



High Level API for Grid Applications (HiLA) 1.0

UNICORE Team
unicore-support@lists.sourceforge.net

November 3, 2009

Contents

1	Introduction	3
1.1	Current Usage	3
2	Usage	4
2.1	Logging	4
3	Concepts	4
3.1	Location and Locatable	4
3.2	Overview	5
3.3	Grid	5
3.4	Site	6
3.5	Storage	6
3.6	Task	6
3.7	File	6
4	Implementations	7
4.1	UNICORE 5	7
4.1.1	Configuration	7
4.2	UNICORE 6	7
4.2.1	Configuration	7
4.3	OGSA-BES	9
4.3.1	Overview	9
4.3.2	Configuration	10
5	Examples	10
5.1	Factory as main entry point	11
5.2	List children	11
5.3	Job Submission	11
5.3.1	Job Submission Description Language – JSDL	12
5.3.2	Abstract Applications	12
5.3.3	File Staging	12
5.3.4	The actual submission	13

References	14
Glossary	14

1 Introduction

Highly dynamic features and security concerns often make it difficult for programmers to easily develop clients accessing Grids. There is a lot of programming overhead involved for each call to a Grid resource, although many of these tasks are structured similarly. A universal, high-level API can alleviate the need to perform the tedious and repetitive tasks of accessing Grid resources and let the developer focus on the purely functional aspects of programming. The extra amount of work necessary to access a site is hidden behind a concise interface.

Another advantage of a cleanly defined API is the opportunity to implement for multiple backend Grid environments. The HiLA API currently supports two implementations, one for UNICORE 5 and more recently another one for the Web services based UNICORE 6. An implementation for OGSA-BES is in a development state. Additional backend implementations are conceivable to support other Grid middlewares in the same uniform way.

Rather than designing a task focused API, HiLA takes a resource oriented approach to the definition of the Java interfaces. The linkages between the resources on the Grid are approached in a consistent manner. A resource is Locatable and possesses a Location. As such, there is a generic mechanism in place for navigating the linkages. For example, a Site has a number Storages, a Storage contains a number of Files and a Site references its running Tasks. So different underlying Grid infrastructures become browsable in a uniform way. Analysing and using Grid resources can then be as easy as browsing a UNIX file system.

The API uses a Factory mechanism to dynamically load particular implementations. Multiple implementations of the API can co-exist within the lifetime of a program execution, and it encourages late binding to a specific implementation, selectable at runtime. Location of the HiLA resources are based on URIs, where the scheme is used to select a particular implementation.

This document describes the HiLA API from a user's point of view. Of course, users of APIs are developers. There is a separate developer guide available for those, who want to develop additional implementations of the abstract interfaces.

Originally, HiLA was named Roctopus, which is why you will find references to this name as well [2, 3].

1.1 Current Usage

HiLA is a convenient toolkit for building web interfaces or command line environments for Grids, and is used by the DESHL suite of tools developed in the JRA7 activity of the DEISA project. In the A-WARE project, we are using HiLA as the basis for a Grid agent component to be plugged into a service bus, which is then used as for building higher-level orchestration services for Grids.

HiLA is an API for accessing Grid resources of different middleware in a consistent manner. It has been implemented for UNICORE versions 5 and 6, and an OGSA-BES implementation is also available. HiLA is in use in various places in different Grid access libraries. The DEISA Services for Heterogeneous management Layer (DESHL) has been implemented using HiLA right from the start. This severely eased the shift to UNICORE 6, as HiLA provides the same interface for both middlewares. In the German AeroGrid project [1], a JavaGAT adaptor has been implemented on top of HiLA, thus enabling the use of UNICORE 5 and UNICORE 6 through GAT and this adaptor.

2 Usage

Those of you, who are eager to get a hand at HiLA, may want to try the Groovy scripts available in the `hila-groovy` package. The most useful script may be `hila_ls.groovy`, with which you can browse any `Locatable` object, i. e. anything that has a `Location`.

