



# EUROPEAN MIDDLEWARE INITIATIVE

## LOGGING AND BOOKKEEPING – DEVELOPER'S GUIDE

| | |
|---|---|
| Document version: | **1.4.12** |
| EMI Component Version: | **4.x** |
| Date: | **March 29, 2013** |

This work is co-funded by the European Commission as part of the EMI project under Grant Agreement INFSO-RI-261611.

# CONTENTS

## L&B DOCUMENTATION AND VERSIONS OVERVIEW

The Logging and Bookkeeping service (L&B for short) was initially developed in the EU DataGrid project[1] as a part of the Workload Management System (WMS). The development continued in the EGEE, EGEE-II and EGEE-III projects,[2] where L&B became an independent part of the gLite[3] middleware [1], and then in the EMI Project.[4]

The complete L&B Documentation consists of the following parts:

- **L&B User's Guide** [2]. The User's Guide explains how to use the Logging and Bookkeeping (L&B) service from the user's point of view. The service architecture is described thoroughly. Examples on using L&B's event logging commands to log user tags and change job ACLs are given, as well as L&B query and notification use cases.
- **L&B Administrator's Guide** [3]. The Administrator's Guide explains how to administer the Logging and Bookkeeping (L&B) service. Several deployment scenarios are described together with the installation, configuration, running and troubleshooting steps.
- **L&B Developer's Guide** – this document. The Developer's Guide explains how to use the Logging and Bookkeeping (L&B) service API. Logging (producer), querying (consumer) and notification API as well as the Web Services Interface is described in details together with programing examples.
- **L&B Test Plan** [4]. The Test Plan document explains how to test the Logging and Bookkeeping (L&B) service. Two major categories of tests are described: integration tests (include installation, configuration and basic service ping tests) and system tests (basic functionality tests, performance and stress tests, interoperability tests and security tests).

**The following versions of L&B service are covered by these documents:**

- *L&B version 4.0*: included in the EMI-3 *Monte Bianco* release
- *L&B version 3.2*: included in the EMI-2 *Matterhorn* release
- *L&B version 3.1*: an update for the EMI-1 *Kebnekaise* release
- *L&B version 3.0*: included in the EMI-1 *Kebnekaise* release
- *L&B version 2.1*: replacement for *L&B version 2.0* in gLite 3.2
- *L&B version 2.0*: included in gLite 3.2 release
- *L&B version 1.x*: included in gLite 3.1 release

**L&B packages can be obtained from two distinguished sources:**

- **gLite releases**: gLite node-type repositories, offering a specific repository for each node type such as *glite-LB*. Only binary RPM packages are available from that source.

[1] http://eu-datagrid.web.cern.ch/eu-datagrid/
[2] http://www.eu-egee.org/
[3] http://www.glite.org
[4] http://www.eu-emi.eu/

- **emi releases**: EMI repository[5] or EGI's UMD repository,[6] offering all EMI middleware packages from a single repository. There are RPM packages, both source and binary, the latter relying on EPEL for dependencies. There are also DEB packages (starting with EMI-2) and `tar.gz` archives.

*Note:* Despite offering the same functionality, binary packages obtained from different repositories differ and switching from one to the other for upgrades may not be altogether straightforward.

Updated information about L&B service (including the L&B service roadmap) is available at the L&B homepage: http://egee.cesnet.cz/en/JRA1/LB

[5] http://emisoft.web.cern.ch/emisoft/
[6] http://repository.egi.eu/

# 1 INTRODUCTION

This document is intented to guide the reader through basic steps of writing, compiling and running programs communicating with the L&B service using the L&B library. It is not intended as a complete API reference; for this, the reader is referred to the C or C++ header files, which are thoroughly documented using the doxygen–style comments.

The L&B API can be divided by functionality into two independent parts:

- *L&B Producer API* (section 3) is used to create and send events to the L&B server (proxy),
- *L&B Consumer API* (section 4) and *L&B Notification API* (section 5) are used to obtain information from the L&B server (proxy).

These two parts (and in fact the whole L&B service implementation) share a number of common concepts, design principles, data types and functions which we will describe first. Most of common data types and functions are separated in its own SW module called `org.glite.lb.common` and are described in section 2

Example code

Source code for examples shown in this guide is distributed together with the document. The examples contain excerpts from the actual files with reference to the file name and line numbers. All the examples can be compiled using attached Makefile.

Recommended reading

Before you start reading this guide, it is recommended to accomodate yourself with the L&B architecture described in the first part of the L&B user's guide ([2]).

## 1.1 LANGUAGE BINDINGS

The L&B library itself is developed in C language, the C API covers all the L&B services. There are bindings for other languages (C++, Java) as well as web-service (WS) based interface, but these cover only subsets of L&B functionality and internally they use the C API themselves (in the C++ case the C API is also exported).

We describe the C API first and then the differences between C and the other languages, as the C constructs often reflect directly.

As for the L&B WS interface, it reflects only the functionality of L&B Querying API (see Sect. 4.4).

There exist also HTML and plain text interfaces to L&B. We do not expect anybody using them in a programming language (though it is possible), they might be useful rather in scripts. Their usage is rather straightforward as it is described in the User's Guide [2].

## 1.2 GETTING AND BUILDING CLIENT LIBRARIES

All C and C++ L&B API's are implemented in L&B client library (`glite-lb-client` package of standard gLite distribution), and L&B common library (`glite-lb-common`). These bring in other gLite dependencies:

- `glite-lb-client-interface` (*L&B version 1.x* only)
- `glite-security-gsoap-plugin` (*L&B version 1.x* only)
- `glite-security-gss` (only *L&B version 2.0* and higher)

and external dependencies:

- globus – only GSS library is needed, we use `vdt_globus_essentials` package from VDT if available.
- expat – XML parser, available in most operating systems
- c-ares – asynchronous resolver library
- cppunit – unit tests library, required for build only
- classads – ClassAd parser and matchmaking library from Condor

For platforms supported by gLite officially all the required packages can be downloaded from http://www.glite.org. However, L&B is fairly portable and it can be built on other platforms fairly smoothly.

Detailed instructions on getting the sources, including the required dependencies, are available at https://erebor.ics.muni.cz/wiki/lb_build.html [7].

## 1.3 General Guidelines

Naming conventions

All names exported by the L&B library (function names, symbolic constants) are prefixed to avoid name clashes. The prefix is `edg_wll_` for function names and `EDG_WLL_` for symbolic constants[8]. In C++ the namespace `glite::lb` is used instead.

Symbolic constants

Symbolic constants (that is enumerated types) are used at various places in the L&B API. There is a user–friendly string representation of each constant and for each enumerated type there are two functions that convert strings to enum values and vice versa. Example is given in section 2.1.5

Input and output arguments

All input arguments in L&B API are designated `const` (for simple types) or have `const` in type name (for structures).

If pointers are passed in output of function call (either as a return value, output argument or part of structure), the corresponding objects are *always* allocated dynamically and have to be freed when not used anymore. Structures defined in L&B API can be deallocated by calling convenience `edg_wll_Free`*Type*`()` functions. *This deallocates members of the structure, but not the structure itself. It has to be* `free()`*'d explicitly.*

Opaque and transparent types

Types used in L&B API are either opaque or transparent. *Opaque types* are considered internal to the library, their structure is not exposed to users and is subject to change without notice. The only way to modify opaque objects is to use API calls. Example of opaque type is `edg_wll_Context`.

Structure of *transparent types* is completely visible to user, is well documented and no incompatible changes will be done without notice. Example of transparent type is `edg_wll_Event`.

Return values

The return type of most of the API functions is `int`. Unless specified otherwise, zero return value means success, non-zero failure. Standard error codes from `errno.h` are used as much as possible. In a few cases the error can not be intuitively mapped into standard code and L&B specific error value greater than `EDG_WLL_ERROR_BASE` is returned.

Few API function return `char *`. In such a case `NULL` indicates an error, non-null value means success.

[7] The location may change but we will keep it linked from official L&B pages http://egee.cesnet.cz/en/JRA1/LB/.

[8] The EDG_WLL_ stands for European DataGrid, the original EU project, and Workload Logging, the subsystem identification.

### 1.4 CONTEXT AND PARAMETER SETTINGS

The L&B library does not maintain internal state (apart of network connections, see 1.5), all the API functions refer to a *context* argument instead. Context object preserves state information among the various API calls, the state including L&B library parameters (for example security context, server addresses, timeouts), reference to open connections (connection pool), error state etc.

The API caller can create many context objects which are guaranteed to be independent on one another. In this way thread–safety of the library is achieved as long as the context is not used by more threads at the same time. One thread may use more than one context, though. w Upon context initialization, all the parameters are assigned default values. If not set explicitly, many of the parameters take their value from environment variables. If the corresponding environment variable is set, the parameter is initialized to its value instead of the default. Note that a few parameters cannot be assigned default value; consequently setting them either in environment or with an explicit API call is mandatory before using the appropriate part of the API.

The context also stores details on errors of the recent API call.

