COMP Superscalar

User Guide

Version 1.1.1

Barcelona Supercomputing Center

Centro Nacional de Supercomputación

1. Developing COMPSs applications

In this section the steps to develop a COMPSs application will be illustrated; the sequential **Simple application** will be used to explain an application porting to COMPSs. The user is required to select a set of methods, invoked in a sequential application, to be run as remote tasks on the available resources.

A COMPSs application is composed of three parts:

- **Main application code:** the code that is executed sequentially and contains the calls to the user-selected methods that will be executed on the Cloud.
- **Remote methods code:** the implementation of the remote tasks.
- **Java annotated interface:** It declares the selected methods to be run as remote tasks and metadata used to schedule the tasks.

The main application code (sequential) will have the name of the application, always starting with capital letter, in this case will be **Simple.java.** The Java annotated interface will be named as *application name+ Itf.java* in this case will be **SimpleItf.java.** And the code that implements the remote tasks will be called as *application name + Impl.java*, in this case will be **SimpleImpl.java**.

All code examples are in the **/home/user/workspace/** folder of the development environment.

1.1. Main application code

In COMPSs the application is kept completely unchanged, i.e. no API calls need to be included in the main application code in order to run the selected tasks on the nodes.

The COMPSs runtime is in charge of replacing the invocations to the user-selected methods with the creation of remote tasks also taking care of the access to files from the main application code.

Let's consider the *Simple* application example that takes an integer as input parameter and increases it by one unit.

The main application code of Simple app (**Simple.java**) will be executed in a sequential way except the **Increment()** method. COMPSs, as mentioned above, will replace at execution time the call to this method generating a remote task on the remote node.

```
package simple;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import simple.SimpleImpl;

public class Simple {

  public static void main(String[] args) {
    String counterName = "counter";
    int initialValue = args[0];

    /*-------------------------------------------------------
    Creation of the file which will contain the counter variable
    -------------------------------------------------------*/
    try {
      FileOutputStream fos = new FileOutputStream(counterName);
      fos.write(initialValue);
      System.out.println("Initial counter value is " + initialValue);
      fos.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }

    /*-----------------------------------------------
              Execution of the program
    -----------------------------------------------*/
          SimpleImpl.increment(counterName);
```

```
    /*-----------------------------------------------
          Reading from an object stored in a File
    -----------------------------------------------*/
    try {
      FileInputStream fis = new FileInputStream(counterName);
      System.out.println("Final counter value is " + fis.read());
      fis.close();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

1.2. Remote methods code

The following code is the implementation of the remote method of the *Simple* application (**SimpleImpl.java**) that will be executed remotely by COMPSs.

```
package simple;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.FileNotFoundException;

public class SimpleImpl {
  public static void increment(String counterFile) {
    try{
      FileInputStream fis = new FileInputStream(counterFile);
      int count = fis.read();
    fis.close();

