependent queens. One such service is provided by the `IceQueen` class. Ice queens are instantiated from a comet and block all incoming MIRs. Lower Saturns can instantiate them in the same manner, by loading the comets from a permanent media, thus saving the network bandwidth. Another option, semi-statical in nature, is given by the `FileQueen` class. File queens are spawned over a ROOT file and provide a mapping from a full pathname to a lens identifier. When sending such queens to a moon it is enough to stream this mapping and the moon can instantiate the objects from the local storage devices. A similar concept could be used for connecting GLED's object-spaces to a database system.

5 Representation and Management of a Gled Cluster

A GLED cluster is composed of Saturns organized in a tree structure. Inside GLED each Saturn is represented by a lens of glass `SaturnInfo`. It provides the network identity of a given node and is

[5]The basic queen implementation of MIR blessing checks that all context arguments belong to the queen or to its dependencies. As all arguments of link and list modifications enter into the context part of a MIR, the consistency of object-spaces with respect to dependencies is guaranteed.

also the natural place to store access privileges and other static or dynamic properties of a node.[6] Further, `SaturnInfo` has three links: one to its master Saturn (for SunAbsolute this is link is `null`) and two to lists holding its direct client Saturns and connected Eyes (viewers). Representants of all Saturns are instantiated and maintained under a special queen, called the *sun-queen*. It is the first queen of the sun-king of Sun Absolute and is pushed down to all connecting Saturns. This means thateach node in a GLED cluster is always aware of the whole cluster structure. Furthermore, this structure is rooted in the object-space itself and directly accessible using the standard mechanisms of the GLED paradigm.

A Saturn can accept connections from any number of Eyes. They also have a corresponding `EyeInfo` structure allowing each viewer to have its own access privileges. Various messages can be sent directly to an Eye, thus providing an efficient way for delivery of messages, warnings and errors resulting from execution of MIRs originating from the Eye in question.

Structure of a GLED cluster is therefore exported to all Saturns but is managed centrally at the Sun Absolute. Any lens can get information about any Saturn or Eye in the cluster and send it requests or messages. A direct bridge can be opened between any two Saturns to allow for node-to-node data transmissions.

The `SaturnInfo` structure also holds unique identifications of both sun-king and fire-king of its Saturn. By combining these IDs with MIRs directed at the Saturn in question, kings can create queens in their object-spaces on request from any Saturn in the cluster (although the requesting Saturn might not be able to *see* this queen). These queens can then perform some computation on behalf of the caller, either locally or by spreading it further down to lower level Saturns.

In a similar manner connections between several GLED clusters could be implemented.

6 Operators & Threads

Each Saturn can run many threads simultaneously. A glass that needs to acquire its own thread for a certain task or computation must be sub-classed from the `Eventor` glass. The common base provides mechanisms for thread control and identification. Each `Eventor` contains a directory-like structure of `Operators` (also glasses) representing the actual algorithmic elements of a thread. The operators can be traversed once or several times until some exit condition is met. All thread operations are therefore encapsulated in the framework and can be controlled with MIRs as well as from GLED GUI.

Execution of and descending into individual operators can be controlled at run-time, so the operator graph traversal is determined by the state of its elements. Thread methods are exception throwing to provide error recovery and thread state management facilities.

When a thread is created, its owner becomes the sender of the MIR that triggered thread's execution. This is rather important as threads can create MIRs and post them directly into a Saturn's routing system. By accessing this information the Saturn can determine the owner of the thread and set the caller ID of MIR accordingly.

Any method can be called from a thread (there is no built-in restriction mechanism) and it is rather easy to spoil the consistency of object-spaces by direct modification of lenses when a MIR should have been used. Also, locks must be used when accessing data of lenses which are not guaranteed to remain constant. Threads are no toys: they are high-speed execution agents that would, if limited in their execution power, hinder the computational performance of GLED.

[6]E.g. node architecture, amount of RAM, CPU speed, load averages and other monitoring data. Fig. 6 shows all GUI enabled data members of the `SaturnInfo` glass.

Threads can be used to perform computations, certain periodical tasks (such as monitoring of the node activity) or can be input-driven if they perform I/O operations. They can also spawn new processes and act as their supervisors, faithfully representing their state and monitoring the output.