For use with the *producer* calls (see section 3) the context has to be assigned a single *JobId* (with the `edg_wll_SetLoggingJob()` call), and keeps track of an event *sequence code* for the job (see also L&B Architecture described in [2]).

The context object and its API functions are described more thoroughly in section 2.1.3

### 1.5 CONNECTION POOL

The L&B library maintains pool of client–server connections to improve performance (creating SSL connection is heavy–weight operation). The connections are transparently shared and reused by all contexts/threads to eliminate the overhead of secure channel establishment. This behaviour is completely hidden by the library.

## 2 L&B COMMON COMPONENTS

### 2.1 C LANGUAGE BINDING

#### 2.1.1 HEADER FILES

Header files for the common structures and functions are summarized in table 1. If you use the producer and/or consumer API described further in this document, you do not have to include them explicitly.

| | |
|---|---|
| `glite/jobid/cjobid.h` | Definition of job identifier. |
| `glite/lb/context.h` | Definition of context structure and parameters. |
| `glite/lb/events.h` | L&B event data structure. |
| `glite/lb/jobstat.h` | Job status structure returned by consumer API. |

**Table 1:** Header files for common structures

### 2.1.2 BUILDING CLIENT PROGRAMS

The easiest way to build programs using the L&B library in C is to use GNU's libtool to take care of all the dependencies:

```
flavour=gcc32dbg
libtool --mode=compile gcc -c example1.c util.c \
        -I\$GLITE_LOCATION/include -D_GNU_SOURCE
libtool --mode=link gcc -o example1 example1.o util.o \
        -L$GLITE_LOCATION/lib -lglite_lb_client_$flavour
```

The library comes in different flavours (with/without debugging symbols, with/without thread support) which are in turn linked with (and depend on) the correct Globus library flavours. When linking threaded programs you have to use the library flavour with thread support.

The RPM package needed is `glite-lb-client` and its dependencies which contain all necessary libraries.

### 2.1.3 CONTEXT

Context initialization

Opaque data structure representing L&B API context (see section 1.4) is named `edg_wll_Context`. The context must be initialized before the first L&B API call:

```
#include <glite/lb/context.h>

edg_wll_Context ctx;
edg_wll_InitContext(&ctx);
```

Parameter setting

The context parameters can be set explicitly by calling

```
int edg_wll_SetParam(edg_wll_Context *, edg_wll_ContextParam, ...);
```

function. The second argument is symbolic name of the context parameter; parameters specific for producer and consumer API are described in respective API sections, the common parameters are:

| C name | Description |
|---|---|
| EDG_WLL_PARAM_X509_KEY | Key file to use for authentication.<br>*Type:* **char** *<br>*Environment:* X509_USER_KEY |
| EDG_WLL_PARAM_X509_CERT | Certificate file to use for authentication.<br>*Type:* **char** *<br>*Environment:* X509_USER_CERT |
| EDG_WLL_PARAM_CONNPOOL_SIZE | Maximum number of open connections maintained by the library.<br>*Type:* **int**<br>*Environment:* |

**Table 2:** Common context parameters

The third argument is parameter value, which can be of type `int`, `char *` or `struct timeval *`. If the parameter value is set to `NULL` (or 0), the parameter is reset to the default value.

If you want to obtain current value of some context parameter, call

```
int edg_wll_GetParam(edg_wll_Context, edg_wll_ContextParam, ...);
```

function:

```
char *cert_file;

edg_wll_GetParam(ctx, EDG_WLL_PARAM_X509_CERT, &cert_file);
printf("Certificate used: %s\n", cert_file);
free(cert_file);
```

The third argument points at variable with type corresponding to the requested parameter. Do not forget to free the result.

**TODO:** *sitera: Mame odkaz kde jsou popsany defaulty a vazby na promenne environmentu (ty jsou v LBUG Appendix C)*

Obtaining error details

When L&B API call returns error, additional details can be obtained from the context:

```
char    *err_text,*err_desc;

edg_wll_Error(ctx, &err_text, &err_desc);
fprintf(stderr, "LB library error: %s (%s)\n", err_text, err_desc);
free(err_text);
free(err_desc);
```

Context deallocation

If the context is needed no more, deallocate it:

```
edg_wll_FreeContext(ctx);
```

For more information see file `glite/lb/context.h`

### 2.1.4 JOBID

The primary entity of L&B is a job, identified by JobId – a unique identifier of the job (see also [2]). The type representing the JobId is opaque `glite_jobid_t`. The JobId is in fact just URL with `https` protocol, path component being unique string with no further structure and host and port designating the L&B server holding the job information. The JobId can be:

- created new for given L&B server (the unique part will be generated by the L&B library):

```
glite_jobid_t jobid;
int ret;
if(ret = glite_jobid_create("some.host", 0, &jobid)) {
        fprintf(stderr, "error creating jobid: %s\n", strerror(ret));
}
```

- parsed from string (for example when given as an program argument or read from file):

```
if(ret = glite_jobid_parse("https://some.host:9000/OirOgeWh_F9sfMZjnIPYhQ", &jobid)) {
        fprintf(stderr, "error parsing jobid: %s\n", strerror(ret));
}
```

- or obtained as part of L&B server query result.

In either case the jobid must be freed when no longer in use:

```
glite_jobid_free(jobid);
```

For more information see file `glite/jobid/cjobid.h`

**L&B 1.x** *In the older L&B versions (1.x) the structure was named* `edg_wlc_JobId` *and the functions had prefix* `edg_wlc_JobId`*, for example* `edg_wlc_JobIdFree()`*. Exact description can be found in the header file* `glite/wmsutils/cjobid.h`

### 2.1.5 EVENT

The transparent data structure `edg_wll_Event` represents L&B event, atomic data unit received and processed by L&B. It is a union of common structure and structures for all event types:

```
union _edg_wll_Event {
        edg_wll_EventCode       type;
        edg_wll_AnyEvent        any;
        edg_wll_TransferEvent   transfer;
        edg_wll_AcceptedEvent   accepted;
        ...more follows...
}
typedef union _edg_wll_Event edg_wll_Event;
```

The most important common event attributes are listed in table 3, the following example shows access:

```
edg_wll_Event event;

event.type = 0;
event.any.user = "me";
```

| Attribute name | Attribute type | Description |
|---|---|---|
| `type` | `edg_wll_EventCode` | Event type. Values are symbolic constants for example `EDG_WLL_EVENT_DONE` |
| `jobId` | `glite_jobid_t` | Jobid of the job the event belongs to. |
| `user` | `char*` | Identity (certificate subject) of the event sender. |
| `host` | `char*` | Hostname of the machine the event was sent from. |
| `source` | `edg_wll_Source` | Designation of the WMS component the event was sent from, for example `EDG_WLL_SOURCE_USER_INTERFACE` |
| `timestamp` | `struct timeval` | Time when the event was generated. |
| `seqcode` | `char*` | Sequence code assigned to the event. |

**Table 3:** Common event attributes

The `edg_wll_Event` is returned by consumer L&B API job event related calls. The only important operation defined on `edg_wll_Event` itself is

```
edg_wll_FreeEvent(edg_wll_Event *event)
```

to free the event structure.

List of event types

The event structure makes use of enumerated types extensively, starting with the `type` atribute. The following example demonstrates how to convert enumerated values into more user–friendly strings; it will print out the event names known to the L&B library:

```
edg_wll_EventCode ev_type;

for(ev_type = 1; ev_type < EDG_WLL_EVENT__LAST; ev_type++) {
        char *ev_string = edg_wll_EventToString(ev_type);
        if(ev_string) {
           /* there may be holes */
           printf("%s\n", ev_string);
           free(ev_string);
        }
}
```

For more information see file `include/glite/lb/events.h`

### 2.1.6 JOBSTATUS

The transparent data type `edg_wll_JobStat` represents status of a job as computed by the L&B from received events. Much like the `edg_wll_Event` structure it can be viewed as a set of attributes, where some attributes are common and some specific for a given job state (but unlike the `edg_wll_Event` it is not implemented as union of structs but rather as one big struct). Generally speaking, when the attribute value is set, it is a valid part of job state description. Most important common attributes are summarized in table 4.

| Attribute name | Attribute type | Description |
|---|---|---|
| `jobId` | `glite_jobid_t` | Job identifier of this job. |
| `state` | `edg_wll_JobStatCode` | Numeric code of the status, for example `EDG_WLL_JOB_SUBMITTED`. |
| `type` | `enum edg_wll_StatJobtype` | Type of the job, for example `EDG_WLL_JOB_SIMPLE`. |
| `children` | `char**` | List of subjob *JobId*'s |
| `owner` | `char*` | Owner (certificate subject) of the job. |

**Table 4:** Common job status attributes

Job status structure is returned by the L&B consumer API job status queries. When no longer used, it has to be freed by calling

```
void edg_wll_FreeStatus(edg_wll_JobStat *);
```

The following example prints out the states of jobs given in the input list; the job states are printed together with their subjobs on the same input list:

