

DataGrid

Logging and Bookkeeping Service for the DataGrid



Document identifier: **DataGrid-01-TEN-0109-1_0**

Date: **October 8, 2001**

Workpackage: **WP1**

Partner: **CESNET**

Document status: **DRAFT**

Deliverable identifier:



Delivery Slip

	Name	Partner	Date	Signature
From	Aleš Křenek, Daniel Kouřil, Luděk Matyska, Jan Pospíšil, Jiří Sitera, Miroslav Ruda, Zdeněk Salvet, Michal Vocu			
Verified by				
Approved by				

Document Log

Issue	Date	Comment	Author

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files



Contents

1	Overview	5
1.1	Used terms	5
2	Requirements and assumptions	7
2.1	Workload management architecture assumptions	7
2.2	Job identification	8
2.2.1	Different job ID's	8
2.2.2	dg_jobid format	9
2.2.3	Security considerations	10
2.3	Event types and sources	10
2.3.1	Event types	11
2.3.2	Event sources	12
2.4	Requested functionality	13
3	Architecture	13
3.1	Main components	14
3.2	Functional model and other design decisions	15
3.2.1	Data flow in L&B subsystem	15
3.2.2	Local persistence in local logger and inter-logger	15
3.2.3	Bookkeeping server and query cache	16
3.2.4	User notification	16
4	API's specification	16
4.1	Logging API	17
4.2	L&B server API	17
4.2.1	Datatypes	18
4.2.2	Functions	18
5	Implementation	22
5.1	Logging API	22
5.2	Local logger	22
5.3	Inter-logger	23
5.4	Bookkeeping server	23
5.5	Logging server	23
5.6	Open issues	24
	Appendices	25
	Appendix A Events	25



Appendix B	Logging API usage	29
Appendix C	UML diagrams	33
Appendix D	Use cases: L&B server API	37
Appendix E	C++ L&B server API	40

1 Overview

The purpose of this document is to provide description of logging and bookkeeping service which will be available as part of the scheduling/brokerage services developed within the WP1. The document contains definition of basic terms, overview of the logging and bookkeeping service (L&B), API for log generation and API for access to the stored logs and a brief description of the proposed implementation. It also includes a list of pre-defined events together with their “use case” clarification.

Section 2 covers the assumptions and requirements for the L&B service, while the overall architecture is presented in section 3. Following section is devoted to API’s, data format and main events and the last section briefly describes the proposed implementation. In the appendix, more detailed description of the API use can be found as well as UML diagrams covering main features of the L&B service.

The document presents a “work in progress”, where (almost) all presented ideas are still subject to discussion and potential changes. In the same time, this document serves as a basis for Month 6 architecture paper sections on logging and bookkeeping services and should serve as a guide for the implementation of these services in the preliminary version planned between Month 6 and 9. Current version of this document as well as API’s header files and examples of their usage are available in WP1 CVS repository:

```
CVSROOT           :pserver:lindir.ics.muni.cz:/export/cvs-wp1
papers/cesnet-M9  lb_draft.tex – this document
sw/LB/api/include dglog.h – the logging API
                  lbapi.h – L&B server API
sw/LB/clients/src  usage examples
```

1.1 Used terms

Monitoring – grid service collecting information about running jobs. While some of the information provided by the monitoring service is (could be) used by the logging and bookkeeping service, it by itself is not part of logging and bookkeeping service and as such is not described in this document.

In more general view, *monitoring* is any service which takes care of (“monitors”) some component of the system. Under this view logging and bookkeeping are part of monitoring, however, we use term *monitoring* in its more restricted sense as a service taking care of running jobs only.

Bookkeeping – grid service providing users with information about their jobs: job definition (in JDL), job status (see below in Fig. 1) and related information, job resource consumption and possibly user defined data. This service stores short-term (volatile) data about currently active jobs.

Logging – grid service for storing long term (persistent) information about jobs and the scheduling system itself. This data serve mainly for scheduler debugging purpose and post-mortem analysis of job executions, but it may be used for generating statistics about job scheduling. Some data are stored both by bookkeeping and logging service, but they are mainly used for different purposes.

Accounting – grid service providing statistics about resource usage, job resource consumption, etc. This service is not part of logging and bookkeeping services and is not described here.

Job status – every job during its lifetime passes through various phases. These phases depicted on Fig. 1 are:

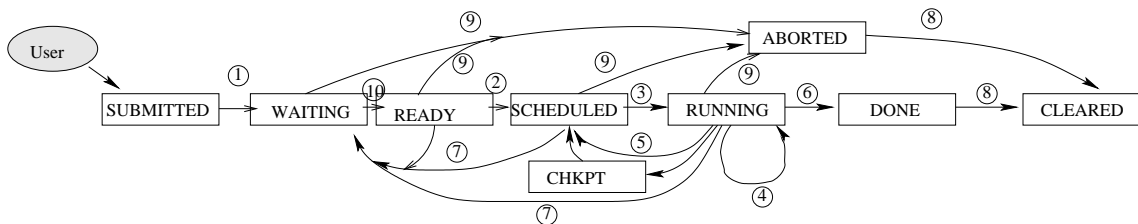


Figure 1: Job Life Cycle

- SUBMITTED ...job is submitted by the user but not yet processed by Resource Broker (for example it may be waiting in User Interface)
- WAITING ...job is waiting in the queue in the Resource Broker for various reasons (no appropriate ComputingElement (cluster) found, required dataset is not available, dependency on other job)
- READY ... appropriate ComputingElement found, job is transferred to the ComputingElement via job submission service (e. g. Condor-G for M9)
- SCHEDULED...job is waiting in the queue on the ComputingElement
- RUNNING ...job is running
- CHKPT ...job is checkpointed for external reason (e. g. cluster maintenance) and is waiting for restart
- DONE ...job exited
- ABORTED ...job was aborted for various reasons (waiting in the queue in Resource Broker or ComputingElement for too long, over-use of quotas, expiration of user credentials, ...)
- CLEARED ...output files were transferred to the user, job is removed from bookkeeping database



Job status is usually not stored directly in the logging and/or bookkeeping databases, but it can be always deduced from the information actually stored. However, information about some states may not be available (e. g. the CHKPT state is not detectable if no relevant information is provided by the local resource management system).

2 Requirements and assumptions

2.1 Workload management architecture assumptions

The mechanism used for logging and bookkeeping is based upon a “push” model, components generating events actively send them to the logging and bookkeeping database.

Proposed architecture of the bookkeeping service is designed to support two basic designs of Resource Broker — a community scheduler running on a highly-available server and a personal scheduler running on a user’s personal machine. This is the main reason for logical separation of Resource Broker and bookkeeping server — the bookkeeping server should be accessible most of the time while the personal scheduler may be used only for the actual job submission and then can be disconnected from a net for a long time.

In PM9 release, several workload management system components are expected to log information to the logging and/or bookkeeping service — Resource Broker itself, Job Submission Service (probably some wrapper around Condor-G because we don’t expect that source code of Condor-G will be available for PM9 release and we also would like to stay independent on Condor-G as much as possible) and Globus job-manager. User Interface (UI) will log just one event — the submission of the job, and it will use the components of L&B service available on the Resource Broker (only logging API will be used directly by User Interface).