    FileOutputStream fos = new FileOutputStream(counterFile);
      fos.write(++count);
    fos.close();
    }catch(FileNotFoundException fnfe){
    fnfe.printStackTrace();
    }catch(IOException ioe){
      ioe.printStackTrace();
    }
  }
}
```

1.3. Java annotated interface

The Java interface is used to declare the methods to be executed remotely along with Java annotations that specify the necessary metadata about the tasks. The metadata can be of three different types:

1. For each parameter of a method, the data type (currently *File* type, primitive types and the *String* type are supported) and its directions (IN, OUT or INOUT).
2. The Java class that contains the code of the method.
3. The constraints that a given resource must fulfil to execute the method, such as the number of processors or main memory size.

Here follows a complete and detailed explanation of the usage of the metadata:

- Method-level Metadata: for each selected method, the following metadata has to be defined:

- **@Method:** Mandatory. It specifies the class that implements the method.
- **@Constraints**: Mandatory. The user can specify the capabilities that a resource must have in order to run a method. The COMPSs runtime will create a VM (in a cloud environment), that fits the specified requirements in order to perform the execution.

- Processor:
  - **processorCPUCount:** Number of required processors.
- Memory:
  - **memoryPhysicalSize**: Amount of GB of physical memory needed.

- Parameter-level Metadata (@Parameter): for each parameter and method, the user must define:

- **Direction:** *Direction.IN, Direction.INOUT* or *Direction.OUT*
- **Type:** COMPSs supports the following types for task parameters:
  - **Basic types:** *Type.BOOLEAN, Type.CHAR, Type.BYTE, Type.SHORT, Type.INT, Type.LONG, Type.FLOAT, Type.DOUBLE*. They can only have **IN** direction, since primitive types in Java are always passed by value.
  - **String**: *Type.STRING*. It can only have **IN** direction, since Java Strings are immutable.
  - **File:** *Type.FILE*. It can have any direction (IN, OUT or INOUT). The real Java type associated with a FILE parameter is a String that contains the path to the file. However, if the user specifies a parameter as a FILE, COMPSs will treat it as such.
  - **Object**: *Type.Object*. It can have any direction (IN, OUT or INOUT).
- **Return type**: Any object, a basic type or a generic class object.
- **Method modifiers:** the method has to be **STATIC**.

The Java annotated interface of the Simple app example (SimpleItf.java) declares the *Increment()* method that will be executed remotely. The method implementation can be found in simple.SimpleImpl class and needs a single input parameter, a string containing a path to the file counterFile. Besides, in this example there are constraints on the minimum number of processors and minimum memory size needed to run the method.

```
package simple;

import integratedtoolkit.types.annotations.Constraints;
import integratedtoolkit.types.annotations.Method;
import integratedtoolkit.types.annotations.Parameter;
import integratedtoolkit.types.annotations.Parameter.Direction;
import integratedtoolkit.types.annotations.Parameter.Type;


public interface SimpleItf {

      @Constraints(processorCPUCount = 1, memoryPhysicalSize = 0.3f)
       @Method(declaringClass = "simple.SimpleImpl")
       void increment(
             @Parameter(type = Type.FILE, direction = Direction.INOUT)
             String file
       );

}
```

1.4. Compiling and Packaging the application

The application can be compiled either using the command line or through the Eclipse IDE tool available on the SDK VM.

```
user@bsccompss:~$ cd /home/user/workspace/
user@bsccompss:~/workspace$ javac simple/src/simple/*.java
user@bsccompss:~/workspace/simple/src$ jar cf simple.jar simple
```

Once the application is compiled and packaged in a **Jar archive** it have to be bundled in a **tar.gz** package; this package will be automatically deployed by COMPSs in a cloud environment, while in the case of a static pool of physical nodes the jar has to be manually pre-deployed.

In this example the application package is stored under **/home/user/workspace/APPNAME/package**

```
user@bsccompss:~/workspace/simple/src$ tar czvf Simple.tar.gz simple.jar
user@bsccompss:~/workspace/simple/src$ mv Simple.tar.gz
~/workspace/simple/package/
```

In case of having to supply external binaries or libraries to the application, as is the case of Blast (see Section 2) they have to be copied into a directory named “binary”, and “lib” respectively and packaged in the ***tar.gz*** package as shown below:

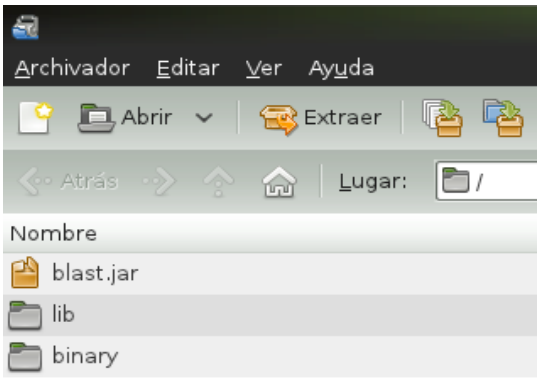


Figure 1 - An example package

1.5. Running the application

To run the application on the SDK VM, the path of the application jar must be added to **CLASSPATH** system variable:

```
user@bsccompss:~$ cp ~/workspace/simple/package/Simple.tar.gz /home/user/
user@bsccompss:~$ tar xzf Simple.tar.gz
user@bsccompss:~$ export CLASSPATH=$CLASSPATH:/home/user/simple.jar
```

Once the execution environment is ready, the application can be launched through the following command:

```
user@bsccompss:~$ runcompss simple.Simple <initial_number>
```

```
user@bsccompss:~$ runcompss simple.Simple 1
-e
----------------- Executing simple.Simple in IT mode total-------------------------

[Loader]  -  Modifying application simple.Simple with loader total
[Loader]  -  Application simple.Simple instrumented, executing...
[   API]  -  Deploying the Integrated Toolkit
[   API]  -  Starting the Integrated Toolkit
[   API]  -  Initializing components
[   API]  -  Ready to process tasks
[   API]  -  Opening file /home/user/IT/simple.Simple/counter in mode WRITE
Initial counter value is 1
Final counter value is 2
[   API]  -  All tasks finished
[   API]  -  Temporary files deleted
[   API]  -  Stopping IT
[   API]  -  Integrated Toolkit stopped

------------------------------------------------------------
```

Figure 2 - Execution of a COMPSs application

1.6. Logging the execution

The **it.log** file, that can be found generally in **$HOME/it.log (/home/user/it.log, in case of SDK VM)**. It shows information on the execution of the application including file transfers and job submission details.

```
user@bsccompss:~$ tail -f it.log
```

On the other hand, the **resources.log** file, that can be found in **$HOME/resources.log**. It shows information about the available resources such as: number of processors of each resource (slots), information about running or pending tasks in the resource queue as depicted in the following picture:

```
user@bsccompss:~$ cat resources.log
localhost has been added to available resource pool as a physical resource with 4 slots
Core 0 can be executed on 4 slots
        |-localhost   (4 slots)
- Resource localhost can execute the cores:
        |-0
        On execution:
                localhost is running core(s): 1
                Pending:
        Task 1 ends.
        On execution:
                localhost is running core(s):
                Pending:
user@bsccompss:~$
```

Figure 3 - Information on the available resources

1.7. Execution results

After the execution, COMPSs stores the files corresponding to the **stdout** and **stderr** of each task in the **/home/user/IT/APPNAME/** directory.

```
user@bsccompss:~$ cd IT/simple.Simple/
user@bsccompss:~/IT/simple.Simple$ ls
counter  job1.err  job1.out
user@bsccompss:~/IT/simple.Simple$
```

```
user@bsccompss:~/IT/simple.Simple$ cat job1.out
WORKER - Parameters of execution:
  * Method class: simple.SimpleImpl
  * Method name: increment
  * Parameter types: java.lang.String
  * Parameter values: d1v2_1320849547354.IT
```

Figure 4 - The log of each task can be retrieved at the end of the execution

The results of an execution will be stored in the **/home/user/IT/simple.Simple/** directory in the example application.

1.8. Exploring the final execution graph

At the end of the execution a dependency graph can be generated representing the order of execution of each type of task and their dependencies.

```
user@bsccompss:~$ gengraph graph.dot
```

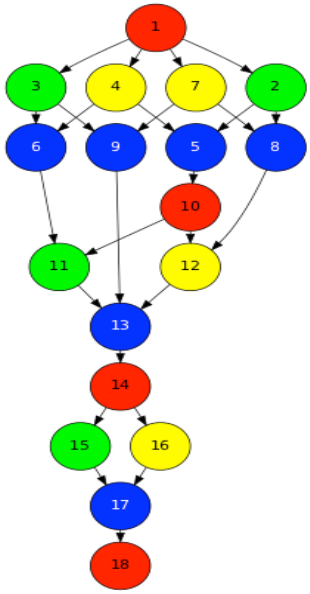


Figure 5 - The dependency graph of the SparseLU application

1.9. COMPSs monitoring system

The COMPSs runtime exposes a Web Service with a graphical interface that can be used to monitor the progress of running applications. In order to see it, a specific URL must be used in a web browser:

```
http://localhost:8080/compss-monitor
```

As it can be seen in Figure 6, the interface gives details about the execution graph (it can be seen how the data dependency graph is built and consumed at real time, and the status of the tasks), the resource usage information, the number of tasks, and the execution time per task.

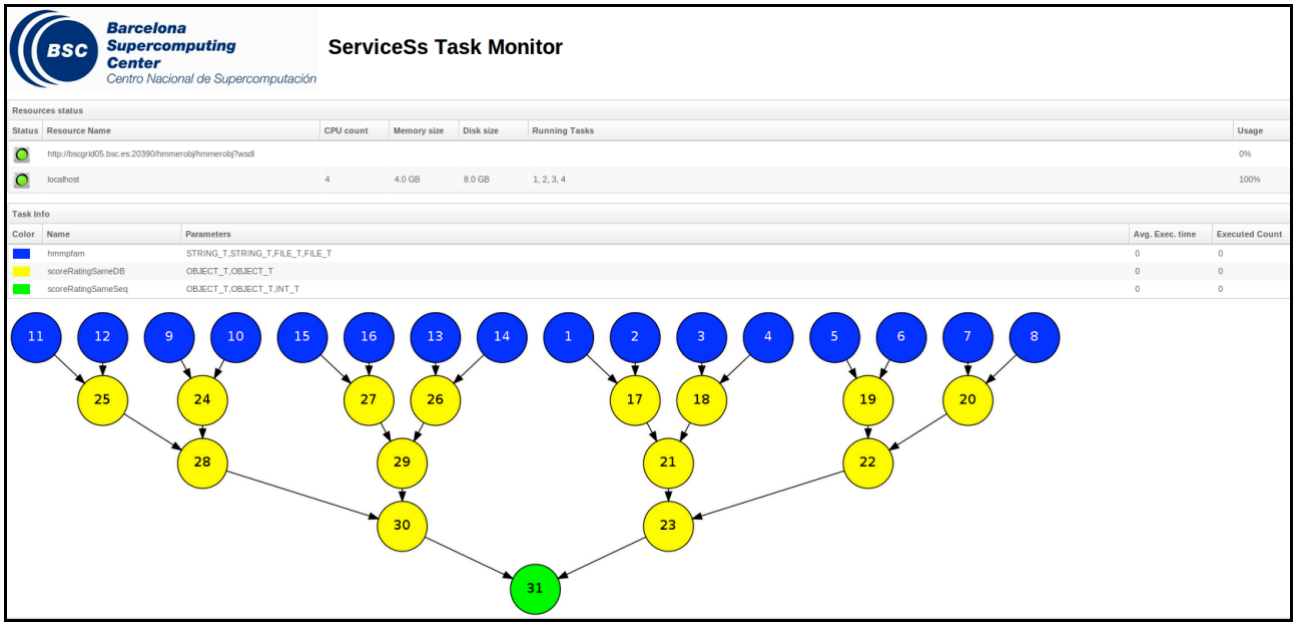


Figure 6 – COMPSs monitoring interface example

2. Sample applications

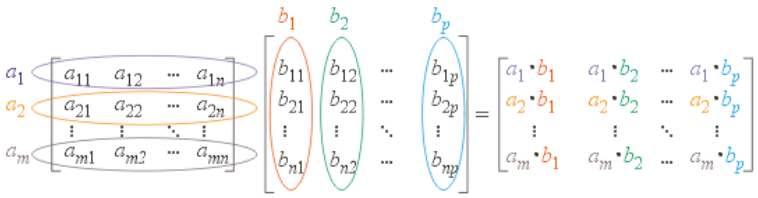
The examples in this section consider the execution in a cloud platform where the VMs mount a common storage on **/sharedDisk** directory**.** This is useful in the case of applications that require working with big files, allowing to transfer data only once, at the beginning of the execution, and to enable the application to access the data directly during the rest of the execution.

The Blast sample workflows available in the development environment and explained in the next section takes advantage of this functionality.

The development environment provides some sample COMPSs applications that can be found in **/home/user/workspace/** directory. The following section describes in detail the development of each of them.

2.1. Matrix multiplication

Matrix Multiplication (Matmul) is a pure Java application that multiplies two matrices in a direct way. The application creates 2 matrices of N x N size initialized with values, and multiply the matrices by blocks of 40 floats (by default).



In this application the multiplication is implemented in the **multiplyAccumulative** that is thus selected as the task that will be executed remotely. In order to run the application the matrix dimension has to be supplied.

