034115

PHOSPHORUS

Lambda User Controlled Infrastructure for European Research

Integrated Project

Strategic objective:
Research Networking Testbeds

Deliverable reference number D5.6

Grid Simulation Environment

Due date of deliverable: 2007-12-31
Actual submission date: 2008-01-04
Document code: <Phosphorus-WP5-D.5.6>

Start date of project:
October 1, 2006

Duration:
30 Months

Organisation name of lead contractor for this deliverable:
IBBT

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission	
RE	Restricted to a group specified by the consortium (including the Commission	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract

The Phosphorus project is addressing some of the key technical issues to enable on-demand, end-to-end network services across multiple domains in an optical Grid environment. Workpackage 5 (Supporting Studies) focuses on the design and evaluation of innovative architectures and algorithms to efficiently manage optical Grid infrastructures. Since the solutions obtained in WP5 can not be easily deployed in the testbed, a simulation environment is developed. The simulator allows to make comparisons between different architectural and algorithmic proposals, and assists in evaluating them in a rapid and straightforward way. In this deliverable, we present a basic simulation environment that implements a multi-domain, optical Grid network and a detailed model to represent a wide range of applications. Furthermore, we show extensions that support the evaluation and analysis of scheduling and fault-tolerance algorithms in Grid environments. Finally, further extensions that implement physical layer parameters are described so that impairment-constrained based routing algorithms can be studied.

Table of Contents

Table of Figures

0 Executive Summary

The Phosphorus project is addressing some of the key technical issues to enable on-demand, end-to-end network services across multiple domains in an optical Grid environment. Workpackage 5 (Supporting Studies) focuses on the design and evaluation of innovative architectures and algorithms to efficiently manage optical Grid infrastructures. Since the solutions obtained in WP5 can not be easily deployed in the testbed, a simulation environment is developed. The simulator allows to make comparisons between different architectural and algorithmic proposals, and assists in evaluating them in a rapid and straightforward way. In this deliverable, we present a basic simulation environment that implements a multi-domain, optical Grid network and a detailed model to represent a wide range of applications. Furthermore, we show extensions that support the evaluation and analysis of scheduling and fault-tolerance algorithms in Grid environments. Finally, further extensions are described that implement physical layer parameters and include additional routing algorithms. This way, impairment-constrained based routing approaches can be studied.

This deliverable presents the architectural view of the simulation environment, and clearly motivates significant design choices. We discuss in detail important variables, methods and portions of the code. Additionally, tutorial style documents are included based on simple scenarios, to ease the understanding and usage of the code. Also note that the simulation environment has been written with extensibility in mind, so it is straightforward to implement additional new features.

The simulation environment consists in large part of Java code, although the inclusion of physical layer parameters is written in the Matlab scripting language. Several important functions are accessible through the use of a Graphical User Interface, allowing intuitive configuration and setup of simulation scenarios. The open and standardized XML format is used for persistency and compatibility issues.

1 Overview

The Phosphorus Grid simulator consists of a basic discrete event simulator (see Figure 1) and grid-specific code (see Figure 2), implemented in Java [1]. For details on the base discrete event simulator, please refer to Section 2.1 (`simbase` package). For details on the grid entities and their interaction, see Sections 2.3 (`grid` package) and 2.4 (`grid.messages` package). The job generation process is described in Section 2.5 (`grid.jobs` package). For a step-by-step tutorial on the use of the simulator, refer to Section 3. Section 4 will introduce a Graphical User Interface for the grid simulator.

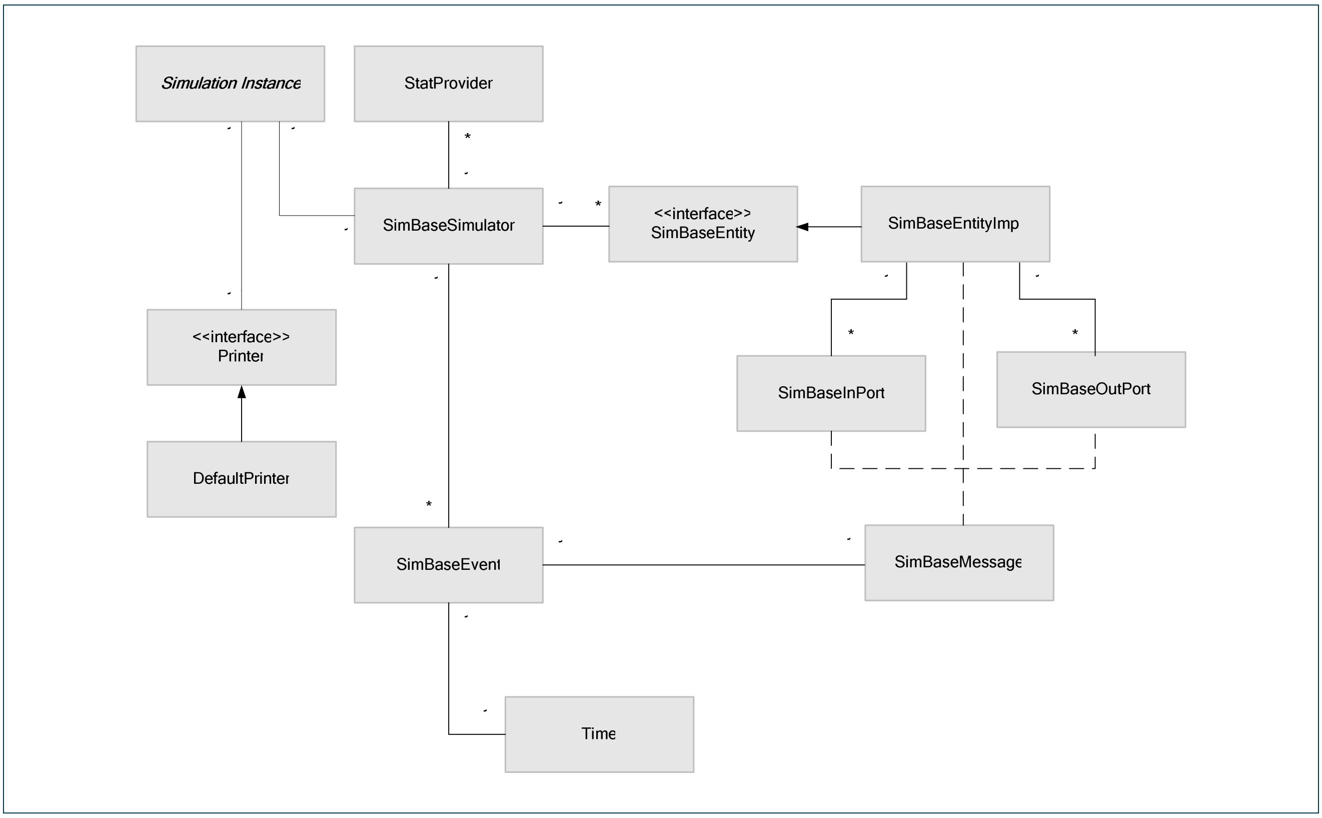


Figure 1 – Base discrete event simulator

The simulator's base class is the `SimBaseSimulator`, which contains several interacting `SimBaseEntity` objects. These are linked through their respective `SimBaseInPort` and `SimBaseOutPort` objects. Every event is represented by a `SimBaseEvent`, which contains a `Time` for the event and the `SimBaseMessage` to be delivered as a result of that event. The `SimBaseSimulator` also contains a `Logger` that provides logging facilities and some `StatProviders` that provide statistics about the simulation. To run a simulation one has to subclass the `SimulationInstance` and provide a `Printer` to print the information relevant to the simulation. Section 2.1 contains a more detailed explanation.

Derived from these base classes are the grid-specific classes (see Figure 2), which implement the grid functionality into the simulator. The `SimBaseEntity` interface descends into a `GridEntity` interface that is implemented by the `GridEntityImpl` class and contains some basic network functionality. Switches are represented by `SwitchImpl` objects, directly derived from `GridEntityImpl` and the interface `Switch`. Other grid entities, such as clients, resources and a service broker, are represented by `ClientNode(Impl)`, `ResourceNode(Impl)` and `ServiceNode(Impl)` interfaces and objects, which are derived from `GridEndEntity(Impl).` `GridEndEntity(Impl)` is a `GridEntity(Impl)` with only one outgoing and incoming link, directly linked to a `Switch`. `SimBaseInPort` and `SimBaseOutPort` objects become `GridEntityInPort` and `GridEntityOutPort`, and are responsible for network-specific tasks, such as keeping track of link usage and wavelength reservations. The `SimBaseMessage`, which is basically an empty placeholder for more specific messages, is replaced by its descendants that contain more specific information. The reason to work with interfaces to describe the different nodes is that one node can fulfill multiple tasks and since Java does not allow multiple inheritance this is the only way out (apart from creating a different node type for every node that is needed).

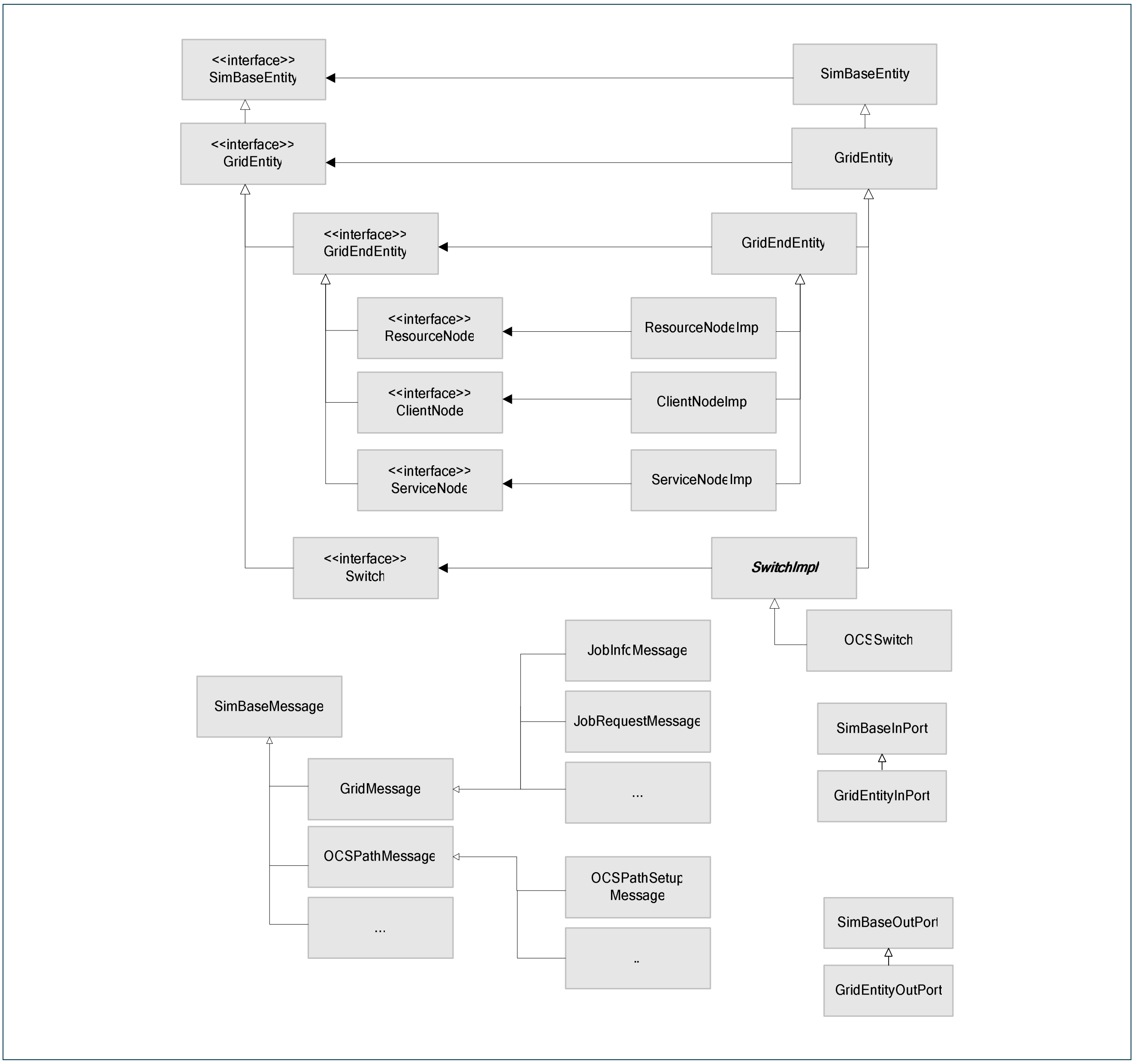


Figure 2 – Grid specific classes, derived from the basic discrete event simulator classes

2 Code

2.1 Package simbase

The simbase package is the core of the simulator. It contains 15 classes, which implement a generic discrete event simulator.

- The Time class is used to represent all measurements of time (both absolute and relative).

- The SimBaseSimulator class contains the logic to keep track of a sorted event list, to run these events, and to keep track of statistics gathered during the simulation run. All entities present in a simulation have to register with an instance of this class (there will be one such instance per simulation run).

- Entities in the simulator (such as Switches, Clients, etc.) are all represented by objects of the SimBaseEntityImpl class. SimBaseEntityImpl objects that need to be able to communicate are linked with ports: SimBaseOutPort on the sending end for outgoing messages, and SimBaseInPort on the receiving end for incoming messages. Thus, bidirectional communication will require two sets of ports. Note that each SimBaseOutPort is linked to one, unique SimBaseInPort. These links are represented by the SimBaseLink class (which is used by the SimBaseSimulator to keep track of the links between the various entities in the simulation).

- Sending messages from one SimBaseEntityImpl to another is handled by SimBaseEvent objects. When one entity wishes to send a message to another entity, this message (derived from the abstract class SimBaseMessage) is packaged into a SimBaseEvent, which is subsequently given a time stamp (the moment of arrival at the destination), and handed over to the SimBaseSimulator, which will add the event to its list.

- Statistics about the simulation are collected by the StatProviders, while the Logger provides a logging facility.

- To make creating a simulation easier a basic simulation has been provided by means of the SimulationInstance class. This is still an abstract class since the network (or traffic pattern, parameters, etc) will differ for each simulation and therefor must be provided by the programmer. To create a simulation, subclass SimulationInstance, provide a network and just call the run() method. The Printer will output the statistics gathered during the simulation.

The detailed class list, with relevant fields and method descriptions can be found in the javadoc folder on the SVN server (http://phosphorus.atlantis.ugent.be/phosphorus).

2.2 Package dist

The dist package contains classes used for generating random numbers from a given distribution. The basic generator used by all the distributions is a MersenneTwister object. For more information on this engine, see http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/random/engine/MersenneTwister.html. The MersenneTwister is part of the COLT library, see [2].

The base abstract class of the dist package, which is used by other objects in the simulator whenever a random distribution is needed, is DiscreteDistribution. This class has the abstract methods sample() for generating a long sample and sampleDouble() for generating a double sample that are called whenever a random number of the distribution is needed. Some of the implemented classes are DDNegExp (negative exponential distribution, for creating Poisson processes), DDNormal (normal distribution with a given average and standard deviation), DDUniform (uniform distribution), etc.