**File:** util.c

```
12  distributed under the License is distributed on an "AS_IS" BASIS,
13  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  See the License for the specific language governing permissions and
15  limitations under the License.
16  */
17
18
19  #include <stdio.h>
20  #include <stdlib.h>
21  #include <string.h>
22  #include <expat.h>
23
24  #include <glite/jobid/cjobid.h>
25  #include <glite/lb/jobstat.h>
26
27  int use_proxy = 0;
28
29  void
30  print_jobs(edg_wll_JobStat *states)
31  {
32    int           i,j;
33
34    for (i=0; states[i].state != EDG_WLL_JOB_UNDEF; i++) {
35      char *id = edg_wlc_JobIdUnparse(states[i].jobId);
36      char *st = edg_wll_StatToString(states[i].state);
37
```

```
38      if (!states[i].parent_job) {
39        if (states[i].jobtype == EDG_WLL_STAT_SIMPLE) {
40          printf("      %s .... %s %s\n", id, st, (states[i].state==EDG_WLL_JOB_DONE) ?
                edg_wll_done_codeToString(states[i].done_code) : "" );
41        }
42        else if ((states[i].jobtype == EDG_WLL_STAT_DAG) ||
43                 (states[i].jobtype == EDG_WLL_STAT_COLLECTION)) {
44          printf("%s  %s .... %s %s\n", (states[i].jobtype==EDG_WLL_STAT_DAG)?"DAG ":"COLL",id,
                 st, (states[i].state==EDG_WLL_JOB_DONE) ? edg_wll_done_codeToString(states[i].
                done_code) : "");
45          for (j=0; states[j].state != EDG_WLL_JOB_UNDEF; j++) {
46            if (states[j].parent_job) {
47              char *par_id = edg_wlc_JobIdUnparse(states[j].parent_job);
48
49              if (!strcmp(id,par_id)) {
50                char *sub_id = edg_wlc_JobIdUnparse(states[j].jobId),
51                  *sub_st = edg_wll_StatToString(states[j].state);
52
53                printf(" `-       %s .... %s %s\n", sub_id, sub_st, (states[j].state==
                      EDG_WLL_JOB_DONE) ? edg_wll_done_codeToString(states[j].done_code) : "");
54                free(sub_id);
55                free(sub_st);
56              }
57              free(par_id);
58            }
59          }
60        }
61      }
62
63      free(id);
64      free(st);
65    }
66
67    printf("\nFound %d jobs\n",i);
68  }
```

For more information see file `include/glite/lb/jobstat.h`

## 2.2 C++ LANGUAGE BINDING

The C++ languague binding now only supports the consumer (querying) API. It is not the (re)implementation of the library in C++; instead it is just a thin adaptation layer on top of the C API, which means all the structures and functions of the C API can be used in C++. The C++ classes wrap up the concepts and structures of C API and provide convenient access to the functionality. The namespace used for L&B C++ API is `glite::lb`.

Exceptions

While the C++ API closely follows the C API functionality, there are also two important differences: error handling and memory management.

When the L&B method call fails, the exception of class `glite::lb::Exception` (derived from `std::runtime_error`) is raised that holds the error description and information about the source file, line and method the exception was thrown from (possibly accumulating information from other exception).

Reference counting

When the C L&B library calls return allocated structures, they are encapsulated within C++ accessor objects. Copying the C++ object does not copy the underlying structure, it increases the reference count instead, making use of the same allocated data. The reference count is decremented with destruction of the wrapper object, when it drops to zero, the allocated memory is freed.

Using this scheme all the data allocated by the L&B library are held in memory only once.

Context The context in C API is part of common components, the C++ API on the other hand differentiates between query context (see Section 4.3.5) and logging context; the description is therefore part of the respective chapters.

### 2.2.1 HEADER FILES

Header files for the C++ version of common definitions are summarized in table 5.

| | |
|---|---|
| `glite/jobid/JobId.h` | Definition of job identifier. |
| `glite/lb/LoggingExceptions.h` | Exception class for L&B–specific errors. |

**Table 5:** Header file for common C++ classes

### 2.2.2 BUILDING PROGRAMS

The recommended way to build programs using the C++ L&B library is, like in the C case, to use the `libtool` utility:

```
flavour=gcc32dbg
libtool --mode=compile gcc -c example1.c util.c \
        -I\$GLITE_LOCATION/include -D_GNU_SOURCE
libtool --mode=link gcc -o example1 example1.o util.o \
        -L$GLITE_LOCATION/lib -lglite_lb_clientpp_$flavour
```

The only difference is the library name, the RPM package required is again `glite-lb-client`.

### 2.2.3 JOBID

The `glite::jobid::JobId` class represents job identification and provides convenient methods for manipulating the data. The `JobId` object can be created:

- from the C structure (this is used mainly internally within the library):

  ```
  using namespace glite::jobid;
  glite_jobid_t cjobid;

  JobId jobid(cjobid);
  ```

  *Note:* This creates copy of the structure, the original structure has to be deallocated as usual.

- parsed from the string:

  ```
  JobId jobid("https://some.host:9000/OirOgeWh_F9sfMZjnIPYhQ");
  ```

- from the components:

  ```
  JobId jobid(Hostname("some.host"), 9000, "OirOgeWh_F9sfMZjnIPYhQ");
  ```

  The last two arguments are optional, so you have to specify only name of the L&B server machine (the `Hostname` class is used to disambiguate the constructors):

```
JobId jobid(Hostname("some.host"));
```

In that case new unique part is generated automatically.

Apart from that there are the usual copy constructor and assignment operator that make deep copy of the object, and the destructor that deallocates the memory.

Data access

The `JobId` class provides methods for obtaining the host, port and unique part of the *JobId* as well as conversion into C `glite_jobid_t` type and into string representation. There is also a defined ordering (`operator<`) on the *JobId*'s, which is just the lexicographical ordering of corresponding string representations. The following example illustrates these features:

```
JobId a(Hostname("me"));
JobId b(Hostname("me"));

cout << "jobid␣host␣and␣port:␣" << a.host() << ",␣" <<
a.port() << endl;
cout << (a < b) ? a.unique() : b.unique() << "␣comes␣first" << endl;
cout << "Complete␣jobid:␣" << a.toString() << endl;
```

### 2.2.4 EXCEPTION

The `glite::lb::Exception` is a base class for all exceptions thrown by the L&B library. It inherits from `std::runtime_error` and adds no additional members or methods except constructors. The typical usage is this:

```
try {
  // some code with LB calls
} catch (glite::lb::Exception &e) {
        cerr << "LB␣library␣exception:␣" << e.what() << endl;
}
```

# 3 L&B LOGGING (PRODUCER) API

## 3.1 C LANGUAGE BINDING

The L&B logging API (or producer API) is used to create and deliver events to the L&B server and/or proxy, depending on the function used:

**TODO:** *kouril: verify ChangeACL*

| Function | Delivers to |
|---|---|
| `edg_wll_LogEvent(...)` | asynchronously through locallogger/interlogger to the L&B server |
| `edg_wll_LogEventSync(...)` | synchronously through locallogger/interlogger to the L&B server |
| `edg_wll_LogEventProxy(...)` | through L&B proxy to the L&B server |
| `edg_wll_Register*(...)` | directly to both L&B server and proxy |
| `edg_wll_ChangeACL(...)` | synchronously to the L&B server |

These general functions take as an argument event format (which defines the ULM string used) and variable number of arguments corresponding to the given format. For each defined event there is predefined format string in the form `EDG_WLL_FORMAT_`*EventType*, for example `EDG_WLL_FORMAT_UserTag`, as well as three convenience functions `edg_wll_LogUserTag(...)`, `edg_wll_LogUserTagSync(...)`, `edg_wll_LogUserTagProxy(...)`.

For most developers (that is those not developing the WMS itself) the `edg_wll_LogUserTag*(...)` and `edg_wll_ChangeACL(...)` are the only functions of interest.

### 3.1.1 CALL SEMANTICS

L&B producer calls generally do not have transaction semantics, the query following succesful logging call is not guaranteed to see updated L&B server state. The typical call – loging an event – is returned immediatelly and the success of the call means that the first L&B infrastructure component takes over the event and queues it for delivery. If you require transaction semantics, you have to use synchronous `edg_wll_LogEventSync(...)` call.

The L&B proxy on the other hand provides a *local view* semantics, events logged into proxy using `edg_wll_LogEventProxy(...)` are guaranteed to by accessible by subsequent queries *on that proxy*.

Job registrations are all synchronous.

### 3.1.2 HEADER FILES

`glite/lb/producer.h` Prototypes for all event logging functions.

### 3.1.3 CONTEXT PARAMETERS

The table 6 summarizes context parameters relevant to the event logging. If parameter is not set in the context explicitly, the L&B library will search for value of corresponding environment variable.