2.1 Logging

Some of the external libraries used by HiLA write quite a bit of logging output to the console. This is one of the reasons why you would like to configure logging different from the default. The other reason is to get more logging output than is usually configured, e. g. for debugging purposes.

HiLA generally uses the SLF4J logging library, which is capable of aggregating the logging output of a number of other logging libraries and also use different logging libraries to produce the final log output. HiLA usually uses the Java Util Logging facility to produce it's final logs. This documentation is only about this particular implementation and describes how to configure it.

Java Util Logging can be configured via a `.properties` file. You can reference your own `logging.properties` file with the Java system property `java.util.logging.config.file`, e. g. by setting it via `-Djava.util.logging.config.file=logging.properties` on the command line when calling Java. You can also set a `JAVA_OPTS` environment variable to automatically have the option set. This is particularly useful if you want to keep the calls to your commands simple. The `JAVA_OPTS` variable contains the setting of the system property as it would appear on the command line.

```
$ export JAVA_OPTS="-Djava.util.logging.config.file=logging.properties"
```

- `slf4j` (over `jul`)
- `logging.properties`
- `JAVA_OPTS`

3 Concepts

3.1 Location and Locatable

The key concept in HiLA is that everything can be referenced by a URI.

The generic URI structure in HiLA looks like this:

```
<scheme>:/sites/<site-name>/tasks/<task-id>/wd/files/<file-name>  
<scheme>:/sites/<site-name>/storages/<storage-name>/files/<file-name>
```

The `<scheme>` placeholder is the implementation specific part of the URI. In real scenarios, it will be something like `unicore6`, `unicore5` or `ogsa`. `<site-name>` will be the name of the site, as it has been discovered by the configuration of HiLA (see ??). `<task-id>` is the id of the task, which is generated by the middleware. This enables us to located particular tasks only from this reference. The same holds for `<storage-name>`, where all storages exposed by the middleware at the site under consideration will be valid values for `<storage-name>`.

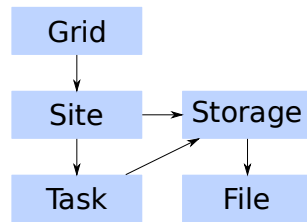


Figure 1: Architectural overview of HiLA

This follows a regular pattern, where collections and their contents alternate in the URI structure. Collections are `sites`, `tasks`, `storages`, and `files`. While all of these do have equivalents in the Java interfaces, they are less important, because they do not offer more functionality than any other Locatable artifact.

The concept of artifacts being locatable is generic and extensible. Which artifacts in the Grid can be referenced by HiLA, will be subject of the following sections.

3.2 Overview

The basic artifacts, which can be accessed by HiLA and referenced via HiLA locations, are:

- Grid
- Site
- Storage
- Task
- File

Figure 1 shows the interrelationships between the various artifacts. A Grid is made up of a number of sites, which in turn contain tasks and storages. Storages contain files, which can be imported to, exported from or transferred among storages. Tasks also have a storage, which is their working directory. One can thus access the working directory of a task at any time, if the underlying implementation supports it. Figure 2 shows a UML class diagram of the main classes within the abstract HiLA API. An implementation of HiLA has to implement the interfaces as appropriate and support the given operations. Operations on the interfaces will be described in the following.

3.3 Grid

Grid locations are structured like `<scheme>:/`. For example, a grid location for the UNICORE 6 implementation of HiLA would look like `unicore6:/`. As shown in figure 1 and more specifically figure 2, a `Grid` consists of an arbitrary number of `sites`. You can query the Sites belonging to a `Grid` by calling the `getAllSites()` method, which returns a List of `sites`. If you know the `Location` of an entity, to which you want to gain access, you can `locate()` it. The `locate()` method returns a `Locatable`, which you may have to cast to the appropriate type.