Any class derived from DiscreteDistribution, implementing any desired distribution, can be plugged into the simulator wherever a random number generator is required.

The detailed class list, with relevant fields and method descriptions can be found in the javadoc folder on the SVN server.

2.3 Package grid

This package contains classes derived from the basic classes of the simbase package. Some of these classes represent Client, Resource, Service, Switch, and other nodes.

This package has some subpackages:

- **grid.interfaces**: contains the interfaces for the different node types
- **grid.jobs**: contains classes for handling resources, applications, client states, and other job related info
- **grid.messages**: contains the different messages one can send during the simulation
- **grid.nodes**: contains the implementation of the different node types
- **grid.scheduler**: contains the job scheduling algorithms
- **grid.senders**: contains the different sending algorithms (Electrical, OCS, etc)

Derived from the SimBaseEntity(Impl) is the GridEntity(Impl) class, which is basically a network-aware version of the SimBaseEntity(Impl) (routing information is stored in this object), containing a RoutingMap object, filled with RoutingEntry instances. The RoutingMap contains functionality to determine the outgoing port to be used when another GridEntity(Impl) is to be reached. Each GridEntity(Impl) object thus has access to its own RoutingMap. These RoutingMap objects can be filled during initialization, or filled and modified on the fly, while the simulation is running. A Route object contains an entire route, either as it's being followed by a message, or as a "path to follow".

In turn derived from the GridEntity(Impl) class are Switch(Impl) and GridEndEntity(Impl). A Switch(Impl) object obviously represents a switch node in the network, and will implement functionality related to the optical network. Each GridEndEntity(Impl) objects is connected to a switch, assuming a non-optical link between the two. The different GridEndEntity(Impl) types are ClientNode(Impl), ResourceNode(Impl) and ServiceNode(Impl) (the package also contains a generic EndNode for testing purposes). The ClientNode(Impl) represents a client in the grid, and implements functionality for generating jobs, using a given job generation pattern (see the grid.jobs package description). The ResourceNode(Impl) represents a storage or computational resource, capable of executing tasks[1]. A queue model is present in the ResourceNode (using the QueuedJob class also present in this package). See the grid.jobs package description for further information. A ResourceInfo object contains the resource's specifications, and is used to register that resource with a ServiceNode. This ServiceNode(Impl) acts as a broker, assigning tasks submitted by clients to a certain resource, and, if needed, reserving a path. It keeps track of resource status and can keep track of link status.

Both the Switch(Impl) and the GridEndEntity(Impl) objects implement an advanced send routine, containing the logic for sending out data (in the form of messages) on their outgoing ports. Port availability and reservations, wavelength reservations and the existence of optical paths are checked and maintained in these functions. When implementing different routing and path setup algorithms, it might be needed to change the functionality of these functions. However, the actual routing algorithm is implemented in the receive routine, where new functionality should be added. The send routines are merely used to actually send a message on the outgoing port chosen earlier in the routine receiving the incoming message.

Other objects in the grid package are GridEntityInPort and GridEntityOutPort, network-aware versions of SimBaseInPort and SimBaseOutPort. Each port pair represents either an optical link between two Switch objects, or a classical link between a Switch and a GridEndEntity. The GridEntityOutPort class is responsible for maintaining information on the wavelengths present in the link, and their status. Each wavelength is represented by a Wavelength object.

The detailed class list, with relevant fields and method descriptions can be found in the javadoc folder on the SVN server.

2.3.1 Scheduling Algorithms

This ServiceNode(Impl) acts as a broker that assigns tasks to resources using either an online or an offline algorithm. Online algorithms assign a task to a resource immediately upon its arrival, while offline algorithms wait for a period of time so that several tasks are accumulated at the broker, before taking the task-to-resource assignment decisions. The algorithms of the latter type usually consist of two phases: the task-ordering phase and the resource-assignment phase. In the first phase the order in which the tasks are processed for

[1] The terms "job" and "task" are interchangeably used in the deliverable, except if otherwise stated.

assignment is determined. In the resource-assignment phase the resource where a task will be assigned is selected. Online algorithms can be considered as special cases of offline algorithms, where the task-ordering phase uses the First Come First Served queuing discipline. Distributed scheduling schemes usually follow a one phase procedure (online algorithms), while centralized scheduling schemes are employed in one or two phases (online or offline algorithms).

The algorithm that the scheduler will use is selected during the `ServiceNodeImpl` object's creation:

```
ServiceNode broker = ServiceNodeImpl.getInstance("broker1", simulator,
ordering, assignment);
```

Where the `ordering` and `assignment` variables indicate which algorithms will be used during the task-ordering and the resource-assignment phase. All the possible values of these variables are defined in the `grid.scheduler.SchedulingAlgorithms` interface. Also, the variable `timeWindow` in the `ServiceNodeImpl` defines the time period for the execution of the two phase task scheduling procedure. If this variable is set to zero, then the task-ordering phase has no meaning and the algorithm becomes an online algorithm.

Any new task-ordering algorithm must implement the `grid.scheduler.OrderingScheduler` interface and the following methods: `queueJobRequeust`, `deQueueJobRequeust`, `queueSize.` We have implemented the following task-ordering algorihtms:

- First In First Out - FIFO (`FifoOrderingScheduler`): Tasks are ordered based on the FIFO queuing policy.

- Weight Fair Queuing - WFQ (`WfqOrderingScheduler`): This task-ordering scheduler was first presented and evaluated in Deliverable 5.2 [3] and provides user-fairness. During a period, tasks belonging to different users arrive at the broker and are handled by a Weighted Fair Queuing (WFQ) scheduler [4]. The WFQ scheduler places these tasks in different queues based on their originating users. When a period expires the tasks are dequeued from the WFQ scheduler and sent to the resource-assignment scheduler.

Any new resource-assignment algorithm must implement the `grid.scheduler.JobScheduler` interface and the following methods: `findResource`, `findDataProvider`, `findDataStorage`, `addResource`. We have implemented the following resource-assignment algorithms, by altering only the `findResource` function and using the same implementations for the rest:

- LEASTBUSY (`DefaultJobScheduler`): The default resource-assignment algorithm, where tasks are assigned to the least busy resources.

- Fair Completion Time Estimation - FETE (FeteJobScheduler): This resource-assignment scheduler was first presented and evaluated in Deliverable 5.2 [3] and provides task-fairness. Tasks are assigned to resources according to their estimated fair completion times.

- Earliest Completion Time - ECT (EctJobScheduler): The ECT algorithm assigns a task to the resource that (at least in theory) will complete it faster than the other resources.

2.4 Package grid.messages

The grid.messages package contains all SimBaseMessage derivates used by the grid simulator. Below are several flowcharts to indicate how these messages normally interact and follow up with each other.

2.4.1 Initialization

The initialization process consists of registering all ResourceNode objects with the ServiceNode, which is accomplished by sending a RegisterResourceMessage (Figure 3). This happens automatically when the register() function is called (see further) on the resource.

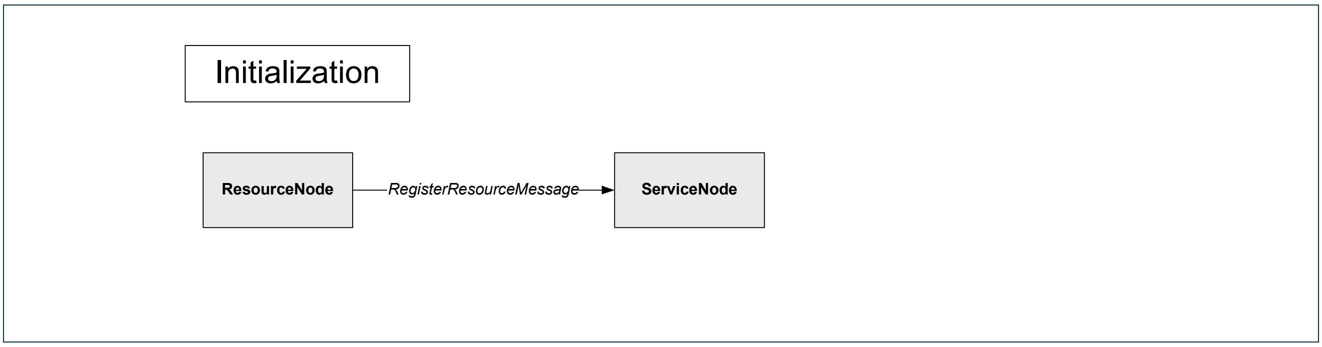


Figure 3 – Initialization process

2.4.2 Job Process

The job process, depicted in Figure 4, starts when the client receives a GeneratorMessage (which is first sent when calling init()). When the GeneratorMessage is received, the ClientNode will determine the time when the next job should be generated (using the interarrival time distribution); and will send itself a new GeneratorMessage, scheduled to arrive at that time. This keeps the job generation process running.

Next, a job request will be created, using the parameters from the current ClientState. This job is packaged into a JobRequestMessage, which is forwarded to the ServiceNode. Upon receipt, the ServiceNode will run its scheduler routine (Section 2.3.1) and appoint a ResourceNode for execution, and when needed, an input data source ResourceNode and an output data destination ResourceNode. It will also initiate any required OCS paths (not shown on the figure; see below). A JobReqAckMessage is then returned to the ClientNode. The ClientNode will then send out the actual job, including any data. Unlike the JobRequestMessage, which is a control plane message without attached data, the JobInfoMessage contains the actual application, the data provided by the client, etc.

The ResourceNode will receive this JobInfoMessage and, if needed, will send out a DataRequestMessage to the resource providing the input data. After the DataReplyMessage has returned with the data (or if no outside data was needed), the ResourceNode will schedule execution of the job, and calculate the time when the job will be ready. A JobCompletedMessage is used to mark this event. Finally, if needed, results are stored on a ResourceNode (the DataStoreMessage), and a JobResultMessage is returned to the client, to confirm successful execution of his job (and to send back the results, again, if required).

There is still one message type that has not been listed in Figure 4: BufferedJobInfoMessage. This message is used when the client sends a JobInfoMessage to the switch it is connected to. Since the link between the switch and the client is a non-optical link the switch will convert the electrical signal into an optical signal. In this simulator the signal is converted after it has been completely received. Upon arrival of the first byte of the JobInfoMessage the switch will send a BufferedJobInfoMessage to itself with the timestamp of the arrival of the last byte of the JobInfoMessage. This way the switch will know when the message has been received and it can start converting the signal from electrical to optical.

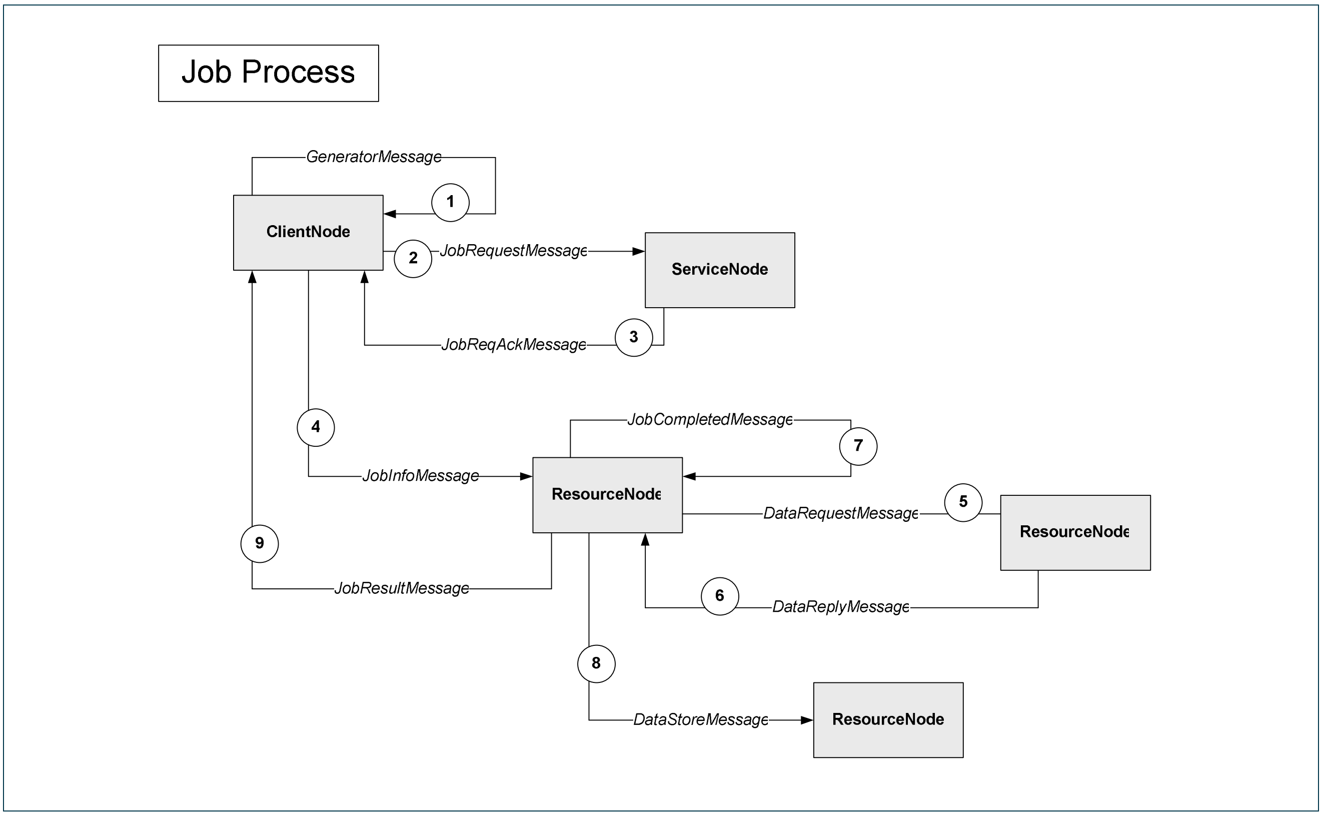


Figure 4 – Job process

2.4.3 OCS Path Setup and Teardown

The OCS path setup must be manually initiated, either by the routing algorithm in the `Switch` objects, or by other `GridEntity` objects, such as the `ServiceNode`.

An OCS path is initiated (Figure 5) by sending out an `OCSPathRequestMessage`, which will be forwarded towards the first switch on the OCS path. Any switches visited on the way (the “not on OCS path” switch on the figure) will simply forward this message, just like any other grid message. When the `OCSPathRequestMessage` arrives at the switch where the requested path begins, all available wavelengths will be determined, and a `OCSPathSetupMessage` will be forwarded to the next hop on the OCS path. This process repeats itself, culminating when the final switch in the path is reached (or when no available wavelength path is found). This final switch will then set up reservations by sending back an `OCSPathAckMessage`, which is forwarded until it reaches the first switch on the path, at each hop making appropriate wavelength reservations.