Job submission and control can be split into several phases. A user submits her job by User Interface using `dg_submit`. The User Interface looks for appropriate Resource Broker and when found, logs the submission event to bookkeeping server announced by Resource Broker. The job is in SUBMITTED (see Fig. 1) state. While the job is within the Resource Broker, it is in state WAITING. When a suitable resource (ComputingElement queue or single node) is found, the job is submitted with `condor_submit` to Job Submission Service (JSS). This corresponds to the change of state to READY. JSS then submits the job to Globus (`globusrun` command). Globus starts job-manager and submits the job to a local resource management system. The job is thus transferred to the SCHEDULED state, which, still under the Globus control, can change to RUNNING, CHKPT, DONE and ABORTED states. The final, CLEARED state is achieved when user transfers all the output data (“clears the job”) or when a pre-specified timeout occurs (user is not reacting to the ABORT or DONE states).

The proposed model supports logging of dynamic values, which describe some speci-



fic property or feature of a running job and may change in time (e. g. the actual CPU time, the actual disk and/or memory consumption etc.). We assume such information will be collected by job-manager (in the worst case by polling the local system) and sent periodically to bookkeeping server as part of a specific event. Currently we distinguish two types of collected information, namely cumulative (e. g. CPU time) and non-cumulative (e. g. memory consumption). For cumulative properties we expect not to store all the values in the database but to overwrite the corresponding cell with the most actual value (naturally timestamped). Non-cumulative properties will be stored as sent in order to allow e. g. creation of appropriate profiles.

For M9 release, several restrictions apply: the CHKPT will not be detected (this will require more close collaboration with local resource management systems) and only those dynamic information which are already available to Globus job-manager will be logged (most notably, it may not be able to log job CPU time if the local resource management system does not provide this information — available in LSF or PBS but not so easily in e. g. fork only systems).

2.2 Job identification

We require a grid-wide unique job ID for the bookkeeping mechanism to work correctly. Moreover, it is desirable that the job ID is unique forever in order the persistent logs remain unambiguous.

2.2.1 Different job ID's

During its life time, each job is assigned several ID's:

dg_jobid – job-id in DataGrid.

This ID is generated by the Grid software (more precisely by User Interface) and is used by the user for communication with a bookkeeping server (and other grid components if necessary). The bookkeeping server address should be derivable from this ID (see section 2.2.2).

jss_jobid – job-id in JSS (Condor-G ID for M9).

This ID is returned by JSS when it accepts a job. It is used for getting information about job from JSS (e. g. job status or `globus_jobid`).

globus_jobid – job-id in Globus.

This ID is granted by job-manager and it can be used to get information about job from job-manager. This ID is used by JSS for job monitoring.

local_jobid – job-id in local resource management system.

The job-manager uses this ID when communicating with local resource management system to obtain a job state.



For simplicity we decided to use `dg_jobid` for unique job identification in logging and bookkeeping service. That means that every component using logging and bookkeeping service has to know `dg_jobid` of the job. This is true for the Resource Broker and can be easily guaranteed in JSS (e. g. the Condor-G wrapper). In case of job-manager it can be guaranteed with small extension of its RSL handling (one new RSL attribute for `dg_jobid` must be defined and discerned by job-manager). Such extension will mean only a minor intervention to a job-manager source code and may also allow us to propagate `dg_jobid` to applications.

2.2.2 `dg_jobid` format

We assume `dg_jobid` is generated by the User Interface on job submission. The proposed `dg_jobid` construction minimizes number of grid components that have to be involved in its generation while preserving a reasonably high probability of uniqueness. Therefore `dg_jobid` should be composed of:

1. IP of the User Interface machine,
2. timestamp (return value of the `time()` system call),
3. process ID (if more UI instances may occur on the same machine),
4. sequence or just random number (if the User Interface submits jobs in batches and more than one per second can be submitted),
5. bookkeeping server name and port.

Items 1–4 serve the only purpose—form a grid-wide unique identification. We suppose those numbers to be combined into a single string, either as one large number or with some URL-compatible separators (dot, slash, ...). Exact form of the string is up to the User Interface which generates it, all the other grid components handle it literally. It should be emphasized that no grid components should rely on either presence of those components or their meaning. `dg_jobid` is just a unique string and its exact structure may be changed without notice.

Unfortunately, in PM9 we cannot rely on information services to provide an efficient mapping of `dg_jobid` to all information that is required by the User Interface as well as the L&B subsystem, mainly the address of the bookkeeping server that handles this job. Anyway, the use of GIS for this purpose is questionable, as it means that the same static information is repeatedly retrieved (e. g. `dg_jobid` to bookkeeping server address mapping should not change during the job life time). Therefore we include the necessary bookkeeping server contact information in the `dg_jobid` as well. Similar situation arises with the Resource Broker address. At certain situations (e. g. job cancel) the Resource Broker that scheduled the job has to be contacted. For this purpose, the RB address is



derivable directly from the PM9 `dg_jobid` format too. It is either identical to the bookkeeping server or included explicitly as a suffix (see below).

Being inspired by Globus, `dg_jobid` is a valid URL pointing to a bookkeeping server¹ that maintains information on the job, composed of the server name (and port if not default) and the unique string generated by the User Interface:

```
https://bserver.host.name[:port]/  
unique_string[?RB=resource_broker]
```

The address of the bookkeeping server is provided by the Resource Broker during the initial handshake with the User Interface.

For post-PM9 releases, we are aware of possible job batches (i. e. a job “forks” into a series of sub-jobs). We assume the “parent” `dg_jobid` will become a prefix of the sub-jobs `dg_jobid`’s. The appropriate API will be extended in order to allow handling of such job hierarchies.

2.2.3 Security considerations

Logging events with faked `dg_jobid`’s might cause serious confusion at the bookkeeping as well as at the logging server. Therefore the servers associate a `dg_jobid` with a user when the first event of the job is logged and check it on any further events.

An alternate solution is a certified *dg_jobid allocation service*. The most natural provider of this service is the bookkeeping server. The service would issue unique `dg_jobid`’s and publish the “ownership” of `dg_jobid`’s via GIS. However, we think drawbacks of this solution (e. g. the dependence on accessibility of GIS to submit a job) still prevail over its attractions.

2.3 Event types and sources

Given the job life cycle (see Fig. 1), we have to identify events that have to be logged to keep the job status information up-to-date. Essentially the bookkeeper needs to know about job state changes (edges in the state graph). State changes involving more than one scheduler component should produce two events (one for each participating component), thus enabling easier detection of communication problems. Additional events important for bookkeeping, but not relevant to job state changes, may have to be defined as well. The verbosity level of log messages can be controlled by user while certain minimal level will be guaranteed and enforced (i. e. the L&B service cannot be switched off).

¹According to preliminary L&B server protocol design, a regular GET https request on this URL should even return job status in some XML-based form.

2.3.1 Event types

Every event should carry information necessary for identifying the time and origin of the event (`dg_jobid` and string identifying the event sender) as well as event type. Event type determines additional data that the event contains. Event types are organized hierarchically as follows:

- Events concerning job transfer between components:

JobTransferEvent – The sender of this event attempted to transfer a job to some other component. This event contains identification of the other component which the job is being sent to and possibly the job description being transferred in language this component accepts (RSL for Globus job-manager, ClassAd for Condor-G). Result, i. e. succes or failure, of the transfer (as seen by the sender) is also included.

JobAcceptedEvent – Receiving component accepted the job. This event should contain the locally assigned job id (`jss_jobid` or `globus_jobid`) (with the exception when job is returned to scheduler via the line 7 in Fig. 1). This event is issued by the receiving component, if possible (in the case of local resource management system accepting job, this event is issued by Globus job-manager).

JobRefusedEvent – Receiving component announces that it could not accept the job, the reason being a part of the event.

- Events concerning job state changes during processing inside a particular component (e. g. CE):

JobAbortEvent – Job processing was stopped due to system conditions or by user request, the event contains abort reason.

