INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Thesis Degree

Assessment of dependability scenarios with failures in large scale grids using GridSim toolkit

Andrea Castiglioni

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Thesis Degree

Assessment of dependability scenarios with failures in large scale grids using GridSim toolkit

Andrea Castiglioni

Supervisor:	Prof. Dr. Alberto Trombetta
Co-examiners:	Prof. Dr. Dieter Kranzlmüller Dr. Michael Schiffers
Deadline:	30. March 2011

Abstract

Grid networks are used to process large amounts of data and use a large number of resources. These resources are embedded in virtual organizations and in coordination with each other, share the computing power to execute certain jobs. Grid systems must exhibit a high degree of dependability, i.e., the ability to appear trustworthy to users. A essential feature of grid networks is their large scalability, which represents a major challenge to understand the "dependability". In this project, we simulate large scale grids using the GridSim toolkit provided by the University of Melbourne. This toolkit is modified to adapt to our needs of having a virtual grid infrastructure on a large scale, in which the components (resources, routers, machines, etc.) can have failures of any kind: natural disaster, malicious attack or normal faults. We study with controllable, repeatable and observable experiments the behavior of the grid network when a single point of failure occurs and when a set of failures affects the grid network. The results of these simulations are compared with the results of the same architecture grid network without failure, to evaluate the differences of behaviour, and to understand the dependability of the network.

Sommario

Le reti grid sono utilizzate per elaborare grandi quantità di dati e impiegano un gran numero di risorse. Queste risorse sono inseriti in organizzazioni virtuali e coordinatamente condividono la potenza di calcolo per eseguire determinati lavori. I sistemi grid devono presentare un alto grado di "dependability", cioè la capacità di apparire affidabili agli utenti. Una caratteristica fondamentale delle reti grid è la loro grande scalabilità, che rappresenta una sfida importante per capire la "dependability". In questo progetto, abbiamo simulato una rete grid su larga scala utilizzando il GridSim toolkit messo a disposizione dall'Università di Melbourne. Questo toolkit è stato modificato per adattarsi alle nostre esigenze di avere un'infrastruttura grid virtuale su larga scala, in cui i componenti (risorse, router, macchine, ecc....) possono avere fallimenti di ogni tipo: catastrofi naturali, attacchi dannosi o normali guasti. Con esperimenti controllabili, ripetibili e osservabili, studiamo il comportamento della rete grid, quando un singolo punto di fallimento si verifica e quando un insieme di fallimenti intacca la rete. I risultati di queste simulazioni sono confrontati con i risultati della stessa architettura di rete grid senza fallimenti, per valutare le differenze di comportamento, e per comprendere la "dependability" della rete.

Contents

1. Introduction

1.1. What is the problem that we want to solve?

We can say that grid computing is and will be the future of networks that require large computational capabilities.

All that glitter is gold? The answer is no, in fact, grid computing brings many advantages over standard networks, but the management of a large scale is not easy. In a realistic grid scenario, resources, routers, machines, and users are distributed in different parts of the world, and this distribution must be done in a meaningful way. For example it is not sensible to put resources in an area of high seismic risk. Earthquakes, natural disasters, malicious attacks or normal network or resources faults are some of the problems that may affect the proper functioning of the network grid. What do we want? We want that the grid network working properly, despite these attacks or failures. For example if I send a gridlet (ie a job to be executed) to a certain resource but this one fails (for whatever reason), then I will have to relocate this gridlet to another resource available. This obviously causes a delay in gridlet execution, dependent on where it has been relocate in the grid network.

So the network will be created and managed, yet must preserve dependability (refers to Chapter 2.2), ie the reliability of a computing system with respect to users, which allows them to be able to trust it and use it safely and without preoccupations of possible failures. We want to analyze the behavior of the grid when these attacks or failures occur and assess the various delays for execute gridlets that depend on the network topology.

1.2. How to solve the problem?

Grid computing has emerged as a potential next generation platform for solving large-scale problems in science, engineering, and commerce. It is expected to involve millions of (heterogenous) resources scattering across multiple organizations, administrative domains, and policies. The management and scheduling of resources in such a large-scale distributed systems is complex and therefore, demands sophisticated tools for analysing and fine-tuning the algorithms before applying them to the real systems [1].

The simulation is a very powerful analysis tool, with which we can represent the real grid network and we can evaluate the events (output) succeeding the imposition of certain conditions by the user that wants to check the behavior of the grid network. Simulation is the only possible way to analyze the behavior of a large scale distributed and heterogenous system and is essential for carrying out research experiments in grid systems.
It allows us to avoid unnecessary workload on the resources which, moreover, in a distributed system must be coordinated, and in addition, having a large scale grids also avoids the involvement of several active users.

So to preseve the dependability and to study the behavior of grid using large scale scenario, we make some simulation using the GridSim toolkit provided by the University of Melbourne[1]. In this way we can have a virtual grid infrastructure that enables experimentation with dynamic resource management techniques and adaptive services by supporting controllable, repetable, observable experiments.
GridSim is an excellent toolkit to perform simulations, but to resolve the problem, parts of it are modified to meet our needs of a large network with the characteristics that we want.

The grid network involves all of Europe and the resources can be strategically placed (for example in areas of low seismic risk), so as to facilitate the dependability of the network. Also the connections between routers and between the various resources will be made strategically, so that if a connection fails, is possible to reach the destination with another connection (probably slower and more expensive). To study the behavior and reliability of the network, the simulations are made by increasing the complexity step by step; the architecture remains the same, but we pass from a single failure to a set of resource failures. Finally, tests are performed, setting the times about resources failure in various combinations, and assess how the network reacts with respect to the various dependabilty parameters (Refers to section 2.2).

1.3. Goals of thesis

In this thesis we study the behavior and the dependability (i.e. the ability to appear trustworthy to users) of a grid large scale scenario using GridSim toolkit. We have created a complex network in which routers, resources and users are allocated in Europe's main cities so we can have a grid that we call "Artificial EU Grid". The project goal is to run several tests to see how the "Artificial EU Grid" behaves when certain events occure. For example, if one or more resources fail, the gridlet (i.e. the job executed by the resource) must be submitted to another available resource, so probably at end, the gridlet will come back to the user with a certain latency. To do this the grid network must support the fault tolerance, i.e enable the system to continue operating properly in the event of the failure of some of its components.

The basic characteristics of fault tolerance required are:

1. No single point of repair
2. Fault isolation to the failing component
3. Fault containment to prevent propagation of the failure

It is obvious that if all the resources on grid network fail I can't submit the gridlet to any resources available, and so I have to wait to repair a resource. So in this project we study the behavior of "Artificial EU Grid" when a failure, natural disasters or malicious attacks happen. Finally the results obtained using "Artificial EU Grid" tipology network compare the grid with failures (discussed in this thesis) to the same grid without failures (discussed in [9]).

1.4. Structure of this thesis

This thesis is organised as follows.
In the Chapter 2 we explain briefly what grid computing is and the important of it in the new networks. We then describe the various aspects of the dependability and how to guarantee it, while the last part deals with the management of failure in grid network, and how the GridSim toolkit (the software used for the simulation) detected the failure.
In Chapter 3 we present the GridSim entity, e.g. users, resources, routers, etc. We then show the existing GridSim classes for support of the failure functionality and finally are present the changes make on some java classes of GridSim to obtain the features that we need.
In Chapter 4 the scenario on which the tests will be run is presented, and a small simulation with a low workload is introduced to facilitate the reader in understanding the behavior of the network.
Chapter 5 presents the simulation of the large scale network grid that we have built. In these simulations, we test the dependability of the network in the cases where we have: one single point of failure, a set of failures and finally a set of failures where we decide when resources fail and how long the failures last.
Chapter 6 shows a comparison between the simulations of the previous chapter and the simulation of the same large scale grid network without failures. Moreover some consideration is made about the simulations and the GridSim toolkit. Finally, the last chapter 7, presents the conclusions about the project and future work intentions. Appendix A reports all part of the Java code modified to meet our needs. Chapter 3, where the changes to the Java code are presented, is closely related to Appendix A .

2. Dependability and management of failures

2.1. What is grid computing?

Before talking about dependability, we introduce briefly the concept of grid computing. This concept was born in the nineties when the first computer networks and the Internet spread. The term grid computing is the act of sharing many works on a distributed computing infrastructure, these works can range from data storage to complex calculations and can be spread over large geographical distances. Usually these computers connect to each other in the network grid and work on the same problems that require large computational power; this grid network has much more power than a single supercomputer. Grid computing is used for applications that require huge data processing, for example, financial, medical and scientific sectors. The largest grid network is CERN grid in Geneva: the work on the LHC[1] project (which recreates the big bang), needs a huge computational power to process the mass of data generated by the experiment, so many machines distributed all over the world constitute the grid network of Cern. The Figure 2.1 [2] shows the idea of grid in a

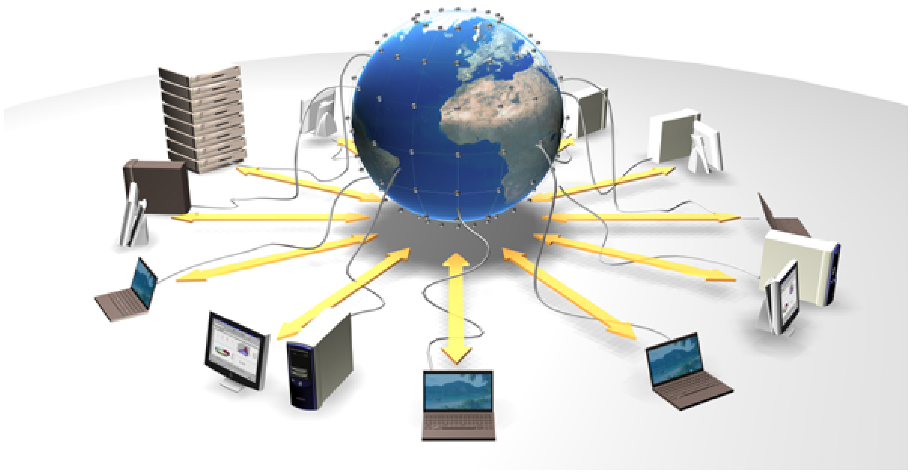

Figure 2.1.: How grid computing Works [2]

good way: computer all over the world can be connected to the grid network and share the computational power of the other machines in the network to perform works.

Networked computers can work on the same problems, traditionally reserved for super-computers, and yet this network of computers is more powerful than the super computers built in the seventies and eighties.

[1]Large Hadron Collider

Modern supercomputers are built on the principles of grid computing, incorporating many smaller computers into a larger whole. We can compare the grid to a power network where we connect the appliance and we use the power of the network to power our unit. The grid is the same: the computer is the appliance that, when connected to the network grid, uses its power.

2.2. Dependability

We want to study the dependability of the grid network, i.e. the ability to appear trustworthy to users. This subchapter gives a short introduction on dependability in distributed systems so the reader can better understand its different aspects. Dependability can be thought of as being composed of three elements, as described in 2.2 [6]:

- *Attributes:* a way to assess the dependability of a system. The following dependapility attributes are qualities of a system:
  - *Availability:* readiness for correct service
  - *Reliability:* continuity of correct service
  - *Safety:* absence of catastrophic consequences on the users and the environment
  - *Integrity:* absence of improper system alteration
  - *Maintainability:* ability for a process to undergo modifications and repairs

- *Threats:* Threats are things that can affect a system and cause a drop in dependability:
  - *Fault:* a fault is a defect in a system
  - *Error:* an error is a discrepancy between the intended behaviour of a system and its actual behaviour inside the system boundary. Errors occur at runtime when some part of the system enters an unexpected state due to the activation of a fault.
  - *Failure:* A failure is an instance in time when a system displays behaviour that is contrary to its specification. An error may not necessarily cause a failure.
- *Means:* ways to increase the dependability of a system:
  - *Fault prevention:* deals with preventing faults being incorporated into a system
  - *Fault Removal:* remove the fault during development or during execution
  - *Fault Forecasting:* predicts likely faults so that they can be removed or their effects can be circumvented
  - *Fault Tolerance:* deals with putting mechanisms in place that will allow a system to still deliver the required service in the presence of faults, although that service may be at a degraded level.

As we mentioned before we want to study the dependability of the grid network, so the grid must satisfy the quality attributes. If a resource fails, the grid must continue to work without altering the entire system. Our grid must always be ready in case a resource is out

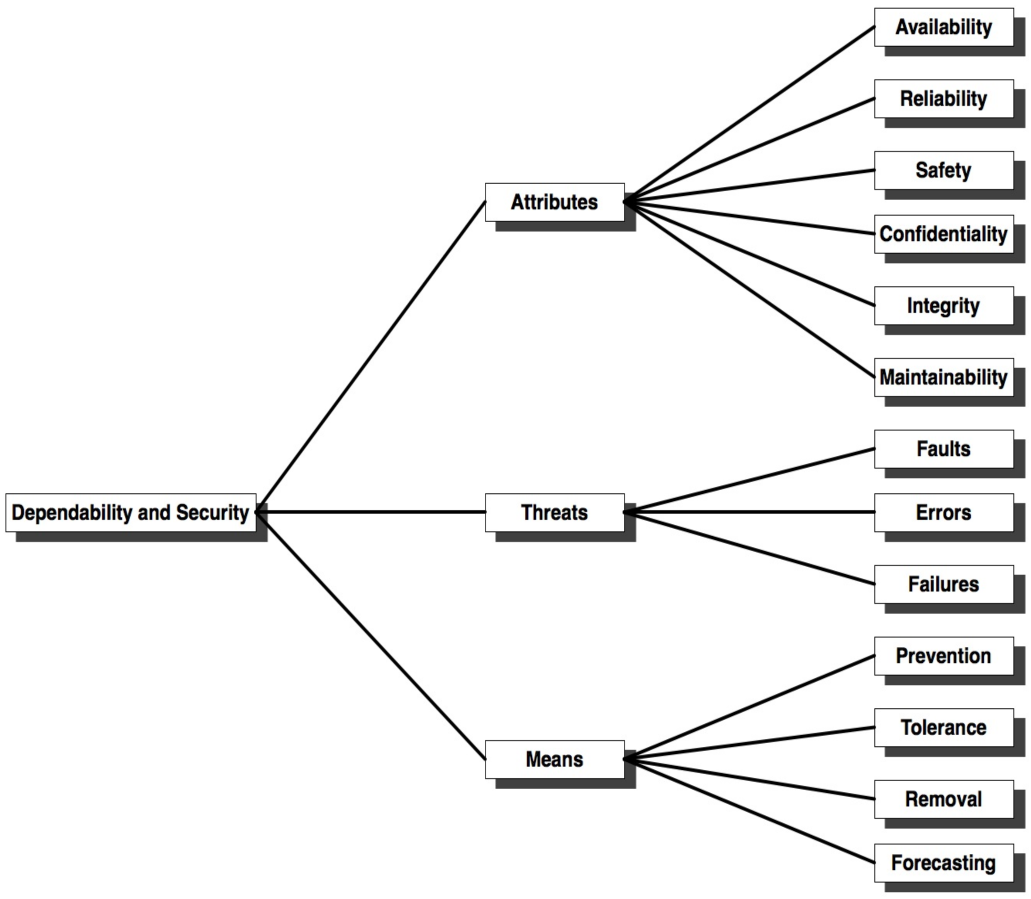

Figure 2.2.: Relationship between "dependability & security" and Attributes, Threats and Means [6]

of order, the work to be done will be delegated to another available resource. If all resources in the network are out of order, then it is necessary to repair the fault as soon as possible so that the system is ready to work.

What may alter the dependability of the grid network are the defects, errors and failures in the system. These three things make the resource, the single machine or router unavailable for a certain period of time. In our simulations on the grid network, we will not distinguish if it is a defect rather than a failure, because more importantly than why a resource is out of order, we want to know how long it is unavailable and the effects that has on the network. In the grid network, usually the components are located in strategic places that can at least avoid failures due to natural events. A situation of failure occurs when the services offered no longer correspond to the specifications previously imposed on the system.

The "means" are useful for increasing the dependability because they allow the system to work also in presence of defects, or better yet, prevent the defects. The fault tolerance is a property that allows the network to continue working properly when there is one or more failures of the components. To guarantee a fault tolerance, it is essential that there isn't a single point of repair, then the system must continue to work without interruption while the fault is repaired. The system must be able to contain and isolate the failure so that it does not propagate to the entire system. The techniques of prevention and fault tolerance allow the achievement of dependability, ie how to ensure that the system has the ability to provide a service that is always faithful to specifications. Additonally, the techniques of prevention and avoidance of faults show how to be confident in the ability of the system to provide a service according to the specifications established.

With many resources in the grid, it is obvious that the possibilities of resource or network failure are very high. The grid network on which we base our test should be able to support the fault tolerance, in order to have a platform that can simulate the real grid enviroment. The system must have a scheme for detecting the failure as well as a recovery scheme. In the next section, we describe the schemes of how it works, with GridSim toolkit, the detection of failures and recovery of them.

2.3. Management of failure

In this section some information is taken from the paper *Extending GridSim with an Architecture for Failure Detection [8].* For more details refer to [8].

In the Grid architecture there are three fundamental entites:

- the resources that execute the users' job.
- the users that submit jobs to the resources and contact the GIS[2] to know the list of available resources.
- the GIS that is responsible for maintaining an up-to-date list of available resources.

For detecting the resource failure, we use the pull method, where the GIS sends a message or a polling request to the resources monitored. When a resource receives this message, it must return it back to the sender. If the message doesn't come back to the GIS after a certain time interval, it means that the resource is not available at the moment.

Figure 2.3 [8] shows a sequence diagram about a failure detection scenario. First of all the two resources, Resource_1 and Resource_2 register to the GIS. The GIS adds the two resources to the list of available resources, but remember also that the GIS have to keep this list update, so it sends a message to the resouces periodically. Now User_1 wants to submit a job to a resource, so he contact the GIS to get the list of available resources. The GIS sends the list to the user that asked for it, and submits the job to Resource_1.

In step 4 we can note that a failure occurs to Resource_1, and the GIS uses the polling mechanism to detect the failure and remove Resource_1 from the list of available resources. User_1 periodically uses a polling mechanism to know if the resource is available, and in this case is discovers that Resource_1 is out of order.

If the failure involves only some machines of the resource, the destiny of the job depends on the allocation policy: if the resource uses a space-shared policy the job will be terminated and sent back to the user, but if it uses a time-shared policy the job continues its execution on other machines available of the resource, and it doesn't fail.

[2]GIS: Grid Information Service is an entity that allow the grid resources registration and provides services

So the User_1 has a job to process, so asks again to the GIS the list of available resources. Since now only Resurce_2 is available, User_1 submits his job to this resource (step 5). In the last step, Resource_1 works again and it's registered to the GIS, but Resource_2 doesn't fail in this simulation so the job executed is send back to User_1.

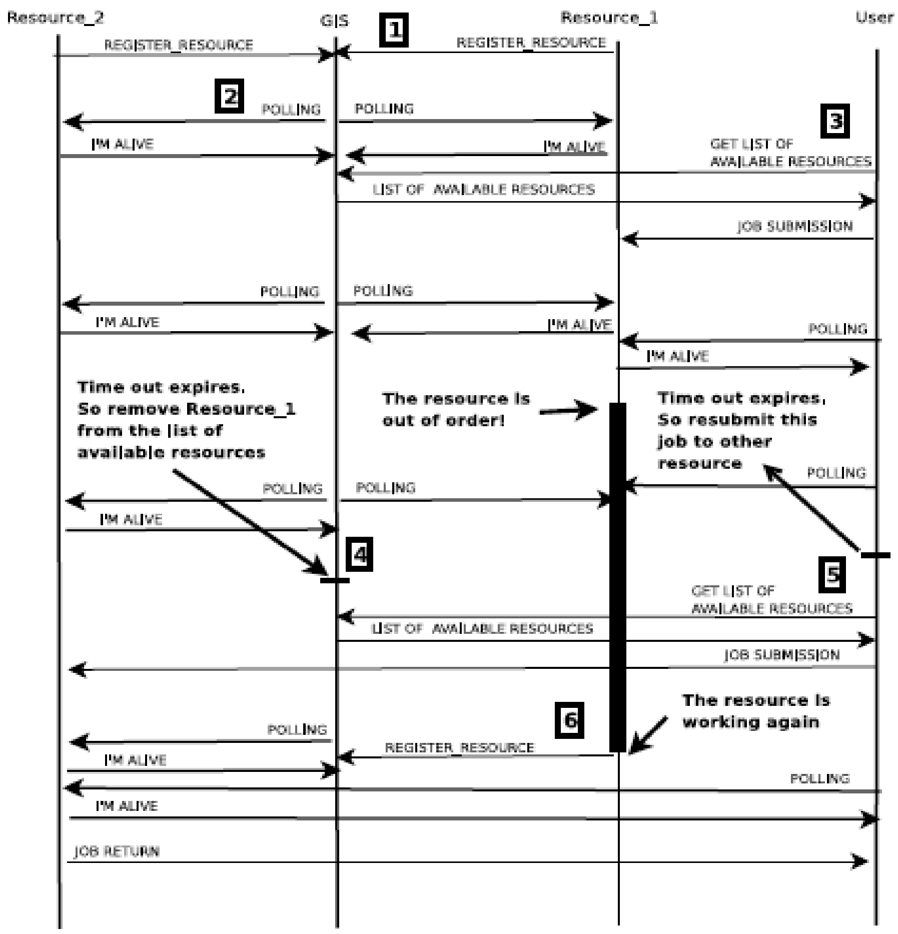
Resource_2
GIS
Resource_1
User
1
2
3
4
5
6
REGISTER_RESOURCE
POLLING
I'M ALIVE
GET LIST OF AVAILABLE RESOURCES
LIST OF AVAILABLE RESOURCES
JOB SUBMISSION
Time out expires. So remove Resource_1 from the list of available resources
The resource is out of order!
Time out expires, So resubmit this job to other resource
The resource is working again
JOB RETURN

Figure 2.3.: Scenario of failure detection [8]

3. Calibration of the GridSim toolkit

3.1. GridSim Entity

The GridSim is an open source software relased under GPL license (Copyright The Gridbus Project, GRIDS Lab, The University of Melbourne, 2002- to date) that can be downloaded from the site `http://www.cloudbus.org/gridsim/` [1] and allows modeling and simulation of entities in parallel and distributed computing (PDC) systems-users, applications, resources, and resource brokers (schedulers) for design and evaluation of scheduling algorithms. It provides a comprehensive facility for creating different classes of heterogeneous resources that can be aggregated using resource brokers for solving computational and data intensive applications. A resource can be a single processor or multi-processor with shared or distributed memory and can be managed by time or space shared schedulers. The processing nodes within a resource can be heterogeneous in terms of processing capability, configuration, and availability. The resource brokers use scheduling algorithms or policies for mapping jobs to resources to optimize system or user objectives depending on their goals.