```
user@bsccompss:~$ cp ~/workspace/matmul/package/Matmul.tar.gz /home/user/
user@bsccompss:~$ tar xzf Matmul.tar.gz
user@bsccompss:~$ export CLASSPATH=$CLASSPATH:/home/user/matmul.jar
```

The command line to execute the application:

```
user@bsccompss:~$ runcompss matmul.Matmul <matrix_dim>
```

2.2. Sparse LU decomposition

SparseLU multiplies two matrices using the factorization method of LU decomposition, which factorizes a matrix as a product of a lower triangular matrix and an upper one.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

The matrix is divided into N x N blocks on where 4 types of operations will be applied modifying the blocks: **lu0, fwd, bdiv and bmod**. These four operations are implemented in four methods that are selecetd as the tasks that will be executed remotely. In order to run the application the matrix dimension has to be provided.

```
user@bsccompss:~$ cp ~/workspace/sparselu/package/SparseLU.tar.gz /home/user/
user@bsccompss:~$ tar xzf SparseLU.tar.gz
user@bsccompss:~$ export CLASSPATH=$CLASSPATH:/home/user/sparselu.jar
```

The command line to execute the application:

```
user@bsccompss:~$ runcompss sparselu.SparseLU <matrix_dim>
```

2.3. BLAST Workflow

BLAST is a widely-used bioinformatics tool for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences with sequence databases, identifying sequences that resemble the query sequence above a certain threshold. The work performed by the COMPSs Blast workflow is computationally intensive and embarrassingly parallel.

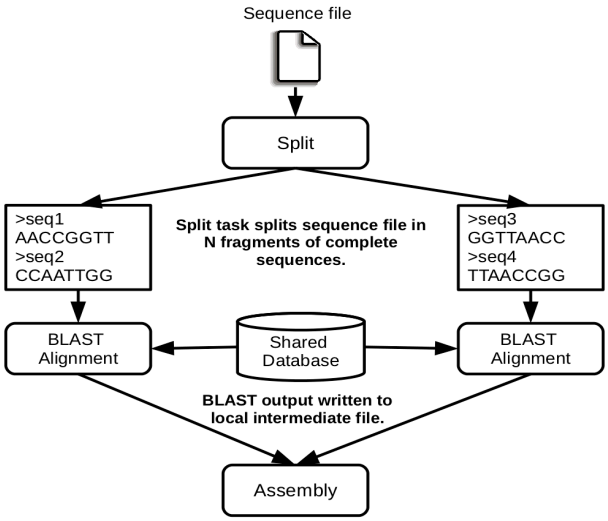


Figure 7 - The COMPSs Blast workflow

The workflow describes the three blocks of the workflow implemented in the **Split, Align** and **Assembly** methods. The second one is the only method that is chosen to be executed remotely, so it is the unique method defined in the interface file. The **Split** method chops the query sequences file in N fragments, **Align** compares each sequence fragment against the database by means of the Blast binary, and **Assembly** combines all intermediate files into a single result file.

This application uses a database that will be on the shared disk space avoiding transferring the entire database (which can be large) between the virtual machines.

```
user@bsccompss:~$ cp ~/workspace/blast/package/Blast.tar.gz /home/user/
user@bsccompss:~$ tar xzf Blast.tar.gz
user@bsccompss:~$ export CLASSPATH=$CLASSPATH:/home/user/blast.jar
```

The command line to execute the workflow:

```
user@bsccompss:~$ runcompss blast.Blast <debug> <bin_location>
                  <database_file> <sequences_file> <frag_number> <tmpdir>
                  <output_file>
```

Where:

- **debug:** The debug flag of the application [true | false].
- **bin_location:** Path of the Blast binary.
- **database_file:** Path of database file; the shared disk **/sharedDisk/** is suggested to avoid big data transfers**.**
- **sequences_file:** Path of sequences file.
- **frag_number:** Number of fragments of the original sequence file, this number determines the number of parallel Align tasks.
- **tmpdir:** Temporary directory (**/home/user/tmp/).**
- **output_file:** Path of the result file.

Example:

```
user@bsccompss:~$ runcompss blast.Blast true
/home/user/workspace/blast/binary/blastall
/sharedDisk/Blast/databases/swissprot/swissprot
/sharedDisk/Blast/sequences/sargasso_test.fasta 4 /tmp/
/home/user/out.txt
```

3. COMPSs Configuration

COMPSs SDK VM is a preconfigured light 64-bit Xubuntu distribution providing the necessary set of tools to develop COMPSs applications. The development environment includes an **Eclipse IDE**, the **COMPSs framework** and a set of **sample applications** in order to ease the comprehension of the programing model in a more straightforward way.

The COMPSs framework is installed in */opt/COMPSs/*;

For the development of new projects in Eclipse please remember to add the reference to the COMPSs runtime adding */opt/COMPSs/Runtime/rt/compss-rt.jar* as referenced library.

Please note that in case of changing the number of available cores in the physical SDK machine, this should be reflected in the COMPSs configurations files, **resources.xml** and **project.xml,** as indicated in the picture.

project.xml

```
user@bsccompss:~$ cat /opt/COMPSs/Runtime/xml/projects/project.xml
<?xml version="1.0" encoding="UTF-8"?>
 <Project>
     <!--Description for any physical node-->
     <Worker Name="localhost">
         <InstallDir>/IT_worker/</InstallDir>
         <WorkingDir>/home/user/</WorkingDir>
         <User>user</User>
         <LimitOfTasks>2</LimitOfTasks>
     </Worker>
 </Project>
```

resources.xml

```
user@bsccompss:~$ cat /opt/COMPSs/Runtime/xml/resources/resources.xml
<?xml version="1.0" encoding="UTF-8"?>
 <ResourceList>
     <!--Description for any physical node-->
     <Resource Name="localhost">
         <Capabilities>
             <Host>
                 <TaskCount>0</TaskCount>
                 <Queue>short</Queue>
                 <Queue/>
             </Host>
             <Processor>
                 <Architecture>IA32</Architecture>
                     <Speed>3.0</Speed>
                     <CPUCount>2</CPUCount>
             </Processor>
             <OS>
                 <OSType>Linux</OSType>
                 <MaxProcessesPerUser>32</MaxProcessesPerUser>
             </OS>
             <StorageElement>
                 <Size>30</Size>
             </StorageElement>
            <Memory>
                 <PhysicalSize>2</PhysicalSize>
                 <VirtualSize>8</VirtualSize>
             </Memory>
             <ApplicationSoftware>
                 <Software>Java</Software>
             </ApplicationSoftware>
             <Service/>
             <VO/>
             <Cluster/>
             <FileSystem/>
```

```
            <NetworkAdaptor/>
            <JobPolicy/>
            <AccessControlPolicy/>
        </Capabilities>
        <Requirements/>
    </Resource>
<ResourceList>
```

In order to use external resources to execute the applications, the following steps have to be followed:

1. Install the COMPSs framework on the new resources following the installation manual available at http://www.bsc.es/compss.
2. Edit the **resources.xml** and **project.xml** files in the master machine (the SDK VM) in order to be aware of the new resources (Section 4).
3. Create/set the WorkingDir in the path specified in **project.xml**
4. Set SSH passwordless access to the rest of the remote resources.
5. In case of cloud resources the application will be deployed automatically, and this, should be set up on COMPSs configuration files. In case of static resources, deploy the application manually on the new ones (see section 3.1 and 3.2).

3.1. Grid/Cluster configuration (static resources)

On the following lines, we provide examples about configuration files for Grid and Cluster environments, which can serve as a reference. They can also be compared to the examples previously provided to see the differences in such scenarios.

project.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Project>
    <!--Description for any physical node-->

    <Worker Name="172.20.200.18">
        <InstallDir>/opt/COMPSs/Runtime/scripts/system/</InstallDir>
        <WorkingDir>/tmp/</WorkingDir>
        <User>user</User>
        <LimitOfTasks>1</LimitOfTasks>
    </Worker>

    <Worker Name="172.20.200.19">
     …
    </Worker>

     …
</Project>
```

