Generalization of ARCLIB

Markus Nordén

December 20, 2007

1 Introduction

The aim of this document is to summarize the curent state of the "generalization of ARCLIB" quasi-task within KnowARC.

Section 2 is an attempt to collect all requirements on the new ARCLIB. The sources are the KnowARC design document (D1.1-1), the documentation of the old ARCLIB, and numerous meetings and discussions.

Section 3 is a draft summary of a very first sketch of a generalized ARCLIB. It was not intended to be used as a design, but rather a starting point for design discussions. The aim when drawing the sketch was to design a new ARCLIB that is modular in a general way, so that interoperability with other middlewares can be achieved by introduction of new classes. All classes in that sketch are not intended for immediate implemention. The first approach should be to implement classes targeted to a specific (server side) middleware (the new ARC1) and then add new classes for interoperation with more middlewares.

The main focus in this sketch has been on execution capability, but hopefully something similar will work also for e.g. storage capability. Furthermore, all class names are very preliminary and should be adapted to reflect conventional terminology.

In section 4, finally, a more detailed design for job descriptions is presented.

2 Requirements

This is a collection of requirements on the new client library - ARCLIB - that is being developed within the KnowARC project. The purpose of this collection is to serve as an aid in developing the design of the new library, and subsequently when that design is implemented.

2.1 Documentation

The new ARCLIB must be well documented. There must be documentation both on an "overview level" (including tutorials) and on a detailed technical

level where the API is presented.

2.2 Interoperability

A general requirement for the new ARCLIB is that it must be general and extensible in order to allow for interoperation with server side middleware from different vendors. Primarily, there must be support for resources available through the ARC1 services developed by KnowARC, such as e.g. A-REX. Initially, there must also be support for ARC0 resources and to some extent[1] gLite resources. The design must also allow for future extensions for resources available through other middlewares, e.g. by addition of new classes.

2.3 Backward compatibility

It would be desirable if the new ARCLIB API could be made backward compatible with previous releases of ARCLIB. To make it fully backward compatible would, however, probably put too severe restrictions on the work for interoperability mentioned above. The requirement regarding backward compatibility is, therefore, that the API of previous versions of ARCLIB shall be preserved whenever possible. If the API needs to be changed, the new API shall be as similar as possible to the old version.

2.4 Language bindings

There must be language bindings for C++, Python, and Java. C++ has the highest priority.

2.5 Installation and maintenance

The new ARCLIB shall be provided as a single software package that contains a fully sufficient set of components needed to utilize all ARC services. It must be possible for an ordinary user to install a standalone personal client as well as for a system administrator to install a system-wide multi-user client. There must be support for various operating systems.

2.6 Handling of Configuration Options

It must be possible to specify a default system-wide configuration as well as personal preferences. Security related configuration must be independent in order not to interfere with possible other grid software on the system. There shall be support for a simplified configuration[2] for users that do not want to or need to configure advanced options.

[1] The extent will be defined by Christian's work.

[2] This refers to the work done by Johan.

2.7 Handling of Grid Credentials

There must be support for certificate requests, proxy creation (including VOMS extensions), credential information query, and credential renewal.

The following kinds of credentials must be supported:

- X.509 credentials
- PKCS12 credentials
- X.509 proxy credentials
- VOMS credentials

The design must be flexible and allow for future extensions to support additional kinds of credentials.

Credentials shall be handled as transparently as possible from a user's point of view. Details about the internal representation and storage of credentials must not be exposed to the user unless necessary.

2.8 Resource Discovery

The new ARCLIB must be able to discover resources behind different server-side middlewares including new ARC, old ARC and gLite. The resource discovery components must be flexible and allow for future extensions to other middlewares.

2.9 Matchmaking

A.k.a. brokering. Possibility to use different optimization algorithms. Extensible. Possible (in the future) to use an external resource broker, which has a "global view" and provides better optimization.

2.10 Service Queries

Job status query, file transfer status etc.

2.11 Job Description

Traditional ARC xRSL and OGF JSDL files, but also creation from within an application by means of function calls. Ivan should know more about this!