JobRunningEvent – The job was started on the selected computing element (its node) by the local resource management system.

JobChkptEvent – The job was checkpointed, the event informs about the reason for checkpoint (if possible).

JobDoneEvent – The job ends; this event should contain process exit code (this may not be achievable for M9 release, as some local resource management systems do not currently provide such information to job-manager).

JobClearedEvent – Job results (output files) were handed over to the user; all temporary files were removed.

- Events associated with Resource Broker only:

JobMatchEvent – Appropriate match (i. e. the CE) was found. This event contains the ID of selected CE.

JobPendingEvent – This event is logged by RB when there was no match found for given job (e. g. not enough information was found, requested resources are not available at the moment, job dependency condition is not satisfied). The event describes reason why the jobs remains in RB queue.

- System specific events:

ComponentStatusEvent – Every component announces its important status change by logging this event; e. g. it should inform if the component was restarted gracefully.

ClusterStatusEvent – Similar notification on important status change of a computing element (node), e. g. reboot and its reason.

Those events are not related to a particular job. Therefore they are not handled by the bookkeeping system, they are logged only. For post-PM9 we assume this functionality will be provided by the monitoring services.

- Dynamic events:

JobStatusEvent – This event is generated periodically and contains information about resources consumed by the job. This event serves for efficient handling of user queries by the bookkeeper by eliminating the need for direct communication with the computing element.

2.3.2 Event sources

The following table describes the correspondence among the workload management architecture components and logging and bookkeeping event types:

UI	<i>JobTransferEvent</i>
Resource Broker	<i>JobTransferEvent, JobAcceptedEvent, JobRefusedEvent, JobAbortEvent, ComponentStatusEvent, JobMatchEvent, JobPendingEvent</i>
JSS	<i>JobTransferEvent, JobAcceptedEvent, JobRefusedEvent, JobAbortEvent, ComponentStatusEvent</i>
Globus job-manager	<i>JobTransferEvent, JobAcceptedEvent, JobRefusedEvent, JobRunningEvent, JobChkptEvent, JobDoneEvent, JobAbortEvent, JobStatusEvent, ComponentStatusEvent</i>

Generating events in the Resource Broker is rather straightforward as the scheduler is being written by the DataGrid project. The same applies, in general, to the JSS. For M9,

this component will probably be made of Condor-G that will be wrapped up by an additional layer responsible for encapsulating Condor-G functionality and for logging events instead of Condor-G itself. This wrapper can obtain necessary events by parsing Condor-G log files or by intercepting some of the Condor-G calls. The Globus job-manager can be easily extended to log job state changes. Moreover, the Globus job-manager periodically polls the job status from the local resource management system and it can thus be easily enhanced to send this information using logging API to logging and bookkeeping service.

2.4 Requested functionality

The logging API should conform to the following requirements:

Non-blocking calls: Applications and services calling the logging API should never be blocked indefinitely by the logging operation (e. g. when logging service daemon is down).

Atomicity: Logging of message should be an elementary operation in the API (i. e. no partial message could be stored).

Reliability: The functionality of the logging service components should be checked by another service — e. g. by globus-daemon-master.

Local persistence: The logging service must ensure that data are not lost locally, before transmission to the log or bookkeeping server. This persistence gives the log a transaction property (see section 3) which will be used during the recovery process (after local disruption of service).

Logging API will not be made available for user applications in the PM9 release. API for logging and bookkeeping information retrieval will be available in order to implement user commands `dg-job-status()` and `dg-get-logging-info()` as described in the Datamat “Job Submission User Interface”.

3 Architecture

This section describes proposed design of logging and bookkeeping service architecture. This architecture design is based on requirements and assumptions summarized in previous sections.

The basic service architecture is presented in Fig. 2. This picture has two main parts: the first describes main architecture components and their interaction and the second outlines data flow through the logging and bookkeeping subsystem.

Supposed use of L&B subsystem and its interaction with other components in scheduling/brokerage services is expressed on UML diagrams in Appendix C, namely Fig. 3 and Fig. 4.

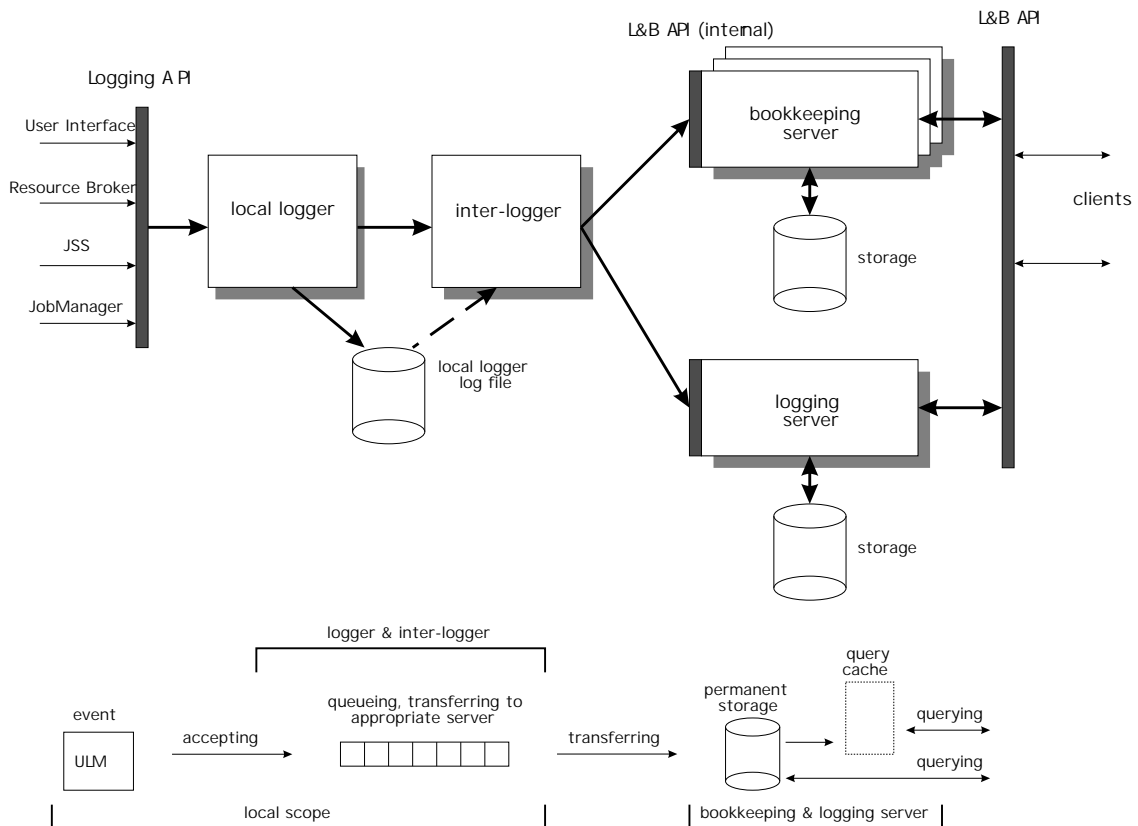


Figure 2: Logging service architecture overview

3.1 Main components

The design of proposed architecture defines main components as follows:

a neni

logging API – This API is used by event sources to pass logging information to the sub-system. The API is implemented as a simple logging library. The source must follow prescribed message format and semantic rules.

logging and bookkeeping server API – User API. It is used for querying the L&B sub-system, e. g. asking for the status of a particular job. This API provides unified access to both logging and bookkeeping information. The API is fully described in the next section.

local logger – Process which accepts the messages from their sources (via logging API) and is responsible for fast and reliable transfer of data to the L&B subsystem. It



is as simple as possible but implements persistence by simple (transaction) log file. The normal information flow is implemented by inter-process communication with inter-logger (see next item), but backed up by log file which acts as source for inter-logger recovery.

inter-logger – Process responsible for transferring data to target storage — bookkeeping or logging server. Inter-logger maintains message queues and implements their handling in environment with possible communication problems.

bookkeeping server Bookkeeping server accepts messages from inter-logger and manages primary data storage. The server is responsible for implementing user queries accepted by L&B API. The bookkeeping server must also implement mechanism for job status queries — i. e. implement message analyses based on a job life cycle finite automaton.

logging server Like bookkeeping server but deals with persistent log type messages — information is not analyzed or managed by any way except that query API access is provided.