The `GLITE_WMS_LOG_DESTINATION` environment variable contains both locallogger host and port separated by colon (that is "host:port").

| Name | Description |
|---|---|
| EDG_WLL_PARAM_HOST | Hostname that appears as event origin.<br>*Type:* **char***<br>*Environment:* |
| EDG_WLL_PARAM_SOURCE | Event source component.<br>*Type:* edg_wll_Source<br>*Environment:* |
| EDG_WLL_PARAM_DESTINATION | Hostname of machine running locallogger/interlogger.<br>*Type:* **char***<br>*Environment:* GLITE_WMS_LOG_DESTINATION |
| EDG_WLL_PARAM_DESTINATION_PORT | Port the locallogger is listening on.<br>*Type:* **int**<br>*Environment:* GLITE_WMS_LOG_DESTINATION |
| EDG_WLL_LOG_TIMEOUT | Logging timeout for asynchronous logging.<br>*Type:* **struct** timeval<br>*Environment:* GLITE_WMS_LOG_TIMEOUT |
| EDG_WLL_LOG_SYNC_TIMEOUT | Logging timeout for synchronous logging.<br>*Type:* **struct** timeval<br>*Environment:* GLITE_WMS_LOG_SYNC_TIMEOUT |
| EDG_WLL_LBPROXY_STORE_SOCK | L&B Proxy store socket path (if logging through L&B Proxy)<br>*Type:* **char***<br>*Environment:* GLITE_WMS_LBPROXY_STORE_SOCK |
| EDG_WLL_LBPROXY_USER | Certificate subject of the user (if logging through L&B proxy).<br>*Type:* **char***<br>*Environment:* GLITE_WMS_LBPROXY_USER |

**Table 6:** Producer specific context parameters

Logging job and sequence numbers

In addition to the above list, there are two more parameters that must be set before logging call is made: *JobId* of the logging job and *sequence number*. There is a special call for this task:

```
extern int edg_wll_SetLoggingJob(
        edg_wll_Context context,        context to work with
        glite_jobid_const_t     job,    jobid of the job
        const char *            code,   sequence code
        int                     flags   flags on code handling
);
```

After setting the logging job identity, all the following logging calls refer to this *JobId* and the sequence code is incremented according to the source component. See [2] for information about sequence codes and event numbering, especially the description, how the sequence codes are updated.

### 3.1.4 RETURN VALUES

The logging functions return 0 on success and one of `EINVAL`, `ENOSPC`, `ENOMEM`, `ECONNREFUSED`, `EAGAIN` on error. If `EAGAIN` is returned, the function should be called again to retry the delivery; it is not guaranteed, however, that the event was not delivered by the first call. Possibly duplicated events are discarded by the L&B server or proxy.

**TODO:** *ljocha: check these*

The synchronous variants of logging functions can in addition return `EDG_WLL_ERROR_NOJOBID` or `EDG_WLL_ERROR_DB_DU`

### 3.1.5 LOGGING EVENT EXAMPLE

In this section we will give commented example how to log an UserTag event to the L&B.

First we have to include neccessary headers:

**File:** prod_example1.c

```
26 #include "glite/jobid/cjobid.h"
27 #include "glite/lb/events.h"
28 #include "glite/lb/producer.h"
```

Initialize context and set parameters:

**File:** prod_example1.c

```
87         edg_wll_InitContext(&ctx);
88
89         edg_wll_SetParam(ctx, EDG_WLL_PARAM_SOURCE, EDG_WLL_SOURCE_USER_INTERFACE);
90         edg_wll_SetParam(ctx, EDG_WLL_PARAM_HOST, server);
91         //edg_wll_SetParam(ctx, EDG_WLL_PARAM_PORT, port);
```

**TODO:** *honik: proper setting of sequence codes*

**File:** prod_example1.c

```
95         if (edg_wll_SetLoggingJob(ctx, jobid, seq_code, EDG_WLL_SEQ_NORMAL)) {
96                 char    *et,*ed;
97                 edg_wll_Error(ctx,&et,&ed);
98                 fprintf(stderr,"SetLoggingJob(%s,%s): %s (%s)\n",jobid_s,seq_code,et,ed);
99                 exit(1);
100        }
```

Log the event:

**File:** prod_example1.c

```
104        err = edg_wll_LogEvent(ctx,
105                               EDG_WLL_EVENT_USERTAG,
106                               EDG_WLL_FORMAT_USERTAG,
107                               name, value);
108        if (err) {
109            char        *et,*ed;
110
111            edg_wll_Error(ctx,&et,&ed);
112            fprintf(stderr,"%s: edg_wll_LogEvent*(): %s (%s)\n",
113                    argv[0],et,ed);
114            free(et); free(ed);
115        }
```

The `edg_wll_LogEvent()` function is defined as follows:

```
extern int edg_wll_LogEvent(
        edg_wll_Context context,
        edg_wll_EventCode event,
        char *fmt, ...);
```

If you use this function, you have to provide event code, format string and corresponding arguments yourself. The UserTag event has only two arguments, tag name and value, but other events require more arguments.

Instead of using the generic `edg_wll_LogEvent()` at line 104, we could also write:

```
err = edg_wll_LogUserTag(ctx, name, value);
```

#### 3.1.6 CHANGE ACL EXAMPLE

**TODO:** *kouril*

### 3.2 JAVA BINDING

**TODO:** *mirek*

# 4 L&B QUERYING (CONSUMER) API

The L&B Consumer API is used to obtain information from L&B server or Proxy using simple query language (see Sect. 4.1). There are two types of queries based on the results returned:

- query for events – the result contains events satisfying given criteria,
- query for jobs – the result contains JobId's and job states of jobs satisfying given criteria.

The potential result sets can be very large; the L&B server imposes limits on the result set size, which can be further restricted by the client.

## 4.1 QUERY LANGUAGE

The L&B query language is based on simple value assertions on job and event attributes. There are two types of queries based on the complexity of selection criteria, *simple* and *complex*. Simple queries are can be described by the following formula:

$$attr_1 \ \mathrm{OP} \ value_1 \wedge \cdots \wedge attr_n \ \mathrm{OP} \ value_n$$

where $attr_i$ is attribute name, $\mathrm{OP}$ is one of the $=$, $<$, $>$, $\neq$ and $\in$ relational operators and *value* is single value (or, in the case of $\in$ operator, interval) from attribute type.

Complex queries can be described using the following formula:

$$\begin{aligned}(attr_1 \ \mathrm{OP} \ value_{1,1} \vee \cdots \vee attr_1 \ \mathrm{OP} \ value_{1,i_1}) \wedge \\ (attr_2 \ \mathrm{OP} \ value_{2,1} \vee \cdots \vee attr_2 \ \mathrm{OP} \ value_{2,i_2}) \wedge \\ \vdots \\ \wedge (attr_n \ \mathrm{OP} \ value_{n,1} \vee \cdots \vee attr_n \ \mathrm{OP} \ value_{n,i_n})\end{aligned}$$

The complex query can, in contrast to simple query, contain more assertions on value of single attribute, which are ORed together.

Indexed attributes

The query must always contain at least one attribute indexed on the L&B server; this restriction is necessary to avoid matching the selection criteria against all jobs in the L&B database. The list of indexed attributes for given L&B server can be obtained by L&B API call.

## 4.2 C LANGUAGE BINDING

### 4.2.1 CALL SEMANTICS

The L&B server queries are, in contrast to logging event calls, synchronous (for asynchronous variant see Sect. 5, notifications). The server response contains *JobId*'s, job states and/or events known to the server at the moment of processing the query. Due to the asynchronous nature of event delivery it may not contain all data that was actually sent; the job state computation is designed to be resilient to event loss to some extent.

Result size limits

When the item count returned by L&B server exceeds the defined limits, the `E2BIG` error occur. There are two limits — the server and the user limit. The user defined limit may be set in the context at the client side, while the server imposed limit is configured at the server and can be only queried by the client. The way the L&B library and server handles the over–limit result size can be specified by setting context parameter `EDG_WLL_PARAM_QUERY_RESULTS` to one of the following values:

- `EDG_WLL_QUERYRES_NONE` — In case the limit is reached, no results are returned at all.
- `EDG_WLL_QUERYRES_LIMITED` — A result contains at most "limit" item count.
- `EDG_WLL_QUERYRES_ALL` — All results are returned and limits have no effect. This option is available only in special cases such as "user jobs query" and the "job status query". Otherwise the EINVAL error is returned.

Default value is `EDG_WLL_QUERYRES_NONE`.

### 4.2.2 HEADER FILES

`glite/lb/consumer.h` Prototypes for all query functions.

### 4.2.3 CONTEXT PARAMETERS

The table 7 shows parameters relevant to the query API.

| Name | Description |
|---|---|
| EDG_WLL_PARAM_QUERY_SERVER | Default server name to query. |
| EDG_WLL_PARAM_QUERY_SERVER_PORT | Default server port to query. |
| EDG_WLL_PARAM_QUERY_SERVER_OVERRIDE | host:port parameter setting override even values in *JobId* (useful for debugging & hacking only) |
| EDG_WLL_PARAM_QUERY_TIMEOUT | Query timeout. |
| EDG_WLL_PARAM_QUERY_JOBS_LIMIT | Maximal query jobs result size. |
| EDG_WLL_PARAM_QUERY_EVENTS_LIMIT | Maximal query events result size. |
| EDG_WLL_PARAM_QUERY_RESULTS | Flag to indicate handling of too large results. |

**Table 7:** Consumer specific context parameters

### 4.2.4 RETURN VALUES

L&B server returns errors which are classified as hard and soft errors. The main difference between these categories is that in the case of soft errors results may still be returned. The authorization errors belong to "soft error" sort. Hard errors like `ENOMEM` are typically all unrecoverable, to obtain results the query must be repeated, possibly after correcting the failure condition the error indicated.