2.12 Job Submission and Management

Status queries, deletion, hold/release, retrieval, re-scheduling, retry, attribute change (ownership, access control, accountability, priority). Transparent data staging.

2.13 Data Manipulation

File transfer, information query, ownership and access control modifications, renaming, replication, deletion. Handling logical names and resolution to physical locations.

2.14 High-level operations

There must be support for high-level operations such as bulk task management and bulk data operations.

2.15 Old ARCLIB

List the functionality from old ARCLIB that must be present also in new ARCLIB!

2.16 Miscellaneous Utilities

Not grid-specific stuff, e.g. URLs. See what else needs to be brought from old ARCLIB!

3 Early Design Proposal

In this section, an early design proposal is presented. It is based on discussions and sketches on the whiteboard in Markus' and Mattias' office in Uppsala. Hopefully, some of the ideas presented here will turn out to be useful, while others will have to be adapted as the design becomes more detailed. The internal representation of job descriptions presented in section 4 is an example of the latter.

3.1 Overview

In Figure 1 all classes and relations are shown simultaneously. All details are not shown. This probably mostly looks like a big mess, but may be useful getting the over all picture when considering specific classes in more detail.

In Figure 2, some classes used for target generation are shown. This Figure is known to be outdated, but still illustrates some thoughts. The idea is that there is one TargetGenerator object, which knows about a couple of information retrievers that it uses as starting points. It then gains knowledge about other information retrievers from the starting point iniformation retrievers, and similarly in a recursive manner. The result is eventually a set of targets that can be considered by a broker.

Figure 3 shows classes that are used for description of jobs and representation of submitted jobs. A SpecificationFile (a better name is urgently needed) is either a JSDL document or an XRSL document. Since (at least) a

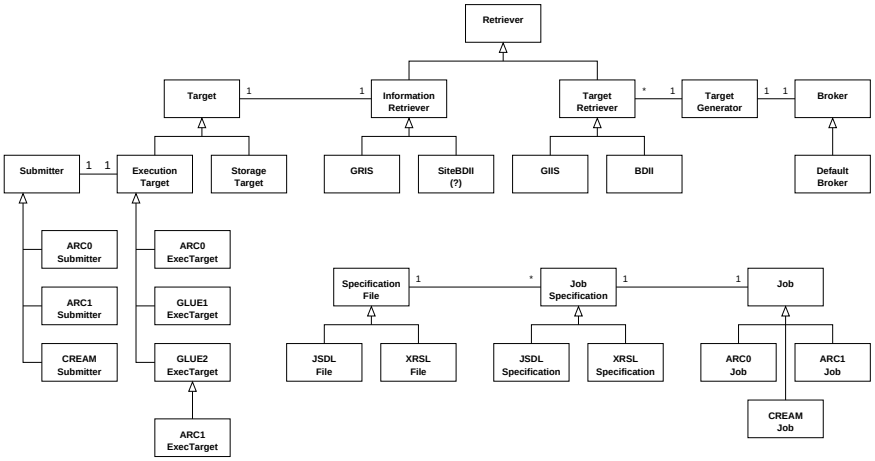

Figure 1: All classes – a big mess...

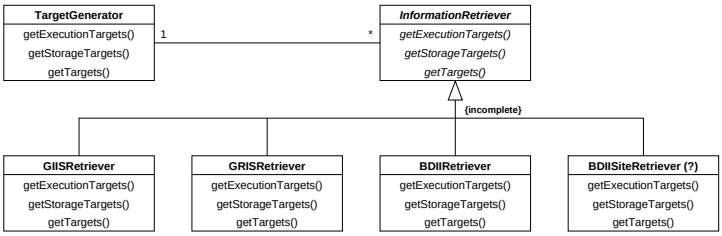

Figure 2: Classes for generation of possible targets.

JSDL document may contain several individual job specifications, individual jobs are represented as separate objects. Furthermore, submitted jobs are represented by classes in the Job hierarchy.

3.2 Relations

Figure 4 is similar to Figure 2 and shows relations between classes used for target generation. The TargetGenerator uses Retrievers to find candidate targets as described above. One option is to let the TargetGenerator use a Broker to rank the possible targets immediately, but this will probably cause trouble if many jobs are to be submitted. Then the generation of candidate targets should be separated from the broker for efficiency reasons.