In this project it was necessary to change several parts of the Java code of GridSim so the toolkit meets our needs. Note that we should represent the "Artificial EU Grid" scenario with failures. The main entity of the GridSim toolkit includes:

- *User:* an application or a broker that schedules jobs onto Grid resources is considered to be a user. Such components are able to query and request dataset transfers, submit jobs and register for events. Within GridSim, these are implemented by creating a specific DataGridUser object for a particular application or scenario [7].
- *Resource:* in Grid computing, any hardware or software component such as a cluster, a supercomputer or a storage repository is called a resource. Computing resources allow users to execute the required application. There is also a ResourceCharacteristics object that stores the properties of a Grid resource: architecture, OS, list of Machines, allocation policy: time- or space-shared [1], time zone.
- *Router:* users and resources are connected to routers.
- *Machine:* a Grid resource contains one or more machines. Similarly, a machine contains one or more PEs.
- *PE:* Processing Elements or CPUs.

[1] If the failure only affects some of the machines in a resource, what happens next depends on the allocation policy of this resource. If the resource runs a space-shared (first come first serve) allocation policy, the jobs that are currently running on the failed machines will be terminated and sent back to users. However, when the resource runs a time-shared (round-robin) allocation policy, no jobs will be failed, as their execution will continue in the remaining machines of the resource. For both allocation policies, the remaining machines are responsible for responding to polling requests from users and GIS.

- *Gridlet:* GridSim already has the ability to schedule compute-intensive jobs, which are represented by a Gridlet class. Each data-intensive job has a certain execution size (expressed in Millions Instructions (MI)) that will be used by a resource to determine how much simulation time is required [7].
- *RegionalGIS:* a Grid Information Service (GIS) is an entity that allow the grid resources registration and provides services. The Grid resources tell their readiness to process Gridlets by registering themselves with this entity. In addition, GIS is responsible for notifying all the registered entities, such as GridResource and network entities to shut down at the end of a simulation.
- *Link:* defines connection between Router-Router, Router-Resources, Router-Users.

3.2. Resource failure classes into GridSim

In this section, we describe the most important classes that GridSim has for support the resource failure functionality, while in the next section (refers to 3.3) we describe how some of them are modified to meet our needs. The information in this section is taken from *Extending GridSim with an Architecture for Failure Detection [8]*, for a detailed description of the GridSim classes that support failure functionality, refer to [8].

The Figure 3.1 [8] shows the GridSim classes to support failures. In the first line there are the main classes and represented under these are the classes that extend them. For example the class *GridUserFailure* extend the class *GridUser*.

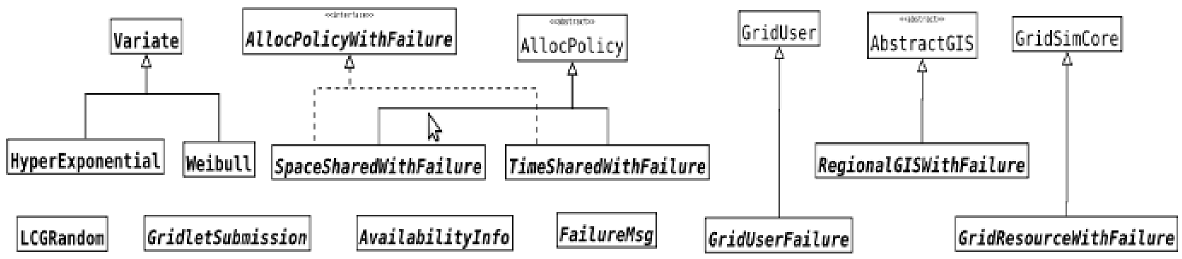

Figure 3.1.: GridSim classes that support failure functionality. [8]

This is a brief explanation of the failure classes of GridSim that we have later modified to create a network with custom features.

- *RegionalGISWithFailure*: this class is based on *RegionalGIS* GridSim class and it extends the class *AbstractGIS*. The class *RegionalGISWithFailure* allow the process of resource failure and the recovery of the resources. In this class the parameters are set for the duration of the failure and for the begining time of failure. This part, as we will see in the next section (refers to 3.3), is modified, so we can customise these parameters (in the orginal class these parameters are set randomly).

  We said that this class support the failure functionality, so we can decide which resource fails. Again, this part of the class is modified, because in the original class the resource that fails is chosen randomly, but according to our needs we want to decide which resource fails (more details in section 3.3).

- *GridUserFailure*: this class implements the behavior of the users of our grid environment, and allow the creation of jobs and the submission of jobs to resources. This part of the class is modified, because in our grid network we want to submit the gridlets to established resources, while in the original code the gridlets are submitted randomly (more details in section 3.3). If a resource fails while executeing a job, this class is responsible for choosing another resource and resubmitting the failed job to it (also this part is modified, for meet our needs).

  The last task of this class is receive the succeeded jobs from the resources.

- *GridResourceWithFailure*: based on *GridResource* GridSim class, it extends the *GridSimCore*. This class interacts with *RegionalGISWithFailure* to set machines as failed/ working. Also it interacts with classes implementing *AllocPolicyWithFailure* to set jobs as failed. *AllocPolicyWithFailure* is an interface, which provides some functions to deal with resource failures. Each allocation policy implementing this interface will have a different behavior with regard to the failures [8].

- *SpaceSharedWithFailure and TimeSharedWithFailure*: these classes are based on *SpaceShared* and *TimeShared* GridSim classes, two of the allocation policies already implemented in GridSim. They extend *AllocPolicy* and implement *AllocPolicyWithFailure* [8].

- *NumResPattern*, *ResPattern*, *TimePattern* and *LengthPattern*, are the four parameters that allow us to set the number of resources that fails, which resource will fail, when they fail and how long the duration of fails are. These parametres are modified because in the original program they are set randomly, based on continuos distribution (like uniform distribution), discrete distribution (as Poisson distribution), and variate distribution (like HyperExponential distribution). To support the generation of these random numbers based on these distributions, GridSim use the following classes: *Variate Random, LCGRandom, HyperExponential and Weibull.*

- Other classes that we don't describe, but only mention are: *AvailabilityInfo*, *GridletSubmission*, *FailureMsg*. For more details of these classes refer to [8]

3.3. What we have modified?

The original toolkit emulates the grid network very well, but it works very randomly. For example, a user or resource is randomly connected with a router, the User_0 can be connected to router_0 or to router_1 with the same probability. Is this what we want? No. We want to completely manage the grid, so we have set all parametres (listed below) to change several numbers of java classes. The only thing that we don't change is the registration time of the resource on the RegionalGIS. We don't set the registration time because after several test we noted that random registration don't cause a change in the output simulation.

In the next sub chapters we describe the changes made.

3.3.1. Changes in User-Router connection

In the original program, users are connected randomly to the router. We want to connect each user to a established router so that our scenario is realistic. Moreover, we want to create a custom network, so it's useless if the routers are randomly distributed. If we want that a user with certain features (eg. high workload) is connected to the Router_0 because it is close to powerful resources, we must avoid the random link between users and resources, because most likely the user will not be connected to Router_0.

In the original code, Listing A.1 (Appendix A.1.1), the java class makes a cycle for each user and links it randomly to Router0 or Router1 depending on the value of random.nextBoolean(). In the modified code, Listing A.2 (Appendix A.1.2), the java class makes a cycle for each user and links it to an established router. There is "statement" variable that is uses in the switch-case structure to decide which user is assign to a certain Router. Note also that in the original code, users are randomly assigned to a RegionalGIS, while in the modified code are assigned to a RegionalGIS with a established criterion.

In the modified code, for each case, a group of users correspond to and must be connected to a specific router. So with the command *linkNetwork*, the user is taken in consideration and is connected to the router specified. In the variable "routerName" the name of the router where the user is connected is saved. This is useful afterward to print to video the message that confirms the connection. We select the RegionalGIS where positioning the users with the command *gisList.get(number fo the RegionalGIS)*, and we set the connection between this and the user with the command *user.setRegionalGIS(gis)*.
Refers to Listing in Appendix A.1.1 to see the original code and refer to Listing in Appendix A.1.2 to see the modified code.

3.3.2. Changes in Resource-Router connection

The toolkit has been modified so that the resources are connected to a specific router and not to a random router. This is very important for the construction of the grid "Artificial EU Grid". It's obvious that for have a custom network the resources can't be distributed randomly. Moreover it is important that resources with some features (eg. high computational power) are connected to a certain router, probably in a zone of high network traffic or in a zone without natural disaster risk, while some other resources (eg with low computational power) are connected to other routers, depending on the needs that we have.

Are proposed the original code (Listing A.3, Appendix A.2.1) where the resources are link to the random router and then the modified code (A.4, Appendix A.2.2). A resource can be link to a Router in a certain RegionalGIS but it can belong to another RegionalGIS. The original toolkit made this randomly,but in the modified toolkit we select the RegionalGIS to be assigned to the resource. In the modified code we can note that there is a switch-case structure that is indispensable for assigning to each resource the features that we want. In each case (so for each resource), a resName is define to be the name of the resource. The command *createGridResource* permits the creation of the resource and specifies the baud rate (refers to 3.4) between resource and router, the propagation delay (refers to 3.4), mtu (Maximum Transmission Unit, refers to 3.4), the total number of cpu for each machine of the resource, the total number of the machines for the resource, the rating of the machines[2]

[2]In our simulation the rating changes from resource to resource, but it is the same for all machines of the resource

and the schedule alghoritm (time or space shared).

Like in the previous modification with the command *linkNetwork* we connect the resource with a specific router. Afterward, the name of the router is saved and we place the resource in a RegionalGIS with the command *res.setRegionalGIS(gis).*
Refers to Listing in Appendix A.2.1 to see the original code and refers to Listing in Appendix A.2.2 to see the modified code.

3.3.3. Changes failure parameters

In Listing A.5 Appendix A.3.1, 4 parameters are set using HyperExponential Java class that generates hyperexponentially distributed random numbers. This parameters are:

- how many resources fail;
- how many machines for each resource fail;
- when the resource will fail;
- how long the failure will be;

In the modified code, Listing A.6 Appendix A.3.2, we change the type of the variables (for each parameter we have a variable) from "HyperExponential" to "Int", and we make a switch-case structure, so for each RegionalGIS we can set the 4 parameters as we want. To do this, we have also modified the Java file (RegionalGISWithFailure.java): in this file we initialise the 4 variables and we set them using a "constructor class".
In this way we can decide for each RegionalGIS how many resources fail and how many machines for each resource fail, moreover we can decide the time when the resource has to fail, and for how much time it isn't available.

These changes in the java class are very important because allow us to manage the "Artificial EU Grid" as we want, for example if we want to simulate a natural disaster that destroyes one resource, and so we know that this resource is unavailable for one day, we can set the variable "how long the failure will be" equal to 24 hours. These code changes will become useful in the last simulations, when the network will be completely customized as we want and according to our needs. To see the description of simulation with time failure customised, refer to Chapter 5.3.

Note that, in the original code, to decide how many resources fail, when and how long the failure is, the code calls the class HyperExponential. This class is one of the class of the GridSim for support failure (refers to section 3.2). In this class there is a mathematical function that generates a hyperexponential distributed random number.
Refers to Listing in Appendix A.3.1 to see the original code and refers to Listing in Appendix A.3.2 to see the modified code.

3.3.4. Change init_time

Another thing changed is the time when we start to submit the gridlets to the resources. If you have many resources and GIS entities, we have to wait for a few minutes to allow GIS to receive registrations from resources. Otherwise, the resource does not exist when we submit. In the original code (Listing A.7, Appendix A.4.1) the initial time is set in a random way, in the modified code (Listing A.8, Appendix A.4.2) we set the variable init_time with an established number, which is very easy. Note that this time is expressed in seconds, so in the code for simplicity, we write the number of minutes multiplied by 60. We change this, because when we do the last simulation (with custom failure parameters), is important also to do tests with a low init_time, so probably not all resources are registered, and with medium and high init_time, the resources are probably out of order or maybe already being recovered, so they work normally.

Refers to Listing in Appendix A.4.1 to see the original code and refers to Listing in Appendix A.4.2 to see the modified code.

3.3.5. Set which resources fail

In the sub chapter above (3.3.3) we saw how to set *how many* resources fail, but we want also to decide *which* resources fail. In the original code (Listing A.9, Appendix A.5.1), the java class chooses for the failure a random resource only considering the set of the available resources in the RegionalGIS. We set this not randomly, because sometimes we want that a resource with high computational power fails to see the behavior of the grid network, sometimes we want that a resource fails in a certain position in the network, due to its geographical zone or to its connection to the routers. So it is impossible to do this in a random way, but we have to modify the code so that we can decide which resources fail. In the modified code (Listing A.10, Appendix A.5.2), we choose the Resource that we want from the RegionalGIS. To do this we have change several lines of code in the 'RegionalGISWithFailure.java' file.

The resources are registered to RegionalGIS in a random way, so when all resources are registered we have a list ordered by the registration time. We want for example to fail the 'Resource_6' because we know that corresponds to the resource located in Milan. We can't send a failure directly to the seventh resource in the List of available resources because it probably doesn't correspond to the 'Resource_6'. So before sending a failure we have to order our available resources in this way: Resource_0, Resource_1, ..., Resource_(n-1), Resource_n. Now if we send a fail to the seventh resource we are sure that is the 'Resource_6'.

To decide which resource fail we use an if-then structure with the support of the variable 'cont'. This variable is very important when we have more than one resource fail in the RegionalGIS.
This integer variable is incremented each time by 1, and counts how many resources fail for the RegionalGIS considered. Be careful, because if 2 resources fail for the RegionalGIS0 and it has 5 resources total, the first time 'cont' is equal to 1 and Resource_0 fails if the variable 'res_num' is set to 0 (now the number of the resources available are 4, and the first resource is: Resource_1) , the second time 'cont' is equal to 2, and if 'num_set' is equal to 0 the resource that fails is Resource_1, otherwise if 'num_set' is 1 the resource that fails is 'Resource_2'. We use this instruction "gisName.equals(*Name of the ArtificialRegionalGIS*)" to know which RegionaGIS we are considering and then we set the variable "Res_num" that indicates the number of the resource that fails.

If we set a number of resources that fail larger then the total resource of the RegionalGIS, then Java code sets the number of resources that fail equals to the total number of resources, so there aren't available resources in this RegionalGIS. If we examine the modified code

(Listing in Appendix A.5.2) we can note that the first thing is a *for cycle* that allows to order by id and in ascending way the resources in the grid network. After, with the command *gisName.equals(name of the RegionalGIS)* we can know which RegionalGIS is considered and set the variable "res_num". If we want a failure for the first resource in the RegionalGIS considered, we set res_num equals to 0. The next code lines are indispensable to send the message of resource failure and to set the recovery time for the resource.
Refers to Listing in Appendix A.5.1 to see the original code and refers to Listing in Appendix A.5.2 to see the modified code.

3.3.6. Submit the gridlet to a specific resource

Scenario like "Artificial EU Grid", a user has to submit gridlets to the resources that he wants, according to his needs. A user have to submits 20 gridlets to Resource_1 because it is the more powerful and other 25 gridlets to Resource_24 because it is safer than the other resources. In the original GridSim toolkit it is not possible to do this because the user submits his gridlets to the resources that are chosen randomly (Listing A.11, Appendix A.6.1). We changed several code lines of 'GridUserFailure.java' file so each user can decide to submit each gridlet to the resource that he wants (Listing A.12, Appendix A.6.2). In addition we edit the java class so that the user can submit some gridlets to a Resource_1, some to Resource_2 and some others to another resource.

If we check the modified code, we can note that with respect to the original code there is a switch-case structure. This is necessary to choose the gridlet that we take in consideration. So the variable "name_actual_user" indicates the user that we are considering. For example, if the user is *ArtificialEU_User_0* the correspondent *case* (in switch-case structure) indicates where to submit the gridlets. The variable "index" specifies the number of the resources where gridlets are sent. If we want to send the first ten gridlet to one resource and the other to another resource we use an if-then-else structure like in Listing A.6.2.
Refers to Listing in Appendix A.6.1 to see the original code and refers to Listing in Appendix A.6.2 to see the modified code.

Another important thing is that in original code the user can submit gridlets only to the resources inside his RegionalGIS, instead in our "Artificial EU Grid" we want that a user can submit his gridlets everywhere, for example a user in the RegionalGIS_0 can submit gridlets to resources available in the RegionalGIS_0 or in the other RegionalGIS. To do this we have to modify the function *'getResList()'* (the original code is this: Listing A.13) so that we can have an array that contains a list of ids of available resources in the grid network ordered by id (Listing A.14). When a resource fails there are two possible beahviors:

1. we wait for a recovery of the resource that failed.

2. we re-submit the gridlet to another resource

In the original code the user resubmits the gridlet to a resource available in a random way, with the changes made the user can re-submit the gridlet to an established available resource. If we check the modified code, it is possible to note that in the first lines, we get the list of the local resources in the RegionalGIS considered, and the list of global resources (the resources out of the RegionalGIS). The variable "lengthtotal" is equal to the sum of the sizes of the local and global lists of resources. The code lines after are necessary to join the two lists, while the two *for cycle* nested are used for order in ascending way the resources by id. Finally we check if there are resources in the list (with an if structure) and we return the array resourceID, using the command *return resourceID*. This array contains all the resources available at this time.

Refers to Listing in Appendix A.6.3 to see the original code and refers to Listing in Appendix A.6.4 to see the modified code.

3.4. Links between routers: the file network_thesis.txt

Different routers spread throughout Europe as well as related resources and users must also be interconnected with each other used order to create the network grid. We must therefore create the connections quite complexly and efficiently so that if a router should fail or a connection is broken, communication can continue diverting traffic to another way. This is important to preserve the dependability, if a resource fails, we must isolate the failure and ensure that the network continues to work as without failure, therefore it is essential that routers and resources are linked in a strategic manner.

To create these connections between routers we use a text file: *network_thesis.txt* (Listing 3.1). This file specifies the total number of the routers, the name of the routers and for each connection between two routers:

- *baud_rate(Gb/s):* in digital communication systems, the baud rate is the total number of physically transferred bits per second over a communication link. More simply, rates of exchange of data between two routers. The higher the cost, less time is used to send and process a gridlet.
- *prop_delay(ms):* in computer networks, propagation delay is the amount of time it takes for the head of the signal to travel from the sender to the receiver over a medium. It can be computed as the ratio between the link length and the propagation speed over the specific medium.
- *mtu(byte):* in computer networking, the maximum transmission unit (MTU) of a communications protocol of a layer is the size (in bytes) of the largest protocol data unit that the layer can pass onwards [5].

Listing 3.1: Network Topology

```
# total number of Routers
12

# specifies each router name and whether to log its activities or
   not
# by default no logging is required
Router0
Router1
Router2
Router3
Router4
Router5
```

```
Router6
Router7
Router8
Router9
Router10
Router11


# specify the link between two Routers
# The format is:
# Router_name1  Router_name2  baud_rate  prop_delay  mtu
                                (GB/s)      (ms)     (byte)

Router0          Router1         1          10       1500
Router1          Router2         1          10       1500
Router2          Router3         1          10       1500
Router2          Router5         1          10       1500
Router3          Router4         1          10       1500
Router5          Router4         1          10       1500
Router5          Router6         1          10       1500
Router6          Router7         1          10       1500
Router6          Router8         1          10       1500
Router5          Router9         1          10       1500
Router9          Router10        1          10       1500
Router9          Router11        1          10       1500
Router10         Router11        1          10       1500
Router11         Router1         1          10       1500
```

When we run the simulation, the first thing that the main class does is to read the topology network by the network_thesis.txt file. Below are shown the instructions in which the class reads the file (Listing 3.2); the Java main class is launched from the terminal with the following statement:

```
macbook-di-andrea-castiglioni:~ Beavis$ java -cp /Users/Beavis/
    Downloads/gridsimtoolkit-4.1/jars/gridsim.jar:.
    ArtificialEUGrid network_thesis.txt>simulation.txt
```

where *ArtificialEUGrid* is the name of Java class, *network_thesis.txt* is the topology network file and *simulation.txt* is the output file with results.