Depending on the setting of context parameter `EDG_WLL_PARAM_QUERY_RESULTS`, the `E2BIG` error may fall into both categories.

### 4.2.5 QUERY CONDITION ENCODING

The L&B query language is mapped into (one- or two-dimensional) array of attribute value assertions represented by `edg_wll_QueryRec` structure:

```
typedef struct _edg_wll_QueryRec {
        edg_wll_QueryAttr       attr;     attribute to query
        edg_wll_QueryOp         op;       query operation
```

```
        union {
                char *                  tag;    user tag name / JDL attribute "path"
                edg_wll_JobStatCode     state;  job status code
        } attr_id;
        union edg_wll_QueryVal {
                int     i;              integer query attribute value
                char    *c;             character query attribute value
                struct timeval  t;      time query attribute value
                glite_jobid_t   j;      JobId query attribute value
        } value, value2;
} edg_wll_QueryRec;
```

The table 8 shows the most common query attributes. For a complete list see `query_rec.h`.

| Name | Description |
|---|---|
| EDG_WLL_QUERY_ATTR_JOBID | Job ID to query. |
| EDG_WLL_QUERY_ATTR_OWNER | Job owner. |
| EDG_WLL_QUERY_ATTR_STATUS | Current job status. |
| EDG_WLL_QUERY_ATTR_LOCATION | Where is the job processed. |
| EDG_WLL_QUERY_ATTR_DESTINATION | Destination CE. |
| EDG_WLL_QUERY_ATTR_DONECODE | Minor done status (OK,failed,cancelled). |
| EDG_WLL_QUERY_ATTR_USERTAG | User tag. |
| EDG_WLL_QUERY_ATTR_JDL_ATTR | Arbitrary JDL attribute. |
| EDG_WLL_QUERY_ATTR_STATEENTERTIME | When entered current status. |
| EDG_WLL_QUERY_ATTR_LASTUPDATETIME | Time of the last known event of the job. |
| EDG_WLL_QUERY_ATTR_JOB_TYPE | Job type. |

**Table 8:** Query record specific attributes.

The table 9 shows all supported query operations.

| Name | Description |
|---|---|
| EDG_WLL_QUERY_OP_EQUAL | Attribute is equal to the operand value. |
| EDG_WLL_QUERY_OP_LESS | Attribute is grater than the operand value. |
| EDG_WLL_QUERY_OP_GREATER | Attribute is less than the operand value. |
| EDG_WLL_QUERY_OP_WITHIN | Attribute is in given interval. |
| EDG_WLL_QUERY_OP_UNEQUAL | Attribute is not equal to the operand value. |
| EDG_WLL_QUERY_OP_CHANGED | Attribute has changed from last check (supported since *L&B version 2.0* in notification matching). |

**Table 9:** Query record specific operations.

### 4.2.6 QUERY JOBS EXAMPLES

The simplest use case corresponds to the situation when an exact job ID is known and the only information requested is the job status. The job ID format is described in [5]. Since *L&B version 2.0*, it is also possible to query all jobs belonging to a specified user, VO or RB.

The following example shows how to retrieve the status information about all user's jobs running at a specified CE.

First we have to include neccessary headers:

**File:** cons_example1.c

```
26 #include "glite/jobid/cjobid.h"
27 #include "glite/lb/events.h"
28 #include "glite/lb/consumer.h"
```

Define and initialize variables:

**File:** cons_example1.c

```
75          edg_wll_Context         ctx;
76          edg_wll_QueryRec        jc[4];
77          edg_wll_JobStat         *statesOut = NULL;
78          edg_wlc_JobId           *jobsOut = NULL;
```

Initialize context and set parameters:

**File:** cons_example1.c

```
84          edg_wll_InitContext(&ctx);
85
86          edg_wll_SetParam(ctx, EDG_WLL_PARAM_QUERY_SERVER, server);
87          if (port) edg_wll_SetParam(ctx, EDG_WLL_PARAM_QUERY_SERVER_PORT, port);
```

Set the query record to *all (user's) jobs running at CE 'XYZ'*:

**File:** cons_example1.c

```
91          jc[0].attr = EDG_WLL_QUERY_ATTR_OWNER;
92          jc[0].op = EDG_WLL_QUERY_OP_EQUAL;
93          jc[0].value.c = NULL;
94          jc[1].attr = EDG_WLL_QUERY_ATTR_STATUS;
95          jc[1].op = EDG_WLL_QUERY_OP_EQUAL;
96          jc[1].value.i = EDG_WLL_JOB_RUNNING;
97          jc[2].attr = EDG_WLL_QUERY_ATTR_DESTINATION;
98          jc[2].op = EDG_WLL_QUERY_OP_EQUAL;
99          jc[2].value.c = "XYZ";
100         jc[3].attr = EDG_WLL_QUERY_ATTR_UNDEF;
```

Query jobs:

**File:** cons_example1.c

```
104         err = edg_wll_QueryJobs(ctx, jc, 0, &jobsOut, &statesOut);
105         if ( err == E2BIG ) {
106                 fprintf(stderr,"Warning: only limited result returned!\n");
107                 return 0;
108         } else if (err) {
109                 char    *et,*ed;
110
111                 edg_wll_Error(ctx,&et,&ed);
112                 fprintf(stderr,"%s: edg_wll_QueryJobs(): %s (%s)\n",argv[0],et,ed);
113
114                 free(et); free(ed);
115         }
```

Now we can for example print the job states:

**File:** cons_example1.c

```
119         for (i = 0; statesOut[i].state; i++ ) {
120                 printf("jobId : %s\n", edg_wlc_JobIdUnparse(statesOut[i].jobId));
121                 printf("state : %s\n\n", edg_wll_StatToString(statesOut[i].state));
122         }
```

In many cases the basic logic using only conjunctions is not sufficient. For example, if you need all your jobs running at the destination XXX or at the destination YYY, the only way to do this with the `edg_wll_QueryJobs()` call is to call it twice. The `edg_wll_QueryJobsExt()` call allows to make such a query in a single step. The function accepts an array of condition lists. Conditions within a single list are OR-ed and the lists themselves are AND-ed.

The next query example describes how to get all user's jobs running at CE 'XXX' or 'YYY'.

We will need an array of three conditions (plus one last empty):

**File:** cons_example2.c

```
74      edg_wll_Context         ctx;
75      edg_wll_QueryRec        *jc[4];
76      edg_wll_JobStat         *statesOut = NULL;
77      edg_wlc_JobId           *jobsOut = NULL;
```

The query condition is the following:

**File:** cons_example2.c

```
90      jc[0] = (edg_wll_QueryRec *) malloc(2*sizeof(edg_wll_QueryRec));
91      jc[0][0].attr = EDG_WLL_QUERY_ATTR_OWNER;
92      jc[0][0].op = EDG_WLL_QUERY_OP_EQUAL;
93      jc[0][0].value.c = NULL;
94      jc[0][1].attr = EDG_WLL_QUERY_ATTR_UNDEF;
95
96      jc[1] = (edg_wll_QueryRec *) malloc(2*sizeof(edg_wll_QueryRec));
97      jc[1][0].attr = EDG_WLL_QUERY_ATTR_STATUS;
98      jc[1][0].op = EDG_WLL_QUERY_OP_EQUAL;
99      jc[1][0].value.i = EDG_WLL_JOB_RUNNING;
100     jc[1][1].attr = EDG_WLL_QUERY_ATTR_UNDEF;
101
102     jc[2] = (edg_wll_QueryRec *) malloc(3*sizeof(edg_wll_QueryRec));
103     jc[2][0].attr = EDG_WLL_QUERY_ATTR_DESTINATION;
104     jc[2][0].op = EDG_WLL_QUERY_OP_EQUAL;
105     jc[2][0].value.c = "XXX";
106     jc[2][1].attr = EDG_WLL_QUERY_ATTR_DESTINATION;
107     jc[2][1].op = EDG_WLL_QUERY_OP_EQUAL;
108     jc[2][1].value.c = "YYY";
109     jc[2][2].attr = EDG_WLL_QUERY_ATTR_UNDEF;
110
111     jc[3] = NULL;
```

As can be clearly seen, there are three lists supplied to `edg_wll_QueryJobsExt()`. The first list specifies the owner of the job, the second list provides the required status (`Running`) and the last list specifies the two destinations. The list of lists is terminated with `NULL`. This query equals to the formula

```
(user=NULL) and (state=Running) and (dest='XXX' or dest='YYY').
```

To query the jobs, we simply call

**File:** cons_example2.c

```
115     err = edg_wll_QueryJobsExt(ctx, (const edg_wll_QueryRec **)jc,
116                                0, &jobsOut, &statesOut);
```

### 4.2.7 QUERY EVENTS EXAMPLES

Event queries and job queries are similar. Obviously, the return type is different —the L&B raw events. There is one more input parameter representing specific conditions on events (possibly empty) in addition to conditions on jobs.