Figure 5 shows relations between classes used for for description of jobs and representation of submitted jobs. It is a less detailed version of Figure 3.

Figure 6 shows the relation between Submitters and ExecutionTargets. Every class that inherits from ExecutionTarget should have a getSubmitter() method that retiurns an object of some class that inherits from Submitter. This object shall have a submit() method that submits a job in a way that matches the kind of CE behind that ExecutionTarget.

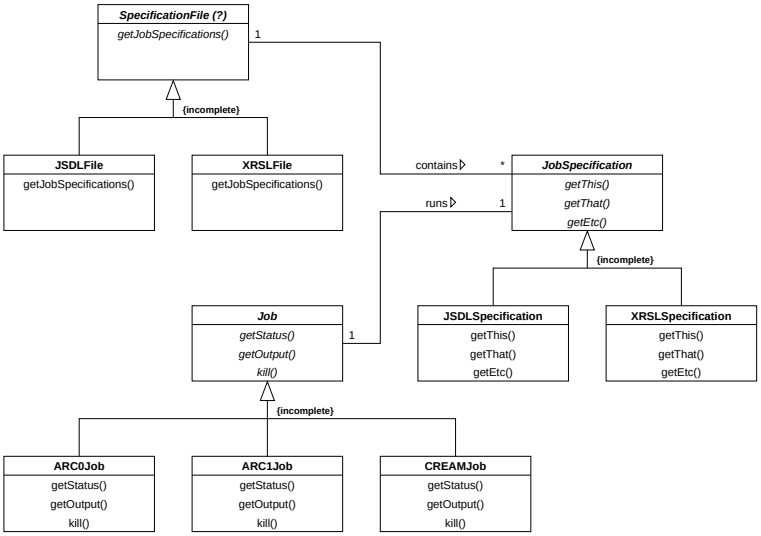

Figure 3: Classes for description of jobs and representation of submitted jobs.

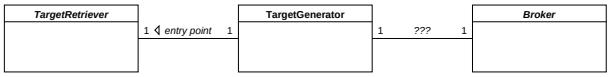

Figure 4: Relations between classes involved in target generation.

For example, one subclass of ExecutionTarget may be ARC0ExecutionTarget, that represents a CE that runs an ARC 0.x server. The getSubmitter() method of that class will return an ARC0Submitter that can submit jobs to ARC 0.x servers. The reason for using a separate class for submission is that ExecutionTargets of different kind may use the same interface (e.g. CREAM).

Figure 7 shows that each Target is associated to an InformationRetriever, that it may query for additional information.

3.3 Class hierarchies

Figure 8 shows a very small class hierarchy for brokering. This is probably the point where users are most likely to want to provide their own taylor made strategies. The hierarchy may therefore be extended with new broker classes that hopefully will be dynamically loadable as "plugins".

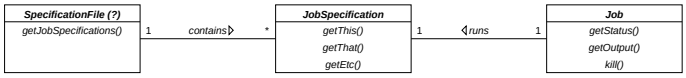

Figure 5: Relations between classes used for for description of jobs and representation of submitted jobs.

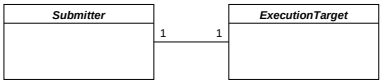

Figure 6: Relation between Submitters and ExecutionTargets.

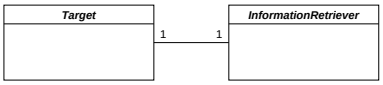

Figure 7: Relation between Targets and their InformationRetrievers

Figure 9 shows the Job class hierarchy for representation of submitted jobs. There is one subclass to Job for every kind of interface to CEs.

Figure 10 shows a class hierarchy for representation of specifications of individual jobs. There are several subclasses intended for representation of jobs specified in different job description languages.

Figure 11 shows classes for information retrieval.

The TargetGenerator starts with a (small) set of TargetRetrievers (i.e. GIISes and BDIIs). They have a getRetrievers() method that returns new Retrievers. If one of those new Retrievers is an InformationRetriever, its getTarget() method is called and the returned Target is added to the list of candidates. If, on the other hand, it is a TargetRetriever, its getRetrievers() method is called recursively in order to find even more candidates.