OCS path teardown is done in a analogous manner: an `OCSPathTearDown` message is sent out and forwarded, freeing up the wavelength reservations at each hop on the path.

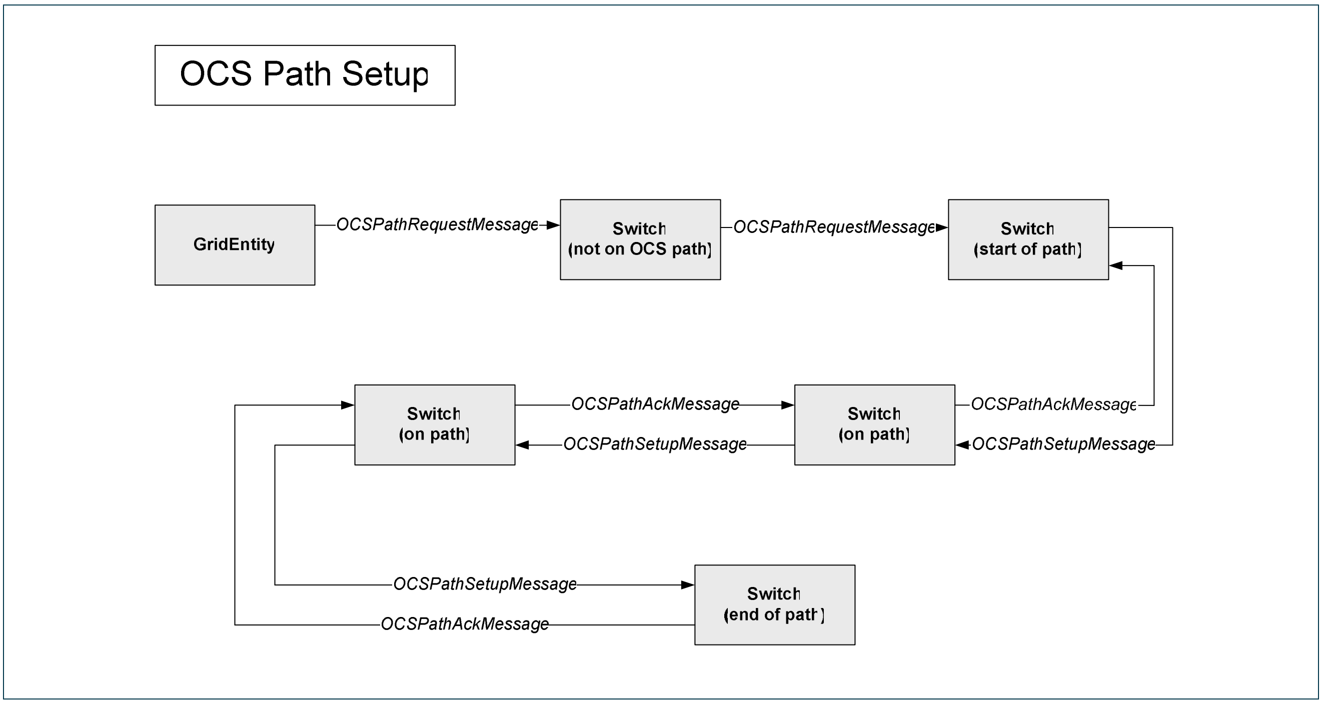


Figure 5 – OCS path setup process

2.5 Package grid.jobs

The `grid.jobs` package implements the job model used for the simulator. Each client is assumed to generate jobs of a given type, with an interarrival time following a given distribution (`DiscreteDistribution`) while in a certain state. Clients may switch to different states on several triggered events, one of which is the time it has spent in its current state. This implements a Markov model (Figure 6).

After having spent an amount of time in a given state, the next state is determined by using a preset transition likelihood. Other triggers for state transitions may be incoming job results etc. When a client is in a given state, it will generate jobs following a job model specific to that state; each state will obviously have its own job model.

This model, for each job type includes:

- job interarrival time
- data size: the size of the input data for the job
- data source, from where to fetch the data: use local data, use data packaged in the job (and thus sent out with the job itself), use data from a certain dataset which needs to be fetched from the grid network
- QoS requirements

- application requirements (e.g. Java vs Perl platform applications, where not every resource in the network supports both platforms)
- job execution time

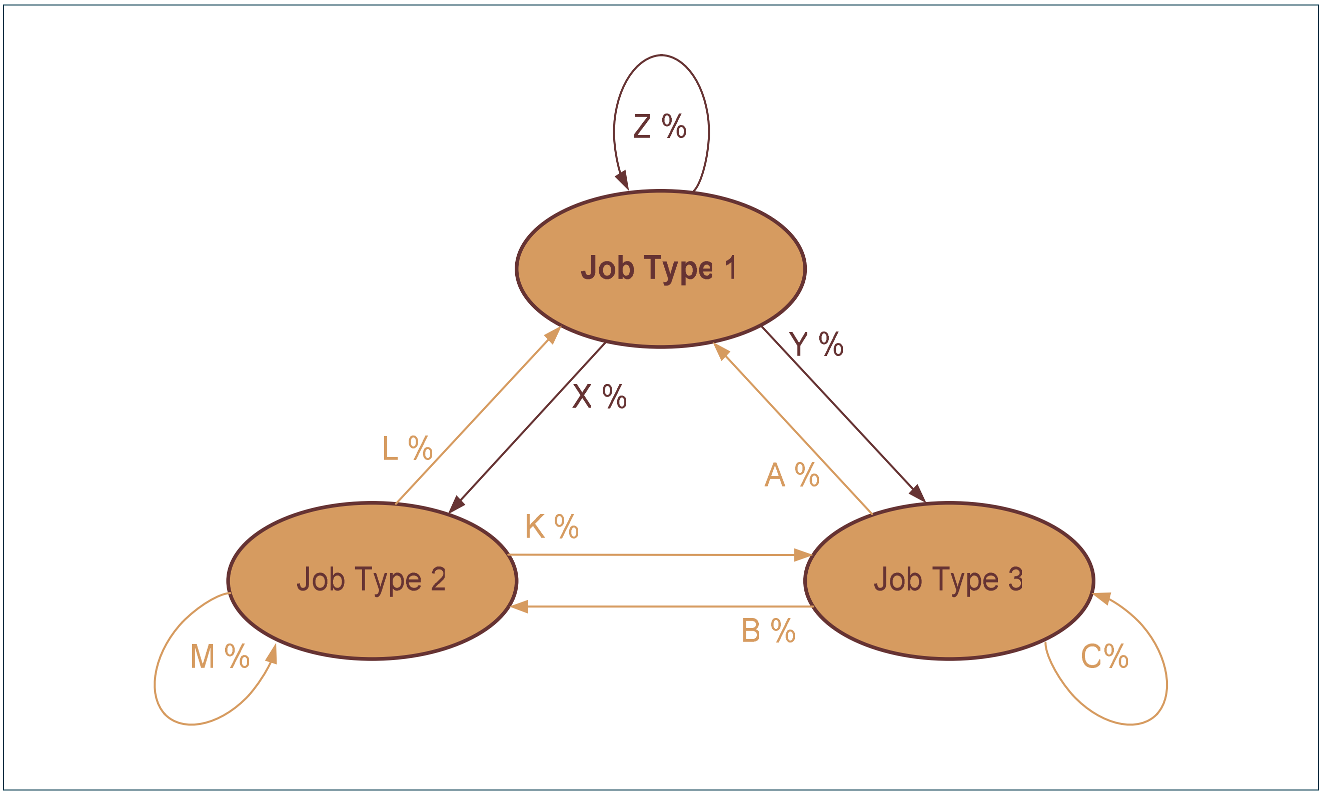


Figure 6 – Markov state job model with state transitions

The different classes in this package are used to define this job model. No specific simulation functionality is contained in these classes.

The Application, QOSClass and DataSet classes are used to represent their namesakes.

The `xyzChance` classes are wrappers used to store the likelihood of a given xyz to be chosen. For example: jobs of a certain job type have a 30% chance to need a certain application1 and 70% chance to need an application2. These values are stored in the `applicationChance` classes.

The `ClientState` class finally, contains all information of the client state, including time-related state transitions and the job model. It has the function generateJob(), which is used to generate a job request, and it is called by the `ClientNode`.

3 Tutorial

In order to use the simulator, it is imperative that the user implements his scheduling and routing algorithms. This can be done by changing the basic algorithms which are present in the `Switch` and `ServiceNode` objects. Refer to Section 2 and to the actual code.

Running a simulation requires an input, the grid network and its variables, and will yield an output, such as the job acceptance rate for the given scenario.

For example, we wish to run a simple simulation for the following grid network (see Figure 7) that contains two users, a resource broker, a resource, and two switches.

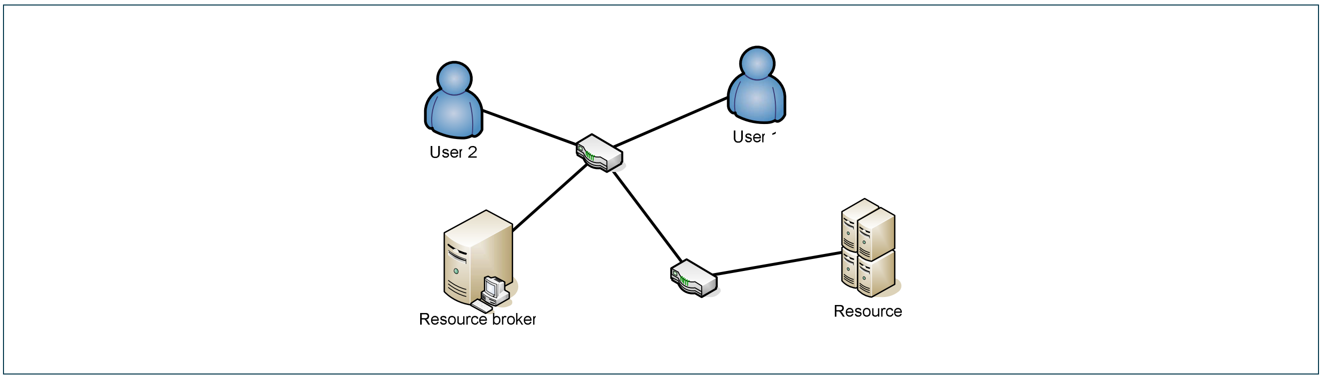


Figure 7 – Simple Grid network

We start out by creating a subclass of the SimulationInstance and denote it by the name MySimulation. Since SimulationInstance is an abstract class at least two methods will have to be implemented: configure() and createNetwork(). In the configure() method some parameters of the simulation have to be specified, such as the number of wavelengths per link, the CPU count for the resources, the job duration, etc. These parameters can be configured manually,

```
Configuration.defaultWavelengths = 8;
Configuration.defaultCPUCount = 1;
Configuration.defaultJobFlopSize= GIGA;
```

or can be loaded from a configuration file, that is a simple Java properties file.

```
PropertyLoader.loadProperties("defaults.cfg");
```

For the createNetwork() method two options are available: either create the entire network programmatically or use the GUI to create the network and then load it into the simulation. The latter is the easy one, needing only two lines of code:

```
Simulation sim = Simulation.read(filename);
simulator = SimulationLoader.loadSimulation(sim);
```

The variable filename holds the name of the file containing the simulation saved by the GUI.

Creating the network in code requires a lot of work, but we provide some useful tools to create nodes and links a lot easier. The first thing to do is to create the simulator:

```
simulator = new SimBaseSimulator();
```

Next we create an application that will run on the grid:

```
Application app = new Application(“application1”);
```

and the required nodes.

```
ServiceNode service =  ServiceNodeImpl.getInstance(“broker”,simulator);
```

This will create a service node with the default parameters. If non-default parameters are required, then one can use other getInstance(…) methods. The first node to create is the service node (broker) because both the client nodes and the resource nodes will require it.

Following, we create the client and resource nodes:

```
ClientNode client1 = ClientNodeImpl.getInstance("client1", simulator,
app, broker);
ClientNode client2 = ClientNodeImpl.getInstance("client2", simulator,
app, broker);
ResourceNode res = ResourceNodeImpl.getInstance("resource", simulator,
app, broker);
```

For setting up the network infrastructure, we create a number of switches:

```
Switch sw1 = OCSSwitch.getInstance("switch1", simulator);
Switch sw2 = OCSSwitch.getInstance("switch2", simulator);
```

and connect the nodes with links, using the createLink() method from GridUtilities. In the createLink() method a number of options (e.g., number of wavelengths) can be specified.

```
//links (optical)
GridUtilities.createLink(sw1, sw2);
GridUtilities.createLink(sw1, sw3);

//links (non-optical)
GridUtilities.createLink(sw1, client1,0,5,0);
GridUtilities.createLink(sw1, client2,0,5,0);
GridUtilities.createLink(sw1, broker,0,5,0);
GridUtilities.createLink(sw2, res,0,5,0);
```

Now our simulation is ready and we can start it by creating an instance and run it:

```
new MySimulation().run(new Time(GIGA));
```

The time argument is not required and if not supplied the default run time will be used.

The simulation we just created is a very simple one: every client node has the same configuration, the same parameters and only one state.

Generally, a number of methods in SimulationInstance can be overridden to provide more flexibility. We provide a list of the methods used in the simulations in the order of their execution:

- `configure():` the parameters configuration
- `createNetwork():` the network creation.
- `fillRoutingTables():` the routing algorithm for filling up the routing tables in the network. By default the Dijkstra algorithm is used
- `registerEntities():` everything that needs to happen before the simulation is started, are specified in this method. For example the transmission of the registration messages from the resources to the service node (broker). Note that the timers should not be started in this method.
- `initEntities():` in this method initialization messages are sent to all entities. Also, various timers can be started here
- `run():` run simulation
- `printInformation():` prints the desired information and statistics gathered during the simulation. By default all statistics are printed in one line separated by tabs. This default behaviour can be overridden using a Printer object.

Two examples are provided in the simulator package: the GUISimulation (using the simulation file created by the GUI) and the NoGUISimulation where the network is created programmatically.

4 Graphical User Interface

As mentioned in Section 3, setting up a new simulation requires the user to adapt the main function of the simulator. This means writing Java-code to create and initialize all the nodes and links in the network as well as setting all their parameters. We realize this is not very user-friendly and can be error-prone. To make things easier, a Graphical User Interface (GUI) for the simulator has been developed. The next paragraphs will discuss the GUI in more detail.

4.1 Simulation

The GUI will be used for creating simulations. Every simulation will contain a network (containing all nodes, delays and link speeds) and a list of applications and datasets on the resource nodes of the network. A simulation can be split up into two or more parts. For example when examining the grid behaviour in function of client the configuration, one may want to create multiple simulation parts, each having a different client state diagram. This way related simulation parts are bundled into one simulation. Simulations are stored in XML-format to make the exchange easier. A simulation part contains the list of all the available client nodes in the network. For every client node a state graph (with the configuration of every state) is given as well as the initial state. Note that when loading the simulation only the first part will be executed, if all parts have to be executed the load routine will have to be adapted.