The following example shows how to select all events (and therefore jobs) marking red jobs (jobs that were marked red at some time in the past) as green.

**File:** cons_example3.c

```
75          edg_wll_Context      ctx;
76          edg_wll_Event       *eventsOut;
77          edg_wll_QueryRec     jc[2];
78          edg_wll_QueryRec     ec[2];
```

**File:** cons_example3.c

```
91          jc[0].attr = EDG_WLL_QUERY_ATTR_USERTAG;
92          jc[0].op = EDG_WLL_QUERY_OP_EQUAL;
93          jc[0].attr_id.tag = "color";
94          jc[0].value.c = "red";
95          jc[1].attr = EDG_WLL_QUERY_ATTR_UNDEF;
96          ec[0].attr = EDG_WLL_QUERY_ATTR_USERTAG;
97          ec[0].op = EDG_WLL_QUERY_OP_EQUAL;
98          ec[0].attr_id.tag = "color";
99          ec[0].value.c = "green";
100         ec[1].attr = EDG_WLL_QUERY_ATTR_UNDEF;
```

This example uses `edg_wll_QueryEvents()` call. Two condition lists are given to `edg_wll_QueryEvents()` call. One represents job conditions and the second represents event conditions. These two lists are joined together with logical and (both condition lists have to be satisfied). This is necessary as events represent a state of a job in a particular moment and this changes in time.

**File:** cons_example3.c

```
104         err = edg_wll_QueryEvents(ctx, jc, ec, &eventsOut);
```

The `edg_wll_QueryEvents()` returns matched events and save them in the `eventsOut` variable. Required job IDs are stored in the edg_wll_Event structure.

**File:** cons_example3.c

```
122         for (i = 0; eventsOut && (eventsOut[i].type); i++ ) {
123                 //printf("jobId : %s\n", edg_wlc_JobIdUnparse(eventsOut[i].jobId));
124                 printf("event : %s\n\n", edg_wll_EventToString(eventsOut[i].type));
125         }
```

In a similar manor to `edg_wll_QueryJobsExt()`, there exists also `edg_wll_QueryEventsExt()` that can be used to more complex queries related to events. See also `README.queries` for more examples.

Last L&B Querying API call is `edg_wll_JobLog()` that returns all events related to a single job. In fact, it is a convenience wrapper around `edg_wll_QueryEvents()` and its usage is clearly demonstrated in the client example `job_log.c` (in the client module).

## 4.3 C++ LANGUAGE BINDING

The querying C++ L&B API is modelled after the C L&B API using these basic principles:

- queries are expressed as vectors of `glite::lb::QueryRecord` instances,
- L&B context and general query methods are represented by class `glite::lb::ServerConnection`,
- L&B job specific queries are encapsulated within class `glite::lb::Job`,
- query results are returned as (vector or list of) `glite::lb::Event` or `glite::lb::JobStatus` read-only instances.

### 4.3.1 HEADER FILES

Header files for the L&B consumer API are summarized in table 10.

| | |
|---|---|
| `glite/lb/Event.h` | Event class for event query results. |
| `glite/lb/JobStatus.h` | JobStatus class for job query results. |
| `glite/lb/ServerConnection.h` | Core of the C++ L&B API, defines `QueryRecord` class for specifying queries and `ServerConnection` class for performing the queries. |
| `glite/lb/Job.h` | Defines `Job` class with methods for job specific queries. |

**Table 10:** Consumer C++ API header files

### 4.3.2 QUERYRECORD

The `glite::lb::QueryRecord` class serves as the base for mapping the L&B query language into C++, similarly to the C counterpart `edg_wll_QueryRecord`. The `QueryRecord` object represents condition on value of single attribute:

```
using namespace glite::lb;

QueryRecord a(QueryRecord::OWNER, QueryRecord::EQUAL, "me");
```

The `QueryRecord` class defines symbolic names for attributes (in fact just aliases to `EDG_WLL_QUERY_ATTR_` symbols described in table 8) and for logical operations (aliases to `EDG_WLL_QUERY_OP_` symbols, table 9). The last parameter to the `QueryRecord` constructor is the attribute value.

There are constructors with additional arguments for specific attribute conditions or logical operators that require it, that is the `QueryRecord::WITHIN` operator and queries about state enter times. The query condition "job that started running between `start` and `end` times' can be represented in the following way:

```
struct timeval start, end;

QueryRecord a(QueryRecord::TIME, QueryRecord::WITHIN, JobStatus::RUNNING,
              start, end);
```

### 4.3.3 EVENT

The objects of class `glite::lb::Event` are returned by the L&B event queries. The `Event` class intgstr roduces symbolic names for event type (enum `Event::Type`), event attributes (enum `Event::Attr`) and their types (enum `Event::AttrType`), feature not available through the C API, as well as (read only) access to the attribute values. Using these methods you can:

- get the event type (both symbolic and string):

```
Event event;

// we suppose event gets somehow filled in
cout << "Event type: " << event.type << endl;

cout << "Event name:" << endl;
// these two lines should print the same string
cout << Event::getEventName(event.type) << endl;
cout << event.name() << endl;
```

- get the list of attribute types and values (see line 34 of the example),
- get string representation of attribute names,
- get value of given attribute.

The following example demonstrates this by printing event name and attributes:

**File:**util.C

```
27 void
28 dumpEvent(Event *event)
29 {
30 // list of attribute names and types
31         typedef vector<pair<Event::Attr, Event::AttrType>> AttrListType;
32
33         cout << "Event name: " << event->name() << endl;
34         AttrListType attr_list = event->getAttrs();
35         for(AttrListType::iterator i = attr_list.begin();
36             i != attr_list.end();
37             i++) {
38                 Event::Attr attr = attr_list[i].first;
39
40                 cout << Event::getAttrName(attr) << " = ";
41                 switch(attr_list[i].second) {
42                 case Event::INT_T:
43                 case Event::PORT_T:
44                 case Event::LOGSRC_T:
45                         cout << event->getValInt(attr) << endl;
46                         break;
47
48                 case Event::STRING_T:
49                         cout << event->getValString(attr) << endl;
50                         break;
51
52                 case Event::TIMEVAL_T:
53                         cout << event->getValTime(attr).tv_sec << endl;
54                         break;
55
56                 case Event::FLOAT_T:
57                         cout << event->getValFloat(attr) << endl;
58                         break;
59
60                 case Event::DOUBLE_T:
61                         cout << event->getValDouble(attr) << endl;
62                         break;
63
64                 case Event::JOBID_T:
```

```
65                          cout << event->getValJobId(attr).toString() << endl;
66                          break;
67
68                  default:
69                          cout << "attribute type not supported" << endl;
70                          break;
71                  }
72          }
73  }
```

### 4.3.4 JOBSTATUS

The `glite::lb::JobStatus` is a result type of job status queries in the same way the `glite::lb::Event` is used in event queries. The `JobStatus` class provides symbolic names for job states (enum `JobStatus::Code`), state attributes (enum `JobStatus::Attr`) and their types (enum `JobStatus::AttrType`), and read only access to the attribute values. Using the `JobStatus` interface you can:

- get the string name for the symbolic job state:

```
JobStatus status;

// we suppose status gets somehow filled in
cout << "Job state: " << status.type << endl;

cout << "State name: " << endl;
// these two lines should print the same string
cout << JobStatus::getStateName(status.type) << endl;
cout << status.name() << endl;
```

- get the job state name (both symbolic and string),
- get the list of job state attributes and types,
- convert the attribute names from symbolic to string form and vice versa,
- get value of given attribute.

The following example demostrates this by printing job status (name and attributes):

**File:**util.C

```
78  void dumpState(JobStatus *status)
79  {
80          typedef vector<pair<JobStatus::Attr, JobStatus::AttrType>> AttrListType;
81
82          cout << "Job status: " << status->name << endl;
83
84          AttrListType attr_list = status->getAttrs();
85          for(AttrListType::iterator i = attr_list.begin();
86              i != attr_list.end();
87              i++) {
88                  JobStatus::Attr attr = attr_list[i].first;
89                  cout << JobStatus::getAttrName(attr) << " = ";
90                  switch(attr_list[i].second) {
91
92                  case INT_T:
```

```
93                      cout << status->getValInt(attr) << endl;
94                      break;
95
96              case STRING_T:
97                      cout << status->getValInt(attr) << endl;
98                      break;
99
100             case TIMEVAL_T:
101                     cout << status->getValTime(attr).tv_sec << endl;
102                     break;
103
104             case BOOL_T:
105                     cout << status->getValBool(attr).tv_sec << endl;
106                     break;
107
108             case JOBID_T:
109                     cout << status->getValJobid(attr).toString() << endl;
110                     break;
111
112             case INTLIST_T:
113                     vector<int> list = status->getValIntList(attr);
114                     for(vector<int>::iterator i = list.begin();
115                         i != list.end();
116                         i++) {
117                             cout << list[i] << " ";
118                     }
119                     cout << endl;
120                     break;
121
122             case STRLIST_T:
123                     vector<string> list = status->getValStringList(attr);
124                     for(vector<string>::iterator i = list.begin();
125                         i != list.end();
126                         i++) {
127                             cout << list[i] << " ";
128                     }
129                     cout << endl;
130                     break;
131
132             case TAGLIST_T: /**< List of user tags. */
133                     vector<pair<string, string>> list = status->getValTagList(attr);
134                     for(vector<pair<string,string>>::iterator i = list.begin();
135                         i != list.end();
136                         i++) {
137                             cout << list[i].first << "=" << list[i].second << " ";
138                     }
139                     cout << endl;
140                     break;
141
142             case STSLIST_T:  /**< List of states. */
143                     vector<JobStatus> list = status->getValJobStatusList(attr);
144                     for(vector<JobStatus>::iterator i = list.begin();
145                         i != list.end();
146                         i++) {
147                             // recursion
148                             dumpState(&list[i]);
149                     }
150                     cout << endl;
151                     break;
```

```
152
153                 default:
154                         cout << "attribute type not supported" << endl;
155                         break;
156
157                 }
158         }
159 }
```