Listing 3.2: Read network grid from txt file

```
[....]
            String filename = args[0];  // get the network
                topology
            System.out.println(filename);
            System.out.println("Reading network from " + filename)
                ;
                LinkedList routerList = NetworkReader.createFIFO(
                    filename);
[....]
```

4. Introduction to simulation scenario

4.1. Failure in Artificial EU Grid

As mentioned in Chapter 1.3 we want to study the behavior and the dependability of grid using large scale scenario using GridSim toolkit. So we created the network grid 'Artificial EU Grid', a network that can represent a European grid: resources, machines, routers and users are located in European states, so that we can cover all Europe with the grid. In the Figure 4.1 [3] we can see, marked with a red dot, where the resource are located, more or less all Europe is covered by the grid.

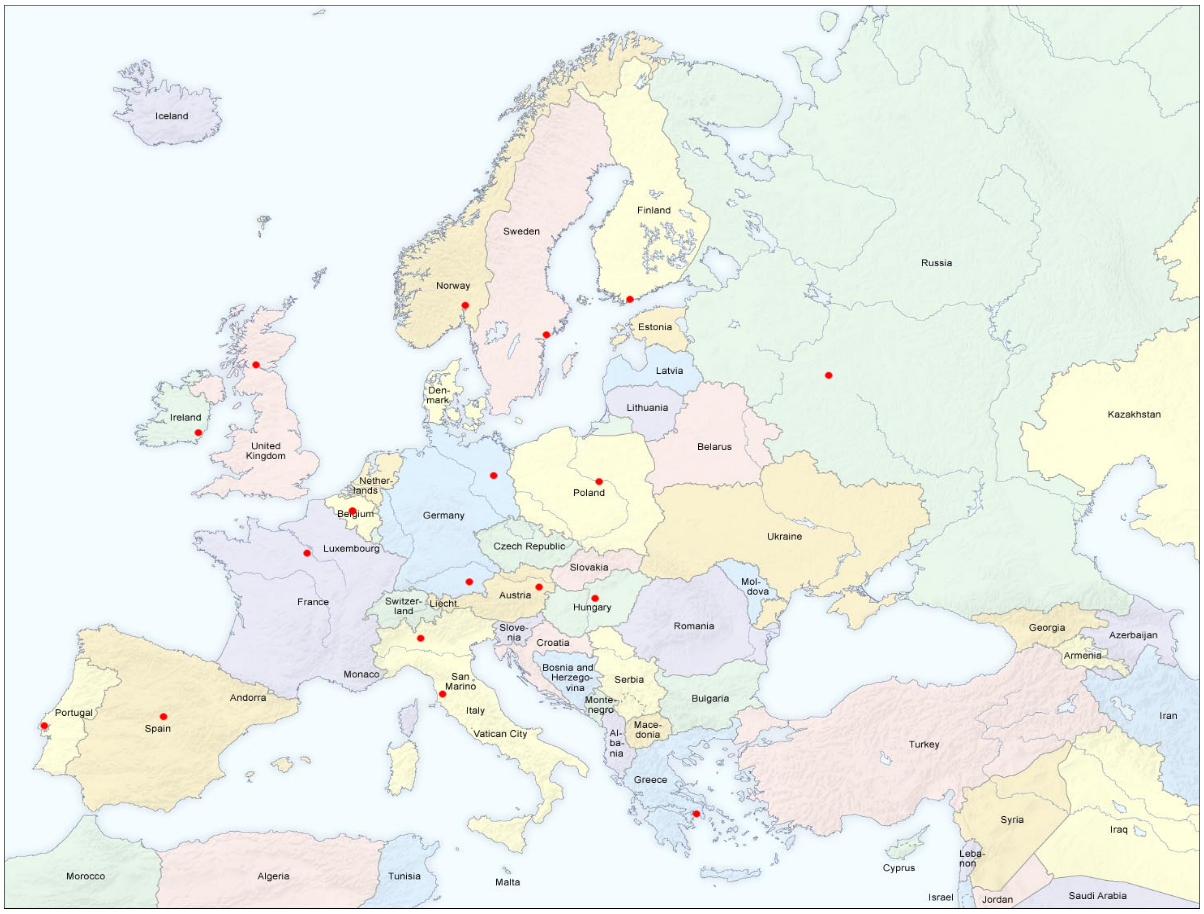

Figure 4.1.: Artificial EU Grid: dislocation of resources [3]

More precisely, the 'Artificial EU Grid' consists of five RegionalGIS (*RegionalGIS_O, RegionalGIS_1, RegionalGIS_2, RegionalGIS_3, RegionalGIS_4*), 12 routers and 18 Resources.

The routers are connected in a certain way so if a router should fail or a connection is broken, the communication can continue following another way. This is only a brief presentation of the 'Artificial EU Grid' structure (Figure 4.2), that can be useful to understand the simulations that we describe in the next sub chapters, the links between routers are set in List of Figures 3.1.

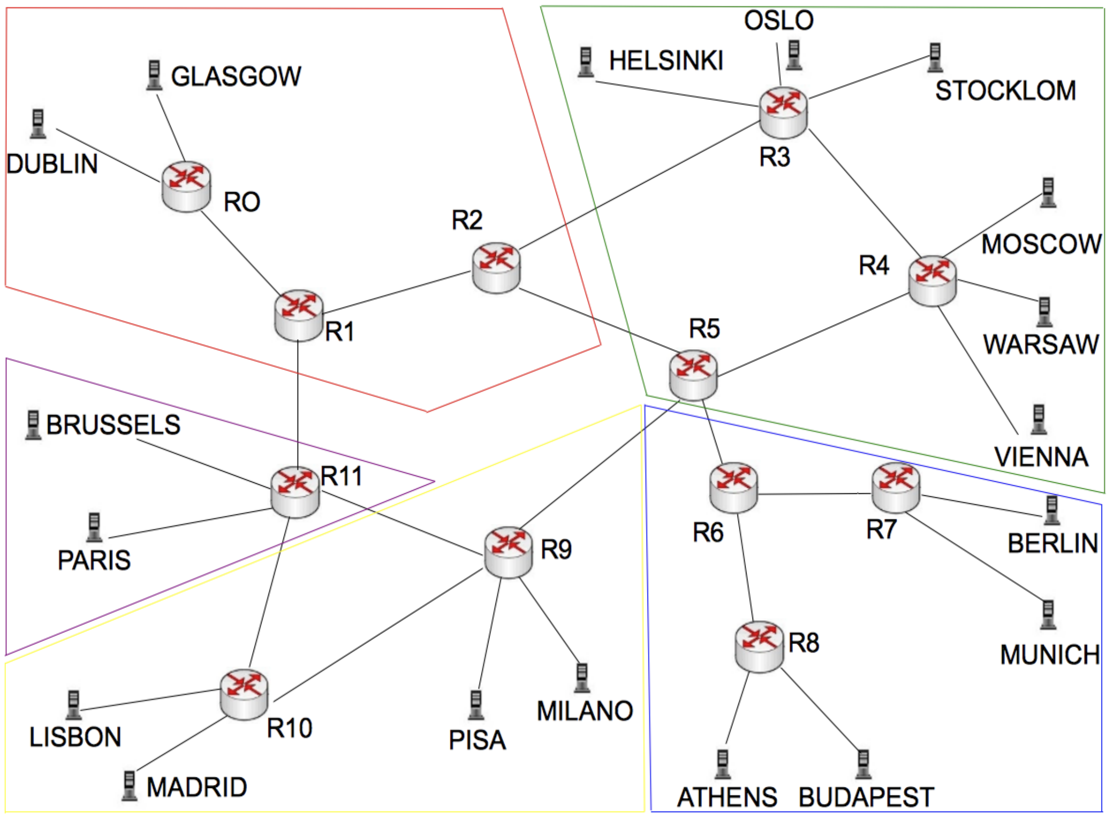

Figure 4.2.: Artificial EU Grid: structure

In the Table 4.1 there is the list of resources for the 'Artificial EU Grid', which specifies how many machines each resource has (column "NODE"), and to which RegionalGIS each one belongs. The computing power of each machine is expressed in MIPS (Million Instructions per second) and it corresponds to the CPU Rating, and also the allocation policy of the resource is declared . The resources have a *Sun Ultra* system architecture and use *Solaris*[1] operating system. The speed between the router and a resource is determined by the baud rate (expressed in Gb), some links resource-network have 1Gb of baud rate some have 0.0001 Gb. Baud rate is very important because it determines the final latency of the gridlet that runs on certain resource. Is important to have a heterogeneous grid network, so in the Table 4.1 we can note that the various resources have different values.

[1]Oracle Solaris. Refers to [4]

In the ArificialEU_Regional_GIS_0, there are only two resources with a good rating and baud_rate, so we decide to set a medium number of machines (30 for each resource).

In the ArificialEU_Regional_GIS_1 there are 6 resources, we decide to set high performances for the first three, and low performances for the last three. Res_Moscow, Res_Warsaw have a low number of machines and a low rating and are connected to the router with a low baud rate. Res_Vienna has a low number of machines but good rating and baud rate. We set these features, so in the simulation we can study the behavior in this RegionalGIS when the resources with good computational power fail, when resources with low computational power fail or when for example, all resources fail except Res_Vienna, which has to process a job with few machines but with high rating.

In the ArificialEU_Regional_GIS_2 we have 4 resources. Also there we set the parameters, so that we can have an heterogeneous RegionalGIS. Res_Berlin has more machines and less rating than Res_Munich so more or less this two resources have the same high computational power. So these two resources have more or less the same high computational power. Similarly, Res_Budapest and Res_Athens have.

In the ArificialEU_Regional_GIS_3 two resources are located in Italy, one in Spain and one in Portugal. So we set for each state one resource with high rating, baud rate and number of machines, and one with medium number of machines but low rating and baud rate with the router.

The two resources in ArificialEU_Regional_GIS_4 are located in France and are identical: they have good rating and good number of machines.

Resource name	**Node**	**RegionalGIS**	**Cpu rating (MIPS)**	**Policy**	**Baud_rate (Gb/s)**
Res_Dublin	30	ArtificialEU_Regional_GIS_0	49000	Space-Shared	1
Res_Glasgow	30	ArtificialEU_Regional_GIS_0	49000	Space-Shared	1
Res_Helsinki	60	ArtificialEU_Regional_GIS_1	49000	Space-Shared	1
Res_Oslo	50	ArtificialEU_Regional_GIS_1	49000	Space-Shared	1
Res_Stocklom	50	ArtificialEU_Regional_GIS_1	49000	Space-Shared	1
Res_Moscow	20	ArtificialEU_Regional_GIS_1	1000	Space-Shared	0.0001
Res_Warsaw	20	ArtificialEU_Regional_GIS_1	1000	Space-Shared	0.0001
Res_Vienna	20	ArtificialEU_Regional_GIS_1	49000	Space-Shared	1
Res_Berlin	100	ArtificialEU_Regional_GIS_2	49000	Space-Shared	1
Res_Munich	80	ArtificialEU_Regional_GIS_2	80000	Space-Shared	1
Res_Budapest	12	ArtificialEU_Regional_GIS_2	700	Space-Shared	0.0001
Res_Athens	10	ArtificialEU_Regional_GIS_2	700	Space-Shared	0.0001
Res_Milano	70	ArtificialEU_Regional_GIS_3	49000	Space-Shared	1
Res_Pisa	40	ArtificialEU_Regional_GIS_3	1000	Space-Shared	0.0001
Res_Madrid	40	ArtificialEU_Regional_GIS_3	49000	Space-Shared	1
Res_Lisbon	40	ArtificialEU_Regional_GIS_3	700	Space-Shared	0.0001
Res_Paris	50	ArtificialEU_Regional_GIS_4	49000	Space-Shared	1
Res_Brussels	50	ArtificialEU_Regional_GIS_4	49000	Space-Shared	1

Table 4.1.: Resources in the Artificial EU Grid

The main components of the network are the users, that want to submit the gridlets to the different resources. Users can have one or more gridlets, usually in our simulations we used 15 gridlets. This is because the length of each grid is 4200 MI (Milion Instructions) and

the total number of users is 125, so we have to submit 1875 gridlets and the simulation is not possible with a greater gridlets' number because of memory limitation of the computer we run the simulation on.
The length of the gridlets is set to 4200 MI, we can also increase this value up to 42000000 MI but the time of simulation becomes too high with this huge number of users and gridlets; for this reason, we prefer to use 4200 MI. The table 4.2 describes how the users are distributed in the grid network.

From *User* to *User*	**Connect to router**
from User_0 to User_9	Router2
from User_10 to User_29	Router3
from User_30 to User_41	Router4
from User_42 to User_71	Router7
from User_72 to User_81	Router8
from User_82 to User_101	Router9
from User_102 to User_113	Router10
from User_114 to User_124	Router11

Table 4.2.: Users in the Artificial EU Grid

If the number of users is set to more than 125, then the excess users submit their gridlets to the resources in a random way. We decide to assign the group of users at these routers, because if we refer to Figure 4.2, we can note that every RegionalGIS has a group of users and these are distributed across the entire network.
Remember that we want to simulate the network grid in case of possible failures of resources, routers or links due to earthquakes, or malicious attacks. In the next section, we will describe the simulations carried out on 'Artificial EU Grid' with failures using only one gridlet.
Then in the next chapter we make simulations on large scale grid (with more than 1 gridlet) with a single point of failure, with a set of failures, and in the end we set the failure time of the resources, the duration of fail, and when the user starts to submit his gridlets (refers to Chapter 3.3.3).
For view the description and the results of the 'Artificial EU Grid' without failures, refers to [9]. In the Table 4.3 there is a small summary of the components of the network:

N° ArtificialEURegionalGIS	**N° Resources**	**N° Users**	**N° Machines**
5	18	125	772

Table 4.3.: Summary of Artificial EU Grid

So the "Artificial EU Grid" has a good number of resources and machines, each machine has 4 PE[2]. The are not very many users, only 125, but we set this number because we use the same grid network setting in the simulation with more than 1 gridlet, and if we increase the number of users that have to submit many jobs, the complexity of simulation increases, and the machine on which we do the simulation doesn't support it.

[2]CPU unit

4.2. Artificial EU Grid: single point of failure, 1 GL

In this simulation we refer to the 'Artificial EU Grid' (Figure 4.2) and we make a failure on resource Res_Helsinki. The network characteristics are described in the previous paragraph 4.3, 4.2. To look the behaviour of the grid in a simple scenario we set the number of gridlets for each user equal to 1. We don't spend much time to describe this scenario because our goal is too simulate large scale grid (so with more than 1 Gridlet), but this first simulation is important to see the behaviour of the network.
Table 4.4 specifies for each user, to which resource he submits the gridlet (every user can submit a gridlet to a different resource, but for simplicity we have combined the users into groups, that submit the gridlet to a certain resource):

From *User* to *User*	**Submit to resource**
from User_0 to User_9	Res_Helsinki
from User_10 to User_29	Res_Budapest
from User_30 to User_41	Res_Budapest
from User_42 to User_71	Res_Budapest
from User_72 to User_81	Res_Budapest
from User_82 to User_101	Res_Madrid
from User_102 to User_113	Res_Budapest
from User_114 to User_124	Res_Budapest

Table 4.4.: Where gridlets are submitted

The different groups of users submit their gridlets to only three resources, because we want to study the behavior of the user when he submits a gridlet to a reosurce with high computational power (Res_Helsinki), low computational power (Res_Madrid) and medium computational power (Res_Budapest). Be careful, because in the next simulations Res_Helsinki fails and in the second variant we decrease the baud rate for the connections that lead to Res_Madrid.

4.2.1. First variant

The configuration of baud_rate between routers is the same as in Listing 3.1, beetwen resource and router is the same as in Figure 4.2.
What do we expect? Ten users submit the gridlets to Res_Helsinki, but this resource fails, so we expect that the gridlet is resubmitted to another resource. We expect that latency is high for the gridlet submitted to Res_Budapest because many gridlets are sent to this resource that has only 12 machines, a low cpu rating and a low baud_rate (with the Router_10)

From Table 4.1 we can note that Res_Helsinki has 60 machines, so if we want that this resource fails completely, all machines of it must fail. We can see in our simulation that all the machines fail from the output file: Listing 4.1.

Listing 4.1: Res_Helsinki no working machines

```
[...]
ArtificialEU\_Res\_Helsinki.processFailure(): receives an event
   GRIDRESOURCE_FAILURE. Clock: 337.13031324800005 which will last
     until clock: 766.9132608756137. There are NO working machines
   in this resource.
[...]
```

If we examine the results of output we see that the first 10 users who send their gridlets to Helsinki now send them in Oslo (due to the fails of Res_Helsinki). Other users send normally the gridlets to the prefixed resources because these are not involved in fails. As we expected the gridlets send to Res_Madrid have a low latency, infact the users are connect directly to the resource and the network traffic is not high.

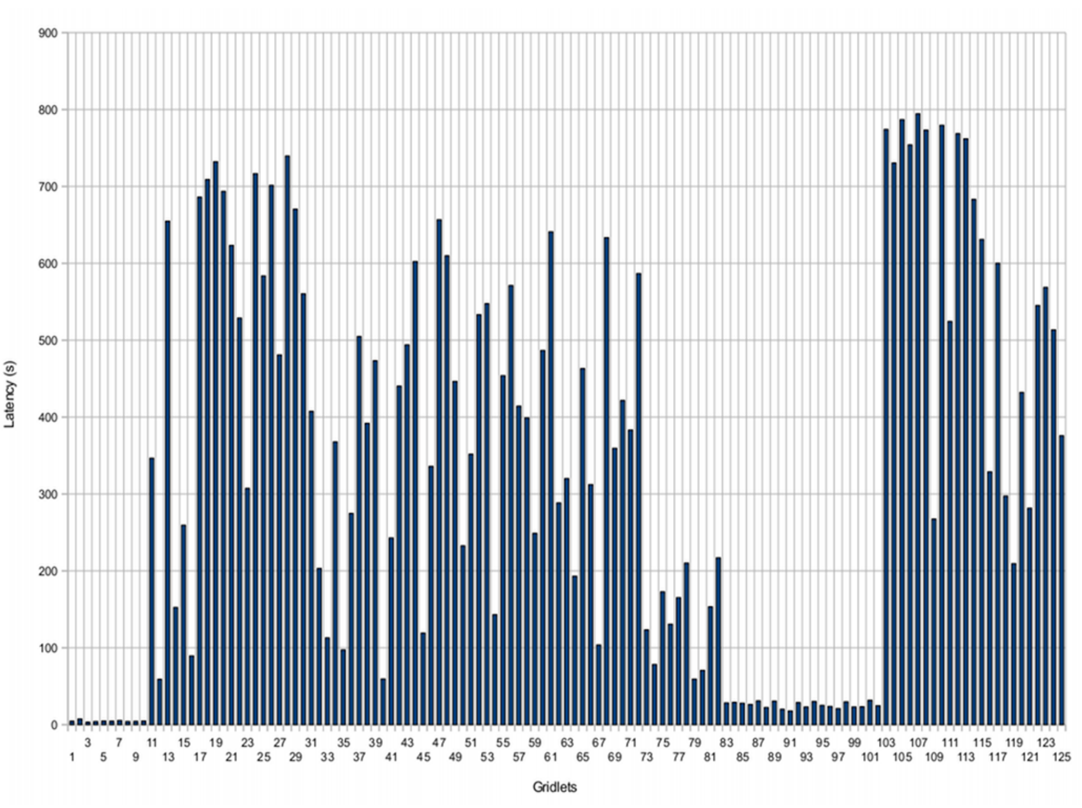

Figure 4.3.: Graph: Res_Helsinki fail, 1gridlet

Infact in graph 4.3 we can see that from users 82 to users 101 the gridlet is submitted to Res_Madrid and their latency is between 20 and 30 seconds (the average latency for these 20 gridlet is 25,82s in this simulation, it is obvious that, even if only slightly, it may vary from one simulation to another).
We can also note (Table 4.5) that the first 10 gridlets are sent to Res_Helsinki which fails, so we decide to submit the gridlets to Res_Oslo which has more or less the same characteristics of Res_Helsinki, so the result doesn't change much.