3.2 Functional model and other design decisions

3.2.1 Data flow in L&B subsystem

Data are accepted by logging API and local logger. They are formatted to predefined format (ULM — see next section) and stored in the appropriate queue. The queue selection is based on `dg_jobid`, message type and logging level. Message queuing is managed by inter-logger process which is also responsible for transferring messages from queues to servers (logging and bookkeeping servers). Each server is responsible for permanent storage of this data and implementation of support for user queries.

3.2.2 Local persistence in local logger and inter-logger

Local logger and inter-logger implement local persistence. Data flowing through queues are logged to log file and this log file is used for queue recovery when inter-logger starts. Local scope on Fig. 2 means that the local logger and inter-logger must be located inside the same possible communication partition of network as appropriate data sources. It is assumed that local logger and inter-logger are implemented as two processes on one node. Separating this functionality to two processes is design decision based on the need of maximal independence between message acceptance and actual processing.

3.2.3 Bookkeeping server and query cache

All events relating to existing (not cleared) jobs will be recorded in bookkeeping server's internal database. This is the primary information stored, the job status is computed on demand from the sequence of events. For efficient implementation of the job status queries we propose query cache mechanism. This cache implements job life cycle finite automaton as asynchronous independent process which retrieves relevant log messages from the bookkeeping server database and maintains current state of each job. The query only reads state of appropriate finite automaton.

It can be considered as optional speed-up mechanism build on top of job life cycle finite automaton implemented (in the first stage) as part of appropriate L&B API query.

The L&B cache may not be available for the PM9.

3.2.4 User notification

Bookkeeping service can be used for user notification. We propose a mechanism where user can register her notification requests (either using the User Interface or, later on, using the monitoring service) with a notification process that runs on top of the bookkeeping server. This process filters events in the bookkeeping server in near real-time and whenever an event that a user registered for occurs, a user specified action is executed.

This notification service will not be available for M9, the first post-M9 version will allow to use only pre-defined set of asynchronous (e. g. sending an e-mail) and synchronous (e. g. sending an Instant message) actions. We assume that the notification will be, at least to some extent (i. e. pre-defined again), parameterizable. The user will be allowed not only to register for an event, but also to specify which parameters associated with the event will make the notification message. A good example is the *JobRunningEvent* — the possible parameterization could include time, the computing element information etc.

After evaluation of user experience with the pre-defined set of actions, and based on user (e. g. WP8 to WP10) requirements, a possible extension of the notification service will include more sophisticated event processing (e. g. "All my jobs finished") and also handling user defined events (generated by the application and sent to the bookkeeping server).

4 API's specification

As shown in Fig. 2 there are two interfaces to the logging & bookkeeping system. The *logging API* collects the events being logged. On the other hand, the *L&B server API* provides interface for querying the logging and bookkeeping servers, as well as control of the bookkeeping process (e. g. clearing a job).

We suppose most of the L&B system network communication to be authenticated. As the proposed authentication layer is GSI, none of the API functions has to contain an



explicit authenticator among their arguments. Instead, the implementation determines the appropriate path to the credentials from its environment.

4.1 Logging API

For the sake of simplicity of use there is only a single autonomous function to be called in this API.

```
int dgLogEvent(char *jobId, char *source, dgLBEventCode event,
               int lvl, char *fmt, ...)
```

`jobId` dg_jobid in its string form (it is passed to the inter-logger literally, there is no need to parse it here)

`source` Event source identification (i.e. Resource Broker, JSS, job-manager, ...)

`event` Type of the event being logged (from the predefined set, see 2.3)

`lvl` Level of the event, as defined in the ULM draft (*draft-abela-ulm-05*), e.g. Debug, Usage, System, ...

`fmt` printf()-like format string containing the assignments *key1 = value1 key2 = value2 ...*. The sets of required and optional keys are specific to each event type.

... List of arguments according to `fmt`

The function constructs a ULM event string, parses it according to the L&B system semantics, and passes the string to the local logger. A version for systems not supporting variable number of arguments (e.g. Python with SWIG) will be also provided.

Successful completion (return value of 0) ensures that the event record has been stored by the local logger. Otherwise an error code is returned. Possible failures on the L&B system level include:

- EINVAL The constructed event string failed to parse.
- ENOSPC The event cannot be logged locally due to lack of disk space.
- EAGAIN The call returns in order not to block. The event still may get logged but this is not guaranteed anymore.

In most cases the logging destination (i.e. the local logger address) is the default `x-netlog://localhost:port` (where `port` is the default local logger port). Therefore it is not included among the function arguments. If necessary (e.g. in User Interface) the default value may be overridden by the environment variable (DGLOG_DEST=host:port).

The logging API usage is more thoroughly described in Appendix B.

4.2 L&B server API

In the following the L&B API is presented in its C binding which is considered to be the primary implementation. The Appendix E sketches a C++ binding as well. A UML



diagram of the class hierarchy is shown in appendix on page 36. Appendix D contains examples of usage of the API calls.

4.2.1 Datatypes

The C L&B server API defines the following types

`dgJobId` Opaque type, internal `dg_jobid` representation.

`dgLBErrCode` Error code returned by most of the functions. Covers standard system errors (specified in `errno.h`) as well as L&B system specific codes (above 1024).

`dgLBContext` An opaque type representing the context of calls to the L&B API functions.

`dgLBEventCode` Numeric code to identify event type.

`dgLBEvent` A union type for representing event. The design of structures representing various event types are inspired by similar types used in the Xlib API. This union contains overlapping entries for each specific type as well as any that can be used to access the common fields and `type` to distinguish actual event type.

`dgLBQueryRec` Structure for specifying general L&B server queries. See the description of `dgLBQuery()` below.

`dgLBJobStatCode` Numeric code to identify job status.

`dgLBJobStat` A union type for representing job status. Similar design to `dgLBEvent`, i.e. contains `stat` field to distinguish the actual status, any for common fields and status specific entries.

4.2.2 Functions

According to their return type all the functions return 0 (or non-NULL pointer) on success and non-zero return code (or NULL pointer) on failure. Further details of failure can be found via the `dgLBError()` API call.

Output arguments to the functions are implemented as “**” types, i.e. the functions expect a pointer to a pointer variable of appropriate type. Passing NULL to those arguments is allowed in general—no object is returned in this case, however, the particular function performs all its other tasks and side effects, e.g. parses the input string. The returned objects are `malloc()`'ed, therefore should be `free()`'ed when not needed anymore. If a function returns a list, it is a `malloc()`'ed, NULL-terminated array of pointers (to `malloc()`'ed objects again).