### 4.3.5 SERVERCONNECTION

The `glite::lb::ServerConnection` class represents particular L&B server and allows for queries not specific to particular job (these are separated into `glite::lb:Job` class). The `ServerConnection` instance thus encapsulates client part of `edg_wll_Context` and general query methods.

There are accessor methods for every consumer context parameter listed in table 7, for example for `EDG_WLL_PARAM_QUERY_SERVER` we have the following methods:

```
void setQueryServer(const std::string& host, int port);
std::pair<std::string, int> getQueryServer() const;
```

We can also use the generic accessors defined for the parameter types `Int`, `String` and `Time`, for example:

```
void setParam(edg_wll_ContextParam name, int value);
int getParamInt(edg_wll_ContextParam name) const;
```

The `ServerConnection` class provides methods for both event and job queries:

```
void queryJobs(const std::vector<QueryRecord>& query,
               std::vector<glite::jobid::JobId>& jobList) const;

void queryJobs(const std::vector<std::vector<QueryRecord> >& query,
               std::vector<glite::jobid::JobId>& jobList) const;

void queryJobStates(const std::vector<QueryRecord>& query,
                    int flags,
                    std::vector<JobStatus> & states) const;

void queryJobStates(const std::vector<std::vector<QueryRecord> >& query,
                    int flags,
                    std::vector<JobStatus> & states) const;

void queryEvents(const std::vector<QueryRecord>& job_cond,
                 const std::vector<QueryRecord>& event_cond,
                 std::vector<Event>& events) const;

void queryEvents(const std::vector<std::vector<QueryRecord> >& job_cond,
                 const std::vector<std::vector<QueryRecord> >& event_cond,
                 std::vector<Event>& eventList) const;
```

You can see that we use `std::vector` instead of `NULL` terminated arrays for both query condition lists and results. The API does not differentiate simple and extended queries by method name (`queryJobs` and `queryJobsExt` in C), but by parameter type (`vector<QueryRecord>` vs. `vector<vector<QueryRecord>>`). On the other hand there are different methods for obtaining *JobId*'s and full job states as well as convenience methods for getting user jobs.

Now we can show the first example of job query from section 4.2.6 rewritten in C++. First we have to include the headers:

**File:** cons_example1.cpp

```
26 #include "glite/jobid/JobId.h"
27 #include "glite/lb/ServerConnection.h"
28 #include "glite/lb/Job.h"
```

Define variables:

**File:** cons_example1.cpp

```
77          ServerConnection            lb_server;
78          glite::jobid::JobId         jobid;
79          std::vector<QueryRecord>    job_cond;
80          std::vector<JobStatus>      statesOut;
```

Initialize server object:

**File:** cons_example1.cpp

```
85                  jobid = glite::jobid::JobId(jobid_s);
86
87                  lb_server.setQueryServer(jobid.host(), jobid.port());
```

Create the query condition vector:

**File:** cons_example1.cpp

```
91                  job_cond.push_back(QueryRecord(QueryRecord::OWNER, QueryRecord::EQUAL, std::
                        string(user)));
92                  job_cond.push_back(QueryRecord(QueryRecord::STATUS, QueryRecord::EQUAL,
                        JobStatus::RUNNING));
93                  job_cond.push_back(QueryRecord(QueryRecord::DESTINATION, QueryRecord::EQUAL,
                        std::string("xyz")));
```

Perform the query:

**File:** cons_example1.cpp

```
97                  statesOut = lb_server.queryJobStates(job_cond, 0);
```

Print the results:

**File:** cons_example1.cpp

```
101                 for (i = 0; i< statesOut.size(); i++ ) {
102                         cout << "jobId␣:␣" <<  statesOut[i].getValJobId(JobStatus::JOB_ID).
                                toString() << endl;
103                         cout << "state␣:␣" <<  statesOut[i].name() << endl << endl;
104                 }
```

The operations can throw an exception, so the code should be enclosed within try–catch clause.

The second example rewritten to C++ is shown here; first the query condition vector:

**File:** cons_example2.cpp

```
91                  jc_part.push_back(QueryRecord(QueryRecord::OWNER, QueryRecord::EQUAL, ""));
92                  jc.push_back(jc_part);
93
94                  jc_part.clear();
```

```
95              jc_part.push_back(QueryRecord(QueryRecord::STATUS, QueryRecord::EQUAL,
                    JobStatus::RUNNING));
96              jc.push_back(jc_part);
97
98              jc_part.clear();
99              jc_part.push_back(QueryRecord(QueryRecord::DESTINATION, QueryRecord::EQUAL, "
                    XXX"));
100             jc_part.push_back(QueryRecord(QueryRecord::DESTINATION, QueryRecord::EQUAL, "
                    YYY"));
101             jc.push_back(jc_part);
```

The query itself:

**File:** cons_example2.cpp

```
105             statesOut = lb_server.queryJobStates(jc, 0);
```

The third example shows event query (as opposed to job state query in the first two examples). We are looking for events of jobs, that were in past painted (tagged by user) green, but now they are red. The necessary query condition vectors are here:

**File:** cons_example3.cpp

```
92              jc.push_back(QueryRecord("color", QueryRecord::EQUAL, "red"));
93              ec.push_back(QueryRecord("color", QueryRecord::EQUAL, "green"));
```

The query itself:

**File:** cons_example3.cpp

```
97              events_out = lb_server.queryEvents(jc,ec);
```

The resulting event vector is dumped using the utility function `dumpEvent()` listed above:

**File:** cons_example3.cpp

```
102             for(i = 0; i < eventsOut.size(); i++) {
103                     dumpEvent(&(eventsOut[i]));
104             }
```

### 4.3.6 JOB

The `glite::lb::Job` class encapsulates L&B server queries specific for particular job as well as client part of context. The `Job` object provides method for getting the job status and the event log (that is all events belonging to the job):

```
JobStatus status(int flags) const;

void log(std::vector<Event> &events) const;
```

**Important!** It is important to notice that `Job` contain `ServerConnection` as private member and thus encapsulate client part of context. That makes them relatively heavy–weight objects and therefore it is not recommended to create too many instances, but reuse one instance by assigning different *JobId*'s to it.

## 4.4 WEB-SERVICES BINDING

**TODO:** *ljocha: Complete review, list of all relevant (WSDL) files, their location, etc.*

In this section we describe the operations defined in the L&B WSDL file (`LB.wsdl`) as well as its custom types (`LBTypes.wsdl`).

For the sake of readability this documentation does not follow the structure of WSDL strictly, avoiding to duplicate information which is already present here. Consequently, the SOAP messages are not documented, for example, as they are derived from operation inputs and outputs mechanically. The same holds for types: for example we do not document defined elements which correspond 1:1 to types but are required due to the literal SOAP encoding.

For exact definition of the operations and types see the WSDL file.

**TODO:** *ljocha: Add fully functional WS examples - in Java, Python, C?*

Aby se na to neapomnelo:

perl-SOAP-Lite-0.69 funguje perl-SOAP-Lite-0.65 ne (stejne rve document/literal support is EXPERIMENTAL in SOAP::Lite ), tak ma asi pravdu

musi mit metodu ns()

# 5 L&B NOTIFICATION API

The L&B notification API is another kind of L&B consumer API which provides streaming publish/subscribe model instead of query/response model. It is designed to provide the same information and use the same query conditions encoding as the consumer API described in sec. 4

Basic usage of the L&B notification API is described in the L&B user's guide ([2]) in section "Tools" as there is described a tool called `glite-lb-notify` which is a command line interface wrapper around the L&B notification API. Its source code can also serve as a complete exaple of the L&B notification API usage.

The L&B notification API have currently fully implemented C language binding and partially implemented C++ binding.

## 5.1 HEADER FILES

`glite/lb/notification.h` Prototypes for all notification API functions.

## 5.2 CALL SEMANTICS

The API have two main parts: notification subscription management and receiving data. Each subscription (registration of notification) have its unique identifier called *Notification ID* represented by type `edg_wll_NotifId`. This ID is returned to the caller when creating a new notification and it is used by receiver to get data from the notification.