7 Viewing & Rendering Infrastructure

GUI elements provided by an application framework/toolkit should be general enough to allow implementation of a wide range of user interfaces. GLED GUI, as the rest of the system, is based on the notion of extended `C++` class: its main purpose is to allow users to manipulate object-data and to invoke object-methods. This implies that the application front-end has to be representable by a collection of classes that hold the relevant configuration data and implement the desired functionality.[7] As a default GUI is provided for all classes the prototype application GUI is readily available.

The GUI architecture is modular and extensible. Each library set can, besides the auto-generated widgets for glasses, define their own GUI elements. This, in principle, allows for construction of custom, hand-made GUI front-ends and restricted user interfaces.

7.1 Standard GUI elements

Viewing objects are completely decoupled from the core of the GLED system. They are managed by the `Eye` class which serves as a router between the local Saturn and actual viewing elements. Towards the Saturn it transmits MIRs generated by lens-views in their GUI callback functions. From Saturn it receives change notifications objects (or `Rays`) containing, among other things, the lens identity and type of change that occurred (change of member data, links, list contents, etc.). As each Eye holds a mapping from viewed lenses to list of their lens-views, it can efficiently update all views of the changed lens. This also implies that each lens can have an arbitrary number of GUI representations.

The base class for lens-views (`A_View`) contains a minimal representation of the observed lens and some abstract methods for handling of change notifications. Further base classes are provided for representations of links and lists. Automatically generated widgets for lenses as well as the hand-made GUI front-end are all built upon these base classes. As details of the implementation are beyond the scope of this article, the emphasis will be put on features of the GUI (implemented using the `FLTK` widget library [8]).

For each lens a window containing a collection of all automatically generated widgets for a given lens can be created (e.g. Fig. 6). It includes widgets for all parent classes, ordered by the class hierarchy.

An important feature of the GLED GUI is a generalized structure browser (*lens-browser*). It displays lenses together with their list members and links. The list members are shown in a collapsible vertical structure (as in most standard file/object browsers) while links are represented in a horizontal structure, allowing more compact views of selected lenses (e.g. Fig. 7). The lens-browser provides a means for manipulation of links, modification of list contents and instantiation of new lenses. Lenses can also be exported into the C++ interpreter shell for manual operations.

The lens-browser has another mode in which its right part contains a custom set of widgets for each lens (sub-selection of those from a full-lens-view). The desired glasses and their members can be specified using a simple syntax. A specialized parser then creates the appropriate widgets for

[7] In OO applications this is always the case, anyway.

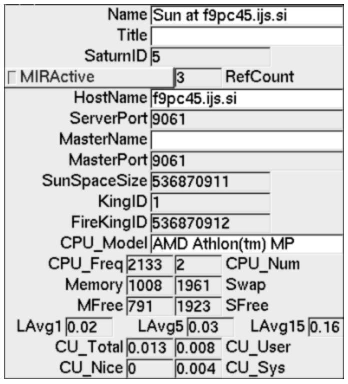

Figure 6: A *full-lens-view* for the `SaturnInfo` glass.