Internal dg_jobid representation

Create internal dg_jobid from its components (see Sect. 2.2.2)

```
int dgJobIdCreate(  
    char * bserver,          IN    bookkeeping server hostname (and  
                                port if not default)  
    char * unique,          IN    unique dg_jobid component  
    char * resourceBroker, IN    resource broker hostname (and port) if  
                                not identical to bookkeeping server  
    dgJobId * jobId         OUT   created internal dg_jobid  
)
```

Parse/unparse string and internal dg_jobid representations

```
int dgJobIdParse(  
    char * jobIdString, IN    string dg_jobid to be parsed  
    dgJobId * jobId     OUT   parsed dg_jobid  
)  
char * dgJobIdUnparse(  
    dgJobId jobId IN    dg_jobid to be unparsed  
)
```

Extract bookkeeping server and resource broker addresses. In post-PM9 implementation we suppose this will encapsulate GIS queries.

```
char * dgJobIdBserver(  
    dgJobId jobId IN  
)  
char * dgJobIdResourceBroker(  
    dgJobId jobId IN  
)
```

Currently the dg_jobid internal representation functions are designed independently on the core L&B server API. Therefore the context argument is not included as well as the return codes are int rather than dgLBErrCode.

Context handling

Allocate and initialize a context object

```
dgLBErrCode dgLBInitContext(  
    dgLBContext * context OUT   allocated and initialized context  
)
```

Free the context



```
void dgLBFreeContext(  
    dgLBContext context IN context to be freed  
)
```

Error handling

Return error code, standard error message, and optional more detailed error description (e. g. exact filename) that may be filled in by the API implementation. It is supposed to be human readable rather than processed automatically.

```
dgLBErrCode dgLBError(  
    dgLBContext context IN context  
    char ** errText OUT standard error text  
    char ** errDesc OUT additional error description  
)
```

Events

Free the appropriate structure contained within the dgLBEvent union according to the type field.

```
void dgLBFreeEvent(  
    event IN event to be freed  
)
```

Parse ULM event string

```
dgLBErrCode dgLBParseEvent(  
    dgLBContext context, IN context  
    char * eventStr, IN ULM event string to parse  
    dgLBEvent ** event OUT parsed event union  
)
```

Unparse event, i. e. reconstruct a ULM string from a structured event

```
char * dgLBUnparseEvent(  
    dgLBContext context, IN context  
    dgLBEvent * event IN event to unparse  
)
```

Connection to server

Connect to a server, fill in appropriate entries in the context

```
dgLBErrCode dgLBOpen(  
    dgLBContext context, IN context  
    char * server IN server hostname and port  
)
```



Close the connection, free associated data

```
dgLBErrCode dgLBClose(  
    dgLBContext context IN context  
)
```

Querying servers

General Query

```
dgLBErrCode dgLBQuery(  
    dgLBContext context,      IN    context  
    int nrec,                 IN    number of query records  
    dgLBQueryRec *queryRec,  IN    query records  
    dgLBEvent ** events      OUT   list of matching events  
)
```

The dgLBQueryRec structure represents an atomic query condition in the form

attribute_name <operation> value.

Currently supported attributes include dg_jobid, job owner, time, and logging level; operation is one of =, <, >. All the conditions are AND'ed and all matching events available at this server returned.

Convenience wrappers on top of dgLBQuery(): all events of a given job, all events of all user's jobs

```
dgLBErrCode dgLBJobLog(  
    dgLBContext context,  IN    context  
    dgJobId jobid,        IN    dg_jobid of the job of interest  
    int level,             IN    required level, currently debug/nodebug  
    dgLBEvent ** events  OUT   list of events  
)
```

```
dgLBErrCode dgLBUserLog(  
    dgLBContext context,  IN    context  
    char * userId,        IN    user's certificate subject, NULL means  
                            the authenticated user  
    int level,            IN    required level, currently debug/nodebug  
    dgLBEvent ** events  OUT   list of events  
)
```

Bookkeeping specific queries

Current user's jobs



```
dgLBErrCode dgLBUserJobs(  
    dgLBContext context, IN context  
    char * owner, IN user's certificate subject, NULL means  
                    the authenticated user  
    dgJobId ** jobs OUT list of dg_jobid's  
)
```

Status of a given job (always returns full job status information):

```
dgLBErrCode dgLBJobStatus(  
    dgLBContext context, IN context  
    dgJobId jobId, IN dg_jobid of the job of interest  
    int resolution, IN required resolution of the job life  
                    cycle state machine, currently only  
                    DGLB_STAT_STD—the 9-state diagram  
                    shown in this document  
    dgLBJobStat * status OUT returned job status  
)
```

Clear the job, i. e. purge all bookkeeping information

```
dgLBErrCode dgLBClearJob(  
    dgLBContext context, IN context  
    dgJobId jobId IN dg_jobid of the job to be cleared  
)
```

5 Implementation

In this section we present suggested implementation and important implementation decisions.

5.1 Logging API

We suppose that the implementation of logging API will be based on NetLogger, sending the events over TCP. Several NetLogger calls are wrapped into the single logging API call for the sake of simplicity of use. The associated overhead is acceptable as a single event source is not supposed to log at high rate. We provide a C library implementation as well as a shell command.

5.2 Local logger

The local logger will be based on `netlogd` (older C version is favored). The daemon listens on a TCP port, stores incoming messages (one file per `dg_jobid`), and forwards



them to the inter-logger. As the two components run on the same node, no network communication between them has to be considered.

5.3 Inter-logger

There are two main parts in the proposed inter-logger implementation:

1. *Main* – collects messages from local logger and assigns them their destination, where they should be sent; starts new threads
2. *Slave threads* – contact destination servers and look after proper delivery of messages.

The implementation of inter-logger must also include:

- queue recovery mechanism when the inter-logger starts,
- purging stale local log files,
- mechanism for maintaining message queues and their handling in environment with possible communication problems (PM9: we propose one thread per one destination server).

5.4 Bookkeeping server

As shown in Fig. 2 there are two interfaces to the bookkeeping server. Messages are received from inter-logger(s) using internal L&B API and protocol (ULM over subset of http). The L&B server API provides interface to bookkeeping server clients (mainly User Interface programs). L&B server API implementation encapsulate network communication, the communication protocol will be based on http(s) as well.

The events will be stored in a SQL database. The relational data model enables more complex future queries and is more scalable. We consider using the Grid SQLDatabase-Service if this is adequate solution, mySQL otherwise.

As the communication is http-based, we consider using a generic http server (Apache) for the front-end implementation (this remains an open issue).

5.5 Logging server

The implementation of communication protocols handled by logging server is supposed to be very similar to the bookkeeping server. However, using SQL database is still open, another storage type may prove more appropriate for the purpose.

For PM9 release we assume the logging server address being identical to the bookkeeping server.



5.6 Open issues

There are the following open issues that have to be further discussed:

- Authentication and authorization — what are the mechanisms (for brief explanation see section 4), who is authorized to retrieve what information (PM9: user can query her jobs only). The security issues are discussed in another draft (`sec_draft`).
- Logging server address — various requirements on logging destination may appear after PM9. Events may be logged according to the `dg_jobid`, user id, cluster community etc. as well as simultaneously to multiple destinations. We assume the further inter-logger releases will determine the logging destination from the grid information services in some way.
- User defined events — the L&B system should provide support for handling user defined events, including notification on the basis of those. For the inter-logger a mechanism for determinig destination of those events has to be defined. We have to specify also, what processing by the bookkeeping server is supported.



Appendices

Appendix A Events

In the following we define structure of individual logged events, i. e. fields which should accompany each predefined event. This is not an exhaustive list, it is more a “use case” specification, which should serve as a guidance to developers and implementors of programs where the events are generated.