In order not to send unnecessary queries, the TargetGenerator should contain a datastructure for storing information about allready queried Retrievers. Furthermore, retrieval should preferably be done in parallel by several threads in order to increase efficiency.

Figure 12 shows a class hierarcy for representation and interpretation of specificatino files of different kind, e.g. JSDL files and XRSL files.

Figure 13 shows a class hierarcy for job submission. There is one subclass for every supported kind of submission interface.

Figure 14 shows a hierarchy of classes that are used for representation of targets. There are different subclasses for execution targets and storage targets. The class for execution targets also has a set of subclasses used for

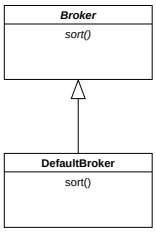

Figure 8: There is a default broker, but other broker classes may be added later allowing for new broker strategies.

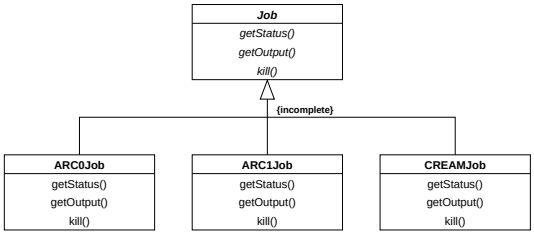

Figure 9: Classes for representation of submitted jobs.

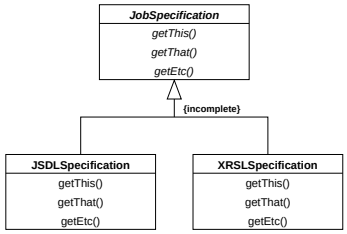

Figure 10: Classes for representation of specifications of single jobs.

different kinds of CEs. In a similar way, the class for storage targets should have a set of subclasses for different SEs, Storage managers, data indexes etc.

3.4 Putting everything together

In Figure 15 some pseudocode is shown that is intended to illustrate the use of the classes presented above. First, job specifications are loaded. Then a broker is created and a set of candidate targets is generated. Finally, the set of candidate targets is sorted by the broker for efery job specification and the job is submitted until it is accepted.

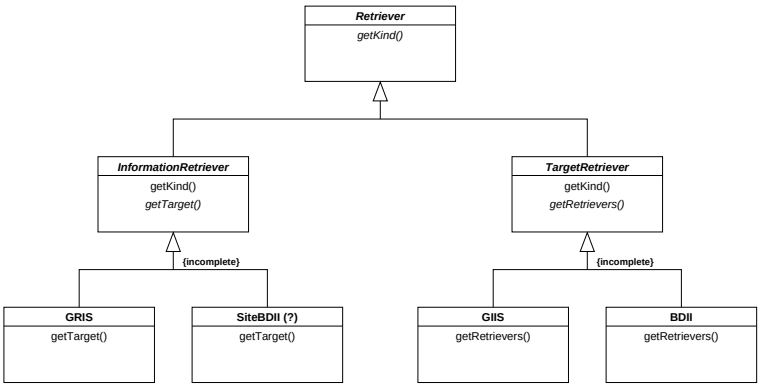

Figure 11: Classes for information retrieval.

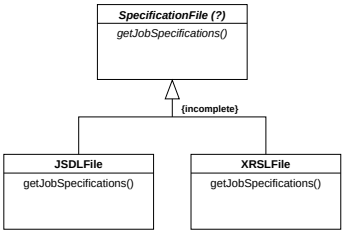

Figure 12: Classes for representation of specification files, which may contain several jobs.

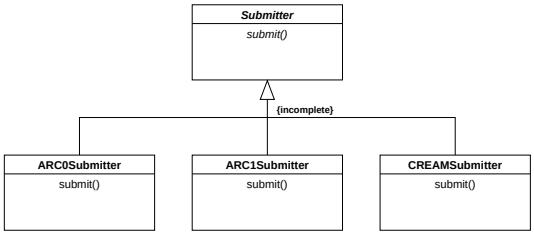

Figure 13: Classes for submitting jobs.