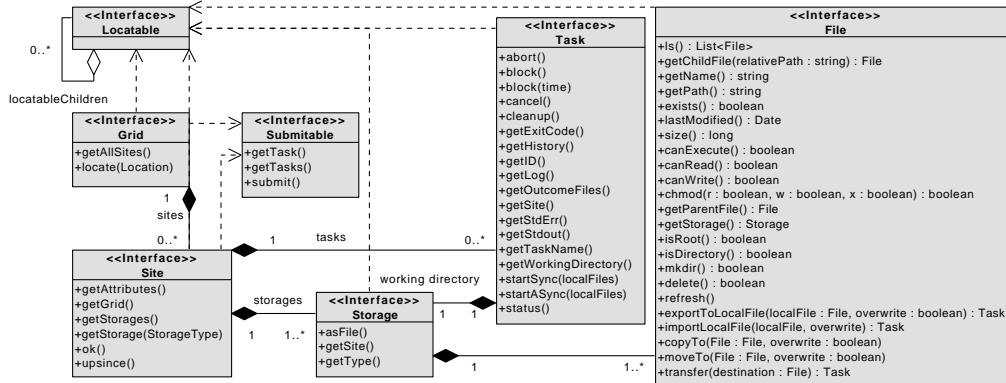


Figure 2: UML class diagram of HiLA API

3.4 Site

Site locations are structured like `<scheme>:/sites/<site-name>`. A Site location for the UNICORE 6 implementation looks like `unicore6:/sites/Alpha-Site`. Sites implement both the `Locatable` and `Submittable` interfaces. While `Locatable` has been explained before, the `Submittable` interface is used for anything that one can submit jobs to. Hence, it implements the methods `submit()`, `getTasks()`, and `getTask()`.

3.5 Storage

Storage locations are structured like `<scheme>:/sites/<site-name>/storages/<storage-name>`. There is a special twist to Storage Locations, as a `Task` also has a `Storage`, where its execution takes place. Thus, a Storage Location can also look like `<scheme>:/sites/<site-name>/tasks/<task-name>/wd` where `wd` means working directory. Storages belonging to Sites can be retrieved by calling `getStorages()` on a Site object. From a `Task`, one can directly jump into the working directory by calling `getWorkingDirectory()` on the `Task`, which actually returns a `File`, but that is what you want anyway.

Examples of Storage Locations for the UNICORE 6 implementation would be `unicore6:/sites/Alpha-Site/storages/home` and `unicore6:/sites/Alpha-Site/tasks/79258434-f295-11dd-b58a-001f160cb28c/wd`.

3.6 Task

Task locations are structured like `<scheme>:/sites/<site-name>/tasks/<task-name>`.

An example Task Location for the UNICORE 6 implementation looks like this `unicore6:/sites/Alpha-Site/tasks/79258434-f295-11dd-b58a-001f160cb28c`.

3.7 File

File locations are structured like `<scheme>:/sites/<site-name>/storages/<storage-name>/files/<file-name>`. Again, as `Tasks` have associated `Storages` as well, and these `Storages` also contain `Files`, an alternative structure of a `File`'s location is `<scheme>:/sites/<site-name>/tasks/<task-name>/wd/files/<file-name>`.

Example File Locations for the UNICORE 6 implementation look like the following: `unicore6:/sites/Alpha-Site/storages/home/files/.bashrc` and `unicore6:/sites/Alpha-Site/tasks/79258434-f295-11dd-b58a-001f160cb28c/wd/files/stdout`.

4 Implementations

The actual implementation to be used for Grid access is chosen at runtime based on the scheme part of the artifact's Location. All implementations available on the classpath register the schemes that they are responsible for. Schemes that are available for current implementations are:

- `unicore5`
- `unicore6`
- `ogsa`

The next sections all follow the same pattern. For each implementation, we will first give an overview followed by a description about how to configure it. As HiLA has originally been developed for the UNICORE 5 middleware, its interface is still somewhat related to the UNICORE 5 capabilities. In consequence, not all features of HiLA may be supported by all implementations and secondly, not all features of a middleware may have their equivalent in HiLA. These discrepancies will be reported on in the respective *limitations* section.