In graph 4.3 we can see that the others gridlets have an high latency because they are

Gridlet number	Res_Oslo	Res_Helsinki
0	4,5	4,33
1	7,26	4,49
2	3,42	7 ,93
3	4,06	4,03
4	4,74	6,04
5	4,6	6
6	5,54	7,72
7	4,18	3,42
8	4,41	5,83
9	4,7	5,51

Table 4.5.: Comparison latency between first 10 gridlet, first time submit to Res_Oslo (Res_Helsinki fail) and second time submit to Res_Helsinki (no fail)

submit to Res_Budapest which has only 12 machines, so if some machines are free, the gridlet is immediately processed and the latency is low (approximately 100-120), otherwise we have to wait free machines to process the gridlets and the latency of gridlets increase.

4.2.2. Second variant

The configuration beetwen resource and router is the same as in Figure 4.2, but in this second simulation we change the baud rate between the routers, so in the file network_thesis.txt (refers to 3.1) we change this connection:

Listing 4.2: network_thesis file modified

```
[...]
Router9       Router10         0.00001       10       1500
Router11      Router10         0.00001       10       1500
[...]
```

We decrease the baud rate for the connections between Router9-Router10 and Router11-Router10, from 1 Gb/s to 0.00001 Gb/s. The users are distributed and the gridlets are submitted to the resources like in the "first variant" simulation, so *User_82, User_83, ..., User_101* submit the gridlets to Res_Madrid that is connected to Router10. So the only two ways to arrive at Res_Madrid are from link Router9-Router10 or Router11-Router10 (refers to Figure 4.2). We set these links with a low baud_rate (0,00001 Gb/s) so we expect that the latency for the gridlets submitted to Res_Madrid is much higher than in "first varian" simulation.

The graph 4.4 explains very well the results of simulation:

- The first 10 gridlets are submitted to Res_Helsinki, but this fail, so they are resubmitted to Res_Oslo as in the "first variant" simulation. In fact, the values of latency for these 10 gridlets are more or less the same as in the first variant.
  Note that in these simulations we set the program in a way that when a resource fail occurs, there aren't *resource_recovery* signals, so the resource is off for all the simulation's time.

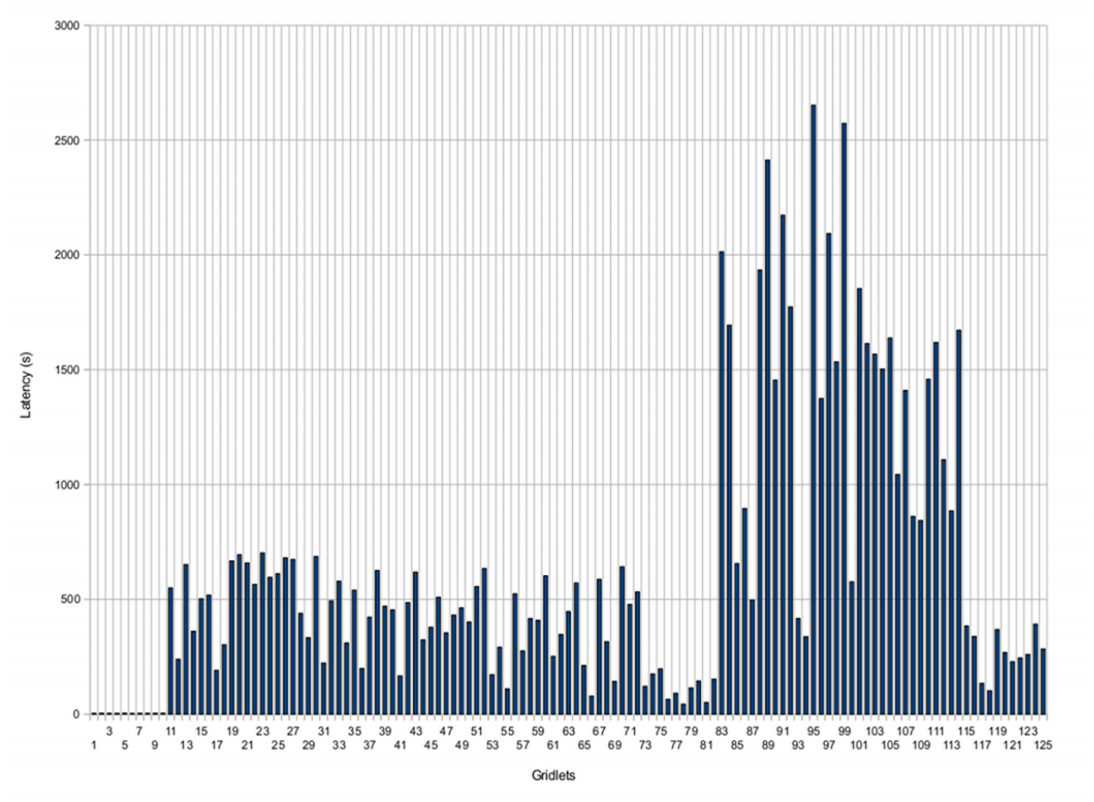

Figure 4.4.: Graph: Res_Helsinki fail, 1gridlet, change the network settings

- Same speech for the gridlets submitted to Res_Budapest, this part of the network is the same that in "first variant", so the output latency for the gridlets is like in the first simulation.
- The 20 gridlets sent to Res_Madrid are submitted by users that are connected to Router9 (as described before we have 2 ways to arrive to this resource in the network). As we can see in the graph 4.4 the latency for these gridlets increases dramatically. The values of latency for these gridlets oscillate between 1000s and 2000s with an average of 1526,53 seconds. This is precisely due to the low baud rate of connections to the resource.

It is very useful to keep track of users during simulations. In this case we consider for example the User_94 that submitted his gridlet to Res_Madrid. In the trace file of this user we can see the sending and receiving time for the gridlet in the "first variant" simulation and in the "second variant".
Obviously, the time difference, produces latency gridlet. In Listing 4.3 we can see the numerical results for User_94 that are described in the two graphs 4.3, 4.4.

Listing 4.3: User_94 trace file

```
Event    GridletID     Resource            GridletStatus  Clock
Sending     0      ArtificialEU_Res_Madrid  Created  967.420383
Receiving  0      ArtificialEU_Res_Madrid  Success  997.870431

Event    GridletID     Resource           GridletStatus  Clock
Sending     0      ArtificialEU_Res_Madrid  Created  966.580383
Receiving  0      ArtificialEU_Res_Madrid  Success  3070.300151
```

There is a big difference for the User_94 between the first simulation where the latency is 30,450048s and the second simulation where the latency is 2103,719768s. This is precisely due to the low baud rate.

5. Artificial EU Grid: large scale simulation

As mentioned in the previous chapter we simulate a large grid scenario (Artificial EU Grid) with one single point of failure, with a set of failures, and in the end we set the failure time of the resources, the duration of fail, and when the user starts to submitted his gridlets.

5.1. Artificial EU Grid: single point of failure

We want to simulate a real large grid scenario, so we always refer to Artificial EU Grid (Figure 4.2), but the number of gridlets that every user has to submitted is set to 15. We have only one point of failure, so as in the previous scenario we decide to send a fail signal to Res_Helsinki. Note that unlike the previous simulation, now it is possible to recover the resource. So when we submitted gridlets to Res_Helsinki and this is "fail", then we send gridlets to Res_Oslo, but if Res_Helsinki begins to work again, the gridlet will be submitted here last.

We slightly modified the file network_thesis.txt (Listing 5.1), so the network is more heterogeneous. We set the baud rate equal to 0,00001 for the links between Router0-Router1, Router9-Router10, Router10-Router11[1][2]

Listing 5.1: Network Topology - single point of failure

```
\# total number of Routers
12

\# specifies each router name and whether to log its activities or
    not
\# by default no logging is required
Router0
Router1
Router2
Router3
Router4
Router5
Router6
Router7
Router8
Router9
Router10
Router11

```

[1]These changes can make an increase of latency if gridlets travel one of these link connections between routers

[2]These changes increase the duration of the simulation

```
\# specify the link between two Routers
\# The format is:
\# Router_name1  Router_name2  baud_rate  prop_delay  mtu
                                (GB/s)      (ms)     (byte)

Router0          Router1        0.00001      10       1500
Router1          Router2          1          10       1500
Router2          Router3          1          10       1500
Router2          Router5          1          10       1500
Router3          Router4          1          10       1500
Router5          Router4          1          10       1500
Router5          Router6          1          10       1500
Router6          Router7          1          10       1500
Router6          Router8          1          10       1500
Router5          Router9          1          10       1500
Router9          Router10       0.00001      10       1500
Router9          Router11         1          10       1500
Router10         Router11       0.00001      10       1500
Router11         Router1          1          10       1500
```

The simulation takes a long time due to the large number of gridlets sent and lower baud rate between some links. In the table 5.1 we can see how users submitted gridlets to the resources, we decide for each user to split his set of gridlets, so the first 10 gridlets are submitted to a certain resource the other 5 are submitted to another resources.

From *User* to *User*	**submitted first 10 to**	**submitted last 5 to**
from User_0 to User_9	Res_Dublin	Res_Glasgow
from User_10 to User_29	Res_Warsaw	Res_Vienna
from User_30 to User_41	Res_Berlin	Res_Munich
from User_42 to User_71	Res_Budapest	Res_Athens
from User_72 to User_81	Res_Milano	Res_Pisa
from User_82 to User_91	Res_Madrid	Res_Lisbon
from User_92 to User_101	Res_Madrid	Res_Lisbon
from User_102 to User_113	Res_Paris	Res_Paris
from User_114 to User_124	Res_Paris	Res_Helsinki

Table 5.1.: Where gridlets are submitted in Artificial EU Grid

The setting of the resources is the same as Table 4.1. When Res_Helsinki fails we decide that the gridlets will be submitted not to the Resource established but to the next resource, for example if one gridlet was submitted to Res_Moscow now (with the failure) is submitted to Res_Warsaw (for network grid structure refers to Figure 4.2)

5.1.1. Results of simulation

This section presents the results of the simulation. Obviously it is impossible to describe for each of the 120 people what happens, but be consider the most significant aspects of the simulation, ie, those that allow us to understand the functioning of the network in case of grid failures.

First, we examine the times of Res_Helsinki failure. If we go to check the output file of the simulation (Listing 5.2) we can see that:

Listing 5.2: Failure and recovery of Res_Helsinki

```
[...] ArtificialEU_Regional_GIS_1: sends an autogenerated
   GRIDRESOURCE_FAILURE to itself. Clock(): 0.0. resTimeFail:
   20.918443148412194 seconds[...]

[...] ArtificialEU_Regional_GIS_1: 1 resources will fail in this
   simulation. Num of failed machines on each resource will be
   decided later[...]

[...] ArtificialEU_Regional_GIS_1: sends an autogenerated
   GRIDRESOURCE_FAILURE to itself. Clock: 20.918443148412194,
   resTimeFail: 27.138488245452017 seconds[...]

[...] ArtificialEU_Regional_GIS_1: sends a GRIDRESOURCE_FAILURE
   event to the resource ArtificialEU_Res_Helsinki. numMachFailed:
    100. Clock: 337.090157248. Fail duration:0.11938415211878156
   hours. Some machines may still work or may not[...]

[...] ArtificialEU_Res_Helsinki.processFailure(): receives an event
    GRIDRESOURCE_FAILURE. Clock: 337.13031324800005 which will
   last until clock: 766.9132608756137. There are NO working
   machines in this resource.[...]

[...] ArtificialEU_Res_Helsinki - Machine: 0 is set to FAILED
ArtificialEU_Res_Helsinki - Machine: 0 is FAILED
ArtificialEU_Res_Helsinki - Machine: 1 is set to FAILED
ArtificialEU_Res_Helsinki - Machine: 1 is FAILED[...]

[...] ArtificialEU_Regional_GIS_1: sends a GRIDRESOURCE_RECOVERY to
    the resource ArtificialEU_Res_Helsinki. Clock: 961.0[...]

[...] Clock: 962.92039
ArtificialEU_Res_Helsinki - Machine: 0 is set to WORKING
ArtificialEU_Res_Helsinki - Machine: 0 is WORKING[...]
```

Some calculations:

- Res_Helsinki fails at clock 337.13031s

- fail duration is 0.11938 hours, equal to 0.11938 * 3600(sec) = 429.768s
- all machines of Res_Helsinki fail so the resource becomes unavailable
- failure will last 766.9132608756137s in fact 337.13031 + 429.768 = 766.89831s (approximately equal)
- but, the machines of the network restart to work at clock: 962.92039s

So Res_Helsinki has to be registered again to the ArtificialEU_Regional_GIS_1, and this is not immediately, we have to wait some seconds, so the resource doesn't become available at 962.92039s precisely.

from User_0 to User_9 the first 10 gridlets are submitted to Res_Dublin and the other 5 to Res_Glasgow. The two resources have the same features. We can analyze for example the trace file for User_8 (Listing 5.3) and see the sending time of the gridlet to the resource and when it is received, the difference of these two times is latency. The latency increases for each gridlet, this is due to a high number of gridlets, a computing power of machines that are not high (rating) or a low number of processors. This was demonstrated by testing on a network with only one user and one single resource (simple network) and studying the behavior:

- with 50 gridlets, 1 processor and a small machine rating the latency increases from first gridlet to the last
- with 50 gridlets, small machine rating but high number of processors, the latency decreases from first gridlet to the last.
- obviously, with 50 gridlets, high machine rating, high number of processors the latency decreases from first gridlet to the last and the value of latency is low.
- if the number of machines of the resource increases, obviously the latency is low and tends to decrease from first to last gridlet executed.

Listing 5.3: Helsinki fail: trace file User_8

```
Event   GridletID   Resource              GridletStatus   Clock

Sending  0  ArtificialEU_Res_Dublin   Created  686.3601319999988
Sending  1  ArtificialEU_Res_Dublin   Created  688.320131999997
Sending  2  ArtificialEU_Res_Dublin   Created  690.2801319999952
Sending  3  ArtificialEU_Res_Dublin   Created  692.0901319999936
Sending  4  ArtificialEU_Res_Dublin   Created  694.0201319999918
Sending  5  ArtificialEU_Res_Dublin   Created  695.98013199999
Sending  6  ArtificialEU_Res_Dublin   Created  697.7901319999884
Sending  7  ArtificialEU_Res_Dublin   Created  699.7201319999866
Sending  8  ArtificialEU_Res_Dublin   Created  701.6801319999848
Sending  9  ArtificialEU_Res_Dublin   Created  703.4901319999832
Sending10  ArtificialEU_Res_Glasgow  Created  705.4201319999814
```

```
Sending11  ArtificialEU_Res_Glasgow  Created  707.3801319999797
Sending12  ArtificialEU_Res_Glasgow  Created  709.190131999978
Sending13  ArtificialEU_Res_Glasgow  Created  711.1201319999763
Sending14  ArtificialEU_Res_Glasgow  Created  712.9601319999746
Receiving0ArtificialEU_Res_Dublin    Success  1629.6401920000346
Receiving1ArtificialEU_Res_Dublin    Success  2429.640191999994
Receiving2ArtificialEU_Res_Dublin    Success  3229.6401919998757
Receiving3ArtificialEU_Res_Dublin    Success  4029.6401919997575
Receiving4ArtificialEU_Res_Dublin    Success  4829.64019199964
Receiving5ArtificialEU_Res_Dublin    Success  5629.640191999521
Receiving6ArtificialEU_Res_Dublin    Success  6429.640191999403
Receiving7ArtificialEU_Res_Dublin    Success  7229.640191999285
Receiving8ArtificialEU_Res_Dublin    Success  8029.640191999167
Receiving9ArtificialEU_Res_Dublin    Success  8829.640191999522
Receiving10ArtificialEU_Res_GlasgowSuccess  9629.640191999995
Receiving11ArtificialEU_Res_GlasgowSuccess  10429.640192000468
Receiving12ArtificialEU_Res_GlasgowSuccess  11229.640192000941
Receiving13ArtificialEU_Res_GlasgowSuccess  12029.640192001414
Receiving14ArtificialEU_Res_GlasgowSuccess  12829.640192001887
```

In the graph 5.1 we can see that the latency always increases for all 15 gridlets submitted by User_8 to Res_Dublin and Res_Glasgow. Also, below (Listing 5.4) the output of User_8 regarding the CPU time, the cost of processing gridlet (cost = CPU Time * the cost of using this resource (that is set to 3$/s)), and the latency:

Listing 5.4: Helsinki fail: output User_8

```
============= OUTPUT for ArtificialEU_User_8 ==========
Gridlet ID   STATUS   Resource ID  Cost  CPU Time   Latency
    0        Success          51    6.0     2.0     943.2800600000359
    1        Success          51    6.0     2.0     1741.3200599999968
    2        Success          51    6.0     2.0     2539.3600599998804
    3        Success          51    6.0     2.0     3337.550059999764
    4        Success          51    6.0     2.0     4135.620059999648
    5        Success          51    6.0     2.0     4933.660059999532
    6        Success          51    6.0     2.0     5731.850059999415
    7        Success          51    6.0     2.0     6529.920059999298
    8        Success          51    6.0     2.0     7327.960059999182
    9        Success          51    6.0     2.0     8126.150059999539
   10        Success          56    6.0     2.0     8924.220060000014
   11        Success          56    6.0     2.0     9722.260060000488
   12        Success          56    6.0     2.0     10520.450060000963
   13        Success          56    6.0     2.0     11318.520060001438
   14        Success          56    6.0     2.0     12116.680060001912
==========================================================
```

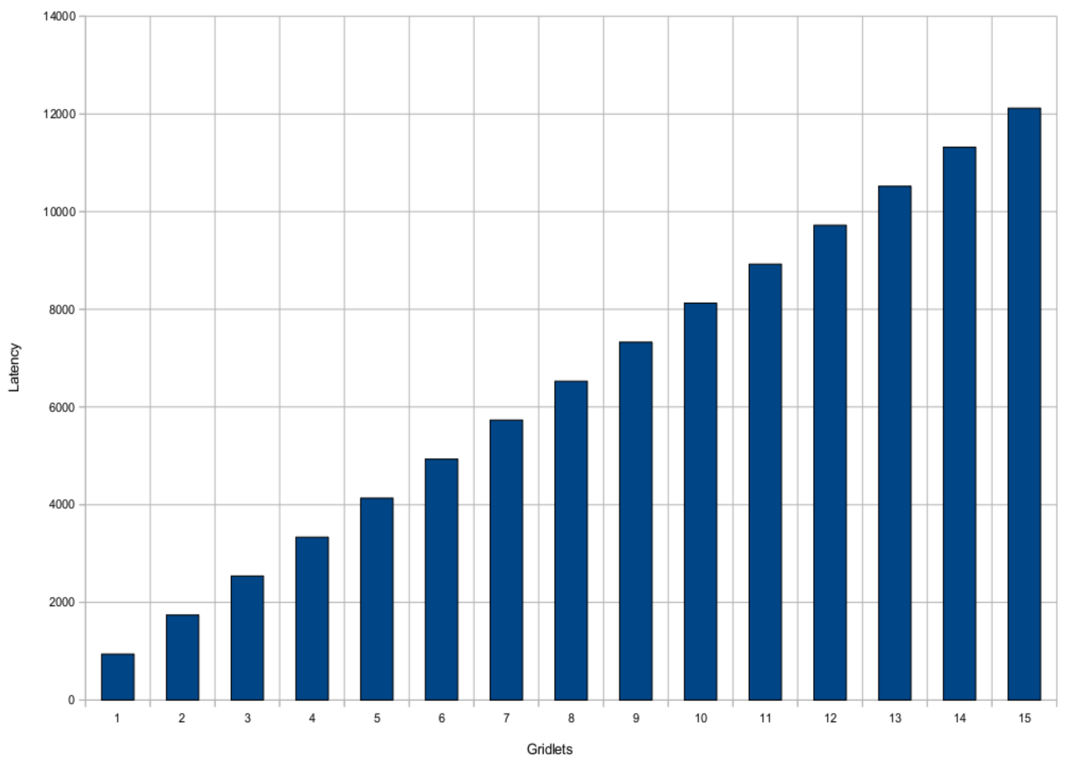

Figure 5.1.: Graph: User_8 latency

from User_10 to User_29 the first 10 gridlets are submitted to Res_Warsaw and the other 5 to Res_Vienna. The features are completly different in fact, Res_Warsaw is connected with Router4 with a low baud rate (0.0001 Gb) while Res_Vienna is connected with high baud rate (1 Gb), moreover Res_Warsaw has a rating equal to 1000 MIPS, while Res_Vienna has rating set to 49000 MIPS. We consider the User_23 and we examine the output (Listing 5.5).