According to the UML specification draft the following fields are required for all log events

DATE = timestamp of the log event (date and time, with millisecond precision)

HOST = name of the issuing host (its canonical DNS name)

PROG = ID or specification of the program issuing the log event (`argv[0]`) in C programs)

LVL = severity level (for PM9 only DEBUG and SYSTEM levels will be allowed)

With the exception of the **LVL** field, the other fields could be “hardwired” into the API and added automatically to any `dgLogEvent` call (this is considered to be more error prone solution).

Precise `dgLogEvent()` calls are presented in the following appendix, here a more descriptive language was used. Only the mandatory fields are commented here, the components are free to add more logged fields, e. g. for debugging purposes.

The job life cycle with respect to individual components is followed:

UI issues just one call, logging a specific instance of *JobTransferEvent* event. The call contains the following fields:

jobID = unique job ID assigned to this job

source = `DGLB_SOURCE_UI`

event = `DGLB_EVENT_JOBTRANSFER`

lvl = `DGLB_LOG_DEFAULT`

fmt = `DGLB_FORMAT_JOBTRANSFER` which specifies the Resource Broker address, the result code of the transfer and the copy of the full ClassAd job specification

RB-accept, the *JobAcceptedEvent* is the first event which should be logged by Resource Broker when the job is accepted. The call contains the following fields:



jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOBACCEPT

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBACCEPT specifying once again the Job ID.

RB-refuse, the *JobRefusedEvent* is the event which should be logged by Resource Broker when the job submitted to it is not accepted. The call contains the following fields:

jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOBREFUSE

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBREFUSE specifying the reason for the job rejection (e. g. not authenticated).

RB-match, the *JobMatchEvent* should be logged by Resource Broker when the match succeeded and the appropriate computing element is found. The call contains following fields:

jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOBMATCH

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBMATCH with the information about the CE selected

RB-pending, the *JobPendingEvent* occurs when the Resource Broker can not complete some operation for other reasons that those already specified. This may happen when, e. g. the GIS is not temporarily available, when waiting for data transfer to the selected SEs etc. The call should contain the following:

jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOB_PENDING

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOB_PENDING specifying the reason of this event



RB-transfer, the *JobTransferEvent* occurs when the job is transferred to JSS. The call contains the following fields:

jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOBTRANSFER

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBTRANSFER which specifies the

RB-abort, the *JobAbortEvent* is issued when job is being aborted for any reason. The call contains:

jobID

source = DGLB_SOURCE_RB

event = DGLB_EVENT_JOBABORT

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBABORT specifying the abort reason

JSS-accept is the JSS instance of the *JobAcceptedEvent* event. The corresponding RB call is closely followed:

jobID

source = DGLB_SOURCE_JSS

event = DGLB_EVENT_JOBACCEPT

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBACCEPT with the Job ID.

JobRefusedEvent, *JobAbortEvent*, and *JobTransferEvent* within JSS are just variants of the corresponding RB calls and are no further discussed here.

Jobmanager logs are mostly just variants of the RB and JSS logs. This is especially true for *JobAcceptedEvent*, *JobRefusedEvent* and *JobAbortEvent*. The “new” events are described below.

Jobmanager-running, the *JobRunningEvent* is issued when the job actually starts to be processed on a particular CE (or CEs). The call should include:

jobID

source = DGLB_SOURCE_JOBMgr



event = DGLB_EVENT_JOBRUN

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBRUN which specify the CE where job runs

Jobmanager-done, the *JobDoneEvent* is issued when job completes. The call should contain:

jobID

source = DGLB_SOURCE_JOBMgr

event = DGLB_EVENT_JOBDONE

lvl = DGLB_LOG_DEFAULT

fmt = DGLB_FORMAT_JOBDONE with exit code if available.

The fields specified are just the mandatory fields and we expect that more detailed information will be logged with all events.



Appendix B Logging API usage

In this section we will describe in more detail the recommended use of logging API by various workload management components. The logging API is defined in header `dglog.h` which has to be included by every L&B client:

```
#include <dglog.h>
```

This header file includes following definitions of event type codes and event sources:

```
/* Predefined event types */
typedef enum _dgLBEventCode {

/* job transfer events */
DGLB_EVENT_JOBTRANSFER,
DGLB_EVENT_JOBACCEPT,
DGLB_EVENT_JOBREFUSE,

/* other job events */
DGLB_EVENT_JOBABORT,
DGLB_EVENT_JOBRUN,
DGLB_EVENT_JOBCHKPT,
DGLB_EVENT_JOBDONE,
DGLB_EVENT_JOBCLEAR,
DGLB_EVENT_JOBSTATUS, /* dynamic job status */

/* system events */
DGLB_EVENT_SYSCMPSTAT, /* component status */
DGLB_EVENT_SYSCLSTAT, /* cluster status */
} dgLBEventCode;

/* Event sources */
#define DGLB_SOURCE_UI           "UserInterface"
#define DGLB_SOURCE_RB          "ResourceBroker"
#define DGLB_SOURCE_JSS         "JobSubmissionService"
#define DGLB_SOURCE_JOBMgr      "GlobusJobmanager"
#define DGLB_SOURCE_LRMS        "LocalResourceManager"
#define DGLB_SOURCE_APP         "Application"
```

For the purposes of logging levels, there is a definition of `dgLBLogLevel` and concerning constants in `dgfbevents.h`²:

```
typedef enum _ULMLogLevel dgLBLogLevel;

#define DGLB_LOG_SYSTEM SYSTEM
#define DGLB_LOG_DEBUG DEBUG

#define DGLB_LOG_DEFAULT DGLB_LOG_SYSTEM
```

where

²automatically included by `dglog.h`



```
/* Log levels (according to draft-abela-ulum-05) */
typedef enum _ULMLogLevel {
    EMERGENCY,
    ALERT,
    ERROR,
    WARNING,
    AUTH,
    SECURITY,
    USAGE,
    SYSTEM,
    IMPORTANT,
    DEBUG,
} ULMLogLevel;
```

These `DGLB_EVENT_*`, `DGLB_SOURCE_*` and `DGLB_LOG_*` constants should be used for the second, third and fourth argument of `dgLogEvent()`. The first argument is string encoded `dg_jobid`, which is assigned by the UI component and is present in all the job-related events. The logging component should extract this string from the job description and use it *as is* for the first argument of `dgLogEvent`. The next arguments are formed according to the event type as follows:

```
/* Event formats */
#define DGLB_FORMAT_COMMON      "DATE=%s HOST=%s PROG=%s LVL=%s DG.EVNT=\"%s\" "
                                "DG.JOBID=\"%s\" "
#define DGLB_FORMAT_JOBTRANSFER "DG.JOB.TRANSFER.DEST=\"%s\" DG.JOB.TRANSFER.RESULT=\"%s\" "
                                "DG.JOB.TRANSFER.JOB=\"%s\" "
#define DGLB_FORMAT_JOBACCEPT  "DG.JOB.ACCEPT.SRC=\"%s\" DG.JOB.ACCEPT.NEWID=\"%s\" "
#define DGLB_FORMAT_JOBREFUSE  "DG.JOB.REFUSE.REASON=\"%s\" "
#define DGLB_FORMAT_JOBABORT   "DG.JOB.ABORT.REASON=\"%s\" "
#define DGLB_FORMAT_JOB RUN    "DG.JOB.RUN.NODE=\"%s\" "
#define DGLB_FORMAT_JOBCHKPT   "DG.JOB.CHKPT.STATUS=\"%s\" "
#define DGLB_FORMAT_JOBDONE    "DG.JOB.DONE.RETCODE=%d"
#define DGLB_FORMAT_JOBCLEAR   "DG.JOB.CLEAR.REASON=%d"
#define DGLB_FORMAT_JOB PENDING "DG.JOB.PENDING.REASON=\"%s\" "
#define DGLB_FORMAT_JOBMATCH   "DG.JOB.MATCH.DEST=\"%s\" "
#define DGLB_FORMAT_JOBSTATUS  "DG.JOB.STATUS=\"%s\" "
#define DGLB_FORMAT_SYSCMPSTAT "DG.SCHED.STATUS=\"%s\" "
#define DGLB_FORMAT_SYSCLSSTAT "DG.SCHED.NODE=\"%s\" DG.SCHED.STATUS=\"%s\" "
```

The suggested use of logging API by the workload management components is described in the following.