4.2 Design of the GUI

The GUI (see Figure 8) consists of six major parts: simulation properties, network topology, resources, part configuration, state graph and state properties. We will discuss each part in the next paragraphs.

4.2.1 Simulation Properties

The simulation properties contain the name for the simulation, the file in which the simulation is stored, the file in which the network is stored and the list of simulation parts. The user can alter the name of the simulation and add/delete simulation parts. Note that the network file has to be entered on the creation of the simulation and cannot be changed afterwards, since further configuration depends on the unique ID's of the network nodes.

4.2.2 Network Topology

The network topology contains the network layout as well as the configuration of every node and link of the network. Since you cannot change the network used by the simulation the user will have to create the network before creating the simulation. We implemented a NetworkEditor based on JGraph Free (see www.jgraph.com). This editor allows the user to create, edit and manage network configurations.

4.2.3 Resources

This part of the GUI contains the list of resources (for now only applications and datasets) available in the network. Every resource is displayed as <resourceNode>-<resourceName>, where resourceNode is the node offering the given resource. Note that this list is common for all simulation parts.

While the previous paragraphs described the properties common for all simulation parts, the next paragraphs will handle the simulation part-specific configuration.

4.2.4 Part Configuration

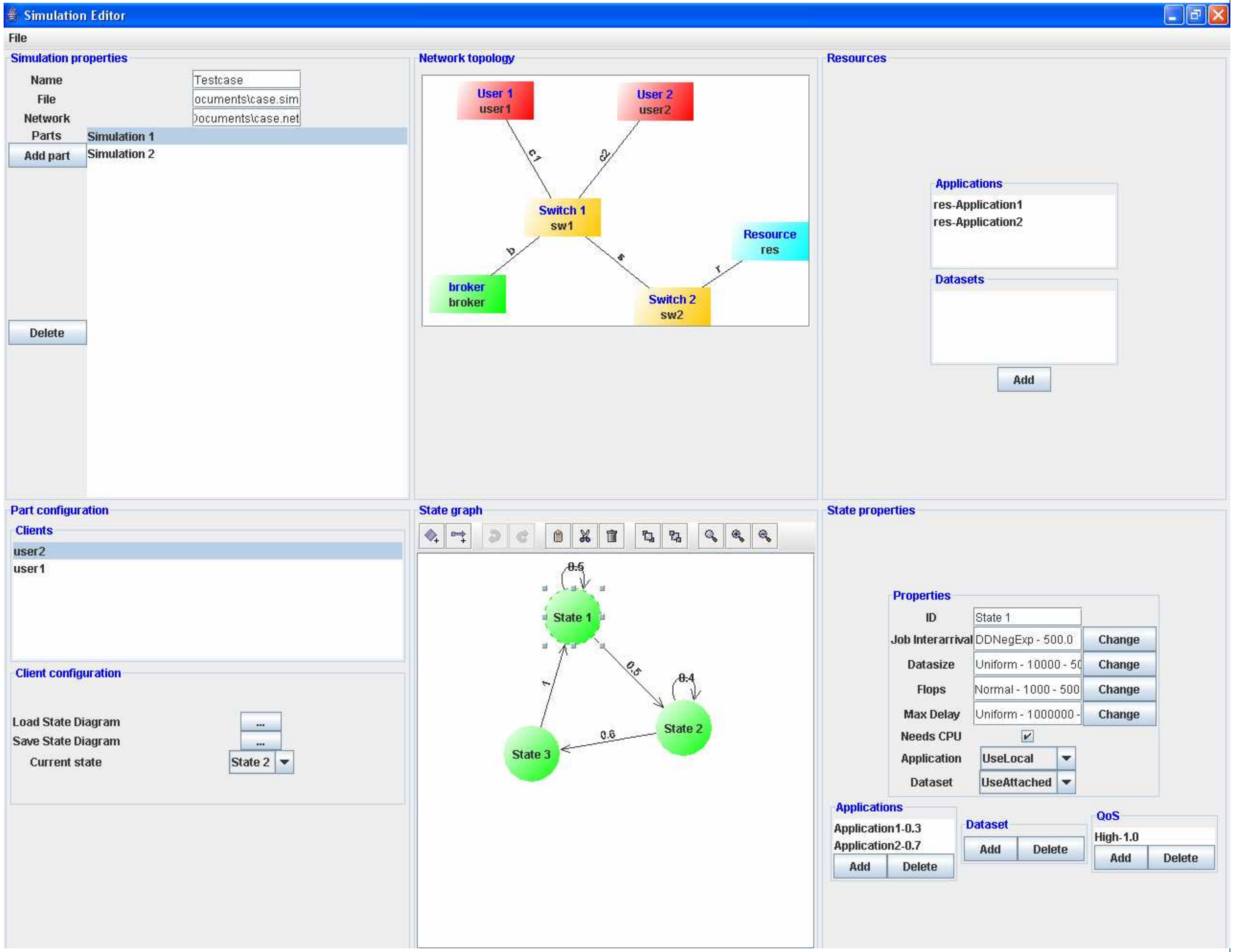
The part configuration contains the list of clients in the network. By clicking on a client in this list the corresponding stategraph and initial state are displayed. The user also has the possibility to load/save state diagrams for reuse.

4.2.5 State Graph

This is an editor for state graphs which allows the user to create, modify and delete states and state transitions. Each state transition has a given transition probability and every state has a unique ID. When clicking on a state the corresponding state properties are shown.

4.2.6 State Properties

Here the user can change the different properties of the state (see Figure 8), add applications, datasets, QoS's and their probabilities.



Simulation Editor

File

Simulation properties

Name: Testcase
File: ocuments\case.sim
Network:)ocuments\case.net
Parts: Simulation 1, Simulation 2

Part configuration

Clients: user2, user1

Client configuration

Load State Diagram ...
Save State Diagram ...
Current state: State 2

Resources

Applications: res-Application1, res-Application2

Datasets

State properties

Properties

ID: State 1
Job Interarrival: DDNegExp - 500.0
Datasize: Uniform - 10000 - 50
Flops: Normal - 1000 - 500
Max Delay: Uniform - 1000000 -
Needs CPU: ☑
Application: UseLocal
Dataset: UseAttached

Applications: Application1-0.3, Application2-0.7

Dataset

QoS: High-1.0

Figure 8 – The Graphical User Interface

5 Extending GridSim

GridSim is a comprehensive toolkit for simulation of various heterogeneous resources (whether single or multiprocessor, shared or distributed memory machines), users and applications [5]. It can also be used to simulate application schedulers for single or multiple administrative domain distributed computing systems such as clusters and Grids.

Application schedulers in multiple administrative domains like Grids, called resource brokers, perform resource discovery, selection, aggregation, and status retrieval of a diverse set of geographically distributed resources for an individual user. Each user has his own private resource broker that can be targeted to operate according to his requirements. On the contrary, schedulers in single administrative domains like clusters, have complete control over the allocation of resources. This means that all users need to submit their jobs to the central scheduler, which can be targeted to perform global optimization based on criteria such as higher system utilization, overall user satisfaction, optimization for high-priority users, and so on. In other words, schedulers in clusters focus on enhancing overall system performance, while schedulers in Grids focus on enhancing performance of a specific application in such a way that its users' requirements are met.

The GridSim toolkit provides tools for the modelling and the simulation of not only resources, applications, and users, but also of network connecting devices (links and routers) with different capabilities and configurations. It also supports application composition, resource discovery, task assignment, and resource management. These features can be used to simulate and evaluate the performance of various scheduling algorithms. The developers of GridSim have also incorporated a Nimrod-G resource broker for the evaluation of deadline and budget constrained scheduling algorithms with cost and time optimizations.

However GridSim contains only the simplest of the operations that real users and schedulers perform. First of all, user entities create all their Gridlets (Gridlets are objects that represent real Grid tasks[2]) at the beginning of their operation. This is opposite to what happens in a real Grid, where a user can create a new task at any time. In addition, Nimrod-G is a user-centric scheduler. This means that each user sends his scheduling requests to his own broker, and therefore each scheduler tries to greedily satisfy the requirements of his own user, without caring about load balance or possible congestion in the infrastructure. Furthermore, Nimrod-G provides only three scheduling policies (optimization strategies); optimization of time, cost, or both. Generally, there are numerous scheduling policies to choose from, and each user should be able to choose the most appropriate for his application.

Taken into account the above, we decided to extend GridSim in order to implement a concise Grid simulator. Users should be able to create a new Gridlet whenever they want, and in addition they should be able to add deadlines to them. Also, system-centric schedulers should be available in the simulator. These schedulers receive requests by numerous registered users, perform the scheduling and resource assignment of their tasks, and in the end return the scheduling results back to the users. Finally, each user should be able to choose among various scheduling policies.

[2] The terms "job" and "task" are interchangeably used in the deliverable.

The extended GridSim simulator, presented in this section, was used in many of the experiments conducted in the deliverable D.5.2 [3].

5.1 GridSim Basics

5.1.1 Features

The basic features of GridSim are the following:

- It allows modeling of heterogeneous types of resources (PCs, workstations, SMPs and clusters).
- Resources can be modeled to operate under space- or time-shared mode. In time-shared mode, a single processor is able to execute more than one tasks at once, in a round-robin matter. Each task is given a share (usually fair) of the total computational power of the processor. In space-shared mode, a processor cannot execute more than one tasks at once; it must finish execution of a task before commencing execution of another one.
- A single resource can contain any number of discrete machines. Each machine within a resource can contain any number of processors (called Processing Elements - PEs). The processing power of each PE is expressed in terms of Million Instructions Per Second (MIPS) as per the Standard Performance Evaluation Corporation (SPEC) benchmark
- Resources can be located in any time zone.
- Weekends and holidays can be mapped depending on resource's local time to model local workload.
- Resources can be booked for advance reservation.
- Applications with different parallel application models can be simulated.
- Application subtasks can be heterogeneous; they can be CPU or data intensive.
- There is no limit on the number of tasks that can be submitted to a resource.
- Multiple users can submit tasks for execution simultaneously in the same resource. This helps in developing schedulers that can use different economic models for selecting services competitively.
- Network speed between resources can be specified. In fact, a whole network can be created that consists of users, resources, schedulers, links (with various transmission rates and propagation delays), and routers (with different switching policies).

- Simulation of both static and dynamic schedulers is supported.

- Statistics of all or selected operations can be recorded and they can be analyzed using statistics analysis methods.

5.1.2 Architecture

GridSim's architecture is implemented as a stack of layers (see Figure 9), each one providing an interface to the above layer. The first and bottom layer is concerned with the Java interface and runtime machinery, called Java Virtual Machine (JVM), whose implementation is available for both single and multiprocessor systems. The second layer contains the basic discrete-event simulation package called SimJava [6]. Recently, a distributed implementation of SimJava was also made available. The third layer consists of the basic GridSim toolkit. It is concerned with modelling and simulation of core Grid entities such as resources, information services, and so on, as well as with application modelling, resource allocation, statistics recording, and a framework for creating higher level entities. The fourth layer is concerned with the simulation of resource brokers and schedulers. The fifth and higher layer is concerned with application and resource modelling with different scenarios, using the services provided by the third and the fourth layers.

In more detail, SimJava is a discrete-event simulation package written in Java. Simulations in Simjava contain a number of entities, each one running in parallel in its own thread. Entities are represented by the class Sim_entity, which encapsulates all the functionality that should be available to entities in the simulation. To define an entity, or rather an entity type, a subclass of Sim_entity must be created. This subclass will contain the entity's desired behaviour in the body() method, which must be overridden in the subclass. GridSim supports a number of different entities including users, brokers, resources, information services, statistics recorder, and network I/O. Each individual entity is an instance of a subclass of the Sim_entity class. This means that each entity runs in parallel in its own thread. In addition, users and brokers extend the GridSim class and belong to the gridbroker package.

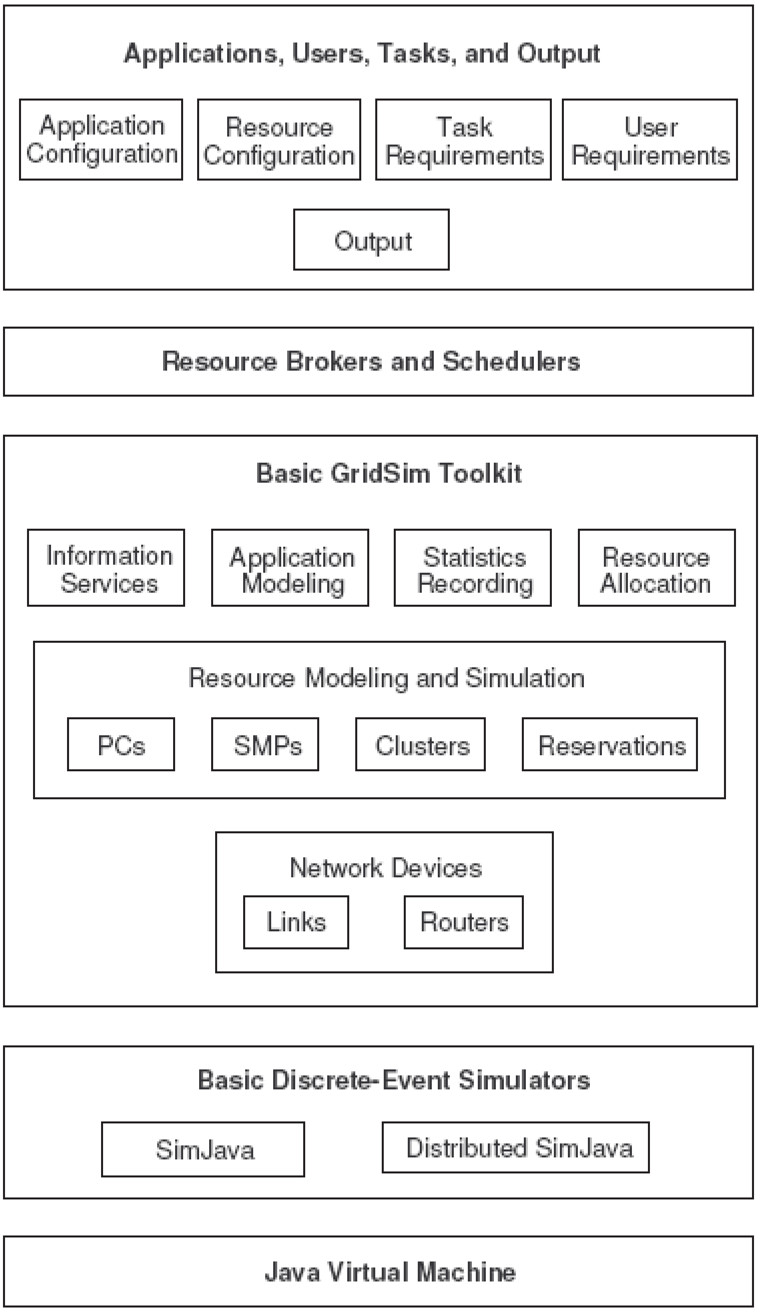


Figure 9 – Layered GridSim's architecture

5.1.3 Entities

5.1.3.1 *Users*

Each real Grid user is represented by an instance of the User entity. He is discriminated from the rest of the users according to the following requirements and characteristics:

- task properties (number of tasks, execution time of each task, etc.)
- scheduling optimization strategy (minimization of time, cost, or both)
- activity rate (how often he creates new tasks)
- time zone
- deadline or budget

5.1.3.2 *Brokers*

Each User entity is connected to an instance of the Broker entity. Each user task is first sent to the broker and then the broker submits the task to a resource for execution according to the user's optimization strategy. Prior to submitting the tasks, the broker gets the characteristics of all resources from the global directory entity. Because brokers try to fulfill the requirements of their own users, they may face extreme competition when trying to gain access to resources.