4 Detailed Job Description Design

An important requirement on the new ARCLIB is that it must have support for Job Descriptions in various formats such as traditional ARC xRSL and OGF JSDL files. It must also be possible to create a Job Description in a program by means of method calls without involving any file.

4.1 Representation of Job Descriptions

In order to allow for a uniform handling of job descriptions there will be one class that is used for all Job Descriptions (as opposed to different classes for xRSL, JSDL etc.). This class may, however be an aggregate of sub-descriptions of the resources needed for the job and the task (what to do). The name of this class will be JobDescription, and the parts may be called JobResourceRequirement and JobTaskDefinition.

4.2 Attributes

The JobDescription class (and its sub-description classes) will have an appropriate set of attributes to describe a job. Exactly which set of attributes to use needs further investigation. Most notably, xRSL and JSDL descriptions must be possible to represent. Furthermore, it must be decided which attributes are mandatory, optional (for a user), and ignorable (for a target). It must also be decided which types to use to represent the different attributes, which values are allowed, and whether any attributes may be as-

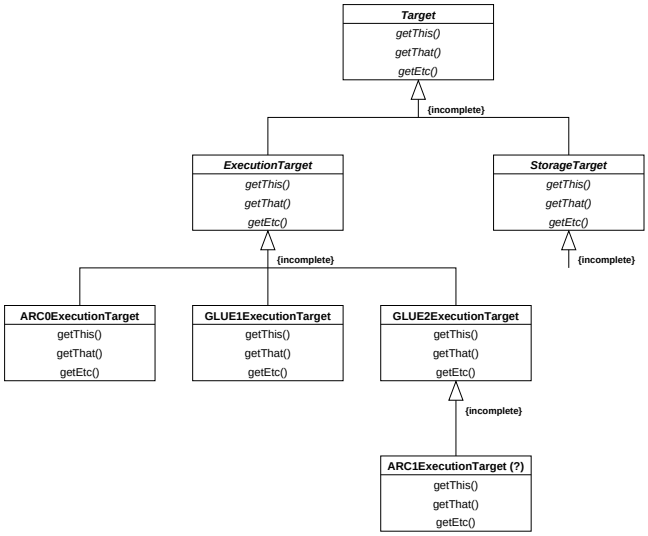

Figure 14: Classes for representation of different targets.

signed conflicting values. The set of attributes used in xRSL has served as a basis for discussions and are presented in the following sections. A more detailed comparison between the attributes of different description formats is presented in appendix A.

4.2.1 General Attributes

The following attributes will be present in the JobDescription class itself.

- Job Name: A user specified name of the job.
- Access Control List: Specifies which users, in addition to the owner, who are allowed to access and control the job. (This attribute may fit better in some other class?)
- Credential Server: Specifies a URL where delegated proxy of a job can be renewed or extended. (This attribute may fit better in some other class?)

4.2.2 Resource Requirement Attributes

The following attributes will be present in the JobResourceRequirement class.

- CPU Time: The maximal CPU time needed by the job.
- Wall Time: The maximal wall clock time needed by the job.

```
void main(int argc, char** argv){

   SpecificationFile specfile(argv[1]);
   List<JobSpecification> jobspecs = specfile.getJobs();

   Broker broker(argv[2]);

   TargetGenerator generator;
   List<ExecutionTarget> candidates = tg.getExecutionTargets();

   for (JobSpecification spec: jobspecs){
      PriorityQueue<ExecutionTarget> preferred =
         broker.sort(candidates, spec);
      Job job = NULL;
      while(job==NULL)
         job = preferred.dequeue().getSubmitter.submit(spec);
   }

}
```

Figure 15: Some pseudocode intended to illustrate the use of the classes presented above.