Listing 5.5: Helsinki fail: output User_23

```
================ OUTPUT for ArtificialEU_User_23 =============
GridletID STATUS ResourceID Cost CPU Time    Latency

0  Success  86  1.86000000000376  0.62000000000125  2103.44023999965
1  Success  86  3.93000000000188  1.31000000000062  2077.19023999967
2  Success  86  0.4200000000050   0.14000000000169  1815.84027599965
3  Success  81       15.0                5.0        1791.8302759996768
4  Success  81       15.0                5.0        1482.270239999652
5  Success  81       15.0                5.0        1464.0102399996686
6  Success  81       15.0                5.0        1445.1402399996857
7  Success  81       15.0                5.0        1118.6102279999782
8  Success  81       15.0                5.0        1099.9602279999951
10 Success  86  4.6200119999698 1.5400039999899    664.54996399979
```

```
9  Success  81        15.0                  5.0             687.5499640002959
11 Success  86         6.0                  2.0             350.00996400028407
12 Success  86  0.3299999998762  0.1099999999587     330.87996399986
13 Success  86         6.0                  2.0             16.63000000036299
14 Success  86  2.4299999999220  0.80999999997402    15.5400000003392
==================================================================
```

The resourceID number 81 is Res_Warsaw, the resourceID 86 is Res_Vienna. The latency decreased gradually gridlet after gridlet, because there is a lot of computing power to process the gridlet, but it is evident that the gridlets submitted to Res_Warsaw have a cost greater than the cost of the gridlets submitted to Res_Vienna because the rating of Res_Warsaw is low.
After we do another simulation where the Res_Vienna (ResourceID=86) has rating equal to 100 MIPS. In Listing 5.6 we have the output for User_23 and we can see that with a low rating (100 MIPS) the CPU Time increases, and consequently the total cost.
For example, gridlet0 in Listing 5.5 has a cost equal to 1.86000000000376, in Listing 5.6, setting the gridlet of the resource to 100 MIPS the cost is 128.04000000000087. This is due to an increase of CPU time, because the time for processing the gridlet with a low latency is greater. The following listing (Listing 5.6) represent the output of User_23 using for Res_Vienna 100MIPS as rating:

Listing 5.6: Helsinki fail: output User_23 Res_Vienna rating 100 MIPS

```
================ OUTPUT for ArtificialEU_User_23 =============
GridletID STATUS ResourceID Cost      CPU Time           Latency
0  Success  86  128.04000000000087  42.68000000000029  2307.18027599974
1  Success  86  127.11000000000172  42.37000000000057  2284.23026399973
2  Success  86  127.50000000000136  42.50000000000045  2018.90023999974
3  Success  81        15.0                  5.0            1994.3602399997649
4  Success  81        15.0                  5.0            1681.9502399997425
5  Success  81        15.0                  5.0            1658.6502399997637
6  Success  81        15.0                  5.0            1345.3599640000443
7  Success  81        15.0                  5.0            1319.7200000000676
8  Success  81        15.0                  5.0            1006.6699640000459
9  Success  81        15.0                  5.0            979.3300000000706
10Success86  128.3399999999856   42.7799999999952  668.6499999998832
11Success86  127.85999999997512  42.61999999999170  642.2599639993073
12Success86  128.15999999998166  42.71999999999389  331.0099999998692
13Success86  128.129999999981    42.70999999999367  311.0599999994338
14Success86  127.07998799995812  42.35999599998604  47.69003200010229
==================================================================
```

Is interesting also to see that we submitted the first 10 gridlets to Res_Warsaw and the other 5 to Res_Vienna, but in our simulation (Listing 5.5) we see that the gridlets with id 0, 1, 2 are submitted to Res_Vienna. This is because, as mentioned above, when Res_Helsinki fails, the gridlets that had to be submitted to a certain resource, now are sent to the next resource until Res_Helsinki return to work.
In the simulation the first three gridlets are submitted while Res_Helsinki is out of order, in

fact are not submitted to Res_Warsaw but to Res_Vienna. This means that the first three gridlets are processed by a resource with greater power so the cost of these gridlets is less than the cost of gridlets 3, 4, 5, 6, 7, 8, 9 (refers to Graph 5.2).
In the graph the green bars refer to the gridlets submitted to Res_Vienna, and we can see that they are more or less the same height as the bars of the first 3 gridlets (in fact these last are submitted again to Res_Vienna). The high blue bars refer to the gridlets submitted to Res_Warsaw.

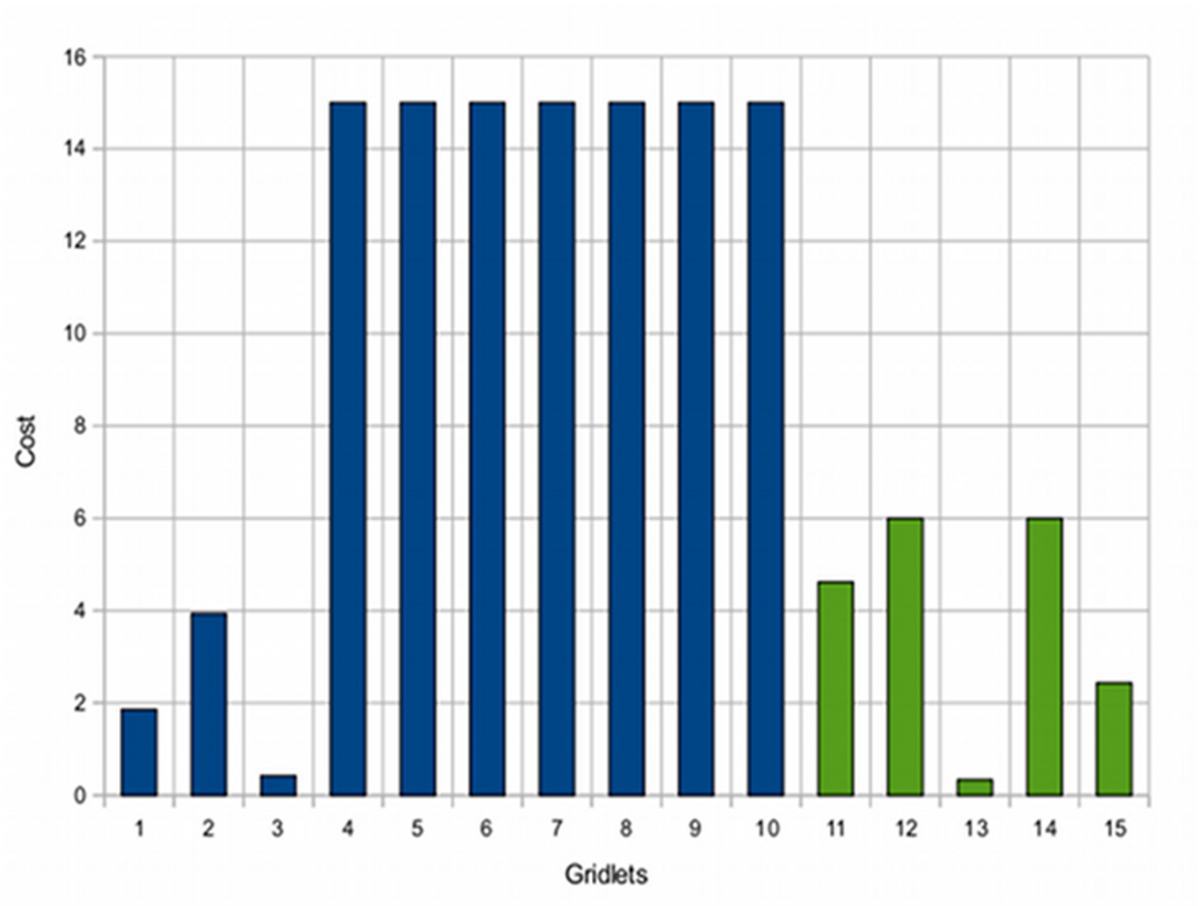

Figure 5.2.: Graph: cost of the gridlets

from User_30 to User_41 the first 10 gridlets are submitted to Res_Berlin and the other 5 to Res_Munich. The output for these users is very similar to the output of User_23. The first three gridlets are submitted to Res_Berlin while Res_Helsinki is out of order, so they are submitted to the next resource that is Res_Munich. As in Listing 5.5 these 3 gridlets have a low cost because Res_Munich has a rating equal 80000 MIPS, that is almost double the rating of Res_Berlin (49000 MIPS).

from User_42 to User_71 the first 10 gridlets are submitted to Res_Budapest and the other 5 to Res_Athens.

In this case all the gridlets are submitted while Res_Helsinki is down, according to our setting the gridlets will be submitted to the next resources, so the first 10 gridlets are submitted to Res_Athens and the other 5 to Res_Milan.
This improves the performance of the grid network, because Res_Budapest and Res_Athens have the same features, but are not very powerful (rating=700, number of machines = 10 and 12, baud rate of link Router8-resouce=0,0001 Gb), while Res_Milano has a greater rating (49000 MIPS), 70 machines and baud rate equal to 1Gb.

Listing 5.7: Helsinki fail: output User_45

```
=============== OUTPUT for ArtificialEU_User_45 ============
GridletID STATUS ResourceID  Cost    CPU Time         Latency
0         Success  106        21.0      7.0        2693.700179999003
1         Success  106        21.0      7.0        2663.700179999030
2         Success  106        21.0      7.0        2412.430239998969
3         Success  106        21.0      7.0        1980.319999999328
4         Success  106        21.0      7.0        1955.049999999351
5         Success  106        21.0      7.0        1641.639999999629
6         Success  106        21.0      7.0        1616.829999999652
7         Success  106        21.0      7.0        1175.699999999929
8         Success  106        21.0      7.0        1150.629999999382
10        Success  111  2.760047999  0.920015999  670.449999999
9         Success  106        21.0      7.0        695.560000000366
11        Success  111         6.0      2.0        271.2900000007485
12        Success  111         6.0      2.0        241.8800000001065
13        Success  111  3.929999999  1.309999999  35.52000000077
14        Success  111         6.0      2.0        22.59000000049309
=========================================================
```

Listing 5.7 reports the output for User_45. The gridlets submitted to Res_Athens (ResourceID = 106) have a greater latency than gridlets submitted to Res_Milan (ResourceID = 111). Every machine of the resources has four processors so gridlet after gridlet the latency decreases. Obviously the gridlets submitted to Res_Athens have a cost greater than gridlets submitted to Res_Milan, this is due to the different features of the resources.

from User_72 to User_81 the first 10 gridlets are submitted to Res_Milan and the other 5 to Res_Pisa. Like in the group of users before, also here the gridlets are submitted while Res_Helsinki is down. So the gridlets will be submitted to Res_Pisa and to Res_Madrid, the output cost for these 10 users is always the same, obviously the gridlets submitted to Res_Pisa have a greater cost due to the low rating of the resource (1000 MIPS).

from User_81 to User_101 the first 10 gridlets are submitted to Res_Madrid and the other 5 to Res_Lisbon. It is the same discourse of the previous users. The gridlets are submitted while Res_Helsinki is down. So the first 10 gridlets will be submitted to Res_Lisbon and the other 5 to Res_Paris (that is situated in another RegionalGIS).
The gridlets submitted to Res_Lisbon have a greater cost then gridlets sent to Res_Paris, this is usually because the rating of Res_Lisbon is low. Moreover, the connection between Res_Lisbon and Router10 has a low baud rate (0,00001 Gb/s).

from User_102 to User_113 all the gridlets are submitted to Res_Paris, but these gridlets are sent from the users when Res_Helsinki is down. According to our setting the gridlets will be submitted to the next resource that is Res_Brussels. This resource and Res_Paris have the same features so there isn't an increase of latency or cost. In Listing 5.8 we report the ouput for the user 103, the ouputs for User_102, User_103, ..., User_113 are the same.

Listing 5.8: Helsinki fail: output User_103

```
 ================ OUTPUT for ArtificialEU_User_103 ===========
Gridlet ID    STATUS   Resource ID  Cost  CPU Time      Latency
    0         Success         136     6.0     2.0     23943.31235200712
    1         Success         136     6.0     2.0     4670.752112000744
    2         Success         136     6.0     2.0     4372.891848000698
    3         Success         136     6.0     2.0     3937.561584000723
    4         Success         136     6.0     2.0     3640.301320000580
    5         Success         136     6.0     2.0     3209.501320000607
    6         Success         136     6.0     2.0     3461.948984002796
    7         Success         136     6.0     2.0     3991.14898400339
    8         Success         136     6.0     2.0     4653.888720003815
    9         Success         136     6.0     2.0     5183.088720004409
   10         Success         136     6.0     2.0     5845.828456004834
   11         Success         136     6.0     2.0     6375.028456005428
   12         Success         136     6.0     2.0     7037.768192005853
   13         Success         136     6.0     2.0     7566.968192005177
   14         Success         136     6.0     2.0     8229.707928002765
==========================================================
```

We try also to modify the features of Res_Brussels in this way (Table 5.2):

	N° machines	rating
original	50	49000
modified	5	50

Table 5.2.: Res_Brussels: modified features

The result, as seen in the previous example (Listing 5.6) is that the cost for processing the gridlet by the resource increases significantly: the CPU time for each gridlet becomes equal to 85 and consequently the cost is equal to 255.

from User_114 to User_124 the first 10 gridlets are submitted to Res_Paris and the other 5 to Res_Helsinki. The gridlets are submitted while Res_Helsinki is out of order, so the gridlets will submitted to Res_Brussels and Res_Oslo. The feautures of these 2 resources are the same but the latency of the first 10 gridlets is higher than the last 5 because in the same time at Res_Brussels are submitted also the gridlets from User_102, User_103, ..., User_113, while Res_Oslo is "free", so the latency is minor.

5.2. Artificial EU Grid: set of failure

In this section we simulate the "Artificial EU Grid" with, not one, but a set of failures. It is a little more difficult than the scenario before (Chapter 5.1) because we have a lot of users and gridlets, and so it is complicated to manage the resources that fail and check the output file. This section, like the previous scenario, describes for each group of users the behavior of the grid network (resources, machines, gridlets).

For the network topology we use the same file network_thesis.txt (Listing 5.1) that we use for the previous scenario.
The resources that fail are distribuited in all "Artificial EU Grid" and one machine fails for each ArtificialEU_Regional_GIS, except for ArtificialEU_Regional_GIS_1 where 2 resources fail (refers to Table 5.3):

Resource Name	**Regional GIS**
Res_Dublin	ArtificialEU_Regional_GIS_0
Res_Helsinki	ArtificialEU_Regional_GIS_1
Res_Warsaw	ArtificialEU_Regional_GIS_1
Res_Munich	ArtificialEU_Regional_GIS_2
Res_Pisa	ArtificialEU_Regional_GIS_3
Res_Brussels	ArtificialEU_Regional_GIS_4

Table 5.3.: Resources that fail

For setting this failure we modified the java file "RegionalGISWithFailure.java" (Chapter 3.3.5) so that we can decide the resource that fails based on the Table 5.3. The Listing 5.9 shows the block If-Then, use for set the "res_num", that is the index of the resource in the RegionalGIS that we want to fail. The Listing 5.10 shows the block Switch-Case in java file "ArtificialEUGrid.java" that we used for setting for each RegionalGIS the number of the resources that fail and the number of machines involved in the failure (in our case all machines in the resource fail so the resource will be out of order).

Listing 5.9: RegionalGISWithFailure.java: block If-Then

```
if (gisName.equals("ArtificialEU_Regional_GIS_0")) {res_num=0;}
       \*fails Dublin
if (gisName.equals("ArtificialEU_Regional_GIS_1"))
       {cont=cont + 1;
       if (cont== 1) {res_num=0;}     \*fails Helsinki
          else {res_num=3;}} \*fails Warsaw
if (gisName.equals("ArtificialEU_Regional_GIS_2")) {res_num=1;} \*
   fails Munich
if (gisName.equals("ArtificialEU_Regional_GIS_3")) {res_num=1;} \*
   fails Pisa
if (gisName.equals("ArtificialEU_Regional_GIS_4")) {res_num=1;} \*
   fails Brussels
```

Listing 5.10: ArtificialEUGrid.java: block Switch-Case

```
switch (i){
case 0: failureNumResPattern=1; /* 1 Res fails (Dublin)
        failureNumMacPattern=30; /* all machines of Dublin fail
            break;

case 1: failureNumResPattern=2; /*2 Res fails (Helsinki-Warsaw)
        failureNumMacPattern=60; /*all machines of Helsinki and
             Warsaw fail
              break;

case 2: failureNumResPattern=1;   /* 1 Res fails (Munich)
        failureNumMacPattern=100; /* all machines of Munich
            fail
              break;

case 3: failureNumResPattern=1;   /* 1 Res fails (Pisa)
        failureNumMacPattern=40; /* all machines of Pisa fail
              break;

case 4: failureNumResPattern=1; /* 1 Res fails (Brussels)
        failureNumMacPattern=50; /* all machines of Brussels
            fail
              break;
}
```

In this simulation we submitted the gridlet to certain resources. In the previous scenario if Res_Helsinki failed we submitted the gridlets to the next resource. It is not the same in this scenario with a set of failures. In this case if we send the gridlets to resource with index equal to 5 (Res_ Moscow, refers to Figure 4.2), we have to be careful that there aren't fail resources with less index.
If there is a resource out of order with a smaller index than which we consider, the gridlets will be submitted to the resource after the resource considered. If there are 2 resources out of order with a smaller index than which we considerd, the gridlets will be submitted 2 resources after the resource considered. For example, refer always to Figure 4.2, Res_Dublin has index equal to 0 (the index depends on the order in which the resources are written in our java code), this resource and Res_Helsinki are down. Want to submit the gridlet to Res_Vienna (normaly index equal to 7), so we have to submit them not to the resource with index equal to 7 but to the resource with index equal to 5, because Res_Dublin and Res_Helsinki aren't in the list of the available resources.
Table 5.4 shows how the gridlets are submitted to the resources. Note that in this simulation the gridlets are submitted to different resources and the users from #71 to #101 are gathered in the same groups, like the users from 102 to 124 for a better understanding of the simulation output file. Anyway the gridlets are submitted to resources in all RegionalGIS. In the next section, we can show with the results of the simulation, that due to the failure of 6 resources, the gridlets will be submitted to other resources (following the index specified).

From *User* to *User*	**submitted first 10 to (index)**	**submitted last 5 to (index)**
from User_0 to User_9	Res_Dublin (0)	Res_Dublin (0)
from User_10 to User_29	Res_Glasgow (1)	Res_Helsinki (2)
from User_30 to User_41	Res_Oslo (3)	Res_Stocklom (4)
from User_42 to User_71	Res_Warsaw (6)	Res_Vienna (7)
from User_72 to User_101	Res_Berlin (8)	Res_Munich (9)
from User_102 to User_124	Res_Budapest (10)	Res_Dublin (0)

Table 5.4.: Artificial EU Grid, set of failure: gridlets submitted to resources

5.2.1. Results of simulation

This section presents the results of the simulation when more than one resource fails in the network. As described in the previous scenario, we analyze the simulation results considering the main aspects that describe the behaviour of the grid network.
As described in the Table 5.3, 6 resources fail, so we have to analyze for each group of users where the gridlets are submitted effectively, and how the output file changes for the users.

from User_0 to User_9, all the gridlets are submitted to Res_Dublin. But according to our setting (refers to Table 5.3) this resource fails, so all gridlets will be submitted to Res_Glasgow. This resource has the same features of Res_Dublin so the output doesn't change much.
If we analyze the output for these 10 users we can see that, as mentioned, the gridlets will be submitted to the resource with id equal to 56 (Res_Glasgow), accordingly the cost for each gridlet is equal to 6 (remember: cost = cost using the resource (set to 3) * CPU Time, cost=3*2 = 6s). This cost is low, this happen when the power of resource's machines is higher than the load of work.

The latency (sending time - receiving time) for the ten users is quite high, this is because the baud rate between router0 and router1 is very low (like 0.0001 Gb/s, refers to Listing 5.1). Normaly, the latency from the first gridlets to the last gridlet submitted increases, but for User_1, User_7 and User_8 it decreases, this is due to the fact that the gridlets are submitted in different time (first gridlet0, then gridlet1, ecc....) but sometimes when we have many gridlets, these are stopped in a queue and are received from the user in the same time. In our case we have to consider 465 gridlets that are submitted to Res_Glasgow: 15 for each of the ten first users, 5 for each of 23 users (from User_102 to User_124) and another 10 gridlets for each user from the 10th to 29th.

from User_10 to User_29, the first ten gridlets are submitted to Res_Glasgow and other 5 to Res_Helsinki. Before the submission of gridlets, Res_Dublin and Res_Helsinki fail. So we have to check the index set to submit the gridlets, and send these to the available resource with the corresponds index.
In our case (refers to Table 5.4), the index for the first 10 gridlets is equal to 1, that corrispond, in the list of available resources to Res_Oslo, while the last 5 gridlets are submitted to the resource with index equal to 2 that is Res_Stocklom. In this way we have a small improvement of the performance for the gridlets that were submitted to Res_Glasgow, in

fact we are now sending to Res_Oslo, which has more machines (20), so it can manage more gridlets in the same time. Listing 5.11 shows the output of one of the 20 users, all of their output are very similar, changing only some decimal values. The cost of each gridlet is very very low, this is due to the high number of machines and high rating of the both resources. Listing 5.12 shows the output of the same user as before, but the Res_Dublin is not out of order, so index equal to 1 now corresponds to Res_Glasgow.
Comparing these outputs, it is clear that in the second, the first 10 gridlets have a higher cost due to the less machines of Res_Glasgow and the latency is higher because as mentioned before, many gridlets are submitted to it.
So if the failure of the resource is managed correctly, it does not always involves an aggravetion of grid network performance.