5.1.3.3 *Resources*

Each Grid resource is represented by an instance of the Resource entity. It is discriminated from the rest resources by the following characteristics:

- number of machines, each one consisting of a different number of processors called Processing Elements (PEs)
- speed of each processor (expressed as MIPS as per the SPEC benchmark)
- cost of processing
- internal allocation policy (space- or time- shared)
- local load factor
- time zone

- weekends and holidays mapping
- operating system (Linux, Solaris, Irix, etc.) and system architecture (Intel Pentium, Sun Ultra, SGI Origin, etc.)

5.1.3.4 *Grid Information Service*

Grid Information Service represents a global directory entity that keeps track of all resources of the Grid environment. Upon creation, each resource registers itself to the information service. When a broker needs information about a resource, he gets the resource's contact details from the information service and then contacts the resource for more info.

5.1.3.5 *Input and Output*

The flow of information between GridSim entities is carried out through their Input and Output entities. Note that these entities are subclasses of the Sim_entity class, which means that they run in parallel, each one in a different thread. As a result, GridSim entities are full-duplex. In addition, I/O entities have buffers, providing in this way a simple mechanism for the modeling of communication delays.

5.1.4 Applications

In GridSim, user tasks are represented by Gridlet objects. A Gridlet contains all the information related to a task, such as its file size (the size of the file that is sent from the user to the resource), its length (expressed in Million Instructions), and its output size (the size of the file that is returned from the resource to the user). The above parameters help in determining transmission and execution times. However, each Gridlet object carries various other auxiliary parameters, such as the Gridlet’s originator, the start and completion times of its execution, its current status, a string containing its history, and other information. GridSim supports a wide range of management tools that allow scheduling of a Gridlet to a resource and managing it throughout its life cycle.

5.1.5 Interactions between Entities

Since GridSim is an event-driven simulator, the interactions between entities are implemented using events (messages). These events can be raised by any entity to be delivered immediately or with a specific delay to itself or to other entities. Events that are destined to the same entity are called internal, while those which are destined to other entities are called external. Events can be further categorized into synchronous and asynchronous. If an event is synchronous, its source halts until the event is delivered to its destination. If an event is asynchronous, its source continues its operation without waiting for the event to be delivered to its destination. It is noted that internal events should be asynchronous in order to avoid undesired deadlocks.

The complete set of events between GridSim entities is shown in Figure 10. Entities send events to themselves or other entities in order to request for service, send data, deliver results, or raise internal events.

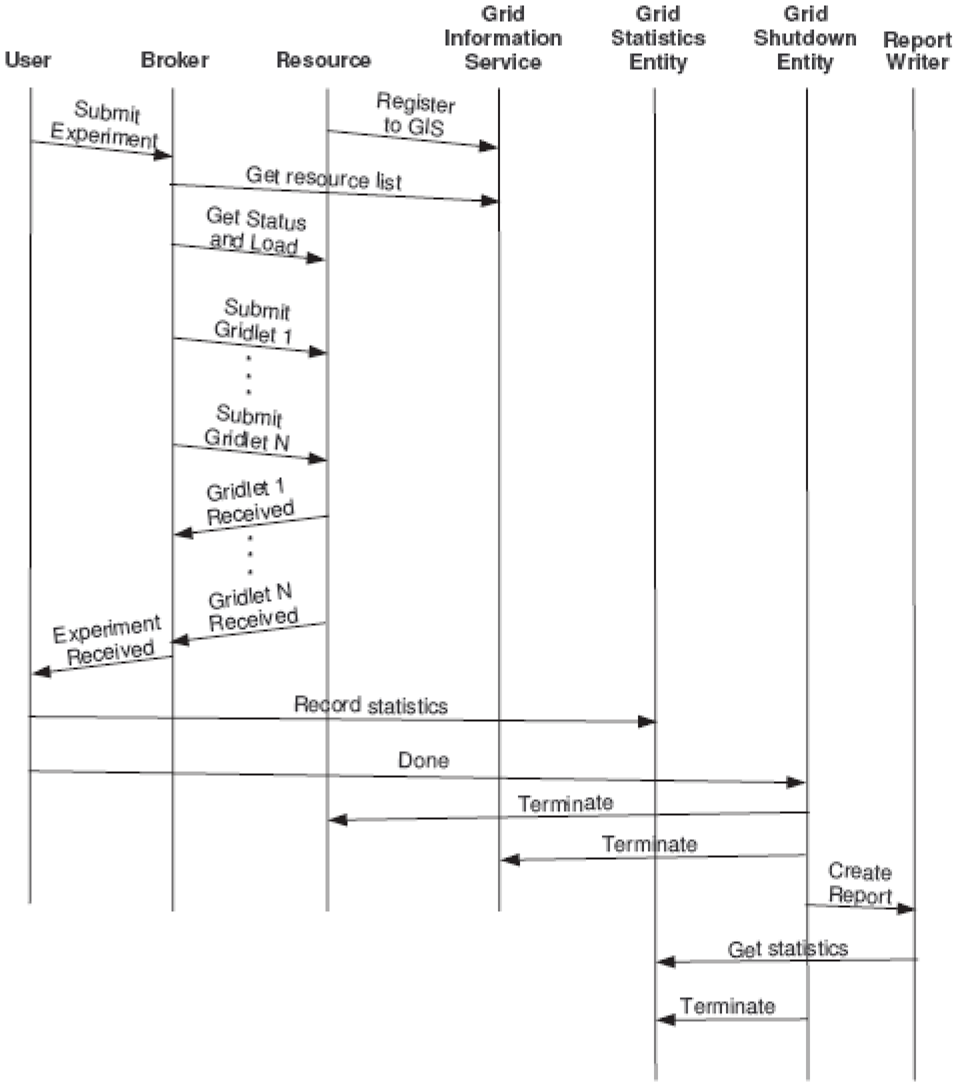


Figure 10 – Interactions between GridSim entities

When GridSim starts simulation, resources register themselves to the Grid Information Service (GIS) by sending an event to it. This registration process is similar to GRIS registering with GIIS in the Globus toolkit. Simultaneously, the users send their experiment (i.e. all their Gridlets) to their resource brokers. The brokers will assign the Gridlets to resources and submit them for execution.

The brokers send an event to the GIS requesting for the list of resources. The GIS replies back with the list of resources and their contact info. The brokers then may send an event to some resources requesting for their characteristics (number of machines and PEs in each machine, MIPS per PE, cost, stc.) and status (number of available or busy PEs, current load, etc.). The resources reply back and then the brokers are ready to send the Gridlets to them for execution. When a Gridlet arrives at a resource, it may begin execution immediately or be put in a waiting queue, depending on the resource allocation policy or current load. The resource will raise an internal event to imply the end of the Gridlet's execution. When the Gridlet finishes execution, the resource sends the Gridlet back to the corresponding broker.

When all Gridlets are finished and returned back to the originating user, then the user can send various interesting statistical data to the Grid Statistics Entity. After that he informs the Grid Shutdown entity that he has finished. When all users have finished, the Shutdown entity terminates the resources and the GIS, and finally requests from the Report Writer to create a report based on the data stored in the Statistics entity.

5.2 Modifications to the Original Code

5.2.1 Basic modifications

Initially, we made the following additions/modifications to the original GridSim code:

- Deadlines in Gridlets: We created a new Gridlet characteristic (apart from the length, the file size, the originator, etc.) that denotes the task's deadline. A task deadline can be critical or not. In general, if a critical deadline expires, the corresponding Gridlet is removed from the Grid, while if a non-critical deadline expires the Gridlet remains in the Grid and in the end the user can choose whether to keep the Gridlet's result or reject it as outdated.

- Resources' Lists Info: Another major addition to the GridSim code was the ability for the resources to send to other entities (mostly schedulers) the exact list of the Gridlets being executed in each PE, as well as the list of the queued Gridlets waiting for execution. Through this feature, a scheduler is able to get dynamically the exact load of each resource and perform a much more precise task assignment. These lists together with the information about the status and load of the resource (the number of machines and PEs within each machine, the number of busy or free PEs, the MIPS rating of each PE, etc.), are sent from the resource to the scheduler in a single message.

- File Logger: SimJava provides a tracer that records every event of the simulation. This tracer is much too detailed, its results are too many for the needs of a Grid environment and the overhead of such recording is extravagant. For this reason, we created our own file logger that records only the Gridlets' related data, as well as their starting and completion times. In addition, the implemented file logger provides some auxiliary functions for further analyzing the simulation results, such as counting how many Gridlets missed their deadline or what is the total makespan of the simulation.

- Gridlet Originator: The Gridlet class contains a private variable called userID_, which naturally denotes the entity to which the Gridlet will return after its execution at a resource. This entity can be anything, from the user who created the Gridlet, to the user-centric scheduler that has undertaken the dispatching and the gathering of the Gridlet. However, for statistical reasons, the resource needs to know the creator of the Gridlet. This is achieved by adding a new characteristic to the Gridlets that denotes their creator/originator.

5.2.2 New Entities

Our main contributions to the GridSim simulator were the new user and scheduler entities. The entities contain a concise set of operations and characteristics, in order to simulate the behavior of real Grid users and schedulers. We implemented two types of schedulers. The first type is the user-centric scheduler, which serves only one user. This type of scheduler is much like the Nimrod-G resource broker implemented by the creators of GridSim, except that our scheduler is much more precise and provides numerous scheduling policies for the user to choose from. The second type is the system-centric scheduler, which accepts Gridlet assignment requests from several users.

5.2.3 Users

Users create Gridlets that are submitted to resources for execution. These Gridlets, as we mentioned, have different needs in computational power and transmission rate. Users use schedulers (resource brokers) in order to determine the best resource for their Gridlets' execution. A user that is registered to a system-centric scheduler sends to it the characteristics of the Gridlets he creates, and the scheduler finds the best (according to a specific policy) assignment of Gridlets to resources. Next, the scheduler returns to each user the assignment decisions for his Gridlets. If a user feels that his needs cannot be satisfied by the scheduling policy of any system-centric scheduler, he can create his own user-centric scheduler that will interact exclusively with him. However, the main drawback of user-centric schedulers is that they try to greedily satisfy the requirements of their own users, without considering the needs of the remaining users and the possible overhead in the Grid. The larger the number of users in a Grid environment, the less accurate are the scheduling decisions of the user-centric schedulers.

For the needs of the simulation, each user entity specifies upon creation a number of Gridlets that he will create. These Gridlets, as already mentioned above, have several characteristics such as length (expressed in million instructions), size (expressed in Bytes), and deadline. The users also specify the relative deadline of their Gridlets. A relative deadline contains the length of the time interval that begins with the creation of the Gridlet and ends with its deadline expiration. If a user does not want to give a deadline to a Gridlet, he can set its relative deadline to -1. Furthermore, deadlines can be critical or non-critical. If a critical deadline expires, the corresponding Gridlet is removed from the Grid, while if a non-critical deadline expires the Gridlet remains in the Grid and in the end the user can choose whether to keep the Gridlet's result or reject it as outdated. Finally, users can specify their Gridlets creation rate.This way a user can create a new Gridlet at any time and not only in the beginning of the simulation, just as in a real Grid.

A user can define a number of different characteristics for his Gridlets: 1) length, 2) input data size, 3) output data size, 4) rate, 5) deadline. These characteristics can be defined in a probabilistic or deterministic manner. Specifically:

- Probabilistic Gridlet Characteristics: Users can specify the probability distributions for the characteristics of their Gridlets. Each characteristic can have a different probability distribution type. The distribution types supported are: uniform, Gaussian (normal), and fixed. Uniform distributions include a minimum and a maximum limit, between which the random value is chosen. Gaussian distributions include a mean and a deviation value, and the random number generator follows the normal probability distribution. Finally, fixed distributions produce only fixed values. This means that if a user sets a Gridlet characteristic to follow the fixed distribution, then the value of this characteristic will be the same for all the Gridlets he creates. Generally, probabilistic Gridlet characteristics are more suitable in simulations where the exact behaviour of the users is not known a priori, and the effectiveness of the infrastructure is tested for many different scenarios.

- Fixed Gridlet Characteristics: Apart from probabilistic characteristics, users can specify the exact values for the characteristics of the Gridlets they create. These values are declared using vectors, each corresponding to a characteristic; the *i*-th element of a vector contains the value of the corresponding characteristic for the *j*-th Gridlet. So, the vectors have the same size, equal to the number of Gridlets that the user will create. Fixed Gridlet characteristics are most suitable to situations where the behavior of users is more or less predictable.

5.2.4 Schedulers

The schedulers (resource brokers) are the entities that are responsible for assigning Gridlets to the most appropriate resources for execution. As we already mentioned above, we have implemented two types of schedulers, user-centric and system-centric. User-centric schedulers serve only one user, like the Nimrod-G resource broker implemented in the gridbroker package. On the contrary, system-centric schedulers accept Gridlet assignment requests from several users, perform the scheduling (assignment) of these Gridlets, and finally return the scheduling results to the users. The main drawback of user-centric schedulers is that they try to greedily satisfy the requirements of their own users, without caring about the needs of the other users and the possible overhead in the network. The larger the number of users in a Grid environment, the less efficient are the scheduling results of user-centric schedulers. Next, we describe the basic operational properties of the system-centric schedulers

System-centric schedulers begin their operation by acquiring the contact info of all resources from the GIS, and querying all resources for their characteristics (number of machines and PEs within each machine, internal allocation policy, etc.). After this step, they enter a loop of accepting requests and processing them, until an event arrives that denotes the end of the simulation. The types of event that arrive to a system-centric scheduler are the following:

- Initialization request: Whenever a new user is created, he registers to the system-centric scheduler. For this purpose the user sends a message to the scheduler requesting the permission to begin his operation. The scheduler replies back with an ACK and the user is ready to create his Gridlets.

- A Gridlet assignment request: Whenever a user that is registered to a system-centric scheduler creates a new Gridlet, he immediately sends the characteristics of the Gridlet to the scheduler in order for the latter to determine the best (according to the scheduling policy) resource for the Gridlet execution. When such a request arrives at the scheduler, it can process it immediately or wait for a ``window'' period during which the scheduler collects (gathers) assignment requests in order to process them as a batch. This way Gridlets are assinged to resources in a more efficient way. There are two types of gathering windows:

  - Time Window: A time interval is specified, during which the scheduler is in the state of receiving assignment requests in order to process them all together at the end of the interval.