- Grid Time: The maximal normalized CPU time (to a 2.8 GHz Pentium 4 CPU) needed by the job.
- Memory: The maximal amount of memory needed by the job.
- Disk: The maximal amount of disk space needed by the job.
- Runtime Environment: Specifies a runtime environment needed by the job.
- Middleware: Specifies a list of allowed middlewares, to be used as a requirement on computing nodes during matchmaking/brokering.
- Operating System: Specifies an operating system requirement for computing nodes.
- Cluster: Specifies which cluster (not) to submit the job to.
- Queue: Specifies which remote batch queue to use.
- Architecture: Request a specific (instruction set) architecture.
- Node Access: Request cluster nodes with inbound and/or outbound IP connectivity.

- Count: Specifies the number of processes/threads to be used by a parallel job. (This can be either a resource requirement attribute or a task definition attribute.)

4.2.3 Task Definition Attributes

The following attributes will be present in the JobTaskDefinition class.

- Executable: The name of the executable to run.
- Arguments: A list of arguments for the executable.
- Input Files: A list of files that need to be transferred to the computing node before execution.
- Executables: A list of all files that need to have execution rights set.
- Output Files: A list of files that will be retrieved by the user or shall be transferred to some storage after execution of the job.
- Standard Input: Specifies a file to be sent as standard input to the executable.
- Standard Output: Specifies a file to which the standard output from the executable shall be sent.
- Standard Error: Specifies a file to which the standard error from the executable shall be sent.
- Join: Specifies that the standard error shall be merged into the standard output from the executable.
- Start Time: Specifies a time, before which the job must not start.
- Life Time: Maximal time to keep job files for retrieval.
- Notify: Request e-mail notifications on job status changes.
- Environment: Specifies execution shell environment variables.
- Count: Specifies the number of processes/threads to be used by a parallel job. (This can be either a resource requirement attribute or a task definition attribute.)

4.2.4 Other Attributes

The following attributes are also available in xRSL. They need further investigation/discussion before deciding whether they shall be included and in which class.

- Cache: Specifies whether files should be placed in the cache, in which case the will be read-only.
- Benchmarks: Specifies that the job shall be used as a benchmark.
- GM Log: Specifies a directory where grid-specific diagnostics of the job will be stored.
- FTP Threads: Defines the number of parallel threads to be used for file transfer.
- Replica Collection: Location of a logical collection in the Replica Catalog.
- Rerun: Number of reruns in case of system failures.
- Dry Run: Specifies that the job shall not be submitted. Used for validation purposes.
- RSL Substitution: String substitution for internal RSL use.
- Job Report: Specifies a logging service to which reports about the job shall be sent.

4.3 Methods

There will be methods to set and get all attributes mentioned in the previous section, i.e. setExecutable(), getExecutable(), setArguments(), getArguments(), etc. Possibly, there should also be an isValid() method, which validates that all mandatory attributes are set, that all attributes that are set have sensible values and that there are no contradictory attributes.

4.4 Creation of Job Descriptions

A job description can be created either by reading a description file or setting the attributes directly by means of the methods mentioned in the previous section. To allow for several file formats, there will be a set of reader classes, each of which has a method for reading the specific kind of file. When reading a file, the reader class will create a JobDescription object and set its attributes by means of the methods mentioned in the previous section.

4.5 Candidate set Generation

When generating the candidate set for submission of a job, the resource requirement is needed. The resource requirement part of the job description is sent to the candidate set generator (not yet designed), which returns a candidate set in some way (e.g. a list of feasible targets or an iterator that returns one feasible target at a time).

4.6 Submitting a Job

When a suitable target has been selected, its submit method is called and the job description passed as a parameter. The selected target will be an instance of a subclass of the Target base class. The details of the class hierarchy for targets remains to be designed, but there will be one subclass for each kind of target (new ARC or other BES/JSDL compliant CE, ARC Classic CE, or some gLite CE). The job description will have to be transformed into some format that is accepted by the specific target. This is done by the target class or possibly some writer class in a manner similar to how reading of job description files is handled. When the job has been successfully submitted a Job object is returned that will be used for subsequent operations on the job.