Listing 5.11: Set of fails: output User_16

```
================ OUTPUT for ArtificialEU_User_16 ============
GridletID STATUS ResourceID Cost        CPU Time        Latency
 0       Success 66 4.2600000000015 1.4200000000005 829.3602879996
 1       Success 66 0.5100000000049 0.1700000000016 824.7502879996
 2       Success 66 1.2900000000042 0.4300000000014 820.1702519996
 3       Success 66 2.8800000000028 0.9600000000009 815.7002879996
 4       Success 66 1.8600000000037 0.6200000000012 811.2102879996
 5       Success 66 1.6500000000039 0.5500000000013 806.7302879996
 6       Success 66 1.2000000000043 0.4000000000014 802.3002879996
 7       Success 66 0.9000000000046 0.3000000000015 797.8802879996
 8       Success 66 2.1600000000034 0.7200000000011 793.4002879996
10       Success 71 3.1800000000025 1.0600000000008 17.75999999998
 9       Success 66 1.4700000000041 0.4900000000013 26.30003599997
11       Success 71 2.2800000000033 0.7600000000011 73.91003599993
12       Success 71 1.8000000000038 0.6000000000012 86.71003599992
13       Success 71 0.5400000000049 0.1800000000016 95.04003599991
14       Success 71 0.7500000000047 0.2500000000015 103.9800359999
=============================================================
```

Listing 5.12: Set of fails: output User_16 (Res_Dublin works)

```
================ OUTPUT for ArtificialEU_User_16 ============
GridletID STATUS ResourceID Cost        CPU Time        Latency
10       Success 66 0.2700000000052 0.0900000000017 211.13000000002
11       Success 66 0.3300000000051 0.1100000000017 208.13000000002
12       Success 66 0.3300000000051 0.1100000000017 205.13000000002
13       Success 66 1.2300000000043 0.4100000000014 21.520035999980
14       Success 66 4.9500000000009 1.6500000000003 32.240035999970
 0       Success 56     6.0                   2.0       10432.660056000
 1       Success 56     6.0                   2.0       14770.009792003
 2       Success 56     6.0                   2.0       16367.009792004
 3       Success 56     6.0                   2.0       17963.899792005
 4       Success 56     6.0                   2.0       19560.839792006
 5       Success 56     6.0                   2.0       21157.839792007
```

```
6       Success  56     6.0                      2.0         22754.839792008
7       Success  56     6.0                      2.0         24351.839792009
8       Success  56     6.0                      2.0         25948.839792010
9       Success  56     6.0                      2.0         27545.839792011
==================================================================
```

from User_30 to User_41 the first 10 gridlets are submitted to Res_Oslo and the last to Res_Stocklom. These are submitted while 3 resources are out of order:

1. Res_Dublin
2. Res_Helsinki
3. Res_Warsaw

So we have to send the gridlets to the resource indicated by index (in GridUserFailure.java), selecting it from the set of available resouces. In this case the indeces are 3 and 4, so the first 10 gridlets will be submitted to Res_Moscow and the other 5 to Res_Vienna.

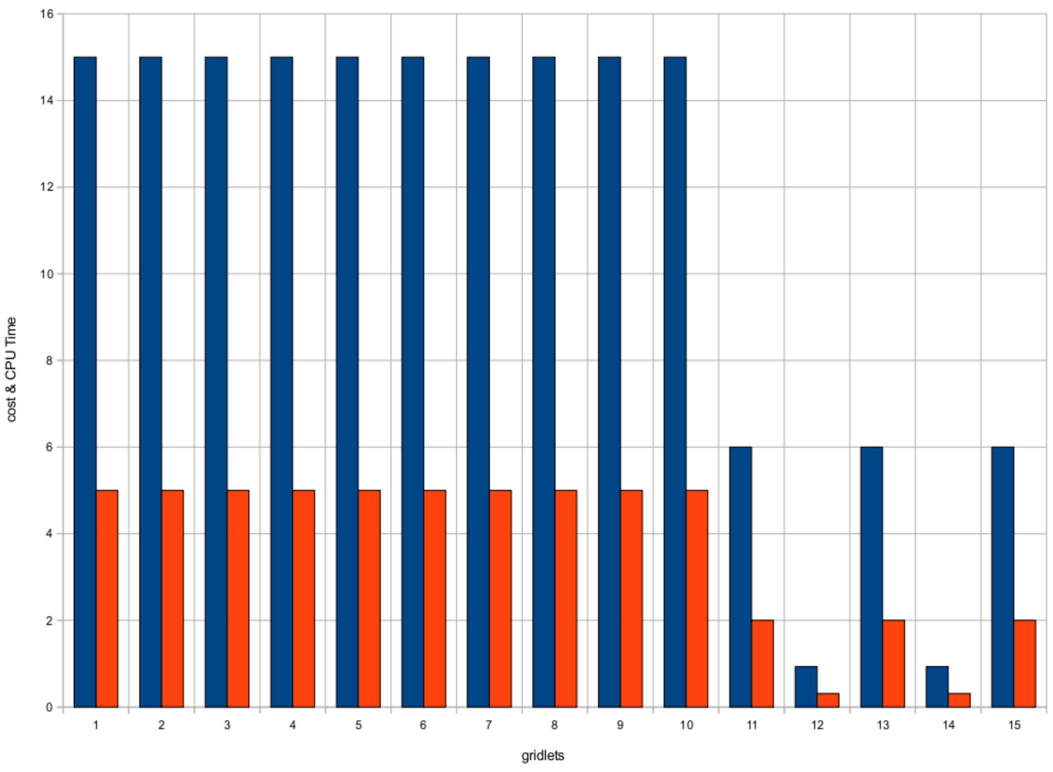

Figure 5.3.: Graph: User_40, cost CPU Time

Performance worsens because these two resources have fewer machines and have lower rating then Res_Oslo and Res_Stockolm. The graph 5.3 explains very well the situation of the CPU time and cost of each gridlet for User_40 (it is a random choice, the output of these 12 users are similar). Res_Moscow and Res_Vienna have the same number of machines

(20) but this last has a rating equal to 49000 MIPS, instead the first is equal to 1000MIPS. We expected that the cost of gridlets submitted to Res_Moscow is greater than the cost of gridlets sent to Res_Vienna. Graph 5.3 confirms that, the blue bar is the cost and the orange bar is the CPU time.

More precisly, the first 10 gridlets have a CPU Time equal to 5s and consequently the cost is 15s, while the cost for processing gridlet_10, gridlet_12 and gridlet_14 is equal to 6s, and the cost for processing the other 2 gridlets is very low and is equal to 0.93s.

from User_42 to User_71 the first 10 gridlet are submitted to Res_Warsaw and the last 5 are submitted to Res_Vienna. The gridlets are submitted while 4 resources are down, these are:

- Res_Dublin
- Res_Helsinki
- Res_Warsaw
- Res_Munich

The indices of resources where these users submitted the gridlets are set in the Java code and are: index equal to 6 for the first 10 gridlet and index equal to 7 for the other gridlets. In the list of available resources these 2 indeces correspond to Res_Budapest and Res_Athens. These two resources have a low rating (700 MIPS) and are connected with the router8 with a baud rate equal to 0.0001 Gb/s, the latency will be high.
The cost for executing the gridlet is equal to 21s, we can compare this value with the cost for processing gridlets by Res_Moscow in the group of previous users (from User_30 to User_41). As we said before, the lower the rating of the resource, the higher the cost of processing the gridlets will be. Res_Moscow has a rating equal to 1000 MIPS, so not much higher than Res_Budapest and Res_Athens, in fact the cost for processing one gridlet is equal 15s, a little less than the cost of processing gridlets (21s) for the two resources that we are considering.

From the output Listing 5.13 it is possible to note that when the gridlets are submitted to a resource connected with the router with a low baud rate, the latencies tend to decrease[3] gridlet by gridlet, but is value is high.

Listing 5.13: Set of fails: output User_64

```
================ OUTPUT for ArtificialEU_User_64 ============
Gridlet ID STATUS  Resource ID  Cost  CPU Time    Latency
    0      Success      101     21.0    7.0     3104.9903479977493
    1      Success      101     21.0    7.0     3079.770347997772
    2      Success      101     21.0    7.0     2725.79008399775
    3      Success      101     21.0    7.0     2699.9600839977734
    4      Success      101     21.0    7.0     2269.3500839981652
    5      Success      101     21.0    7.0     2245.430083998187
```

[3]After several simulations based also on simple scenarios we can say that with a low baud rate connection, the gridlets submitted by a user to a resource have a high latency, but this tends to decrease gridlet after gridlet. From the trace file of the user it is possible to note that the gridlets are sent one after the other and the "sending time" everytime is increases. After they are processed and finally received by the users in the same "receiving time"

```
    6        Success          101      21.0      7.0      1795.1600839985895
    7        Success          101      21.0      7.0      1769.8300839986125
    8        Success          101      21.0      7.0      1306.9200839988625
    9        Success          101      21.0      7.0      1281.240083998302
    10       Success          106      21.0      7.0      831.9700839992979
    11       Success          106      21.0      7.0      806.6500839987452
    12       Success          106      21.0      7.0      475.45003599956135
    13       Success          106      21.0      7.0      507.0797479987955
    14       Success          106      21.0      7.0      296.9897119995694
==================================================================
```

from User_72 to User_101 the first 10 gridlets are submitted to Res_Berlin and the last to Res_Munich. These are submitted while 5 resources are out of order:

- Res_Dublin
- Res_Helsinki
- Res_Warsaw
- Res_Munich
- Res_Pisa

The index is setting equal to 8 for the first ten gridlets and 9 for the 5 gridlets so the resource with index 8 in the set of available resources is Res_Milano and with index 9 is Res_Madrid. Referring to the Figure 4.2 it is simple to know at which resource the gridlets will be submitted when a set of fails happens. In this case some gridlets will be submitted to Res_Madrid, and if we look at the network_thesis.txt (refers to Listing 5.1) we can note that the ways to arrive to Res_Madrid, i.e. router11-router10 and router9-router10 have a lower baud rate than the other. This cause an increase of the latency of gridlets submitted to this resource.

Surely we have a worsening performance, because without failure the gridlets would send to Res_Berlin and Res_Munich that have more machines than the actual resource, moreover as already mentioned Res_Madrid is connected with the router with a low baud rate. Listing 5.14 shows the output file of User_100.

Listing 5.14: Set of fails: output User_100

```
================== OUTPUT for ArtificialEU_User_100 ============
Gridlet  ID STATUS   Resource ID   Cost   CPU Time     Latency
   0       Success       111        4.2     1.40     274.1702639999837
   1       Success       111        2.01    0.67     269.52026399998795
   2       Success       111        2.01    0.67     264.8502639999922
   3       Success       111        2.28    0.76     260.4402639999962
   4       Success       111        1.89    0.63     256.09026400000016
   5       Success       111        1.56    0.52     251.73026400000413
   6       Success       111        2.55    0.85     247.43026400000804
   7      Success     111      2.37    0.79     243.09026400001198
   8      Success     111      3.78    1.26     29.57981999973981
   9      Success     111      4.56    1.52     38.21981999973195
  10      Success     121      6.0     2.0      1824.1000159999471
  11      Success     121      6.0     2.0      3444.9600159997117
  12      Success     121      6.0     2.0      5041.690015999479
  13      Success     121      6.0     2.0      6640.810015999246
  14      Success     121      6.0     2.0      8282.49001599977
==================================================================
```

In Listing 5.14, we can note that the gridlets submitted to the Res_Madrid (Resource ID equal to 121) have a latency greater than the latency of the gridlets submitted to Res_Milano (Resource ID equal to 111), this is due to low baud rate between Res_Madrid and the router. Moreover, Res_Milano has 30 machines more than Res_Madrid, and, as we can note in the next group of users, more gridlets are submitted at Res_Madrid. So if we compare the cost of processing gridlet we can note that it is very low (between 2s and 4s) for the gridlets submitted to Resource ID 111, and it is equal to 6 for gridlets submitted to Res_Madrid.

from User_102 to User_124 the first 10 gridlets are submitted to Res_Budapest and the last to Res_Dublin. These are submitted while 6 resources are out of order:

- Res_Dublin
- Res_Helsinki
- Res_Warsaw
- Res_Munich
- Res_Pisa
- Res_Brussels

In this case the indices are 10 and 0, in the list of available resources these indexes correspond to Res_Lisbon and Res_Glasgow. If we examine the output of the simulation we can note that at different times Res_Dublin, Res_Helsinki and Res_Pisa receive a GRIDRESOURCE_RECOVERY signal.
Some of these last gridlets are submitted to Res_Lisbon before the recovery of Res_Pisa, but some are submitted after the recovery, so Res_Pisa becomes an available resource, then index equal to 10 will not be Res_Lisbon but Res_Madrid.

Table 5.5 lists the users that submitted gridlets before and after the failure of Res_Pisa, note that here the gridlets 7, 8, 9 are submitted after the recovery, gridlet 6 is submitted before.
Table 5.6 lists the users that submitted gridlets before and after the failure of Res_Pisa, note that here the gridlets 6, 7, 8, 9 are submitted after the recovery, while Table 5.7 lists the users that submitted the gridlets before the recovery of Res_Pisa .

Artificial EU User
ArtificialEU_User_107
ArtificialEU_User_112
ArtificialEU_User_113

Table 5.5.: List of users that submitted gridlets before and after recovery of Res_Pisa (gridlets 7, 8, 9 submitted after the recovery)

Artificial EU User
ArtificialEU_User_102
ArtificialEU_User_103
ArtificialEU_User_104
ArtificialEU_User_105
ArtificialEU_User_106
ArtificialEU_User_108
ArtificialEU_User_109
ArtificialEU_User_110
ArtificialEU_User_111

Table 5.6.: List of users that submitted gridlets before and after recovery of Res_Pisa (gridlets 6, 7, 8, 9 submitted after the recovery)

Artificial EU User
ArtificialEU_User_114
ArtificialEU_User_115
ArtificialEU_User_116
ArtificialEU_User_117
ArtificialEU_User_118
ArtificialEU_User_119
ArtificialEU_User_120
ArtificialEU_User_121
ArtificialEU_User_122
ArtificialEU_User_123
ArtificialEU_User_124

Table 5.7.: List of users that submitted gridlets before recovery of Res_Pisa

Listing 5.15 shows the output file for User_121 (Table 5.7). As we expected the first 10 gridlets that are submitted to Res_Lisbon have a greater cost for processing the gridlet. This is because the resource has a rating of 700 MIPS, that is low. Also the latency is quite high because Res_Lisbon and Res_Madrid can be reached only with router10-router11 way and router9-router11 way. These two ways have a low baud rate that causes the increase of

latency.
The other 5 gridlets are submitted to Res_Glasgow that has a rating of 49000 MIPS in fact the cost of processing gridlets is lower than the first 10 gridlets. Obviously the latency is high because also to reach Res_Glasgow the gridlets have to pass from router0-router1 way that has a low baud rate.

Listing 5.15: Set of fails: output User_121

```
================ OUTPUT for ArtificialEU_User_121 ============
Gridlet ID   STATUS   Resource ID   Cost   CPU Time   Latency
     0       Success     126        21.0     7.0     1074.5100200000425
     1       Success     126        21.0     7.0     1950.1300199999737
     2       Success     126        21.0     7.0     2825.510019999848
     3       Success     126        21.0     7.0     3700.9500199997224
     4       Success     126        21.0     7.0     4576.630019999597
     5       Success     126        21.0     7.0     5452.24001999947
     6       Success     126        21.0     7.0     6327.890019999344
     7       Success     126        21.0     7.0     7203.560019999218
     8       Success     126        21.0     7.0     7867.079755999486
     9       Success     126        21.0     7.0     8742.18975600001
    10       Success      56         6.0     2.0     11746.893513335135
    11       Success      56         6.0     2.0     12622.553513335659
    12       Success      56         6.0     2.0     13498.063513336185
    13       Success      56         6.0     2.0     14373.623513336708
    14       Success      56         6.0     2.0     15249.603513337232
==============================================================
```

Listing 5.16 shows the output file for User111 (Table 5.6). We can see that the first six gridlets are submitted before the recovery of Res_Pisa so are sent to Res_Lisbon, like User_121, the cost is equal to 21s and the latency is quite high.
The gridlet 6, 7, 8, 9 are submitted after the recovery of Res_Pisa so are send to Res_Madrid. So the performance increases because this resource has the same machines of Res_Lisbon but it has a rating of 49000 MIPS (refers to Table 4.1), in fact the cost for processing gridlet is lower than the first six gridlets (6s). The latency is quite high because the ways for the resource have a low baud rate.
The last five gridlets are submitted to Res_Glasgow, and it doesn't matter if Res_Pisa is out of order or is recovered, because these gridlets are sent to a resource that has an index smaller than its own. Res_Madrid and Res_Glasgow have the same baud rate in fact the cost is always 6, the latency for these 5 gridlets is quite high because also Res_Glasgow is reached with a way that has a low baud rate.

Listing 5.16: Set of fails: output User_111

```
================ OUTPUT for ArtificialEU_User_111 ============
Gridlet ID   STATUS   Resource ID   Cost   CPU Time   Latency
     0       Success     126        21.0     7.0     16497.081316002503
     1         Success   126        21.0     7.0     9363.311052003324
     2         Success   126        21.0     7.0     3793.650792000277
     3         Success   126        21.0     7.0     3568.3905280001673
```

```
   4        Success   126     21.0    7.0    3274.260264000115
   5        Success   126     21.0    7.0    3011.660264000069
   6        Success   121      6.0    2.0    2786.3999999999596
   7        Success   121      6.0    2.0    2402.400000000014
   8        Success   121      6.0    2.0    2092.19999999997
   9        Success   121      6.0    2.0    1821.5999999999494
  10        Success    56      6.0    2.0    2011.1724773356673
  11        Success    56      6.0    2.0    2692.1724773362093
  12        Success    56      6.0    2.0    3226.1724773368114
  13        Success    56      6.0    2.0    3907.1724773373535
  14        Success    56      6.0    2.0    4459.182077337962
==============================================================
```

In the Table 5.8 we can compare the time, when the gridlet is submitted, and when Res_Pisa is recovered. We report only the first ten gridlets because the other 5 are submitted to Res_Glasgow, so are not interested in the recovery of Res_Pisa.
Highlighted in red, are the times of the gridlets submitted before the recovery of Res_Pisa, while higlighted in green, are the time of the gridlets sent after the recovery of the resource, so submitted to Res_Madrid.

N° gridlet	**Time gridlet submission (s)**	**Time Res_Pisa recovery(s)**	**submitted to**
0	758.3201	14352.404	ArtificialEU_Res_Lisbon
1	7892.0903	14352.404	ArtificialEU_Res_Lisbon
2	13461.7506	14352.404	ArtificialEU_Res_Lisbon
3	13687.010	14352.404	ArtificialEU_Res_Lisbon
4	13981.141	14352.404	ArtificialEU_Res_Lisbon
5	14243.741	14352.404	ArtificialEU_Res_Lisbon
6	14469.001	14352.404	ArtificialEU_Res_Madrid
7	14853.001	14352.404	ArtificialEU_Res _Madrid
8	15163.201	14352.404	ArtificialEU_Res _Madrid
9	15433.801	14352.404	ArtificialEU_Res _Madrid

Table 5.8.: Artificial EU Grid, set of failure: comparison of gridlets submission time and Res_Pisa time recovery

5.3. Artificial EU Grid: set of failure with custom time of failure

In the previous simulations, we made some tests on "Artificial EU Grid" varying the number of resources that fail, and setting the various parameters of the resources' machines. The only parametres that we don't vary are the intial time for submitted gridlets, the time when a resource fails and how many time the resources are out of order. In this section we want to customize these parameters that before are setted in a random way. In the chapter 3.3.3 we have already mentioned the changes to Java code for setting these parameters.

5.3.1. First testbed: high duration of resources failure

In the previous simulation, Chapter 5.2.1, from User_102 to User_124 the gridlets are submitted to Res_Lisbon, Res_Glasgow but for some users also to Res_Madrid, because these gridlets are submitted after the recovery of Res_Pisa.
In this simulation we want to change the fail duration time of Res_Pisa so that all users from User_102 to User_124 submitted their gridlets to Res_Lisbon and Res_Glasgow. Obviously, the resources that fail are the same of the previous simulation (Chapter 5.2). To do this we set the variable "failureLengthPattern" equal to 10 hours, so we are sure that the gridlets submition is before GRIDRESOURCE_RECOVERY signal. The resources, as specified in Listing 5.17, fail after 180s (3 minutes) and the gridlets are submitted after 15 minutes (900s, this value is set with the variable init_time in the java code).