  - Gridlet Window: Instead of a time interval, the maximum number of assignment requests gathered is specified. However, if no new requests have arrived after a time period, then the scheduler proceeds with the assignment of the Gridlet requests arrived until then.

  When the gathering window expires, the scheduler raises an internal event to process the arrived assignment requests. If the scheduler uses no gathering window, this event is raised immediately with the arrival of every new Gridlet assignment request.

- Process all arrived Gridlet assignment requests: The process of assigning Gridlets to resources passes through two discrete phaces; the queueing phase and the dispatching phace.

  - Queueing Phase: The scheduler takes the list of the arrived assignment requests and sorts it according to a specific queueing policy (see below). The sorted list is then used as input to the dispatching phase.

  - Dispatching Phase: The scheduler uses the sorted list of the Gridlet requests, and assigns each Gridlet to a resource. This resource is chosen according to a specific dispatching policy (see below).

5.2.4.1 *Queueing Policies*

The queueing policies supported are the following:

- First Come First Served (FCFS): The Gridlets keep the order with which they have arrived at the scheduler.

- Earliest Deadline First (EDF): The Gridlets are ordered according to their deadlines. The earlier a Gridlet's deadline expires, the higher is the order given to that request. The Gridlets with critical deadlines have the highest priority and are put in the top of the sorted list, followed by the Gridlets with non-critical deadlines and the ones with no deadlines at all

- Least Length First (LLF): The smaller the length of a Gridlet, the higher its order in the sorted list. This way, Gridlets that require large execution times do not delay other smaller Gridlets from being executed.

- Maximum Length First (MLF): The larger the length of a Gridlet, the higher its order in the sorted list.

5.2.4.2 *Dispatching Policies*

After all Gridlet assignment requests have been sorted, the dispatcher chooses the resource in which each Gridlet will be executed. The elements of the list are processed in a FCFS manner. The dispatching policies supported by the schedulers are based on some well-known heuristics and are the following:

- Earliest Start Time (EST): Each Gridlet is assigned to the resource where its execution will start earlier than in other resource. This policy is suitable for situations where Gridlets need to communicate with each other, and thus synchronization is important.

- Minimum Processing Time (MPT): Each Gridlet is assigned to the resource where its processing time will be smaller than in any other resource. This policy is suitable for situations where we want to minimize the processing costs of the Gridlets (the cost is analogous to the time the Gridlet uses the PE for processing).

- Earliest Completion Time (ECT): Each Gridlet is assigned to the resource where it will complete its execution earlier than in any other resource.

- Load Balance (LB): Let CT_1, CT_2, … ,CT_N the completion times "offered" to a Gridlet by the resources R_1, R_2, … , R_N respectively. Then the mean completion time of the Gridlet is:

$$\overline{CT} = \frac{CT_1 + CT_2 + ... + CT_N}{N} .$$

The Gridlet is assigned to the resource R_j that satisfies the following:

1. $CT_j \leq \overline{CT}$

2. $CT_j \geq CT_i, \forall i$ such that $i \neq j$ and $CT_i \leq \overline{CT}$

The above hold if the Gridlet has no defined deadline. If the Gridlet comes with a deadline, then from the set of *N* resources we use only the *K* $(K \leq N)$ that execute the Gridlet without missing its deadline. The index of these resources is denoted by *d* : {1, 2, … ,*K*} → {1, 2, …, *N*}, and their completion times are $CT_{d(1)}$, $CT_{d(2)}$, …, $CT_{d(K)}$ respectively. The mean completion time is then:

$$\overline{CT}_d = \frac{CT_{d(1)} + CT_{d(2)} + ... + CT_{d(K)}}{K},$$

and each Gridlet is assigned to the resource R_k that satisfies the following:

3. $k \in d\{1,2,...,K\}$

4. $CT_k \leq \overline{CT}_d$

5. $CT_k \geq CT_i, \forall i \in d\{1,2,...,K\}$ such that $i \neq k$ and $CT_i \leq \overline{CT}_d$.

- Simulated Annealing (SA): This is an iterative technique that aims at minimizing the makespan of all Gridlets in the sorted list. Initially, all Gridlets are randomly assigned to resources. At each step a new random assignment is produced. If the makespan of the new assignment is less than that of the old assignment, then the new assignment replaces the old one. If the new makespan is greater than the old one, and a random number $x \in [0,1]$ is greater than a certain limit *y*, the new assignment replaces the old one. This limit is determined as

$$y = \frac{1}{1 + e^{\frac{old\_makespan - new\_makespan}{temperature}}},$$

where temperature is a parameter of the simulated annealing algorithm. Initially, when temperature is very high, then *y* → 0.5, and assignments with less makespan have approximately 50% probability of getting accepted. However, as time passes and temperature drops (the system "cools"), then *y* → 1, and poorer assignments are more difficult to be accepted. When the makespan does not change for a specific number or steps, or when the temperature drops too low, the algorithm exits.

- Genetic Algorithm (GA): This last assignment technique is based on well-known genetic algorithms. Like the simulated annealing algorithm, it aims at minimizing the makespan of the Gridlets. Each chromosome represents an assignment of the Gridlets to resources. The chromosomes with the greatest fitness (i.e. the chromosomes offering the least makespan) are chosen in order to participate in the crossover and mutation processes. When the best chromosome does not change for a specific number of generations, the algorithm exits.

6 A Simulator for Examining Fault Tolerance in Grids

In this section we present a simulator that was developed in order to examine fault tolerance techniques in Grid Networks. The main characteristics of the simulator are the following:

- Simplicity of model: The model of simulation is simple and easily comprehensible, facilitating in this way the development of new algorithms.

- Facility of development: A new scheduling algorithm can be developed and altered with easiness and speed. Also, the simulator offers ready entities and tools that help in the design of experiments. For example, it provides an entity that creates tasks with concrete statistical characteristics.

- Parameterized experiments: The simulator provides the capability of executing a massive number of experiments that differ only in the values of certain parameters. For example, “execute this experiment for values of λ from 10 to 30”.

- Flexibility in metrics: The simulator includes multiple ready to use metrics and gives the facilities for the easy definition of new metrics.

- Results: The simulator stores the results of experiments in a form easily to process. Tools that process and combine these results are provided, as well as tools for their visualization (automatic production of graphic representations).

6.1 Simulator Model

In this section we present various entities that participate in a simulation.

6.1.1 Epoch

Epoch is the abstract time unit that we use in the simulation. It represents a run of the experiment. In each epoch resources or tasks arrive and depart and statistics are calculated.

6.1.2 Task

The task is a computing work that is scheduled in the Grid. We consider that tasks are non-pre-emptable. Thus, by the moment a task begins its execution in a node[3] (resource, machine, etc), it cannot be stopped. We also assume that tasks are indivisible and cannot be separated in simpler tasks. The basic characteristics of a task are the following:

- taskNumber: A serial number that uniquely identifies a task.
- initialWorkload: The initial computing workload of a task.
- workloadLeft: The workload that remains for computation. When the task is executed in a resource, this number is decreased according to the computation capacity of the resource. When this value reaches zero, then the task is considered as completed and the result of its execution have to be transmitted in the task's originating user.
- inputData: The input data the task will need during its execution. These data are transmitted to the resource where the task will be executed.
- outputData: After a task completes its execution in a resource, then its result (output) data are transmitted back to the task's originating user.
- dataLeftToTrasmit: The remaining task's related data that need to be transmitted over a link. In each epoch this value is decreased according to the link's bandwidth. When the task is ready for transmission, this parameter is initialized with the value of inputData, while when the task has completed in the resource this parameter is initialized with the value of outputData.
- desirableCompletionTime: The desirable completion time of a task or else its non-critical deadline. With the term "completion" we consider the computation of the task and the transmission of input and output data, from and to the originating user.
- absolutCompletionTime: The absolute completion time of a task or else its critical deadline.
- creationTime: The epoch at which the task is created.
- completionTime: The time at which the task is completed.
- nodeOfOrigin: The task's originating user.
- nodeRunningOn: The resource where the task is executed.
- nodeTransmitingFrom: The (storage) resource where the input data of the task is located.

[3] The terms "node" or "computing node" and "resource" are interchangeably used in this section.

- nodeTransmitingTo: The (storage) resource where the output data of the task is stored.
- history: The history of events that relate to the task.
- status: The status of the task.
- taskCategory: The category of the task.

6.1.3 TaskHistory

It is a class that represents a task event. The fields of TaskHistory are the following:

- time: The time (epoch) that the event happened.
- type: The type of the event,
    - CREATION
    - SCHEDULED
    - START_TRANSMIT
    - END_TRANSMIT
    - START_CALC
    - END_CALC
    - ORIGIN_NODE_DEPARTED
    - COMP_NODE_DEPARTED.

6.1.4 Node - Resource

The node - resource is the basic computing unit. Each resource represents either a unique personal computer that offers some percentage of its CPU in the Grid or an entire cluster of powerful computing systems. The resources are connected using a network; a link between two resources (NodeLink) is characterized by its bandwidth and the two nodes that it connects. We assume that each resource has an unlimited storage capacity as well as a network frontend, so a resource can receive, send data or execute a task at the same time. The characteristics of a resource are the following:

- nodeNumber: A serial number that uniquely identifies the resource.

- computationalCapacity: The computation capacity of the resource. In each epoch the remaining workload of an executed task is decreased according to the computationalCapacity parameter.

- creationTime: The epoch at which the resource was created.

- departureSchema: It is the entity that determines the time that the resource departs (fails, stops operating) from the Grid. This entity is used for simulating fault tolerance experimental scenarios.

- runningTask: The task that is currenlty executed in the resource.

- tasksToCompute: A list of tasks ready to be executed in the resource. When the runningTask completes its execution, then the resource will take the first task from this list and make it the runningTask. When a task's input data transmission to the resource has been completed, then it is introduced in the end of the list.

- tasksReceiving: A list of tasks in this resource that receive input data (not ready yet for execution)..

- tasksPending: A list of tasks that have been submitted in the resource and have not yet been completed.

- numberOfEpochsNotUsed: This number is increased by one for each epoch that the resource doesn't process a task.

- connectedNodes: The resources and the corresponding network links connected with this resource.

- nodeStatus: The status of the resource,

    - COMPUTING

    - RECEIVING

    - IDLE

- computatedTasks: The number of tasks that have completed their execution in the resource.

- toDepart: It is set to true when the departureSchema object decides that the resource should depart from the Grid.

- sharedCpu: When it is true, then time-sharing is used.

- weights: A HashMap that maintains the weights of the users in this resource.

- cpuhunger: This value represents the user's (node-user) task generation rate. This is the case where the user is collocated in the resource.

- requestedComputanionalCapacity: The total computational workload that the user has requested.

- userCategory: The category of the node-user.

6.1.5 Scheduler

It is the basic entity that performs the scheduling of tasks to resources. The corresponding classes should implement the Scheduler interface, e.g., implement the schedule() method that takes as input a list of tasks to be assigned and a list of on-line computing resources.

6.1.6 TaskProducer

It is the entity that creates tasks, initiates their characteristics and assigns them to the corresponding originating users. The corresponding classes should implement the generateTasks() method, which is called in every epoch and produces a list of new tasks. The implemented classes are the following:

- NormalTaskProducer: It produces tasks in time intervals that follow a Gaussian distribution. The task characteristics also follow Gaussian distributions.

- PoissonTaskProducer: It produces tasks in time intervals that follow a Poisson distribution. The task characteristics follow Gaussian distributions.

6.1.7 NodeProducer

The NodeProducer entity creates new computing nodes (resources), initializes their characteristics and interconnects them in the network. Different implementations of this entity give us the capability to study the behaviour of the proposed scheduling algorithms in heterogeneous environments. These corresponding classes should implement the NodeProducer interface. The implemented classes are the following:

- SimpleNodeProducer: It produces a new resource (node) in time intervals that follow a Gaussian distribution. The resources characteristics also follow Gaussian distributions.

- StaticNodeProducer: It produces a static number of nodes with static characteristics.

6.1.8 Grid

It is the entity that represents an experiment and it has the following members:

- taskProducer: The entity that creates new tasks.
- nodeProducer: The entity that creates new nodes.
- scheduler: The scheduling entity.
- statsAggregators: A list of entities that compute and collect statistics. Each of them computes only one metric.
- tasksWaitingToBeScheduled: A list of tasks that have to be scheduled.
- completedTasks: A list of tasks that have been completed.
- allTasks: A list of all tasks ever created.
- failedTasks: : A list of failed tasks.
- tasksInTheGrid: A list of tasks that are active in the Grid. These tasks have been submitted in the Grid, but they have not yet been processed.
- nodesOnline: A list of resource that are online and available to execute tasks.
- nodesDeparted: A list of resource that have been departed (failed, stop working) from the Grid.
- currentTime: The current epoch. It is accessible in any entity via the public method getCurrentTime(). It is increased by one unit in each iteration of the basic simulation loop, in the run() method.
- finishTime: The final epoch of the simulation. When currentTime becomes equal to finishTime, the simulation is terminated.

6.1.9 Simulation Cycle

In each epoch a simulation cycle is executed (Figure 11). Initially the nodeProducer entity adds a number of new resources in the system and the taskProducer creates new tasks. The created task list is handled by the scheduler, which tries to schedule the tasks to resources, based on the implemented scheduling policy. In the next step, the nextEpoch() method is called for each active resource, which executes the functions required. Finally, the failed tasks are detected and statistical data for the experiment are collected, by calling the aggregate() method. Epochs are increased by one, and if the maximum simulation time hasn't be reached, then the cycle is repeated.

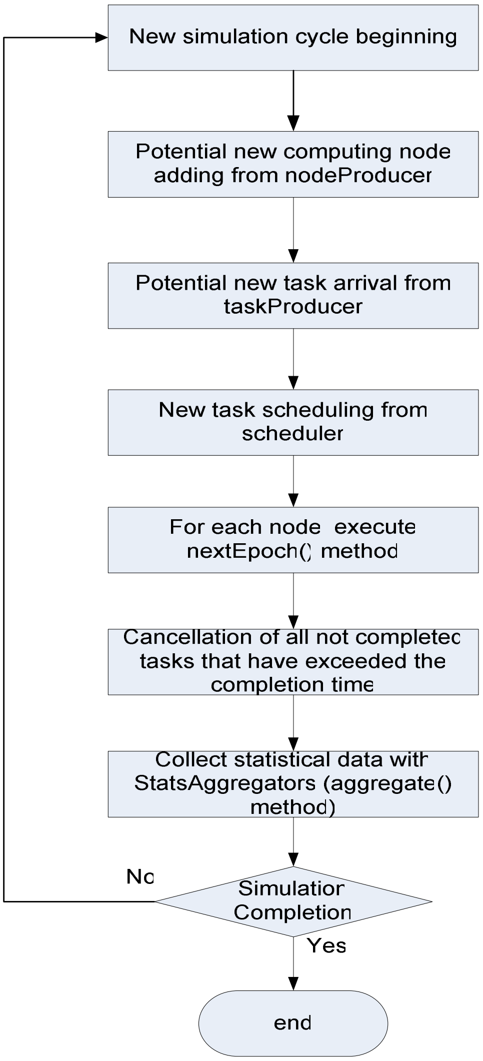

New simulation cycle beginning
Potential new computing node adding from nodeProducer
Potential new task arrival from taskProducer
New task scheduling from scheduler
For each node execute nextEpoch() method
Cancellation of all not completed tasks that have exceeded the completion time
Collect statistical data with StatsAggregators (aggregate() method)
No
Simulation Completion
Yes
end

Figure 11 – Cycle simulation flow chart

Figure 12 shows the simulation cycle of a resource and the functions that are executed when the nextEpoch() method is called. Initially the departureSchema decides whether the resource will depart in the current epoch. If yes, then the system executes the Grid.nodeDeparture () method. If the resource remains in the system, then the remaining computational workload (workloadLeft) of the task under execution is decreased. The amount of the workload decreased depends on whether time-sharing or space-sharing scheduling policy is used. If a task is completed, then it is placed on the computatedTasks list, in order for the task's output data to return in the task's originating node-user. The task's remaining data for transmission (dataLeftToTrasmit) are calculated using the bandwidth of the link between the resource and the originating user.