4.7 Examples

The intended usage of the job description class is illustrated by the pseudocode in Figure 16.

```
// Some declarations
JobDescription jd;
JSDLReader jr;
CandidateSetGenerator csg(...);
CandidateSetIterator csi;
Broker b;
Target t;
Job j;

// Define a job specification by means of method calls...
jd.getTaskDefinition().setExecutable("echo");
jd.getTaskDefinition().setArguments("hello world");
...
Jd.getResourceRequirement().setOperatingSystem("FC3");
...

// ...or read it from a file.
jd = jr.read("my_job.jsdl");

// Find a suitable target and submit the job.
csi = csg.find(jd.getResourceRequirement());
t = b.select(csi);
j = t.submit(jd);
```

Figure 16: Some pseudocode intended to illustrate the use of the job description class.

A Job Description Attributes

The following table of attributes available in different job description formats has been produced by Ivan Marton.

JSDL	JDL	RSL	xRSL
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl:ApplicationName-/text()			
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:Executable/text()	Executable	executable	executable
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:Argument/text()	Arguments	arguments	arguments
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:JobIdentification-/jsdl:JobName/text()			jobName
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:JobIdentification-/jsdl:Description/text()			
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl-arc:gmlog/text()			gmlog
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:DataStaging-/jsdl:Source/jsdl:URI-/text()	InputData		inputFiles
			executables
			cache

/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:DataStaging- /jsdl:Source/jsdl:URI- /text()	OutputData		outputFiles
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Application- /jsdl-posix:POSIX- Application/jsdl- posix:CPUTimeLimit- /text()			cpuTime
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Resource- /jsdl:CPUTime/text()			cpuTime
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Software- Requirements- /jsdl:Limits- /jsdl:WallTimeLimit- /text()			wallTime
			gridTime
			benchmarks
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Application- /jsdl-posix:POSIX- Application/jsdl- posix:MemoryLimit- /text()			memory
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Resource- /jsdl:PhysicalMemory- /text()			memory
/jsdl:JobDefinition- /jsdl:JobDescription- /jsdl:Resource- /jsdl:DiskSpace/text()			disk

			runTime-Environment
			middleware
			opsys
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:StdIn/text()	StdInput	stdin	stdin
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:StdOut/text()	StdOutput	stdout	stdout
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:StdErr/text()	StdError	stderr	stderr
			join
			ftpThreads
			acl
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Resource-/jsdl:CandidateHosts-/jsdl:HostName/text()			cluster
		queue	queue
			startTime
			lifeTime
			notify
			replicaCollection
			rerun
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Resource-/jsdl:CPUArchitecture-/text()			architecture
			nodeAccess

		dryRun	dryRun
			rsl_substitution
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Application-/jsdl-posix:POSIX-Application/jsdl-posix:Environment-/text()	Environment	environment	environment
/jsdl:JobDefinition-/jsdl:JobDescription-/jsdl:Resource-/jsdl:CPUCount/text()		count	count
			jobreport
			credentialserver
			sstdin
			action
			savestate
			lrmstype
			hostName
			jobid
			clientxrsl
			clientsoftware
GRAM RSL Attributes			
		directory	
		jobType	
		maxTime	
		maxWallTime	
		maxCpuTime	
		gramMyjob	
		project	
		maxMemory	
		minMemory	
		hostCount	
		libraryPath	
		gassCache	
		fileStageOut	
		fileStageIn	
		fileStageInShared	
		fileCleanUp	
		remoteIoUrl	
		scratchDir	

DUROC RSL Attributes			
		resourceManager-Contact	
		label	
		subjobComms-Type	
		subjobStartType	
Unique JDL attributes			
	Virtual-Organisation		
	Requirements		
	Rank		
	DataAccess-Protocol		
	Prologue		
	Epilogue		
	AllowZippedISB		
	ZippedISB		
	ExpiryTime		
	PersualFileEnable		
	PersualTime-Interval		
	PersualFilesDest-URI		
	StorageIndex		
	DataCatalog		
	DataRequirements		
	DataCatalogType		
	DataAccess-Protocol		
	OutpuSE		
	RetryCount		
	ShallowRetry-Count		
	LBAddress		
	MyProxyServer		
	HLRLocation		
	JobProvenance		
	NodeNumber		
	JobSteps		
	CurrentStep		
	JobState		

	ListenerPort		
	ListenerHost		
	ListenerPipeName		