4.1 UNICORE 5

4.1.1 Configuration

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns:hila="http://www.unicore.eu/hila-unicore5">
  <hila:unicore5grid id="grid"
    outcomeDirectory="file:${user.home}/.hila/data" config="#config" />

  <hila:csvconfig id="config"
    resource="file:${user.home}/.hila/unicore5.csv" />
</beans>
```

4.2 UNICORE 6

4.2.1 Configuration

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:hila="http://www.unicore.eu/hila-unicore6">
  <hila:unicore6grid id="grid"
    config="#config"
    fileTransferPrefs="#ftp"
    outcomeDirectory="file:/home/user/.hila/data"
    terminationTimeInDays="15" />

  <hila-common:compositeconfig id="config"
    xmlns:hila-common="http://www.unicore.eu/hila-common">
```

```

    <constructor-arg>
      <list>
    <hila:registryconfig
      registryURL="https://omiiei.zam.kfa-juelich.de:6000/Registry/services/
        Registry?res=default_registry"
      grid="#grid" securityProperties="#demo.security" />
      </list>
    </constructor-arg>
  </hila-common:compositeconfig>

  <bean name="demo.security"
    class="de.fzj.hila.implementation.unicore6.Unicore6SecurityProperties">
    <constructor-arg value="/home/user/.hila/unicore6.security" />
  </bean>

  <hila:filetransferpreferences id="ftp">
    <constructor-arg>
      <list>
        <value>SBYTEIO</value>
        <value>BFT</value>
        <value>RBYTEIO</value>
      </list>
    </constructor-arg>
  </hila:filetransferpreferences>
</beans>

```

The above configuration file is rather complex, so we will try to break it down into manageable pieces. First of all, as in any well-formed XML file, there is an all-surrounding root element called `beans` in this case. This is the root of the Spring configuration and is not related to HiLA itself. Just make sure it always exists in your configuration.

Secondly, we have a number of other elements

- `hila:unicore6grid`
- `hila-common:compositeconfig`
- `bean`
- `hila:filetransferpreferences`

`hila:unicore6grid` configures the Grid for UNICORE 6 implementation. The attribute `id` will be needed further down in the file to reference the Grid. The `config` attribute contains a reference to the Config object of the Grid. Please note that it contains the same name as the `id` attribute of the `hila-common:compositeconfig` element prefixed with a `#`. The same applies to the `fileTransferPrefs` attribute, which contains a reference to the `hila:filetransferpreferences` element further down in the file. The `outcomeDirectory` will be used for calls to `getOutcomeFiles()` on a Task. All files retrieved from the working directory of the Task will be transferred into a Task specific directory inside this directory. The name of the Task directory is created from the Task's ID. `terminationTimeInDays` is used to define the termination time of the WS Resources that UNICORE 6 is based on. The default life time of a UNICORE 6 target system service on the server side is one day. Consequently, if you did not use your target system for more than this time, it will be gone and with it all jobs that have been submitted to it. Extending the life time to something longer is advisable. If you omit this attribute, the default in HiLA is 10 days. The UNICORE Rich Client for instance has a default of 30 days. This lifetime is also applied to jobs whenever you access them.

The `hila-common:compositeconfig` element exists to use a number of `Config` objects all associated with the same `Grid`. The meaning of `Config` is described in section ???. The `compositeconfig` is made up of a `hila:registryconfig`, which is a UNICORE 6 specific config, gathering its list of sites from the UNICORE registry given by the `registryURL` attribute. The `grid` attribute on that element needs to point to the `hila:unicore6grid` element by referring to its id, i.e. “#grid” in this case. The `securityProperties` attribute refers to an object definition of a `Unicore6SecurityProperties` object (see below). As `hila-common:compositeconfig` takes a list of arguments as a parameter, there could be more registries defined, each having its own security configuration. The `Unicore6Grid` object does not rely on a `compositeconfig`, so if there is only one registry to be addressed, one can configure HiLA without the extra `compositeconfig` and reference a `registryconfig` in the `unicore6grid`.