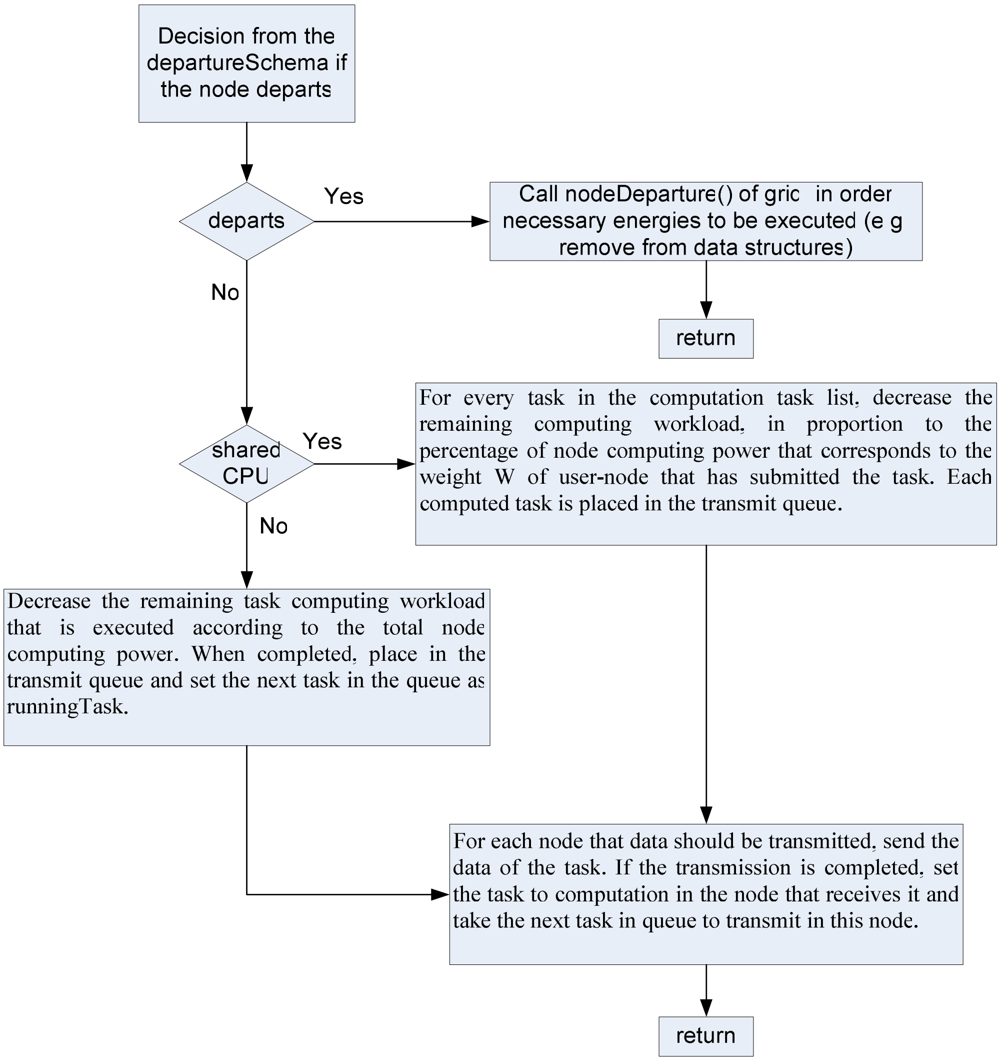

Decision from the departureSchema if the node departs
departs
Yes
Call nodeDeparture() of gric in order necessary energies to be executed (e g remove from data structures)
return
No
shared CPU
Yes
For every task in the computation task list, decrease the remaining computing workload, in proportion to the percentage of node computing power that corresponds to the weight W of user-node that has submitted the task. Each computed task is placed in the transmit queue.
No
Decrease the remaining task computing workload that is executed according to the total node computing power. When completed, place in the transmit queue and set the next task in the queue as runningTask.
For each node that data should be transmitted, send the data of the task. If the transmission is completed, set the task to computation in the node that receives it and take the next task in queue to transmit in this node.
return

Figure 12 – Simulation cycle of a resource

6.2 Technical Description of the Simulator

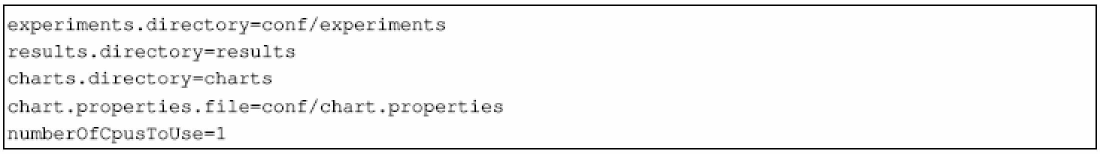
6.2.1 Implementation – Execution

The simulator is implemented in Java and it requires JRE version 1.5 at least in order to be executed. For logging purposes it uses the library commons-logging as a wrapper around log4j. The use of commons-logging as a wrapper around the real system logging, allows the adaptation and use of the simulator in environments that the log4j is not available. For the dynamic creation of objects and their assignment in class fields that implement the various entities we use the BeanUtils library, while for the creation of graphic representations we use the JCCKit library. The necessary distributions and other mathematic calculations were implemented using another piece of the Jakarta Project, that is the commons-math library.

The basic class that controls the entire simulation is the ExperimentsRunner. When the main() method of this class is executed, it creates an ExperimentsRunner object and calls the init() method for the initialization of the simulation. The basic initialization file is the config.properties that is found in the conf/ directory of the simulator. This file contains the following properties:

- experiments.directory: The directory that contains the XML descriptions of the experiments.
- results.directory: The directory in which the results of the experiments (serialized StatsAggregator objects) are stored.
- charts.directory: The directory in which the graphic representations of results are stored.
- chart.properties.file: The file contatining the configurations of the graphic representations library.
- numberOfCpusToUse: The maximum number of simultaneous computing threads that the simulator will create. A thread for each experiment is used.

For example a typical conf.properties file is the following:

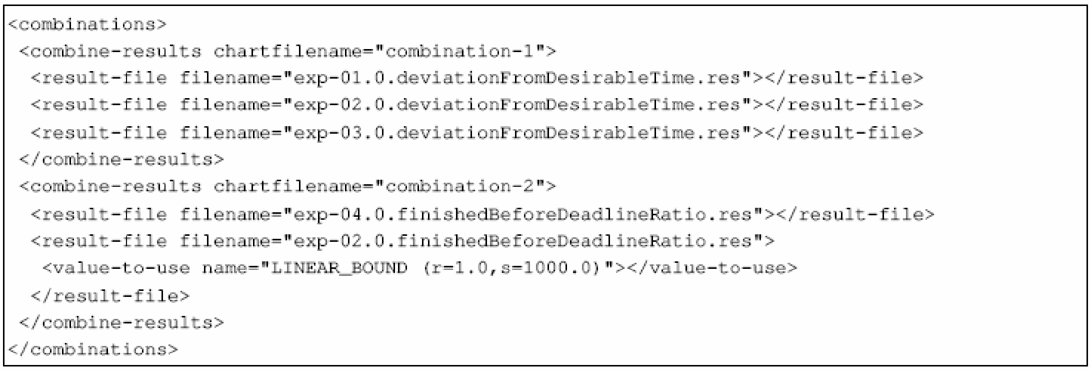


```
experiments.directory=conf/experiments
results.directory=results
charts.directory=charts
chart.properties.file=conf/chart.properties
numberOfCpusToUse=1
```

According to this configuration file, the XML descriptions of the experiments are in the conf/experiments/ directory, the results will be stored in the results/ directory and the graphic representations in the charts/ directory.

The ExperimentsRunner will read all the XML experiment files and for each one will call the getExperiment() method of the ExperimentXMLReader object. This method returns a list of Experiment objects. Each Experiment object represents an experiment and is, in fact, a simple wrapper around a Grid object, which contains all the necessary entities for the experiment (as described previously). Next the ExperimentsRunner will execute sequentially the experiments and will store their results (as serialized StatsAggregator objects), which contain the values of the measured metrics. If the main() function of the ExperimentsRunner is called with the input value “makecharts”, then all the result files of the results.directory will be read and for each one a graphic representation (in png format) will be created and stored in the charts directory. Also, there is the possibility of combining more than one StatsAggregator objects, so as to create a common graphic representation. This way we can compare scheduling techniques in similar experimental conditions. This capability is activated by using the “combineresults” value as input in main() function of the

ExperimentsRunner. In this case, the simulator searches for the file combineresults.xml in the results directory. An example of this file follows:



```
<combinations>
 <combine-results chartfilename="combination-1">
  <result-file filename="exp-01.0.deviationFromDesirableTime.res"></result-file>
  <result-file filename="exp-02.0.deviationFromDesirableTime.res"></result-file>
  <result-file filename="exp-03.0.deviationFromDesirableTime.res"></result-file>
 </combine-results>
 <combine-results chartfilename="combination-2">
  <result-file filename="exp-04.0.finishedBeforeDeadlineRatio.res"></result-file>
  <result-file filename="exp-02.0.finishedBeforeDeadlineRatio.res">
   <value-to-use name="LINEAR_BOUND (r=1.0,s=1000.0)"></value-to-use>
  </result-file>
 </combine-results>
</combinations>
```

Each combine-results tag corresponds to a combination and each result-file tag corresponds to a result file. Also, a result file tag may contain value-to-use tags that indicate the graphic representations that will be imported in the combined graph. If no value-to-use tag exists, then all the graphic representations that are in the result file will participate in the final, combined graph.

In our previous example, two combined graphs will be created. The first will contain all the graphic representations from three files: exp-01.0.deviationFromDesirableTime.res, exp-02.0.deviationFrom DesirableTime.res and exp-03.0.deviationFromDesirableTime.res. The second will contain all the graphic representations from the exp-04.0.finishedBeforeDeadlineRatio.res file, but from the exp-02.0.finishedBeforeDeadlineRatio.res file will only use the graphic representation with the name "LINEAR_BOUND(r=1.0, s=1000.0)". The various result files that participate in one combinational graph are supposed to contain the same metric, otherwise they will be rejected by the system.

6.2.2 Use

The simulator uses the Ant tool of the Apache Software Foundation. It has a central build.xml file, in which all the operations of the simulator are coded in targets. The possible targets can be appeared with the ant -projecthelp command and are the following:

- clean: Deletes the compiled classes.
- compile: Compiles the simulator.
- Run: Executes the experiment and stores the results.
- convert-results-to-charts: Transformes the stored results (serialized StatsAggregator objects) in graphic representations, in PNG form.

- combine-results-to-charts: Combines the stored results into graphic representations, according to the instructions in the file combine-results.xml

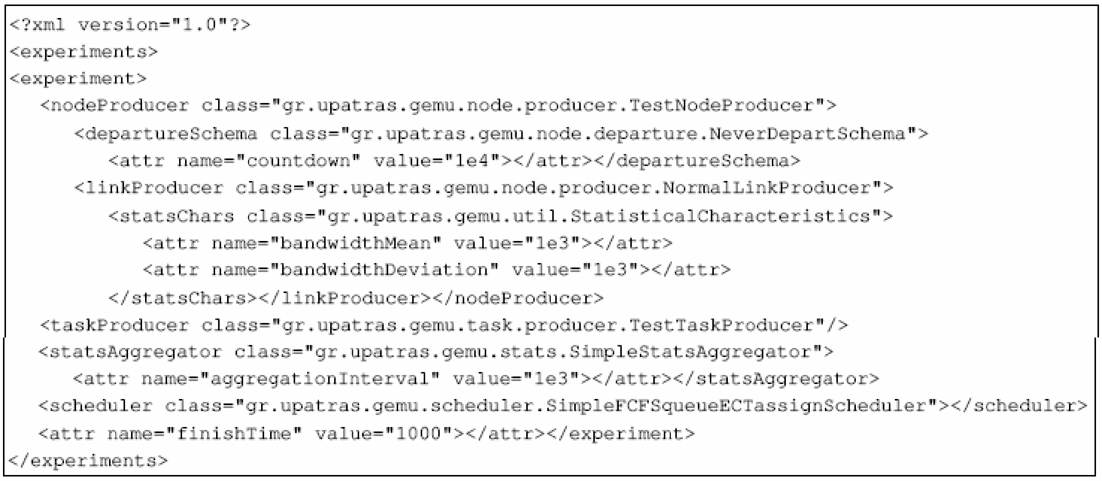
- javadoc: Creates javadoc of the application classes.

- run-all: Executes the target run and then the target convert-results-to-charts.

- all: Executes the target compile and then the target run-all.

6.2.3 XML description of experimental simulation

Experiments are described using a XML file. For example:



```
<?xml version="1.0"?>
<experiments>
<experiment>
   <nodeProducer class="gr.upatras.gemu.node.producer.TestNodeProducer">
      <departureSchema class="gr.upatras.gemu.node.departure.NeverDepartSchema">
         <attr name="countdown" value="1e4"></attr></departureSchema>
      <linkProducer class="gr.upatras.gemu.node.producer.NormalLinkProducer">
         <statsChars class="gr.upatras.gemu.util.StatisticalCharacteristics">
            <attr name="bandwidthMean" value="1e3"></attr>
            <attr name="bandwidthDeviation" value="1e3"></attr>
         </statsChars></linkProducer></nodeProducer>
   <taskProducer class="gr.upatras.gemu.task.producer.TestTaskProducer"/>
   <statsAggregator class="gr.upatras.gemu.stats.SimpleStatsAggregator">
      <attr name="aggregationInterval" value="1e3"></attr></statsAggregator>
   <scheduler class="gr.upatras.gemu.scheduler.SimpleFCFSqueueECTassignScheduler"></scheduler>
   <attr name="finishTime" value="1000"></attr></experiment>
</experiments>
```

Each experiment is enclosed in experiment tags. Also, there are tags that define the various entities; the process is retrospective. For each tag an instance of a class, named in the attribute “class”, is created and assigned with Java bean processes in the tag-parent. If the tag has the name “attr”, then it is not considered as an object, but as a primitive variable that takes the value in the attribute “value”. The initial object that is created is always the Grid object.

