

HiLA 1.0

Roger Menday, Bjoern Hagemeier
{r.menday,b.hagemeier}@fz-juelich.de

February 15, 2008

1 Introduction

Highly dynamic features and security concerns often make it difficult for programmers to easily develop clients accessing Grids. There's a lot of overhead involved for each call to an underlying 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. Other backend implementations are conceivable enabling supporting other Grid infrastructures 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 URI's, where the scheme is used to select a particular implementation.

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 project¹.

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 Grids².

2 Usage

2.1 Configuration

Before starting to use HiLA, a minimal amount of configuration is necessary. This is predominantly due to the fact that no Grid resources can be accessed without user credentials. Any user of the Grid must be authenticated to do anything.

The currently available implementations of the HiLA API differ in configuration. Different backends have different demands for handling security. While for the UNICORE 5 implementation it is perfectly alright to

¹<http://forge.nesc.ac.uk/projects/deisa-jra7/>

²<http://www.a-ware-project.eu/>

select only an X509 private key and decrypt it, parameters for the UNICORE 6 implementation are many. Choices are to use WS-Security, client and server authentication, etc.

The `Config` interface and associated implementations thereof contain this configuration information. HiLA uses Spring³ to declaratively configure the `Config` objects for each implementation.

Each supported implementation is given a name. For UNICORE 5 and UNICORE 6 these are `unicore5` and `unicore6`. The preceding algorithm is followed to

- System property `hila/{scheme}.xml`
- current directory `{scheme}.xml`
- `$/USER/.hila/{scheme}.xml`
- by looking on the classpath for `de/fzj/hila/{scheme}.xml`

2.1.1 UNICORE 5

To be updated

2.1.2 UNICORE 6

According to the discovery rules, an example file (`unicore6.xml`) is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:hila="http://www.unicore.eu/hila-unicore6">
  <hila:unicore6grid id="grid"
    outcomeDirectory="file:${user.home}/.hila/data" config="#config" />

  <hila-common:compositeconfig id="config"
    xmlns:hila-common="http://www.unicore.eu/hila-common">
    <constructor-arg>
      <list>
        <hila:registryconfig
          registryURL="https://host1:1234/REGISTRY/services/Registry?res=default_registry" grid="#grid" securityPro
        <hila:registryconfig
          registryURL="https://host2:1235/REGISTRY/services/Registry?res=default_registry" grid="#grid" securityPro
        </list>
      </constructor-arg>
    </hila-common:compositeconfig>

    <bean name="proawar0.security"
      class="de.fzj.hila.implementation.unicore6.Unicore6SecurityProperties">
      <constructor-arg value="/etc/hila/s1.security" />
    </bean>

  </beans>
```

An example of the referenced file `s1.security` can be found in the HiLA download. Essentially, it contains the low-level credential information, as simple key-value pairs.

2.2 Starting up

First of all you need a `Grid` object to be able to do anything. This is your entrance into the Grid!

³<http://www.springframework.org/>

```

Config config = new CSVConfig(new File("etc/deisa.csv"));
Grid grid = HiLAFactory.getInstance().getGrid(config, new
File("etc/HiLA.properties"));

```

2.3 Location and Locatable

Every significant resource addressable by HiLA is a `Locatable` object, and possesses a `Location`.

A site which has a abstract name of 's1'

```
unicore6:/sites/u1
```

... the home directory which the u1 site

```
unicore6:/sites/u1/storages/home
```

... and file `abc.tex` inside the home directory

```
unicore6:/sites/u1/storages/home/files/abc.txt
```

A running task on the system

```
unicore6:/sites/u1/tasks/1254345
```

2.4 Sites

Having a look at all my sites.

```

List<Site> sites = grid.getAllSites();
for (Site site : sites)
{
    System.out.println( site + "    " + site.ok() );
}

```

2.5 Storages and Files

Listing the contents of the home directory.

```

Site u1 = ...
List<File> fls = u1.getStorage(StorageType.getType("home")).asFile("").ls();
for (File file : fls)
{
    System.out.println( file.getLocation() + "    " + file.size() );
}

```

Doing a transfer from one site to another.

```

File src= (File)hilaf.locate(new Location("unicore6:/sites/U1/storages/home/files/a.txt"));
File dst= (File)hilaf.locate(new Location("unicore6:/sites/U2/storages/home/files/abc/destination.txt"));

assertTrue( sf.exists() ); // perhaps

Task t = sf.transfer(df, true);
t.startASync();

```

2.6 Tasks

```
Site site = ...
Task t = site.submit( jsdl );
t.start(false);
```

You can register a listener for changes in the status of the job.

```
Site site = ...
Task t = site.submit( jsdl );
t.registerStatusChangeListener(new StatusChangeListener()
{
    public void notify(TaskState newstate, Task task)
    {
        log.info(newstate.toString());
    }
});
t.start(false);
```

2.7 More ...

More usage information can be found in the javadocs for the API.

3 Development

3.1 Build system

HiLA is entirely built with Maven 2. Dependencies should be drawn in from the central and unicore.eu repositories. The unicore.eu repository is references inside the top-level POM file.

Tests for individual packages are currently not always working. This is due to limited server availability and lack of configuration. To avoid running tests when installing or deploying artifacts, you can add `'-Dmaven.test.skip'` to the call of `'mvn'`.

3.2 Modules

```
HiLA
├── HiLA-api
├── HiLA-common
├── HiLA-garden
├── HiLA-tests
├── HiLA-unicore5
└── HiLA-unicore6
```

3.3 Profiles

Three modules are only activated, if respective profiles are active during build time. Profiles and modules are

- unicore5 – HiLA-unicore5 implementation
- unicore6 – HiLA-unicore6 implementation
- garden – HiLA-garden
- tests – HiLA-tests

When running Maven, profiles are activated on the command line by adding `'-Pname-of-profilei,j...l'` to the `'mvn'` command. So, multiple profiles are activated by comma separated lists. That would be necessary to activate both the `'unicore5'` and `'unicore6'` profiles.

3.4 New implementations

As a purely abstract API, you need particular implementations to make use of HiLA. The `HiLAFactory` looks for available implementations on the classpath. They are identified by the availability of JAR files that contain `META-INF/services/de.fzj.HiLA.HiLAFactory$HiLAImpFactory`. The contents of a file is the qualified name of the class implementing the interface `HiLAImpFactory`. More documentation about service providers can be found in the JAR file specification⁴.

3.5 Versioning

Proper versioning is essential to enable developers using a library to target the correct version of interfaces. Targetting a specific and fixed version, they avoid the hassle of constantly having to follow changes in APIs and their implementations.

HiLA distinguishes two kinds of versions: developer and user. User versions use fixed interfaces (unless seriously broken), while the interfaces of developer versions are subject to change during the period of development. Once sufficiently stable, developer versions turn into user versions.

The use of Maven as a build tool supports this approach. Using `SNAPSHOT` versions, Maven attempts to download new versions of packages once a day. This is used for developer versions. Fixed user versions usually aren't updated. As an example, consider the current version `1.0-SNAPSHOT`. As a `SNAPSHOT` version, Maven attempts to retrieve new versions once a day. Patches to this version will be provided as `1.0-p<n>`. While this version is fixed and only changed for bug fixes, the new developer version will become `1.1-SNAPSHOT`, until that is sufficiently stable to roll out as version `1.1`.

⁴<http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html>