DRMAA_v0_1_Haas.txt

DRMAA interface specification (draft)

Fritz Ferstl (ferstl@sun.com)
Andreas Haas (ahaas@sun.com)

Version 0.1 12.19.2001 initial revision

1. Interface Form

a) C-API library interface - no protocol

The most compelling argument for a library is, that the chances for a broad adoption of this interface will rise, if it's easy to exploit: ISV's will not start implementing protocols, but they might bind a library to their applications.

b) Use shared library binding

Our suggestion is that each DRM system producer should provide a shared library complying with the interface. This is necesssary because it is not realistic to expect from an ISV to deliver more than one binary-version of their software. Providing the interface in a shared library will even allow ISV's to optionally bind the library to their application using dlopen(3).

c) Support only one DRM system

It was Bill's idea that more than one DRM system should be supported and actually I agree with Bill that this would be nice. The main problem to be solved before multiple DRM systems can be interfaced (at a time) is how to dynamically link multiple libraries implementing the *same* interface to a single application. Though it might be possible to find a technical answer, we regard it as an overly ambitious objective given the focus and tight schedule we wanted to impose on ourselves.

2. Interface specification

All the functions below indicate error conditions by means of an appropriate return value. For error diagnosis purposes there will be a function that returns adequate diagnosis information.

The draft forsees two objects

a) Job template

The job template describes all attributes of a job to be submitted to the DRM system. In this specification the job template is a C-language structure. Job templates can be allocated/destructed only using certain functions

```
job_template *newJobTemplate(void);
void deleteJobTemplate(job_template *jt);
```

The interface does not forsee direct access on this structure, instead set<attrib-name>() and get<attrib-name>() functions are provided for this purpose. We need to agree on a set of job attributes that are supported across all DRM systems and DRM-alike systems.

Assuming we will agree on an attribute specifying the email-adress to be used for job related mail sent by the DRM system we would

add two functions

```
    int setEmail(job_template *jt, char *email);
    char *getEmail(job_template *jt);
```

or if we can agree on an attribute "real time limit" we would add two functions

```
    int setRealTimeLimit(job_template *jt, long rtl);
    long getRealTimeLimit(job_template *jt);
```

Surely the most important attribute of a job template is the path of the binary/script to be executed

```
    int setJobPath(job_template *jt, char *path);
    char *getJobPath(job_template *jt);
```

the semantic of job path should allow for specifying absolute file pathes and unqualified pathes, as both variations appear to be needed.

Another attribute I expect that we can agree on is job arguments.

Naturally there will be a limit regarding on what we can agree on. For passing non-agreed DRM system specific attributes there will be a function for adding those attributes

```
    int setAttrib(char *name, char *value);
```

to a job template. Non-agreed job template attributes are evaluated by the DRM specific library under consideration of site-specific settings provided by a concrete installation.

b) Job instance and identifier

A job instance is a proxy for the job submitted to the DRM system. The job instance is represented by a unique identifier. Due to the flexibility of character strings we propose "char *" as data type for this identifier.

Job instances can then be created using

```
    char *runJob(job_template *jt);
```

which is a non-blocking function (i.e. it does not wait for job's end, but only for jobs submission to the DRM system).

A blocking and non-blocking wait function allows waiting for one or more job instances to finish

```
   char *waitJob(char *jobid, int *status, int options, char **rusage);
```

If 'jobid' is non-NULL waitJob() waits for this job. If 'jobid' is NULL waitJob() waits for the termination of any job instance. 'status' is a pointer to an integer variable in which the reason (signal) of the job's termination will be stored (similar to wait3(2) status argument). The 'options' argument can be used for changing the behavior of waitJob() how to operate (similar to wait3(2) option argument), it allows e.g. blocking and non-blocking modes. waitJob() returns the job handle of the terminated job. Like with wait3(2) system call notification about further job state transitions (pending -> running, running -> suspended) could be interfaced in an analogous fashion.

A set of functions will be defined to interface common operations like

```
int suspendJob(char *jobid);
int resumeJob(char *jobid);
int terminateJob(char *jobid);
```

these functions initiate suspension/release suspend state/initiate termination of running jobs. 'Jobid' is the job handle as returned by runJob(). We need to agree what should be the effect when suspendJob()/ resumeJob() are applied on pending jobs. It might be wise just to map it internally into a DRM hold state.