The `hila:filetransferpreferences` element describes the preferences for the multiple file transfer methods that UNICORE 6 supports. The element needs to be referenced from the `unicore6grid` element only if you desire an influence on the file transfer mechanism. If the reference is omitted, then default preferences will be used. The default preferences are BFT, RBYTEIO, and SBYTEIO in order.

The contents of the `unicore6.security` file, which is referenced in the `demo.security` element should look like the following:

```
unicore.wsrflite.ssl.keystore = /home/user/.hila/demouser.p12
unicore.wsrflite.ssl.keypass = *****
unicore.wsrflite.ssl.keytype = pkcs12
unicore.wsrflite.ssl.keyalias = demo user
# unicore.wsrflite.ssl.truststore = /home/user/.hila/demo.jks
# unicore.wsrflite.ssl.truststorepass = *****
# unicore.wsrflite.ssl.truststoretype = jks
```

The four top properties are the basic ones which have to be there. The commented ones (commented using #) are inferred if they don't exist. If the truststore, the collection of certificates that are considered trusted, is not given explicitly, then the keystore is assumed to be a truststore at the same time. A similar approach holds for the password of the truststore. If it is not given, then the keystore password is used. The default keystore and truststore type is JKS (Java KeyStore format), the alternative is PKCS12.

4.3 OGSA-BES

4.3.1 Overview

There is also an implementation of HiLA for OGSA-BES services. An implementation of these standardized services is also available within UNICORE 6. The acronym BES stands for Basic Execution Services, which is just what they are. BES is a specification of the OGF that specifies standard services for the management and execution of jobs on Grid resources. What BES does not specify is the transfer of files to and from the Grid, which is usually necessary if you want to execute scripts, stage in data that is to be processed, or if you are interested in retrieving the output of your job. Different from UNICORE 6, BES assumes that for any task there will always be data staging specifications in the job description. Thus, the Grid service will take care of the staging, but is up to the user to provide references to the input and output locations and also to ensure that the input files are available. If a user wants to upload local files, he needs to put them to some location accessible by the service prior to submitting a job to a BES implementation.

An essential consequence of the above discussion is that part of the HiLA API is not applicable to the OGSA-BES implementation. As BES does not offer a means to reference remote files and act on them, Storages and Files have been left out of the implementation of HiLA for OGSA-BES.

4.3.2 Configuration

The configuration of OGSA-BES is rather simple. It is made up of a Java properties file, which has to be located in `$user.home/.hila/ogsa.properties`. The following is an example of its contents. You will find an explanation afterwards.

```
ogsa.sites = FZJ_JUMP,ZAM461

ogsa.site.FZJ_JUMP.url = https://deisa-unic.fz-juelich.de:9111/FZJ_JUMP/
services/BESFactory?res=default_bes_factory
ogsa.site.FZJ_JUMP.security = d-grid.security

ogsa.site.ZAM461.url = https://zam461.zam.kfa-juelich.de:9102/DEMO-SITE/
services/BESFactory?res=default_bes_factory
ogsa.site.ZAM461.security = demo.security
```

`ogsa.sites` is a comma-separated list of Site names, which will be used for the `<site-name>` part of the URIs. Following that is a definition of each of the sites specified by name. Two pieces of information need to be specified:

- The URL of the BESFactory endpoint
- A properties file with security settings

The name of the property for a Site is structured like `ogsa.site.<site-name>.<property>` where `property` is a place holder for either `url` or `security`.

The security property points to a file containing security properties. This file has the same structure as in the UNICORE 6 implementation. Please refer to section ?? for information about this.

As with all implementations of the HiLA API, a Spring configuration could be placed in the `.hila` directory as well. However, as the OGSA implementation is currently rather simple, we consider the current approach sufficient. The actual Spring configuration, which is always needed by HiLA, as the implementation independent parts of it require it, is available from the class path.