In our example, a Grid object will be created and his finishTime will be equal to 1000 (< attr name="finishTime" value="1000"></attr></experiment >). A nodeProducer object (<nodeProducer class="gr.upatras.gemu.node. producer. TestNodeProducer">) and departureSchema object (<departureSchema class="gr.upatras.gemu. node.departure.NeverDepartSchema">) will be assigned to this Grid object.

From this example we can see the flexibility of the simulator and how easy it is to experiment with different parameters. Also when implementing new entities and algorithms, it is quite trivial to create experiments that will use the new classes. The only demand is the new class to be placed in the classpath of the simulator and the name of the class in the corresponding class attribute in the XML file that describes the experiment. Another capability of the simulator is the successive execution of the same experiment, changing only the value of one parameter. This is achieved with the use of the iterateOn tag, in which the parameter of interest is defined along with its initial, its final and its step value. In case the iterateOn tag has been used, then the value of a metric is not related to time, instead the mean value of this metric is related to the values of the parameter of interest. Finally, if more than one scheduler tags are declared, then the simulator will execute the same experiment using all the schedulers declared. Thus, a curve for each scheduler will exist in the same graph, giving the chance of a direct comparison of schedulers' performance in relation to a metric of interest.

7 A Simulator for Optical Constraint Routing

A separate simulator has been developed to study the physical layer effects of optical networks. In this section we introduce the basic implementation elements of the impairment constraint based routing simulations that were thoroughly investigated in [7].

The simulator has been developed in Matlab [8] and provides performance studies of the European and the global Phosphorus network topology either for the transparent case or when 2R regeneration is deployed. A number of linear and nonlinear analytical models of optical impairments are implemented and their effects are considered in the path computation process.

7.1 Simulator Code

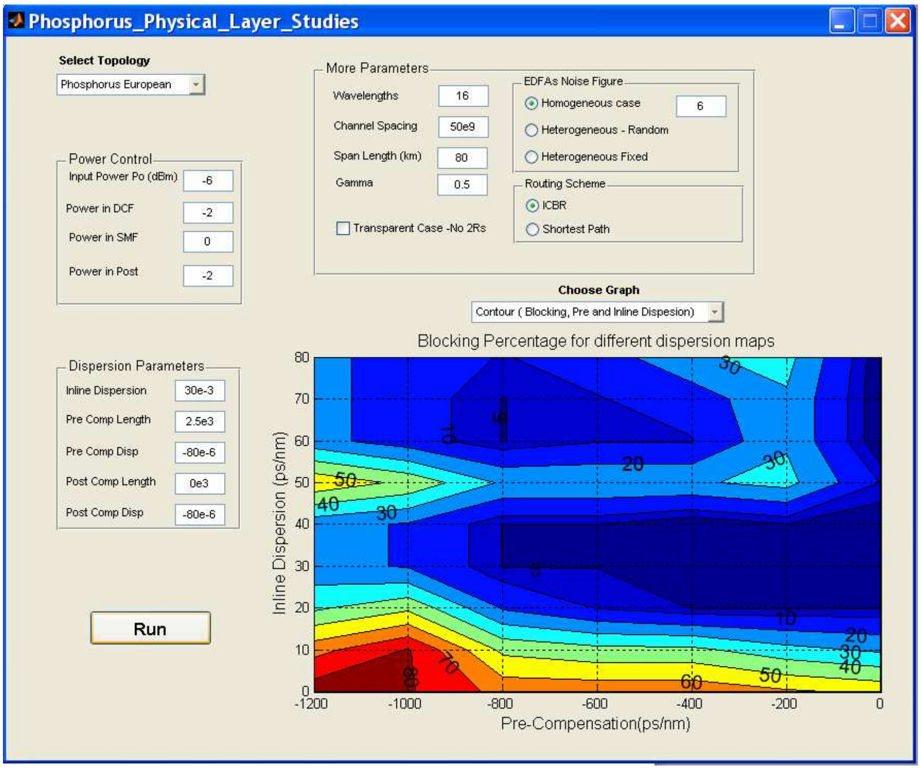
The simulator is separated in two parts. The first part (*European* folder) deals only with the fully transparent European Phosphorus network topology and includes analytical modelling for amplifier spontaneous emission (ASE) noise, filter concatenation (FC) effects, cross phase modulation (XPM), four wave mixing (FWM) and the combined effects of self phase modulation and group velocity dispersion (SPM/GVD) as detailed in [7]. The Q-factor penalty per link is calculated in the *Q_penalty.m* file by calling the files (*SPMGVDpenalty_new.m*, *sigma_find_XPM.m*, *sigma_find_FWM.m*) which perform the analytical calculations of the various impairments. This Q-penalty is used as weight to the Dijkstra algorithm from the *determine_paths*.m file to calculate the path for each request. After the paths are found Linear Programming (LP) is invoked (*main.m*) to determine whether there are enough wavelengths to carry the requests and calculate a blocking percentage due to lack of network resources. Finally the *paths_check.m* file is called to examine the quality of the path that has been selected by the LP and computes the percentage of the paths that will be dropped due to the optical impairments.

The second part (*Global_* folder) of the simulator utilizes the same parameters for the transparent case but it also considers the scenario in which 2R regeneration is deployed. For the 2R case, a penalty due to the accumulation of jitter may arise, confining the maximum reach of the network and thus the *paths_check.m* file is modified to consider also this penalty when evaluating the path performance.

The user can easily perform simulations on the two topologies by altering a number of parameters through a friendly graphical user interface described in the next section.

7.2 Tutorial

The GUI is initiated by running the *runme*.m file which opens the simulation window depicted in Figure 13. First the user selects from the drop down menu the topology to be used for the simulations (European or Global) and provides the values of some of the parameters that can be modified. Figure 14 presents the link structure considered for these simulations and illustrates the power and dispersion parameters that can be varied. The link consists of a number of spans (Single Mode Fiber, SMF) according to the link length where each span is followed by a dispersion compensation fiber (DCF) used to compensate at the requested degree the chromatic dispersion introduced by the SMF. The amount of the dispersion that is introduced in every span is referred as *inline dispersion*. At the beginning of the link a pre-compensation fiber (PRE) is used to insert initial chromatic dispersion whereas at the end, a post-compensation fiber (POST) is used to collect the chromatic dispersion evolved through the link. The pre and post dispersion values are the product of the *chromatic dispersion parameter* and the *length* of the pre and post compensation fibers respectively. The inline amplifiers of the link are required to compensate the losses of each fiber segment and boost the power to the appropriate levels at the entrance of each segment. The noise figure of these amplifiers can be set from the *EDFAs Noise Figure* area of the GUI as *homogeneous* (amplifiers on all links will have the same noise figure) or *heterogeneous* (amplifiers in different links may have a different noise figure but amplifiers of the same link continue to have identical noise figure).



Phosphorus_Physical_Layer_Studies

Select Topology: Phosphorus European

Power Control
- Input Power Po (dBm): -6
- Power in DCF: -2
- Power in SMF: 0
- Power in Post: -2

More Parameters
- Wavelengths: 16
- Channel Spacing: 50e9
- Span Length (km): 80
- Gamma: 0.5
- ☐ Transparent Case -No 2Rs

EDFAs Noise Figure
- ● Homogeneous case: 6
- ○ Heterogeneous - Random
- ○ Heterogeneous Fixed

Routing Scheme
- ● ICBR
- ○ Shortest Path

Choose Graph: Contour (Blocking, Pre and Inline Dispesion)

Dispersion Parameters
- Inline Dispersion: 30e-3
- Pre Comp Length: 2.5e3
- Pre Comp Disp: -80e-6
- Post Comp Length: 0e3
- Post Comp Disp: -80e-6

Figure 13 – GUI for the physical layer studies

Furthermore the power parameters considered on each fibre segment are depicted in Figure 14. The user has the ability to select the *input power* (Po, PinNODE) at each node which is also the power that enters the pre compensation fibre and the power at the input of the SMF (PinSMF), the DCF segment (PinDCF) and the post compensation fibre (PinPOST) in dBm. This is accomplished by altering the gain of the amplifiers to the appropriate level.

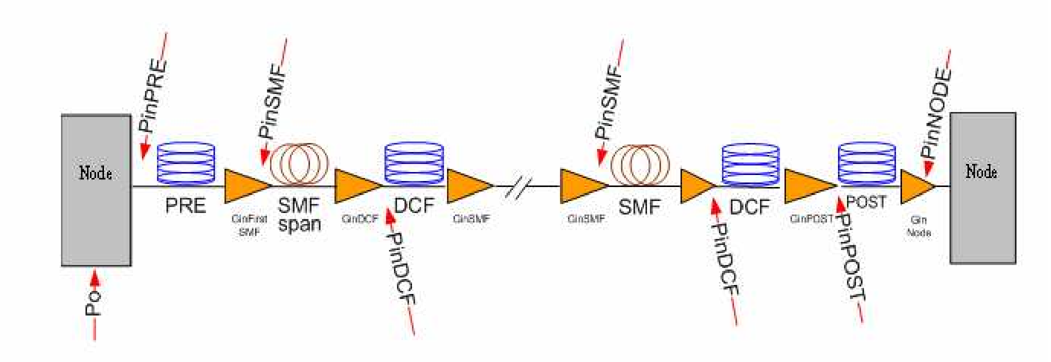


Figure 14 – Simulated link architecture

Table 1 reports some more parameters representing typical values of the two fibre types (DCF and SMF) that form the link and cannot be altered from the GUI but are required for the analytical calculations of the optical impairments.

Parameters	SMF	DCF
Attenuation a (dB/km)	0.25	0.5
Nonlinear index coefficient n (m^2/W)	2.6*10^{-20}	3.5*10^{-20}
Chromatic Dispersion Parameter D (s/m^2)	17*10^{-6}	-80*10^{-6}
Dispersion Slope dD/dλ (s/m)	0.085*10^3	-0.3*10^{-3}
Effective Area A_{eff} (m^2)	65*10^{-12}	22*10^{-12}

Table 1 – Constant fibre characteristics

In addition the user can define easily from the GUI the *number of wavelength channels* per link, the *channel spacing* (in Hz), and the *span length* (a single SMF segment on each link) in km. Also for the global Phosphorus topology a checkbox is available indicating whether the 2R regeneration is deployed on every network node or not. If the 2R case is selected the, the slopes γ (*gamma*) around the “mark” and “space” levels of the time invariant step-wise linear transfer function that models the reshaping properties of the subsystem must be provided. When γ=0 there is full suppression of the amplitude distortions, whilst when γ=1 no regeneration occurs. Finally the routing scheme (either Impairment Constraint Based Routing or Shortest Path) is selected as well as the requested graph from the drop menu for plotting the desired simulation results on the reserved space of the GUI.

The simulation starts by pressing the *run* button initiating the pre-processing phase that collects all the information related to the network and the traffic demands. Also the pre-processing phase assigns costs to the links based on the Q-factor penalties that are calculated utilizing the collected parameters. Once the link costs are found the RWA phase is initiated assigning paths and wavelengths to all the demands. If the RWA formulation is feasible the paths that should be established and the minimum number of wavelengths required to carry the requests are specified. The established lightpaths are processed next by the path check module which verifies the Q-factor constraint considering all the physical impairments involved across the path. This module evaluates the Q-factor at the end of the route for the path designated as the best from the k candidates that satisfy the specific request. A path is accepted when the Q-factor value at the destination node is higher than 11.6dB, which corresponds to a BER of 10^{-15} after forward error correction (FEC) is utilized at 10Gbps and the connection is established, in any other case the path is rejected and the connection is blocked.

8 Conclusions

This deliverable presents the architectural view of the simulation environment, and clearly motivates significant design choices. We discuss in detail important variables, methods and portions of code. Additionally, tutorial style documents are included based on simple scenarios, to ease the understanding and usage of the code. Note that the simulation environment has been written with extensibility in mind, so new features are straightforward to implement and added to it.

The simulation environment consists in large part of Java code, although the inclusion of physical layer parameters is written in the Matlab scripting language. Several important functions are accessible through the use of a Graphical User Interface, allowing intuitive configuration and setup of simulation scenarios. The open and standardized XML format is used for persistency and compatibility purposes.

The simulation environment supports not only the basic multi-domain, optical Grid network operation, but also advanced features and algorithms. These include multiple resource scheduling algorithms, evaluation procedures for fault-tolerance on both the network and resource level, and routing algorithms based on physical layer parameters.

9 References

[1] The Java Platform, http://java.sun.com

[2] The Colt Project, http://dsd.lbl.gov/~hoschek/colt

[3] Phosphorus-WP5-D5.2, “QoS-Aware Resource Scheduling”.

[4] A. Demers, S. Keshav and S. Shenker, “Design and Analysis of a Fair Queuing Algorithm,” Proc. of the ACM SIGCOMM, Austin, Sep 1989

[5] GridSim, “A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing”, http://gridbus.org/gridsim

[6] SimJava, “A Process-Based Discrete Event Simulation Package for Java”, http://www.dcs.ed.ac.uk/home/hase/simjava

[7] Phosphorus-WP5-D5.3, “Grid Job Routing Algorithms”

[8] MATLAB – The Language of Technical Computing, http://www.mathworks.com/products/matlab

10 Acronyms

[2R]	Re-amplification and Reshaping
[3R]	Re-amplification, Reshaping and Retiming
[ACK]	Acknowledgement
[ASE]	Amplifier Spontaneous Emission
[DCF]	Dispersion Compensation Fibre
[ECT]	Earliest Completion Time
[EDF]	Earliest Deadline First
[EDFA]	Erbium Doped Fibre Amplifier
[EST]	Earliest Start Time
[FC]	Filter Concatenation
[FCFS]	First Come First Served
[FEC]	Forward Error Correction
[FWM]	Four Wave Mixing
[GA]	Genetic Algorithm
[GIS]	Grid Information Service
[GUI]	Graphical User Interface
[GVD]	Group Velocity Dispersion
[JRE]	Java Runtime Environment
[JVM]	Java Virtual Machine
[LB]	Load Balanced
[LLF]	Least Length First
[LP]	Linear Programming
[MIPS]	Million Instructions Per Second
[MLF]	Maximum Length First
[MPT]	Minimum Processing Time
[OCS]	Optical Circuit Switching
[PE]	Processing Element
[RWA]	Routing and Wavelength Assignment
[SA]	Simulated Annealing
[SMF]	Single Mode Fibre
[SMP]	Symmetric MultiProcessing
[SPEC]	Standard Performance Evaluation Corporation

[SPM]	Self Phase Modulation
[QoS]	Quality of Service
[XML]	eXtensible Markup Language
[XPM]	Cross Phase Modulation