Listing 5.17: Set of fails: modify the time parameters for resource failure

```
switch (i){
 case 0: failureNumResPattern=1;
           failureNumMacPattern=30;
           failureTimePattern = 180;//when the resource fails(s)
           failureLengthPattern = 10*3600;//for 10 hours is out
              of order
              break;

 case 1: failureNumResPattern=2;
           failureNumMacPattern=60;
           failureTimePattern = 180;//when the resource fails(s)
           failureLengthPattern = 10*3600;//for 10 hours is out
              of order
              break;

 case 2: failureNumResPattern=1;
           failureNumMacPattern=100;
           failureTimePattern = 180;//when the resource fails(s)
           failureLengthPattern = 10*3600;//for 10 hours is out
              of order
              break;

 case 3: failureNumResPattern=1;
           failureNumMacPattern=40;
           failureTimePattern = 180;//when the resource fails(s)
           failureLengthPattern = 11*3600;//for 10 hours is out
              of order
              break;

 case 4: failureNumResPattern=1;
           failureNumMacPattern=50;
           failureTimePattern = 180;//when the resource fails(s)
           failureLengthPattern = 10*3600;//for 10 hours is out
```

```
                    of order
                    break;
}
```

Checking the output file, we can see that there aren't GRIDRESOURCE_RECOVERY messages, this means that all gridlets are submitted while the resources (that we set as fail) are out of order, and the simulation ends before a recovery of any resource. So it is deducible that the processing of all gridlets in the networs takes less than 10 hours. In fact the last gridlet processed is received at time 22573.79s that is equal almost to 6 hours.
Graph 5.4 shows the comparison of the User_111 cost when the Res_Pisa is recovered (last simulation) and when the Res_Pisa is not recovery (this simulation). We can note that the cost is always the same except for the gridlets 6, 7, 8, 9 that in this simulation are sent again to Res_Lisbon, so the cost for processing these gridlets is higher than the cost with the recovery of Res_Pisa. Moreover we can note that these four gridlets when the resource is recovered are submitted to Res_Madrid, that, as we have already mentioned, has the same rating of Res_Glasgow, thus the same cost (6s).

Listing 5.18 shows the output of User_111 without the recovery of Res_Pisa, all first ten gridlets are submitted to Res_Lisbon.

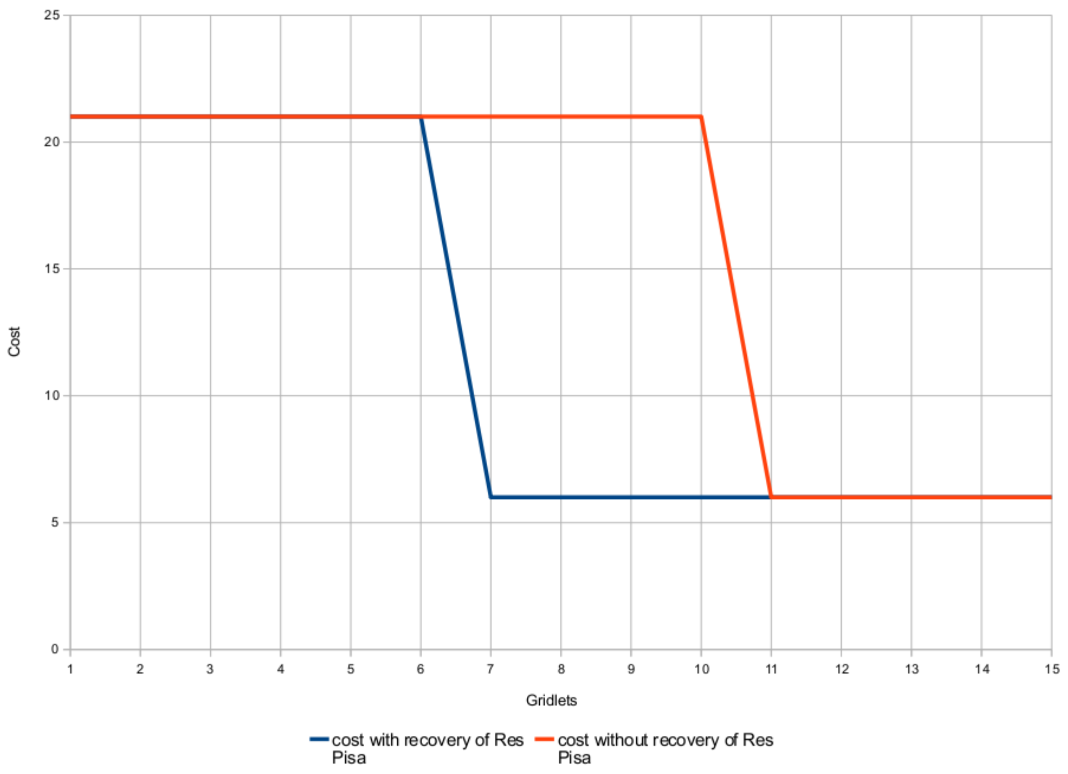

Figure 5.4.: Graph: Compare User_111 cost: with recovery of Res_Pisa and without the recovery

Listing 5.18: Set of fails: output User_111 without recovery of Res_Pisa

```
================= OUTPUT for ArtificialEU_User_111 =============
Gridlet ID  STATUS  Resource ID  Cost  CPU Time   Latency
    0       Success     126      21.0     7.0    15866.802648002978
    1       Success     126      21.0     7.0    12371.222372003596
    2       Success     126      21.0     7.0    3127.9621120006814
    3       Success     126      21.0     7.0    2902.7018480005718
    4       Success     126      21.0     7.0    2676.7715840004475
    5       Success     126      21.0     7.0    2345.9715840004737
    6       Success     126      21.0     7.0    2120.711320000364
    7       Success     126      21.0     7.0    1894.78105600024
    8       Success     126      21.0     7.0    1563.981056000266
    9       Success     126      21.0     7.0    1338.7207920001565
   10       Success      56       6.0     2.0    2137.3582080020697
   11       Success      56       6.0     2.0    2766.5582080026634
   12       Success      56       6.0     2.0    3502.4979440031384
   13       Success      56       6.0     2.0    4236.5676800037145
   14       Success      56       6.0     2.0    4866.967680004309
=======================================================================
```

For the other users the output file doesn't change, because also in the simulation before (set of failure), except Res_Pisa all resources that we set as fail are not recovered before the end of the simulation, so the behavior of the grid network is the same.

5.3.2. Second testbed: medium duration of resources failure

We use the traditional network "Artificial EU Grid", and the users submitted the gridlets to specific resources like in the previous scenario (refers to Table 5.4). The time of failure[4] for all resources is this:

- how long it is out of order = 2 hours (7200 seconds).

The following lines describe the behavior of "Artificial EU Grid" for each group of users, and the results of the test are compared with the results of the test with high duration of fail. Important: in this second test the variable init_time to begin to submitted gridlets is modified, this has a random value between 5 and 10 minutes (we wait this time because we have many resources, so we allow GIS to receive registrations from resources. Otherwise, the resource does not exist when we submitted). Note that, the times of failure that we set in this test, are more or less similar to the times of failure in the simulation "set of failure"[5]; so if the variable init_time doesn't change (before was setting between 10 and 15 minutes),

[4]Remember that the real time of failure is a little bit different from that specified, because a resource has to set its machine as failed, so it takes time. When there is a Grid resource recovery, happens the same, because the resource has to set her machine as "working", and it must register to RegionalGIS, so also here the time is not precisely the time specified in the variable "failureLengthPattern". Moreover, if the baud_rate between the resource and router is low (ex. Res_Dublin - Router 0), the recovery and the new registration take more time.

[5]In "set of failure" simulation the failure times are set in a random way, after an analysis of the output, it is confirm that the times of failures are more or less similar

we have the same grid network features and the final output doesn't change so much. The resources, fail after 180s (3 minutes), like in the previous testbed.

from User_1 to User_9 the gridlets will be submitted to Res_Dublin. In the previous simulations (high duration time failure) all gridlets were submitted to Res_Glasgow because Res_Dublin was out of order. Analyzing the output for this test (second testbed) is evident that the situation doesn't change in fact the gridlets are submitted to Res_Glasgow. The cost for processing the gridlet is equal to 6s (like before, the number of machines and the rating don't change).

from User_10 to User_29 the gridlets are submitted to Res_Glasgow (first 10) an to Res_Helsinki (last 5). In the simulation "high duration time failure" these gridlets will be submitted to Res_Oslo and Res_Stocklom due to the failure of two resources.
Indeed in this test we can note that the first gridlets are submitted to Res_Glasgow correctly because the resource Res_Dublin fails, but when we submitted the gridlets this is still registered in the list of resource available. Listing 5.19 shows the part of code where it is possible to see the list of available resources, the user and the index of resource takes in consideration and the message of gridlet sending.

Listing 5.19: Resource available output for User_18

```
User in consideration: ArtificialEU_User_18
Total number of local resources available: 4
Total number of global resources available: 12
Total number of resources available: 16
ArtificialEU_User_18: resource[0] = 51
ArtificialEU_User_18: resource[1] = 56
ArtificialEU_User_18: resource[2] = 66
ArtificialEU_User_18: resource[3] = 71
ArtificialEU_User_18: resource[4] = 81
ArtificialEU_User_18: resource[5] = 86
ArtificialEU_User_18: resource[6] = 91
ArtificialEU_User_18: resource[7] = 96
ArtificialEU_User_18: resource[8] = 101
ArtificialEU_User_18: resource[9] = 106
ArtificialEU_User_18: resource[10] = 111
ArtificialEU_User_18: resource[11] = 116
ArtificialEU_User_18: resource[12] = 121
ArtificialEU_User_18: resource[13] = 126
ArtificialEU_User_18: resource[14] = 131
ArtificialEU_User_18: resource[15] = 136
ACTUAL USER: ArtificialEU_User_18
ACTUAL INDEX:1
ArtificialEU_User_18: Sending Gridlet \#2 to
   ArtificialEU_Res_Glasgow at clock: 1015.4902159999975
```

As we can see in Listing 5.19 the resource with index equal to 1 has resource ID equal to 51 that corresponds to Res_Glasgow, in fact the last line confirms that the gridlet #2

is send to that resource. In this moment of the simulation the number of resources available is sixteen so 2 resources are out of order. Moreover from the output we know that we are considering the ArtificialEU_User_18 that is connected with Router3 that it's in ArtificialEU_Regional_GIS_1. So the number of local resources available, i.e. the resources available in ArtificialEU_Regional_GIS_1 are 4 (on total of 6). If we check the output, is clear that the Resource ID 61 (Res_Helsinki) and Resource ID 71 (Res_Moscow) are not in this list, so they're out of order. The number of global resources available is the number of resources available that are not in ArtificialEU_Regional_GIS_1. This number is equal to 12 (on total of 12), this means that there aren't global resource out of order.
After this analysis, it is obviously that for the same user and the same list of resources available, the gridlets between #10 and #14 are sent to the resource with index 2 (Table 5.4) that is Res_Oslo (Resource ID = 66). The latency in this test for the first 10 gridlets is greater than the latency in "set of failure" simulations, because Res_Glasgow has fewer machines than Res_Oslo and in this case has a big work load, because many gridlets are submitted to it.

from User_30 to User_41 the gridlets are submitted to Res_Oslo (first 10) and to Res_Stockholm (last 5). In the simulation with high duration of failure these gridlets will be submitted to Res_Warsaw and Res_Vienna due to the failure of three resources.
In this test we can note from the output that the first ten gridlets are correctly submitted to Res_Oslo because, if we check the list of available resources, we see that there aren't resources out of order.

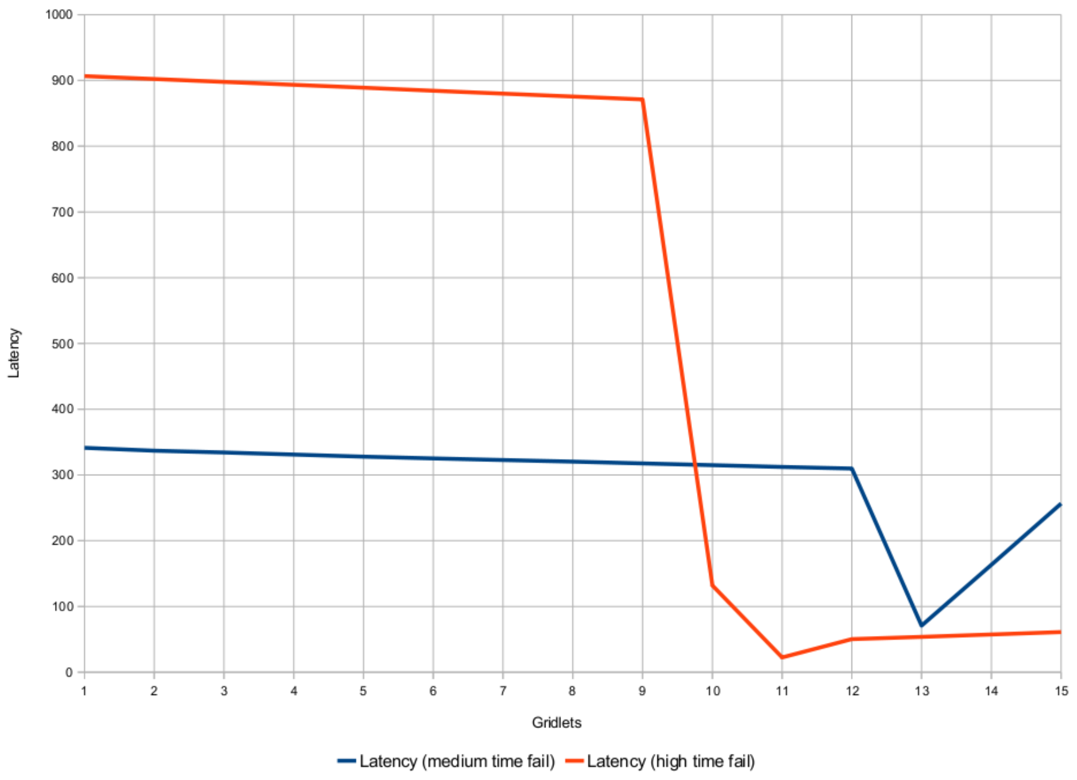

Figure 5.5.: Graph: Compare User_38 latency, high and medium duration of failure

For each of the 12 users, the last 5 gridlets are submitted to Res_Stocklom and Res_Warsaw:

- Gridlet #10 and Gridlet #11 are submitted to Res_Stocklom. In fact, if we check for example the output of User_38 (gridlet #10), we note that this gridlet is submitted before the failure of Res_Dublin and Res_Helsinki, so all resources are available. The gridlet will be submitted to the resource with index 4 in the list of available resources that correspons to Res_Stocklom.

- Gridlet #12, Gridlet #13 and Gridlet #14 are submitted to Res_Warsaw. For example, the gridlet #14 of User_38, is submitted when Res_Helsinki is down, so the gridlet will be submitted to the resource with index 4 in the list of available resources that is Res_Warsaw.

Graph 5.5 compares the output latency of User_38 in the simulation with a high duration time of failure (10 hours) and with a medium duration time of failure (2 hours, this last simulation). It is possible to see that the latency in case of medium duration time (blue line), tends to decrease until gridlet #11, in fact these twelve gridlets are submitted to Res_Oslo and Res_Stocklom, that have a high computational power, moreover it is less than the latency with high duration time of failure (orange line), where the first 10 gridlets are submitted to Res_Warsaw (low computational power).
The blue line tends to increase between gridlet #12 and gridlet #14 because these three are submitted to Res_Warsaw, also the orange line tends to increase between gridlet #10 and gridlet #14, but the values of latency are less than the simulation with medium duration time of failure, this because these gridlets are submitted to Res_Vienna (great CPU rating but few machines).

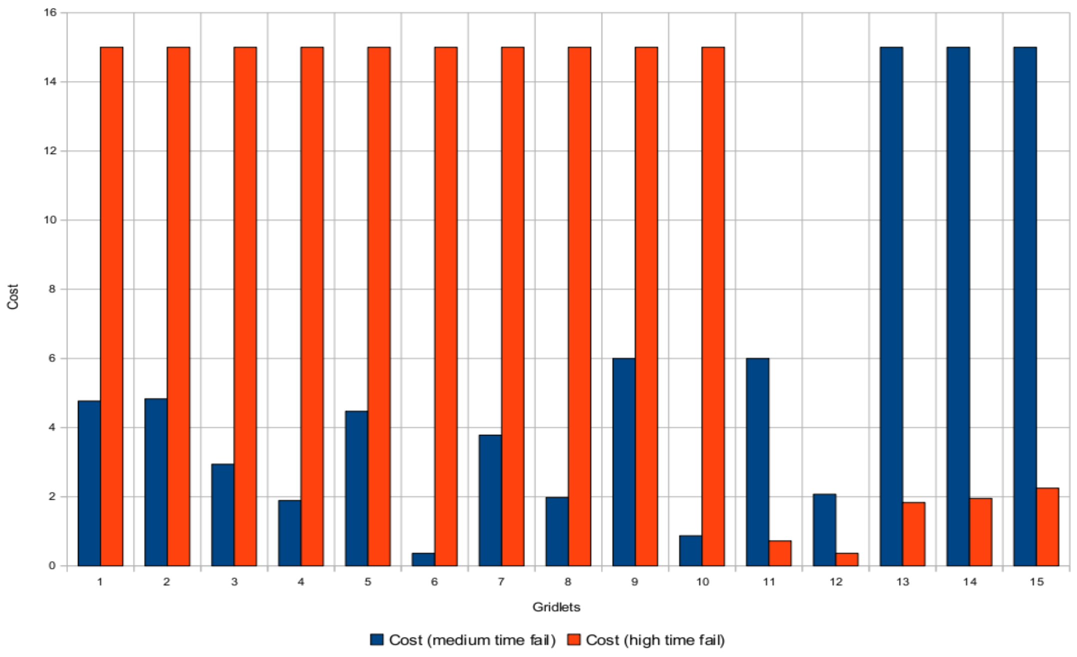

Figure 5.6.: Graph: Compare User_38 cost, high and medium duration of failure

Graph 5.6 compares the output cost of User_38 in the simulation with a high duration time of failure (10 hours) and with a medium duration time of failure (2 hours, this last simulation). As we can note the first 10 gridlets for high duration time of failure simulation have a cost equal to 15s because they are submitted to Res_Warsaw that has a low CPU rating (1000 MIPS), while for the first twelve gridlets the cost of processing gridlet for User_38 in medium time of failure simulation is low (max. 6s), this is becasue these gridlets are submitted to Res_Oslo and Res_Stocklom (high CPU rating: 49000 MIPS). For the last 3 gridlets, the opposite happens, cost equal to 15s for gridlets in "medium time" simulation (submitted to Res_Warsaw) and low cost for gridlets in "high time" simulation (submitted to Res_Vienna).

from User_42 to User_71 the gridlets are submitted to Res_Warsaw (first 10) an to Res_Vienna (last 5). In the simulation with high duration time of failure these gridlets will be submitted to Res_Budapest and Res_Athens due to the failure of four resources.
In this testbed with a medium duration of failure, the gridlets are submitted to Res_Vienna

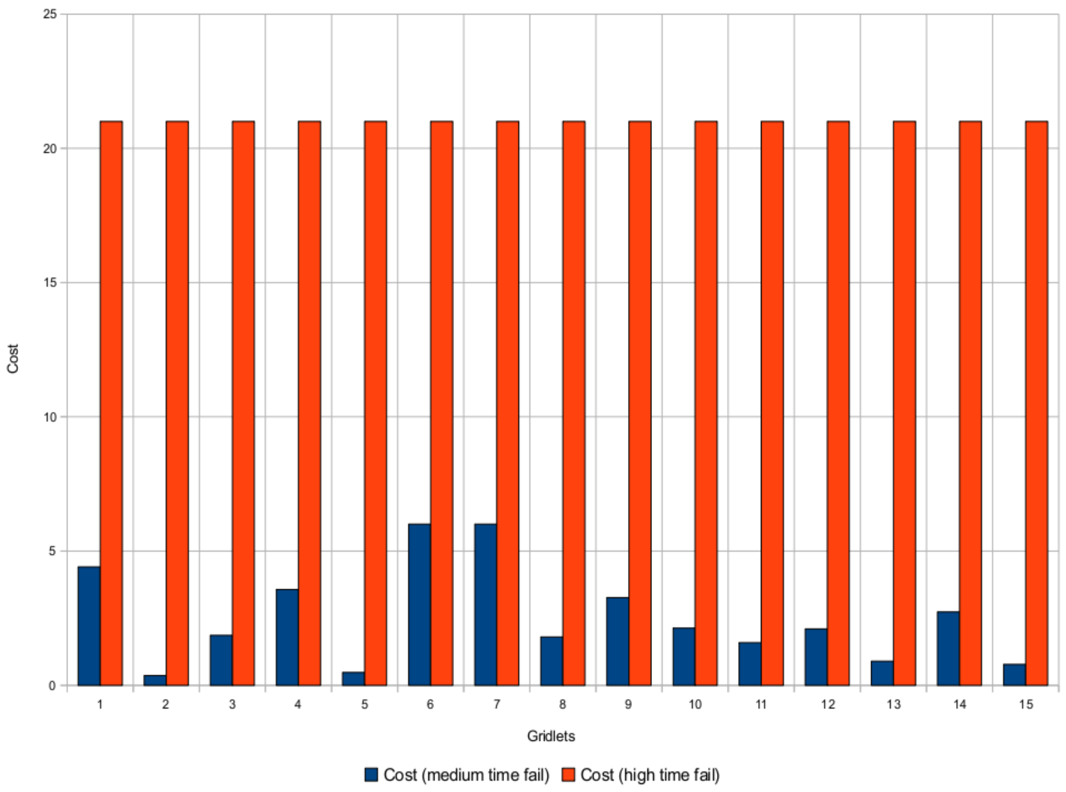

Figure 5.7.: Graph: Compare User_47 cost, high and medium duration of failure

and to Res_Berlin, which are the resources immediately following those established. If we check for example the output of User_47 for each one of his gridlets submitted, we can note that in the list of availbale resources there isn't Res_Dublin, that is out of order. The computational power of Res_Vienna and Res_Berlin is higher than the Res_Budapest and Res _Athens power. So it is obviously that in this case the performances are higher in the medium duration of failure simulation, in fact for all of 30 users the maximum cost for

submitting the gridlets is 6s, while in the simulation with high duration of failure the cost never goes down below 21s.
Graph 5.7 perfectly explains the situation.

from User_72 to User_101 the gridlets are submitted to Res_Berlin (first 10) an to Res_Munich (last 5). In the previous simulations (high duration time of failure) the gridlets were submitted to Res_Milano and Res_Madrid, because five resources are out of order.
In this testbed, if we check the output, it is possible to note that gridlets are submitted to different resources, this is due to the different submission time and to the recovery of the resources. In this testbed, we note also that some gridlets are submitted to a certain resource because it is available when we sent it the gridlet, but during the execution the resource fails. What happens?
The gridlet, after some seconds (resubmission time[6]) is resubmitted many times to the same resource (until it returns available), or to a resource in the list with the correspondent index. For example, if we check the output we can see that User_84 submitted the gridlet #0 to Res_Munich because it is in the list of available resources, but when this receives the gridlet, it is out of order, so the gridlet will be resubmitted after to the same resource.
We have a list of messages (Listing 5.20) about the behavior of gridlet #0 and User_84. The gridlet is resubmitted 7 times, and at time 8701.972052019704 is received with status success.
Remember that Res_Munich is recovered at time 7674.960119999999s (Listing 5.21), so after this time the resource is available again.