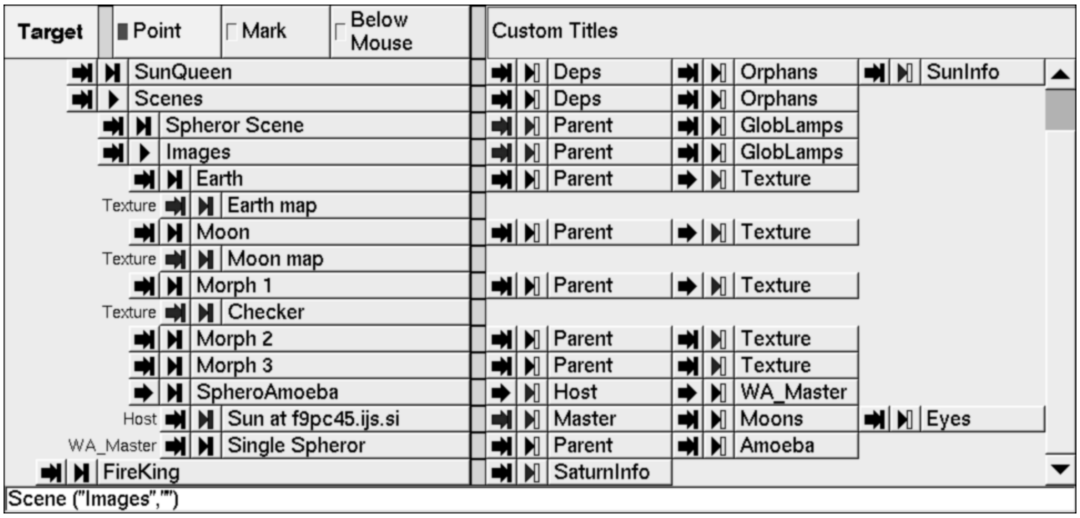

Figure 7: A lens browser displaying a lens graph. Each entry has two collapsing buttons: one for links and the other for list members. Links of each lens are displayed on the right part of the browser.

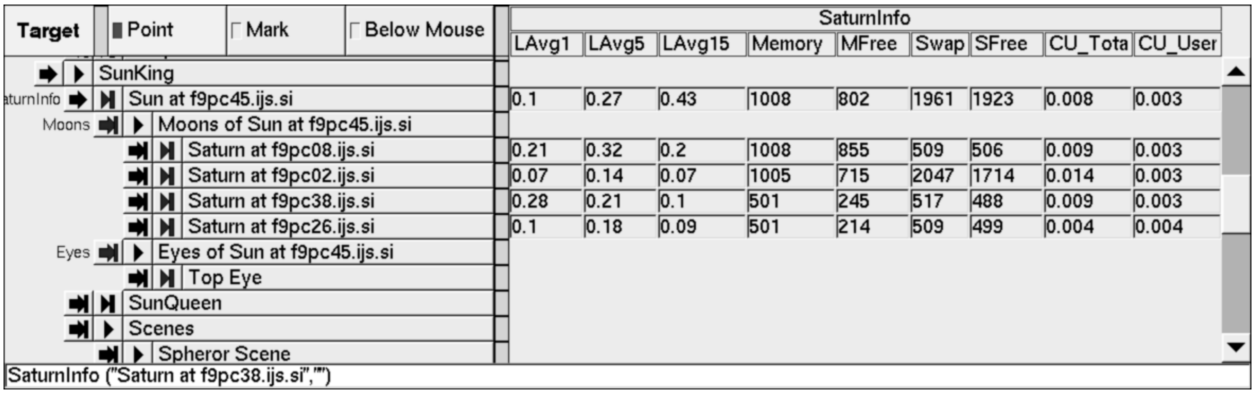

Figure 8: A set of custom-generated widgets inside a lens browser showing `SaturnInfo` structures for a GLED cluster composed of five nodes.

each lens (Fig. 8). Such views provide an efficient way for viewing and editing of a large number of lenses.

7.2 Rendering

The rendering infrastructure offers the representation of lenses and lens-graphs in 3D space. Actual rendering of lenses is done by a parallel class structure that has to be coded manually for each rendering device.[8] It is not mandatory for glasses to have a corresponding rendering class: they can use the renderer of any of their base classes. Rendering classes for each device and each library set are produced as independent libraries and can be dynamically loaded into a running GLED system.

A high level of flexibility is achieved by actual rendering of each lens being split into three functions: `PreDraw()`, `Draw()` and `PostDraw()`. Further, rendering is proceeding on an arbitrary number of levels allowing the draw methods of the lens, its links and list members to be called in any order. This provides a versatile infrastructure for traversal of lens-graphs as well as for management of renderer state (lighting, polygon mode, material properties, texturing, etc.).

More examples of GLED's GUI and can be found at the project's homepage [9].

8 Applications of the Gled system

We have presented the core functionality of the GLED system which, from this perspective, offers an OO framework. However, if one is interested in building a distributed application, the base classes of GLED can be seen as elements of a toolkit. Future work will, among other things, focus on development of library sets (offering particular implementations for selected areas of interest) and preparation of corresponding demonstration programs. Based on these, GLED will also become a toolkit for respective development areas.