User interface

User interface issues this call after an attempt (either successful or not, result is logged using `result` parametr) to send a job to the selected resource broker:

```
status = dgLogEvent(jobid, DGLB_SOURCE_UI, DGLB_EVENT_JOBTRANSFER, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBTRANSFER, resource_broker, result, class_ad);
switch(status) {
    case EAGAIN: /* maybe try again? */
break;

    case EINVAL: /* internal error */
```



```
break;

    case ENOSPC: /* this should not happen in UI... */
break;

    default:
};
```

Resource broker

When receiving job, the resource broker may accept it and log the corresponding event by

```
status = dgLogEvent(jobid, DGLB_SOURCE_RB, DGLB_EVENT_JOBACCEPT, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBACCEPT, from_where, datagrid_jobid);
/* check for status ... */
```

or log the refusal by

```
status = dgLogEvent(jobid, DGLB_SOURCE_RB, DGLB_EVENT_JOBREFUSE, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBREFUSE, error_string);
```

After attempting to send a job to the JSS, the resource broker logs:

```
status = dgLogEvent(jobid, DGLB_SOURCE_RB, DGLB_JOBTRANSFER, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBTRANSFER, submission_service, result, new_class_ad);
```

The `new_class_ad` contains the job description being sent to the job submission service (i. e. with physical file names, etc.), the `result` variable contains the transfer result (success or reason of failure).

Job submission service

The job submission service announces the job reception or refusal by logging appropriate messages (for simplicity only the positive case is shown):

```
status = dgLogEvent(jobid, DGLB_SOURCE_JSS, DGLB_EVENT_JOBACCEPT, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBACCEPT, from_where, condor_jobid);
```

After contacting Globus gatekeeper the job submission service logs this event:

```
status = dgLogEvent(jobid, DGLB_SOURCE_JSS, DGLB_JOBTRANSFER, DGLB_LOG_DEFAULT,
                    DGLB_FORMAT_JOBTRANSFER, computing_element, result, rsl);
```

The `rsl` parameter contains the job description being sent to the Globus gatekeeper, `computing_element` is the gatekeeper contact string, the `result` variable contains the transfer result (success or reason of failure).



Globus job manager

Globus job manager plays two roles here — it logs its own events and at the same time it logs events on behalf of the local resource management system. When accepting job, it calls

```
status = dgLogEvent(jobid, DGLB_SOURCE_JOBMgr, DGLB_EVENT_JOBACCEPT, DGLB_LOG_DEFAULT,  
                    DGLB_FORMAT_JOBACCEPT, from_where, globus_jobid);
```

(the refusal case is not stated here). When submitting the job to the local queue, it logs

```
status = dgLogEvent(jobid, DGLB_SOURCE_JOBMgr, DGLB_JOBTRANSFER, DGLB_LOG_DEFAULT,  
                    DGLB_FORMAT_JOBTRANSFER, queue_name, result, job_description);
```

When the local queue accepts the job, the job manager logs this event:

```
status = dgLogEvent(jobid, DGLB_SOURCE_LRMS, DGLB_JOBACCEPT, DGLB_LOG_DEFAULT,  
                    DGLB_FORMAT_JOBACCEPT, from_where, local_jobid);
```

During the job processing in the local queue the job manager can log various events according to the information obtained by polling the local resource management system, especially

```
status = dgLogEvent(jobid, DGLB_SOURCE_LRMS, DGLB_JOBRun, DGLB_LOG_DEFAULT,  
                    DGLB_FORMAT_JOBRun, node);  
  
/* exit code if obtained from the LRMS, can be logged */  
status = dgLogEvent(jobid, DGLB_SOURCE_LRMS, DGLB_JOBDone, DGLB_LOG_DEFAULT,  
                    DGLB_FORMAT_JOBDone, exit_code);
```