The OGSA-BES implementation does not support the Storage interface. This is due to the fact that the OGSA-BES standard is only about job submission and management, but not about data transfers. Input and output data, which is needed for any job except the most simple ones, needs to be declared in the JSDL file for submission.

If you use the additional import file parameters to the `startSync` and `startASync` methods of a `Task`, these will be ignored for the OGSA-BES implementation. OGSA-BES assumes that an Activity (OGSA-BES terminology for Task) is started directly after submission. Hence, the start methods have no effect, except for blocking on that Task.

5 Examples

The following subsections provide examples of use of the HiLA API. We do not provide full source code, as we consider it to be too long and distracting from the actual purpose.

All examples will be embeddable into Java or Groovy source code. We assume the following imports to exist in all full source files when you want to run the examples.

```
import de.fzj.hila.*;
import de.fzj.hila.exceptions.*;
```

5.1 Factory as main entry point

Whenever you want to do anything with HiLA, you will need to use the HiLAFactory as the main entry point. HiLAFactory takes care of discovering available implementations of the abstract HiLA API. Implementations have to be available on the classpath in order to be discovered.

```
Locatable locatable = HiLAFactory.getInstance().locate(location);
```

5.2 List children

```
import java.util.List;

...

Location location = new Location("unicore6:/sites/");
Locatable locatable = HiLAFactory.getInstance().locate(location);
List<? extends Locatable> locatableChildren =
    locatable.getLocatableChildren();
for (Locatable locatableChild : locatableChildren)
{
    System.out.println(locatableChild.getLocation());
}
```

The output of this code, when targeted at the UNICORE test grid, to which everyone can get access, looks like this:

```
unicore6:/sites/Bravo-Site
unicore6:/sites/Alpha-Site
```

5.3 Job Submission

From the point of view of the abstract HiLA interfaces, jobs could be submitted to any Object implementing the Submittable interface. This is the case for Grid and Site types. However, none of the currently existing implementations supports submitting jobs to Grids, as this would require some intelligent brokering. Thus, Jobs can only be submitted to Sites, which actually support the Submittable interface.

Jobs are called Tasks in HiLA, in the following, we will refer to them in this terminology from now on. Depending on the implementation, you have several options to run a Task on a Site. The `submit()` method on the Submittable interface generously accepts `java.lang.Object` as an input parameter. The actual type of the passed object will be checked in the implementation of this method. All implementations support JSDL (see below) as input. The JSDL has to be encapsulated in special types, but JSDL is your best bet in any case. For UNICORE 5, you can also submit a so called Abstract Job Object (AJO), which is the native job model for UNICORE 5. In this implementation, even JSDL job descriptions will be transformed into AJO, as the backend implementation does not support anything else.

5.3.1 Job Submission Description Language – JSDL

JSDL is specified by the OGF as the least common denominator that Grid middleware can support. It is a description of a task with references to executables or applications, resources required by the job, and data staging.

5.3.2 Abstract Applications

Among the key concepts in the UNICORE Grid middleware is that of abstract applications. Abstract applications take the burden of knowing the exact installation details of certain software from the user and put it on the administrator of a Grid site. The rationale behind this is that administrators know their machines much better than users. Users on the other hand do not need to know installation details of the various software packages available on a site. At the same time, abstract applications can be advertised through the target system's properties.

Any application available on a target system can be wrapped as an abstract application. Sometimes even different configurations of applications can be made different abstract applications. Abstract applications are identified by name and version.