To conclude this paper the GLED system will be discussed from the point of view of four different usage modes. This will serve to recapitulate the main ideas, to merge the presented details into a coherent whole and to discuss future extensions of GLED.

[8]OpenGL is used as the standard real-time renderer, others can be added using a common render-driver API.

8.1 Single machine, single user usage

GLED classes can serve as an interface to configuration and execution of user algorithms as well as for visualization of resulting data. Persistency of configuration data and results of algorithms is achieved by using ROOT's object serialization and I/O infrastructure (ROOT files and trees).

Glass member functions can hold the implementation of the algorithms or can serve as a wrapper layer for a specialized (numerical) library which can be implemented in any language linkable with `C++`. By sub-classing the base thread glass (`Eventor`) finer control over operation and configuration of algorithms can be achieved.

As GUI classes are automatically generated for all glasses, they can immediately serve for composition of object graphs, for configuration and editing of member data and for algorithm control via management of threads. CINT, the `C/C++` interpreter, offers a general scripting interface to all GLED's functionality and thus complements the GUI functionality for complex and repetitive tasks.

Standard data presentation services, such as histograms and graphs, are offered by the ROOT system. Further, ROOT has a rich set of tools and algorithms for statistical analysis, minimization and spectral analysis.

Dynamic real-time 3D visualization can easily be programmed by providing the rendering classes and using the `Eventor`-thread infrastructure. As rendering crucially depends on the underlying data-structures it is hard, if not impossible, to provide a generic and efficient set of rendering atoms. However, for some specific cases the renderers are already available. E.g. a representation of a 2D function on a rectangular mesh (using ROOT's `TMatrix` class) and 3D triangulated surfaces (wrapper over GTS library [10]). Further rendering elements will become available with appearance of new library sets and with their proper structuring GLED will offer a generic multi-purpose rendering toolkit.

8.2 Distributed and `grid` computing

The GLED framework enables prototyping and control of algorithms performing computation or dynamic visualization within a hierarchic server-client-viewer model. As topology of the cluster is not limited and can also change dynamically, sophisticated problem solving strategies can be implemented over heterogenous clusters (MIRs and streamed data are architecture independent).

A particularly interesting feature of GLED is that the cluster structure itself is directly accessible to algorithms using the standard framework mechanisms. On one hand this provides a means for cluster monitoring and visualization while on the other it allows for dynamic optimization of the cluster structure.

Currently there are no central mechanisms for node allocation and acquisition. Such mechanisms will be implemented as daemons or network services with a top-level node manager. Another option is to adopt **`grid`** resource allocation services when they become standardized. Until then standard remote command execution mechanisms can be used (e.g. `rsh` or `ssh`).

Each node in a distributed cluster and each thread within a node analyses and/or produces data. Based on the nature of the problem, different solutions for emitting the resulting data may be applied.

- If the data is segmented into many relatively small fragments or is needed for continuation of computation on some other node, then it can be sent encapsulated into a MIR. The object that receives the MIR can properly place the data into a larger structure or further propagate it to relevant algorithms. Arguments of the MIR can serve as instructions for management of the subsequent data. This way MIRs can serve as carriers for a sophisticated data exchange protocol.

- If the data is produced in large fragments and is not instantly needed for further computation then it can be emitted via standard data-exchange mechanisms, e.g. `ftp`, `rootd` (remote access to ROOT files) or distributed file-systems (nfs, afs). The data can also be cached in the vicinity of the local node and transmitted on request upon completion of the computation on a larger scale. This allows minimal network connection duration and provides means for shaping of the network traffic.

Similar problems arise when algorithms need read access to shared data. Further development of GLED will also address various data exchange mechanisms and will offer some concrete implementations of data traffic structuring and `grid`-aware data addressing protocols.

8.3 Multi-user information systems

Hierarchic structure of the GLED cluster makes it an interesting platform for implementation of collaborative tools and VR systems. Intermediate proxy nodes can provide an effective reduction of network traffic when several nodes from several organizations are connected into a common information system.