resources.xml

```
?xml version="1.0" encoding="UTF-8"?>
<ResourceList>
     <!--Description for any physical node-->

     <Resource Name="172.20.200.18">
         <Capabilities>
             <Host>
                 <TaskCount>0</TaskCount>
                 <Queue>short</Queue>
                 <Queue/>
             </Host>
             <Processor>
                 <Architecture>IA32</Architecture>
                 <Speed>3.0</Speed>
                 <CPUCount>1</CPUCount>
             </Processor>
             <OS>
                 <OSType>Linux</OSType>
                 <MaxProcessesPerUser>32</MaxProcessesPerUser>
             </OS>
             <StorageElement>
                 <Size>30</Size>
             </StorageElement>
                …
             <Memory>
                 <PhysicalSize>1</PhysicalSize>
                 <VirtualSize>8</VirtualSize>
             </Memory>
             <ApplicationSoftware>
                 <Software>Java</Software>
             </ApplicationSoftware>
             <Service/>
             <VO/>
             <Cluster/>
             <FileSystem/>
             <NetworkAdaptor/>
             <JobPolicy/>
             <AccessControlPolicy/>
```

```
            </Capabilities>
            <Requirements/>
        </Resource>

        <Resource Name="172.20.200.19">
         ...
        </Resource>

<ResourceList>
```

3.2. Cloud provider configuration (dynamic resources)

The COMPSs runtime communicates with the Cloud by means of Cloud connectors. Each connector implements the interaction of the runtime with a given Cloud provider, more precisely by supporting four basic operations: ask for the price of a certain VM in the provider, get the time needed to create a VM, create a new VM and terminate a VM.

Connectors abstract the runtime from the particular API of each provider; furthermore, this design facilitates the addition of new connectors for other providers.

The next subsections describe the basic configuration options for Cloud provider connectors and provide a description of each of the connectors currently available.

3.2.1 Configuration

The connectors can be configured by providing some information in the **resources.xml** and **project.xml** files. These files are located at <COMPSs_INSTALL_DIR>/xml/projects and <COMPSs_INSTALL_DIR>/xml/resources. Examples files for different backends are provided in the <COMPSs_INSTALL_DIR>/xml/projects/examples and <COMPSs_INSTALL_DIR>/xml/resources/examples folders.

3.1.1. Resources

The resources.xml file can contain one or more tags **<CloudProvider>** that encompass the information about a particular Cloud provider, associated to a given connector. The tag must have an attribute ***name*** to uniquely identify the provider. Table 1 summarizes the information to be specified by the user inside this tag.

Table 1 – Configuration of resources.xml file, tag <CloudProvider>

`Server`	Endpoint of the provider's server
`Connector`	Class that implements the connector
`ImageList` - **`Image`** ○ **`ApplicationSoftware`** ▪ **`Software`**	Multiple entries of VM templates - VM image ○ Multiple entries of software installed in the VM image ▪ Software installed in the VM image
`InstanceTypes` - **`Resource`** ○ **`Capabilities`** ▪ **`Processor`** ▪ **`StorageElement`** ▪ **`Memory`**	Multiple entries of resource templates - Instance type offered by the provider ○ Hardware details of instance type ▪ Architecture and number of available cores ▪ Size in GB of the storage ▪ PhysicalSize, in GB of the available RAM

3.1.2. Project

The project.xml complements the information about Cloud providers specified in the resources.xml file. This file can contain a **<Cloud>** tag where to specify a list of providers, each with a **<Provider>** tag, whose ***name*** attribute must match one of the providers in the resources.xml file. Thus, the project.xml file must contain a subset of the providers specified in the resources.xml file. Table 2 summarizes the information to be specified by the user in the <Provider> tags of the project.xml file.