JSDL perfectly supports this approach, as you can see in the following example.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jsd1:JobDefinition
  xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl" id="Script">
  <jsd1:JobDescription>
    <jsd1:Application>
      <jsd1:ApplicationName>Date</jsdl:ApplicationName>
      <jsd1:ApplicationVersion>1.0</jsdl:ApplicationVersion>
    </jsdl:Application>
  </jsdl:JobDescription>
</jsdl:JobDefinition>
```

In the listing above, the application to be executed is Date version 1.0. This application is configured in default configurations of UNICORE as an example for abstract applications. Usually, it is configured to execute /bin/date, which is usually available on all systems.

5.3.3 File Staging

JSDL provides a means to describe files to be staged in before running a job and files to be staged out after a job has completed successfully.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsd1:JobDefinition
  xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <jsd1:JobDescription>
    <jsd1:DataStaging>
      <jsd1:FileName>foo</jsdl:FileName>
      <jsd1:CreationFlag>overwrite</jsdl:CreationFlag>
      <jsd1:Source>
        <jsd1:URI>
          uncore6:/sites/Alpha-Site/storages/home/files/.bashrc
        </jsdl:URI>
      </jsdl:Source>
    </jsdl:DataStaging>

    <jsd1:DataStaging>
```

```

<jsd1:FileName>foo</jsdl:FileName>
<jsd1:CreationFlag>dontOverwrite</jsdl:CreationFlag>
<jsd1:Target>
  <jsd1:URI>
    uncore6:/sites/Bravo-Site/storages/home/files/.bashrc
  </jsdl:URI>
</jsdl:Target>
</jsdl:DataStaging>
</jsdl:JobDescription>
</jsdl:JobDefinition>

```

What the JSDL specification does not define are the actual contents of the `jsdl:URI` element. In order to make the most use of HiLA and not have to write different JSDL description files for each target middleware, one can insert HiLA URIs into the `jsdl:URI` element, which will be rewritten by HiLA to match the implementation's native format. This will not actually work for all implementations. In those cases, HiLA will forward the URI unaltered. The same applies of course, if the URI inside `jsdl:URI` is not a HiLA URI.

5.3.4 The actual submission

Having a JSDL document at hand, we need to submit it to a Site for execution. This can be done using the following code.

```

...
import org.ggf.schemas.jsdl.x2005.x11.jsdl.JobDefinitionDocument;

JobDefinitionDocument jsdl = JobDefinitionDocument.Factory.parse(
    new java.io.File(this.args[1]));

Site site = HiLAFactory.getInstance().locate(
    new Location(args[0]));
Task task = site.submit(jsdl);
task.startSync();

```

The first thing we do in the above snippet of code is to parse the contents of a file into a `JobDefinitionDocument`, which is a Java binding type for JSDL documents. Next, we locate a `Site` from a `Location`, which is instantiated with a `String` contained in `args[0]`. The JSDL object is then submitted to the `Site` and we receive a `Task` in return.

In the plain HiLA model, `Tasks` need to be started explicitly. The simple reason for this is that you may want to stage in local files from the client machine to the working directory of the `Task`. This can be done by adding a list of files (`java.io.File` as a parameter to the `startSync` method. The same applies for the `startASync` method.

`startSync` will block on the `Task` until it is finished. This behavior may not always be desirable, e.g. if you want to submit and run multiple jobs simultaneously and monitor the bulk of them all at the same time. Therefore, you can use the `startASync` method, which will use a thread to start and monitor the `Task` in the background and immediately return. Thus, users of the interface do not have to deal with concurrent operations.

References

- [1] Aerogrid. <http://www.aero-grid.de/>.
- [2] Björn Hagemeyer, Roger Menday, Bernd Schuller, and Achim Streit. A Universal API for Grids. In Marian Bubak, Michael Turala, and Kazimierz Wiatr, editors, *Cracow Grid Workshop '06*, pages 312–319, ul. Nawojki 11, 30-950 Krakw 61, P.O. Box 386, Poland, 2007. Academic Computer Centre CYFRONET AGH.
- [3] Roger Menday, Lidia Kirtchakova, Bernd Schuller, and Achim Streit. An API for Building New Clients for UNICORE. In *Proceedings of the Cracow Grid Workshop '05*, Cracow, Poland, 2005.

Glossary

JSDL Job Submission Description Language.

OGSA-BES OGSA Basic Execution Services.