The API uses `EDG_WLL_NOTIF_SERVER` context parameter to set the source server (L&B server name and port).

The typical notification workflow consist of 3 tasks:

- Create a new notification registration based on given conditions.
- Refresh the registration. Each notification registration is soft-state registration and must be regullarly refreshed by the owner.
- Receiving the data from notification. The L&B infrastructure provides data queuing and garanteed delivery (while the registration is valid).

The client notification library contains a code providing a pool of receiving sockets/connections to optimize a parallel receiving of notifications.

For complete reference of all API funcions please see the header file. The next sessions briefly describe main facts about API funcions.

## 5.3 NOTIFICATION SUBSCRIPTION AND MANAGEMENT

- *New notification* is created using `edg_wll_NotifNew` call. The call needs properly initialized context and returns a unique notification ID. To create a new notification the same encoding of conditions as for the L&B query/response API is used (sec. 4.2.5).

  L&B 2 and higher

  In version 1.x there is a restriction that at least one particular JobId must be defined. Since L&B 2.0

you can make a registration based on other attributes without referencing a particular JobId (you can select owner, VO, network server). It is also a feature of L&B 2.0 and higer versions, that you can use attributes derived from JDL (VO).

- *Refresh of a notification*. When a new notification is created using `edg_wll_NotifNew` call, the notification validity parameter is intended to set the refresh period, not the lifetime of the notification itself. The owner of notification must periodically call `edg_wll_NotifRefresh` to ensure validity of the notification. See also next sections.
- It is possible to *change existing notification* (its conditions) by `edg_wll_NotifChange` call.
- If the user does not want to receive notifications anymore, `edg_wll_NotifDrop` call *removes the registration* for notifications from L&B server.

## 5.4 RECEIVE DATA

To receive data from a notificaton the API provides `edg_wll_NotifReceive` call. It returns first incoming notification if at least one is available or waits for a new one. The maximal waiting time is limited to a specified timeout. You can also set the timeout to zero if you want to poll.

If the user wants to move the client receiving the notifications to a different machine than where the registration was done, it is possible. The client must use the `edg_wll_NotifBind` call to inform the notification infrastructure (interlogger) about its location change.

The notification API cleanup procedure should be called when finalizing the client (`edg_wll_NotifClosePool` and `edg_wll_NotifCloseFd` calls – where the later is optional – see the next section).

## 5.5 ADVANCED ASPECTS

### 5.5.1 EXTERNAL VERSUS INTERNAL MANAGEMENT OF NOTIFICATION SOCKET

A notification socket used by `edg_wll_NotifReceive` call to receive the notifications is automatically created during the `edg_wll_NotifNew` or `edg_wll_NotifBind` calls.

It the user wants to use its own socket (for example to be used in main select() call) it can be created and closed by the user and set as a parameter (fd) to all calls mentioned above.

When using automatically created socket it must be closed explicitly by calling `edg_wll_CloseFd`.

### 5.5.2 MULTIPLE REGISTRATIONS

Each user can register for multiple notifications (call `edg_wll_NotifNew` function more than once). Every registration gets its own notification ID and must be managed separately (refresh, change, drop). But the `edg_wll_NotifReceive` call is common for all the registrations created in the same context (all previous `edg_wll_NotifNew` calls).

If the user wants to distinguish between multiple registrations it is needed to inspect a notification ID value of each received notification.

A `edg_wll_NotifBind` works in similar way like `edg_wll_NotifNew`. For each notification ID it must be called once and subsequent `edg_wll_NotifReceive` call will work with the whole set of registrations. Will receive a first notification from any of registations.

### 5.5.3 OPERATOR CHANGED

L&B 2 and higher

The notification events are generated by LB server based on primary events send by grid components. Each of the primary events (called LB events) generates one notification event to be possibly sent to the client but not each LB event for example changes the job state. You can use notification conditions to filter only the notification events you want to receive, for example $jobstatus = done$. If you want to receive all job status changes you need to setup a condition on job status attribute using special unary operator `CHANGED`. Otherwise (without any condition) you will receive more events that you want – even events where job state was not changed. Operator `CHANGED` is available since L&B 2.0.

### 5.5.4 RETURNED ATTRIBUTES

L&B 2 and higher

Each LB notification contains a structure describing job state including job's JDL. For optimization purposes the API user can set the JDL flag in `edg_wll_NotifNew` flags parameter to prevent sending of unnecessary JDL data with each notification.

### 5.5.5 TIMEOUTS

A user of the notification API should distinguish between various timeouts:

- *Registration validity timeout.* Each registration is soft-state entity which must be refreshed in a given timeout. If there is no refresh received by the LB server in validity timeout period the registration is dropped. On the other hand for that timeout all events are queued in the LB infrastructure for the case of client's temporary unavailability.

  The registration validity timeout can be set by the user when creating a new registration but only to a reasonably short time period. The validity of a registration is driven by the refresh process not the timeout itself. For a exaple of registration management via the refresh calls please see the `glite-lb-notify` source code as mentioned above.

- *Receive call timeout.* The timeout used in the `edg_wll_NotifReceive` call is inteded just to control the receiving loop. It is the maximum time the API can spend in the call before returning the control to user code.

## 5.6 REGISTERING AND RECEIVING NOTIFICATION EXAMPLE

The following example registers on L&B server to receive notifications triggered by any event belonging to a given user and waits for notification (until `timeout`).

First we have to include neccessary headers:

**File:** notif_example.c

```
26 #include "glite/security/glite_gss.h"
27 #include "glite/lb/context.h"
28 #include "glite/lb/notification.h"
```

Define and initialize variables and context. During context initialization user's credentials are loaded and environment variable `GLITE_WMS_NOTIF_SERVER` is used as a LB notification server:

**File:**notif_example.c

```
65      edg_wll_Context         ctx;
66      edg_wll_QueryRec        **conditions;
67      edg_wll_NotifId         notif_id = NULL, recv_notif_id = NULL;
68      edg_wll_JobStat         stat;
```

Set the query record to *all user's jobs*:

**File:** notif_example.c

```
85      conditions[0][0].attr = EDG_WLL_QUERY_ATTR_OWNER;
86      conditions[0][0].op = EDG_WLL_QUERY_OP_EQUAL;
87      conditions[0][0].value.c = user;
```

New registration based on prepared query record is created and a unique notification ID is returned:

**File:**notif_example.c

```
91      if (edg_wll_NotifNew(ctx, (edg_wll_QueryRec const* const*)conditions,
92         0, -1, NULL, &notif_id, &valid)) {
93              char    *et,*ed;
94
95              edg_wll_Error(ctx,&et,&ed);
96              fprintf(stderr,"%s: edg_wll_NotifNew(): %s (%s)\n",argv[0],et,ed);
97
98              free(et); free(ed);
99              goto register_err;
100     }
101     fprintf(stdout,"Registration OK, notification ID: %s\nvalid: (%ld)\n",
102             edg_wll_NotifIdUnparse(notif_id),
103             valid);
```

The `edg_wll_NotifReceive` call returns one notification. If no notification is ready for delivery, the call waits until some notification arrival or timeout:

**File:** notif_example.c

```
109     if ( (err = edg_wll_NotifReceive(ctx, -1, &timeout, &stat, &recv_notif_id)) ) {
110             if (err != ETIMEDOUT) {
111                     char    *et,*ed;
112
113                     edg_wll_Error(ctx,&et,&ed);
114                     fprintf(stderr,"%s: edg_wll_NotifReceive(): %s (%s)\n",argv[0],et,ed);
115
116                     free(et); free(ed);
117                     goto receive_err;
118             }
119             fprintf(stdout,"No job state change recived in given timeout\n");
120     }
121     else
122     {
123             /* Check recv_notif_id if you have registered more notifications */
124             /* Print received state change */
125             printf("jobId : %s\n", edg_wlc_JobIdUnparse(stat.jobId));
126             printf("state : %s\n\n", edg_wll_StatToString(stat.state));
127             edg_wll_FreeStatus(&stat);
128     }
```

**TODO:** *zminit http interface - podporujeme ho jeste? tusim ze fila to nejak resuscitoval*

# REFERENCES

[1] E. Laure, F. Hemmer, F. Prelz, S. Beco, S. Fisher, M. Livny, L. Guy, M. Barroso, P. Buncic, P. Kunszt, A. Di Meglio, A. Aimar, A. Edlund, D. Groep, F. Pacini, M. Sgaravatto, and O. Mulmo. Middleware for the next generation grid infrastructure. In *Computing in High Energy Physics and Nuclear Physics (CHEP 2004)*, 2004.

[2] A. Křenek et al. L&B User's Guide. http://egee.cesnet.cz/en/JRA1/LB/.

[3] A. Křenek et al. L&B Administrator's Guide. http://egee.cesnet.cz/en/JRA1/LB/.

[4] A. Křenek et al. L&B Test Plan. http://egee.cesnet.cz/en/JRA1/LB/.

[5] EGEE JRA1. EGEE Middleware Architecture—Release 2. https://edms.cern.ch/document/594698/.