Table 2 - Configuration of project.xml file, tag <Cloud>

`InitialVMs`	Number of VM to be created at the beginning of the application
`minVMCount`	Minimum number of VMs available in the computation
`maxVMCount`	Maximum number of VMs available in the computation
`Provider` - `LimitOfVMs` - `ImageList` o `Image` ▪ `InstallDir` ▪ `WorkingDir` ▪ `User` ▪ `Package` • `Source` • `Target` - `InstanceTypes` o `Resource` - `Property` o `Name` o `Value`	Multiple entries of Cloud providers – Maximum number of VMs allowed by the provider – Multiple entries of VM images available at the provider o VM image ▪ Path of the COMPSs worker scripts in the image ▪ COMPSs working directory in the image ▪ Account username ▪ Multiple entries of user packages to be deployed to the VM • Local path of the package • VM path where to deploy the package – Resource types available at the provider o Instance type offered by the provider – Multiple entries of provider-specific properties o Name of the property o Value of the property

3.2. Connectors

3.2.1. Amazon EC2

The COMPSs runtime features a connector to interact with the Amazon Elastic Compute Cloud (EC2).

Amazon EC2 offers a well-defined pricing system for VM rental. A total of 8 pricing zones are established, corresponding to 8 different locations of Amazon datacenters around the globe. Besides, inside each zone, several per-hour prices exist for VM instances with different capabilities. The EC2 connector stores the prices of standard on-demand VM instance types (*t1.micro*, *m1.small*, *m1.medium*, *m1.large* and *m1.xlarge*) for each zone. Spot instances are not currently supported by the connector.

When the COMPSs runtime chooses to create a VM in the Amazon public Cloud, the EC2 connector receives the information about the requested characteristics of the new VM, namely the number of cores, memory, disk and architecture (32/64 bits). According to that information, the connector tries to find the VM instance type in Amazon that better matches those characteristics and then requests the creation of a new VM instance of that type.

Once an EC2 VM is created, a whole hour slot is paid in advance; for that reason, the connector keeps the VM alive at least during such period, saving it for later use if necessary. When the task load decreases and a VM is no longer used, the connector puts it aside if the hour slot has not expired yet, instead of terminating it. After that, if the task load increases again and the EC2 connector requests a VM, first the set of saved VMs is examined in order to find a VM that is compatible with the requested characteristics. If one is found, the VM is reused and becomes eligible again for the execution of tasks; hence, the cost and time to create a new VM are not paid. A VM is only destroyed when the end of its hour slot is approaching and it is still in saved state.

Table 3 summarizes the provider-specific properties that must be defined in the project.xml file for the Amazon EC2 connector.

Table 3 – Properties of the Amazon EC2 connector

Placement	Location of the amazon datacentre to use
Access Key Id	Identifier of the access key of the Amazon EC2 account
Secret Key Id	Identifier of the secret key of the Amazon EC2 account
Key host location	Path to the SSH key in the local host, used to connect to the VMs
KeyPair name	Name of the key pair to use
SecurityGroup name	Name of the security group to use

3.3. rOCCI Connector

In order to execute a COMPSs application in the EGI Cloud the rOCCI connector has to be used and the configuration files have to be properly edited. The connector uses the rOCCI binary client being not available a Java API. These connector needs an additional file that provides details of each resource template available at the provider and that is located at <COMPSs_INSTALL_DIR>/xml/templates. The user must indicate which virtual images and instance types are offered by the specific provider; thus, when the runtime asks for the creation of a VM, the connector selects the appropriate image and resource template according to the requirements (in terms of cpu, memory, disk, etc) and invokes the rOCCI client through Mixins

Table 4 contains the rOCCI specific properties that must be defined in the project.xml file.

Table 4 - rOCCI extensions in the project.xml file

Provider	
ProxyCA	Path of the x509 certificate used to create the VOMS proxy
UserCert	Path of the generated VOMS proxy
CA	Path to CA certificates directory
AuthType	Authentication method, x509 only supported
Owner	Optional. Used by the VENUS-C Job Manager (PMES)

JobNameTag	

Table 5 - Configuration of the <provider>.xml templates file

Instance	Multiple entries of resource templates.
Type	Name of the resource template. It has to be the same name than in the previous files
CPU	Number of cores
Memory	Size in GB of the available RAM
Disk	Size in GB of the storage
Price	Cost per hour of the instance

Please find more details on the COMPSs framework at

www.bsc.es/compss