The concurrent access of several user requires an elaborate authentication & access control mechanism. Implementation of such infrastructure is under way. Authorization is performed during MIR processing stage (as a part of blessing of a MIR by a `Queen`) and further infrastructure is being implemented that allows individual lenses and `Queens` to specify access policy. At first GLED will use internal user and group databases and in due time interfaces to appropriate LDAP and `grid` services will be created.

As GLED offers a direct correspondence between lenses and their 3D representations, advanced VR systems with immersive control over object graphs can be implemented. This requires further work on UI infrastructure, which has to provide mechanisms for creation of complex object contexts with transparent access to available member functions and their arguments. However, the infrastructure for emitting MIRs from UI and receiving update requests is implemented as a part of the core system. The available GUI elements (full-lens-views, lens-browser and GL rendering window) provide a general interface to mechanisms of the GLED system and offer an example of complexity that can be achieved by indirect mapping of lenses to GUI structures.

8.4 Encapsulated network services

By encapsulating network services within a sub-class of the `Eventor` glass, threads from within the Saturn can offer any kind of service to outside clients, for example:

- Provide information about local Saturn and local computing node (status, monitoring data, catalog of locally available data and availability of other services).
- Export its object-spaces and local data to consumers by using any desired protocol. For example, lens graphs and lens data could be exported via `http` protocol in `html` or `xml` format.
- Serve as entry point for computation or service instantiation requests. After a successful authentication such request can result in creation of a thread that will provide the required service.

- Serve as entry point for data coming from external sources, such as other processes or sensors connected to the node.

In this respect GLED offers monitoring and replication facilities for such services as well as infrastructure for following and reporting the usage patterns (ROOT trees, histograms and graphs). It is also an advantage that multiple services can be spawned from within the same process: it allows for easier load balancing between different services and allows implementation of advanced strategies for resolving complex queries (e.g. `grid` resource allocation requests) or providing parallel file I/O operations.

9 Conclusion

The GLED system brings together a colorful collection of modern computing techniques and binds them within a fairly general implementation of a hierarchic server–client model. By merging cluster creation and configuration tools, object-based RPC, threaded exectuion environment, automatically generated GUI widgets and sophisticated 3D rendering infrastructure, GLED has been shown to represent a useful OO framework/toolkit for development of a number of distinctly different applications that can take advantage of distributed computing environment, object serialization, cooperative multi-user GUI and rich data presentation & visualization infrastructure. In GLED these complex techniques are represented at a fairly abstract level but at the same time a low-level interface is available to allow for further extensions and more specialized implementations.

We hope that GLED addresses the demands and possibilities of modern computing in a fashion that is both interesting as a research project and usable as a concrete framework implementation.

Availability

The implementation of the described framework is publicly available [9]. While GLED is constantly developing, its core functionality is stable and suitable for implementation of production systems. As it is with any open-source project, the future development of GLED largely depends on interests and contributions from users of outside communities.

Acknowledgments

I am infinitely indebted to my friend J. J. Javoršek for his support and enthusiasm for the GLED project. Without his willingness to spend countless hours discussing the arising problems and ideas GLED, most probably, would not exist at all.

Many thanks go to teams developing ROOT, `perl`, FLTK and GNU development tools for excellent packages that make the burdens of software development bearable.

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] SETI@home, *The Search for Extraterrestrial Intelligence*, `http://setiathome.ssl.berkeley.edu/`.

[3] Folding@home, *Distributed protein folding simulations*, `http://folding.stanford.edu/`.

[4] A.D. Birrell and B.J. Nelson, *Implementing Remote Procedure Calls*, XEROX CSL-83-7, October 1983.

[5] R. Srinivasan, *XDR: External Data Representation Standard*, RFC 1832, Sun Microsystems, Inc., August 1995.

[6] W. Yeong, T. Howes and S. Kille, *Lightweight Directory Access Protocol*, RFC 1777, Performance Systems International, University of Michigan, ISODE Consortium, March 1995.

[7] M. Goto, *C++ Interpreter - CINT*, CQ publishing, ISBN4-789-3085-3 (Japanese).

[8] B. Spitzak, et al, *FLTK – A Fast Light ToolKit*, `http://www.fltk.org/`.

[9] The GLED project homepage: `http://www.gled.org/`.

[10] *GTS – GNU Triangulated Surface Library*, `http://gts.sourceforge.net/`.