Appendix C UML diagrams

Figure 3: Logging service architecture — package view

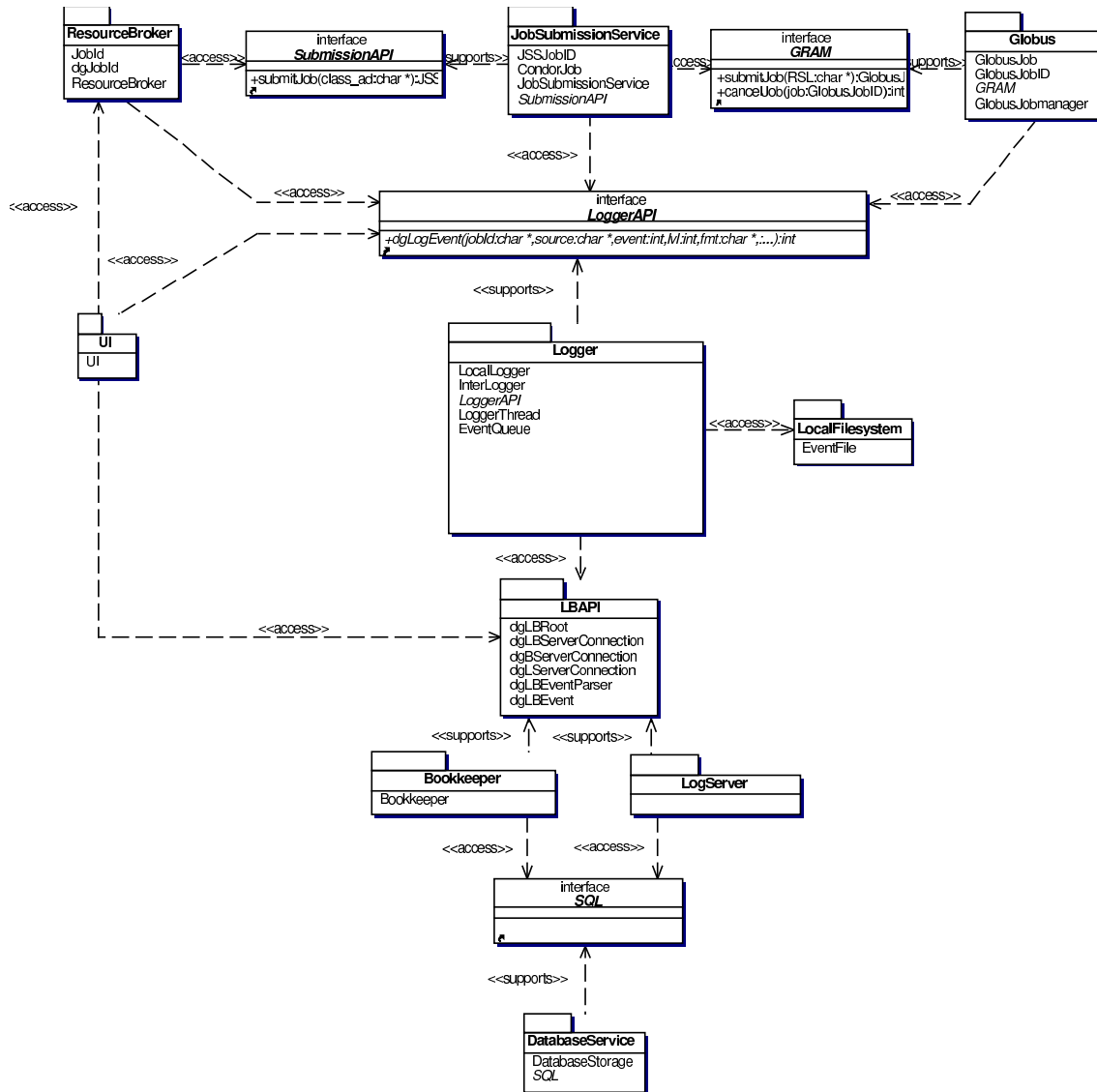
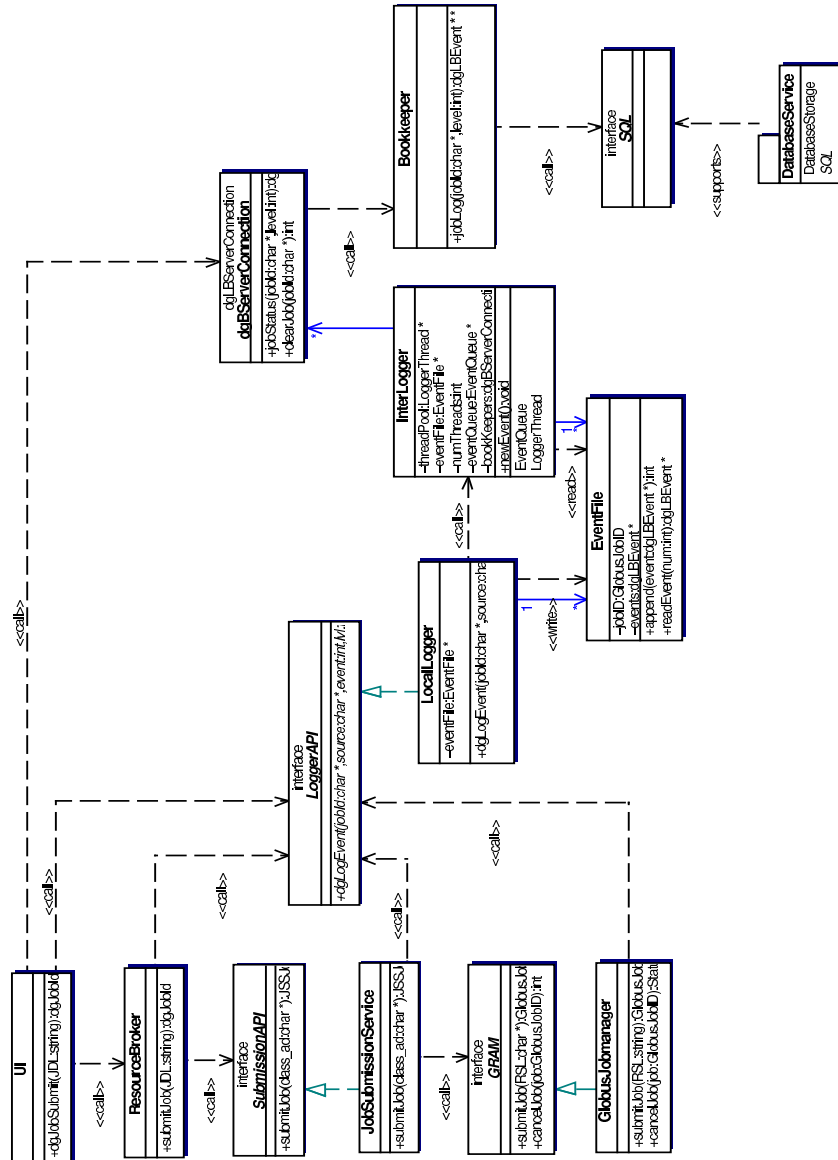


Figure 4: Logging service architecture — class view





Appendix D Use cases: L&B server API

This example code is a skeleton of the *dg-job-status* user interface command.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <lbapi.h>

static void dgerr(dgLBContext, char *);
static void printstat(dgLBContext, dgJobId);

static char *myname;

int main(int argc, char *argv[])
{
    dgLBContext    ctx;
    dgJobId        *jobs, job;
    char           *bserver;
    int            i, result = 0;

    myname = argv[0];

    if (dgLBInitContext(&ctx)) {
        fprintf(stderr, "%s: cannot initialize dgLBContext\n", myname);
        exit(1);
    }

    if (argc >= 2 && strcmp(argv[1], "-all") == 0) {
        bserver = getenv("DG_BSERVER");
        if (!bserver) {
            fprintf(stderr, "%s: no default bookkeeping server\n", myname);
            result=1; goto cleanup;
        }
    }
    /* connect to server */
    if (dgLBOpen(ctx, bserver)) {
        dgerr(ctx, "dgLBOpen");
        result=1; goto cleanup;
    }
    /* retrieve job ID's */
    if (dgLBUserJobs(ctx, NULL, &jobs)) {
        dgerr(ctx, "dgLBUserJobs");
        result=1; goto cleanup;
    }
    /* retrieve and print status of each job */
    for (i=0; jobs[i]; i++) {
        printstat(ctx, jobs[i]);
        dgJobIdFree(jobs[i]);
    }
    free(jobs);
    dgLBClose(ctx);
    } else for (i=1; i<argc; i++) {
    /* parse job ID */
    if (dgJobIdParse(argv[i], &job)) {
        fprintf(stderr, "%s: %s: cannot parse jobId\n",
            myname, argv[i]);
        continue;
    }
}
```



```
/* determine bookkeeping server address */
    bserver = dgJobIdBserver(job);
    if (!bserver) {
        fprintf(stderr,"%s: %s: cannot extract bookkeeping server address\n",
            myname,argv[i]);
        dgJobIdFree(job);
        continue;
    }
/* connect to the server */
    if (dgLBOpen(ctx,bserver)) {
        dgerr(ctx,"dgLBOpen");
        dgJobIdFree(job);
        continue;
    }
/* retrieve and print job status */
    printstat(ctx,job);
    dgJobIdFree(job);

/* We should be more clever here and cache server connections.
 * However, for the sake of simplicity ... */
    dgLBClose(ctx);
}

cleanup:
    dgLBFreeContext(ctx);
    return result;
}

static void
dgerr(dgLBContext ctx,char *where)
{
    char    *etxt,*edsc;

    dgLBError(ctx,&etxt,&edsc);
    fprintf(stderr,"%s: %s: %s",myname,where,etxt);
    if (edsc) fprintf(stderr," (%s)",edsc);
    putc('\n',stderr);
    free(etxt); free(edsc);
}

static void printstat(dgLBContext ctx,dgJobId job)
{
    dgLBJobStat    stat;
    char            *jobid = dgJobIdUnparse(job);

    switch (dgLBJobStatus(ctx,job,DGLB_STAT_STD,&stat)) {
        case 0:
            break;
        case ENOENT:
            printf("%s: not found\n",jobid);
            free(jobid);
            return;
        default:
            dgerr(ctx,"dgLBJobStatus");
            return;
    }
    printf("%s: \n",jobid);
    switch(stat.stat) {
        case DGLB_JOB_SUBMITTED:
            printf("Status:\tSUBMITTED\n");
            /* ... */
            break;
    }
}
```



```
        case DGLB_JOB_WAITING:
            printf("Status:\tWAITING\n");
            /* ... */
            break;
        /* ... */
    }
    dgLBFreeStatus(&stat);
    free(jobid);
}
```



Appendix E C++ L&B server API

Not available yet.