Listing 5.20: Resource unavailable: resubmission of gridlet

```
ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 1792.9303960000118 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.051986361447091792.9303960000118

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 3545.270731999608 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.051986361447093545.270731999608

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 3959.4908160083064 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.051986361447093959.4908160083064

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 4907.361104006151 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.051986361447094907.361104006151

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 6123.341344009502 from
```

[6]The java code is not been modified, so the resubmission time is random

```
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.05198636144709 6123.341344009502

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 6533.891416018123 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.05198636144709 6533.891416018123

ArtificialEU_User_84: Receiving Gridlet \#0 with status
   Failed_resource_unavailable at time = 7607.941680018377 from
   resource ArtificialEU_Res_Munich(resID: 96). Resubmission time
   will be: 73.05198636144709 7607.941680018377

ArtificialEU_User_84: Receiving Gridlet \#0 with status Success at
    time = 8701.972052019704 from resource ArtificialEU_Res_Munich
```

Listing 5.21: Res_Munich recovery time

```
ArtificialEU_Regional_GIS_2: sends a GRIDRESOURCE_RECOVERY to the
   resource ArtificialEU_Res_Munich. Clock: 7674.960119999999
```

This is the history of the gridlet #0 for User_84 (Listing 5.22)

Listing 5.22: User_84: gridlet #0 history

```
history gridlet: Time below denotes the simulation time.
Time (sec)       Description Gridlet \#0
------------------------------------------

0,00    Creates Gridlet ID \#0
0,00    Assigns the Gridlet to ArtificialEU_User_84 (ID \#477)
760,45    Sets Gridlet status from Created to
   Failed_resource_unavailable
760,45    Allocates this Gridlet to ArtificialEU_Res_Munich (ID
   \#96) with cost = $3.0/sec
2503,47    Sets the length's finished so far to 0.0
2522,561    Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
3805,061    Sets the length's finished so far to 0.0
3848,321    Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
4844,131    Sets the length's finished so far to 0.0
4878,181    Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
5095,151    Sets the length's finished so far to 0.0
5225,341    Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
6338,441    Sets the length's finished so far to 0.0
```

```
6468,791   Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
7370,682   Sets the length's finished so far to 0.0
7445,402   Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
8642,752   Sets the length's finished so far to 0.0
8695,632   Moves Gridlet from ArtificialEU_Res_Munich (ID \#96) to
    ArtificialEU_Res_Munich (ID \#96) with cost = $3.0/sec
8695,632   Sets the submission time to 8695,632
8695,632   Sets Gridlet status from Failed_resource_unavailable to
    InExec
8695,632   Sets the execution start time to 8695,632
8695,802   Sets Gridlet status from InExec to Success
8695,802   Sets the wall clock time to 0,17 and the actual CPU
   time to 0,17
8695,802   Sets the length's finished so far to 4200.0
```

Now we can analyze the output for these 31 Users:

- from User_71 to User_81 the first ten gridlets are submitted to Res_Budapest and the other five to Res_Athens, for both the rating is low (700 MIPS) so the cost for processing gridlets is always equal to 21s. The gridlet is submitted to the resources with indeces equal to 8 and 9, in that case these indeces correspond to Res_Budapest and Res_Athens because if we check the list of available resources we can see that Res_Dublin (Resource ID 51) and Res_Munich (Resource ID 96) fail.

- from User_81 to User_101 the first ten gridlets are submitted to Res_Munich and the last five to Res_Budapest. In this case for each user the first ten gridlets are submitted to Res_Munich, but when this receives the gridlets, it is out of order, so the gridlets will be resubmitted many times, until Res_Munich works again (Listing 5.20). This causes a delay, because the gridlets are processed after the resource recovery (in this case only 2 hours, but sometimes it can be more).
  Here the cost for processing gridlets is different, because Res_Munich has a high rating (80000 MIPS), while Res_Budapest only 700 MIPS, so for each user the cost of the first ten gridlets is low (maximum 4s), while for the last five is high (21s, like for the gridlets submitted to Res_Budapest by the users between #71 and #81).

Comparing the output of this simulation, with the output of the simulation with high duration time of failure, we can say that the performances decrease when the failure time is 2 hours because the gridlets are submitted to resources with low rating, so the cost is greatest. Moreover in this last testbed the gridlets are submitted to Res_Budapest and Res_Athens, that are connected with the routers with a low baud rate, this causes an increase of latency.

from User_102 to User_124 the gridlets are submitted to Res_Athens (first 10) an to Res_Glasgow (last 5). In the previous simulations (high duration time of failure) the gridlets were submitted to Res_Lisbon and Res_Glasgow, because six resources are out of order. The indexes are 10 and 0, so if we go to check the list of available resources we note that

Res_Dublin is out of order so the resource number 10 will be Res_Athens and the resource with index 0 corresponds to Res_Glasgow.
The features of Res_Lisbon and Res_Athens are very similar, same rating and same baud rate connection with the routers, they differ only in number of machines. Consequently there are no different performances for these 23 users with high or medium duration time of failure. Obviously, the first ten gridlets submitted have always a greater cost than the last five, due to the rating of the machines.

5.3.3. Third testbed: medium duration of resources failure and high init_time

In this testbed we use the same settings of the second testbed, so the duration of failure is again 2 hours, but we set the variable init_time for begin to submitted gridlets at 120 minutes plus variable "pause" that is set to 10 minutes, so totally 130 minutes (Listing 5.23). (we wait this time because we have many resources, so we allow GIS to receive registrations from resources. Otherwise, the resource does not exist when we submitted[7]).

Listing 5.23: Setting init_time to 130 minutes

```
int PAUSE = 10*60;  // 10 mins
int init_time = PAUSE + (120*60); //120 mins
```

Analyzing the output we note that all gridlets are submitted to the resources that we are setting in the Table 5.4, only the users from User_30 to User_41 send the gridlet #0 to a resource different than what we had decided. So the initial submitted gridlets signal is for all users at 7800 seconds (Listing 5.24), all gridlets are submitted after the recovery of the resources, so they are sent to the correct destination.

Listing 5.24: Initial submitted gridlets after 7800 seconds

```
Starting GridSim version 4.0
Entities started.
ArtificialEU_User_56: initial submitted_GRIDLET event will be at
   clock: 7800. Current clock: 0.0
ArtificialEU_User_62: initial submitted_GRIDLET event will be at
   clock: 7800. Current clock: 0.0
ArtificialEU_User_64: initial submitted_GRIDLET event will be at
   clock: 7800. Current clock: 0.0
[....] //for all users, we have an initial submitted gridlet event
   .
```

For users between #30 and #41 the gridlets have to be submitted to Res_Oslo (the first ten) and Res_Stocklom (the last five), but if we check the output we note that the gridlets are sent to the correct resource with the exception of gridlet#0 that is submitted to Res_Stocklom. This is right because if we check the output we note that the gridlet #0 is sent when Res_Helsinki (Resource ID 61) is not in the list of available resources, so the resource with index equal to 3 is Res_Stocklom (Listing 5.25).

[7]In our grid network 10 minutes is enough for the registration of resources, so there is no problem with more than 120 minutes

Listing 5.25: Resource available output for User_36

```
User in consideration: ArtificialEU_User_36
Total number of local resources available: 4
Total number of global resources available: 12
Total number of resources available: 16
ArtificialEU_User_36: resource[0] = 51
ArtificialEU_User_36: resource[1] = 56
ArtificialEU_User_36: resource[2] = 66
ArtificialEU_User_36: resource[3] = 71
ArtificialEU_User_36: resource[4] = 81
ArtificialEU_User_36: resource[5] = 86
ArtificialEU_User_36: resource[6] = 91
ArtificialEU_User_36: resource[7] = 96
ArtificialEU_User_36: resource[8] = 101
ArtificialEU_User_36: resource[9] = 106
ArtificialEU_User_36: resource[10] = 111
ArtificialEU_User_36: resource[11] = 116
ArtificialEU_User_36: resource[12] = 121
ArtificialEU_User_36: resource[13] = 126
ArtificialEU_User_36: resource[14] = 131
ArtificialEU_User_36: resource[15] = 136
ACTUAL USER: ArtificialEU_User_36
ACTUAL INDEX:3
ArtificialEU_User_36: Sending Gridlet \#0 to
   ArtificialEU_Res_Stocklom at clock: 8221.150144000005
```

The following listing shows the output for the User_36, gridlet #0 as the last five gridlets are submitted to Resource ID 71 that correspond to Res_Stocklom (Listing 5.26).

Listing 5.26: Third testbed: medium duration of resources failure and high init_time

```
============== OUTPUT for ArtificialEU_User_36 ===========
GridletID STATUS ResourceID Cost      CPU Time     Latency
   0      Success    71         6.0          2.0     342.1400000001122
   1      Success    66         6.0          2.0     336.9699999999993
   2      Success    66         6.0          2.0     334.7299999999504
   3      Success    66         6.0          2.0     332.1999999998952
   4      Success    66  2.1899999999 0.72999999997 329.66999999984
   5      Success    66         6.0          2.0     327.0499999997828
  10      Success    71         6.0          2.0     313.5699999994886
   6      Success    66  2.1299999999 0.70999999997 324.42999999972
  11      Success    71  2.4599999999 0.81999999997 310.26999999941
   7      Success    66         6.0          2.0     321.8099999996684
  12      Success    71  2.1599999999 0.71999999997 307.19999999934
   8      Success    66  2.2199999999 0.73999999997 319.11999999960
   9      Success    66         6.0          2.0     316.4099999995505
  13      Success    71         6.0          2.0     6.630032000146457
```

```
    14      Success    71  0.3299999998  0.10999999995  23.590036000516
================================================================
```

5.3.4. Fourth testbed: duration of resources failure equal to 1 hour and high init_time

In this testbed we decrease the duration of failure to 1 hour, and we set the variable init_time to begin to submitted gridlets at 120 minutes plus "pause" that is set to 10 minutes, so totally 130 minutes like in the previous testbed.
What happens?
All the gridlets now are sent after the recovery of all resources, so in this case, unlike the previous scenario, these are submitted to the correct resource, also the users between #30 and #41 submitted the gridlet #0 to Res_Oslo.

With regard to the Java code, remember that we need to set the variable "failureLength-Pattern", so that the duration of failure lasts only 3600 seconds. Listing 5.27 shows the output for User_36, note that gridlet #0 is submitted to Resource ID 66, equal to Res_Oslo.

Listing 5.27: Fourth testbed: duration of resources failure equal to 1 hour and high init_time: output for User_36

```
================ OUTPUT for ArtificialEU_User_36 ==============
GridletID STATUS ResourceID  Cost     CPU Time       Latency
    0       Success    66           6.0         2.0        336.7399999999943
    1       Success    66  2.1899999999  0.72999999997  331.93999999988
    2       Success    66  2.6399999999  0.87999999997  329.74999999984175
    3       Success    66  3.9899999999  1.32999999998  327.33999999978914
    4       Success    66  2.0999999999  0.69999999997  324.949999999737
   10       Success    71  4.1399999999  1.37999999998  310.4099999994196
    5       Success    66  3.9899999999  1.32999999998  322.53999999968437
   11       Success    71           6.0          2.0       307.8499999993637
    6       Success    66  1.7399999999  0.57999999996  320.139999999632
    7       Success    66           6.0          2.0       317.7399999995796
   12       Success    71  0.7799999998  0.25999999996  305.1499999993048
    8       Success    66  1.4399999999  0.47999999996  315.3399999995272
   13       Success    71  1.4699999999  0.48999999996  302.2499999992415
    9       Success    66  4.7099999999  1.56999999999  312.8799999994735
   14       Success    71  3.7200119999  1.24000399998  9.640036000211694
================================================================
```

5.3.5. Fifth testbed: low duration of resources failure

We use the traditional network "Artificial EU Grid",and we do a simulation like in first and second testbed but using a low time of failure. The test has been executed two times, with different values of the variable "failureLengthPattern", however the results are the same:

- how long it is out of order = 15 minutes (900 seconds).
- how long it is out of order = 30 minutes (1800 seconds).

In this test the variable init_time to begin to submit gridlets has a random value between 5 and 10 minutes, like in the testbed with medium duration of failure, in this way we can compare the results.
For this testbed, the behavior of the gridlets on the grid network is not reported for each group. This because analyzing the output file we can note that the users send the gridlets to the same resources of the testbed with medium duration of failure. Table 5.9 shows effectively at which resource the users submitted the gridlets, in the three simulations: low, medium and high duration of failure.

Group of users	**Low time fail**	**Medium time fail**	**High time fail**
from User_0 to User_9	Res_Glasgow	Res_Glasgow	Res_Glasgow
from User_10 to User_29	Res _Glasgow Res_Oslo	Res_Glasgow Res_Oslo	Res_Oslo Res_Stocklom
from User_30 to User_41	Res_Oslo Res_Stocklom Res_Warsaw	Res_Oslo Res_Stocklom Res_Warsaw	Res_Warsaw Res_Vienna
from User_42 to User_71	Res_Vienna Res_Berlin	Res_Vienna Res_Berlin	Res_Budapest Res_Athens
fromUser_72 to User_81	Res_Budapest Res_Athens	Res_Budapest Res_Athens	Res_Milano Res_Madrid
from User_82 to User_101	Res_Munich Res_Budapest	Res_Munich Res_Budapest	Res_Milano Res_Madrid
from User_102 to User_124	Res_Glasgow Res_Athens	Res_Glasgow Res_Athens	Res_Glasgow Res_Lisbon

Table 5.9.: Comparison 3 testbed: where gridlets are submitted?

As already mentioned the first 2 columns are equal, this is because we submitted the gridlets more or less in the same time (between 5 and 10 minutes), moreover the gridresource failure signals are sent for all resources at the same time (180 seconds, 3 minutes).

The duration of failure in this fifth testbed, one time is 900 seconds (15 minutes) and one time is 1800 seconds (30 minutes), but in both case the gridlets are submitted to the same resources because:

- The resources fail after 3 minutes, and minimum is recovered at time 18 minutes (with 15 minutes of failure).
- The gridlets are send between 5 and 10 minutes so when the resources are down, the list of available resources in the two test are the same.
- Changing the variable "init_time" to 20 minutes (while "failureLengthPattern" doesn't change and is equal 900 seconds), the gridlets are submitted to different resources, this because are sent when some failure procedures are finished and some not, so the

resource is in the list of available resources. If we use the same "init_time" but we set the duration of failure equal to 1800 seconds, the grid network has more or less the same behavior of the tests with high duration of failure, because the gridlet are submitted after 20 minutes and all resources are out of order, the first gridresource recovery signal can arrive only after minimum 33 minutes (180s + 1800s).

In our simulation, however, it is not important to do tests when resources are working (they probably are interesting for check the correct functioning of the network), the important thing is to observe the behavior when we have small time failures, with the original settings of the network, so in "Artificial EU grid", we have proved that the situation doesn't change with respect to the situation with a medium duration of failure.

We increase the hours of failure for the "medium duration failure" test. We can consider 2 hours, while a low time of failure would be something like thirty or fifteen minutes. After the simulation made with "failureLengthPattern" equal to seven hours, we note that the behavior is the same of the grid network with high duration fail (the resources fail and the simulation finishes before the recovery of any resource), so 7 hours can be considered a high time of fail in this case, while we can consider 2 hours a great time for the class "medium fail time".

Another test that we made is a simulation with a time of failure setting to 26 hours, so for more than one day, the six resources that we want to fail are out of order. The output confirms the expectations, in fact the grid network behaves like in the "high duration failure" testbed, because all resources that are failed received a grid resource recovery after 26 hours, when all gridlets are already worked from other resources.
So for the analysis of high duration failure, we can consider ten hours a great value for the variable "failureLengthPattern".

The last test is a simulation that confirms the behavior of grid network. In this testbed we submitted the gridlet after a time between 5 and 10 minutes, as set in the variable init_time, while the grid resource failure signal is set to 36000s (10 hours) (refers to Listing 5.28). The duration of failure is low (1800 seconds, 30 minutes), however, as we expected, we will see that this feature is not very important, because all gridlets are submitted before the failure of the resources.

Listing 5.28: Setting: submitted the gridlets before the failure of the resources

```
 failureTimePattern = 18000;
 failureLengthPattern = 1*1800;
[....]
 int PAUSE =5*60;  // 5 mins
 Random random = new Random();
 int init_time = PAUSE + random.nextInt(5*60);
```

So it is obvious that all gridlets are submitted to the correct resources that we had set (refers to Table 5.4), this is due to the fact that when users submitted them, the list of available resources is complete, no resources fail.

6. Discussion of the results

6.1. Simulation analysis

In the previous chapter, we show many simulations based on the failure of the resources. In this section we summarize and compare in an overall view the output of various simulations, to see what happens in a large grid scenario with different failures of resources.
There is no reference to the test where there is only one gridlet, because we use it only to introduce the Artificial EU Grid, but our goal is to study large scale scenario, so we will consider the following scenarios:

1. single point of failure
2. set of failures
3. set of failures with custom times of failure

For each of these three scenario, we have already analyzed the outputs of simulation, so we know the behavior of each user (remember that the users are gathered in groups of users), but now we want to have a general view of the three scenario and compare them also with scenario without failure. Obviously, the last scenario (set of failure with custom time of failure) is treated in a certain way because it is compose of five testbeds, that depending on settings, can get closer to "single point of failure " or "set of failure" scenario.

6.1.1. Single point of failure simulation

In the first test only Res_Helsinki fails, some gridlets are submit while the resource is down, some other gridlets are submit after the recovery of the resource. In general when there is a failure of the resource, the gridlets destined to this resource are sent to the next available resource. For example for User_34 the first four gridlets are submit to the next resource available respect the resource setted, because they are submitted while Res_Helsinki is out of order. Now we compare the cost of processing gridlets in the same scenario, but one time without the fail of Res_Helsinki, and one time with the failure of it (refers to chapter 5.1). In Table 6.1 we can see the comparison of average costs for processing gridlets for each group of users.

Graph 6.1 is reports the table in a graphic way, so it is easier to see that the costs are more or less the same without failure or with the failure of Res_Helsinki.

In the group of users from #10 to #29, we can note that the cost is greater with the scenario without failure. This is because for this group of users the first ten gridlets are submit to Res_Warsaw and the other five to Res_Vienna, while in "single point of failure" scenario the gridlets are submit to these same resources, except the first three gridlets that are sent to Res_Vienna, due to the failure of Res_Helsinki. Res_Vienna as a rating higher

From *User* to *User*	**Cost without failure**	**Cost with Res_Helsinki fails**
from User_0 to User_9	6	6
from User_10 to User_29	11,248	8,610
from User_30 to User_41	4,533	4,962
from User_42 to User_71	21	15,076
from User_72 to User_81	8,010	12
from User_82 to User_91	11	14,980
from User_92 to User_101	11	14,800
from User_102 to User_113	6	6
from User_114 to User_124	3,242	2,828

Table 6.1.: Comparison cost without and with failure of Res_Helsinki in "single point of failure" scenario

than Res_Warsaw so the costs for the first three gridlets are less in "single point of failure" scenario. This, leads to a slight decrease of average cost in "single point of failure" scenario.

It is also evident in the graph that the cost for users from #42 to #71 is greater in the scenario without failure.

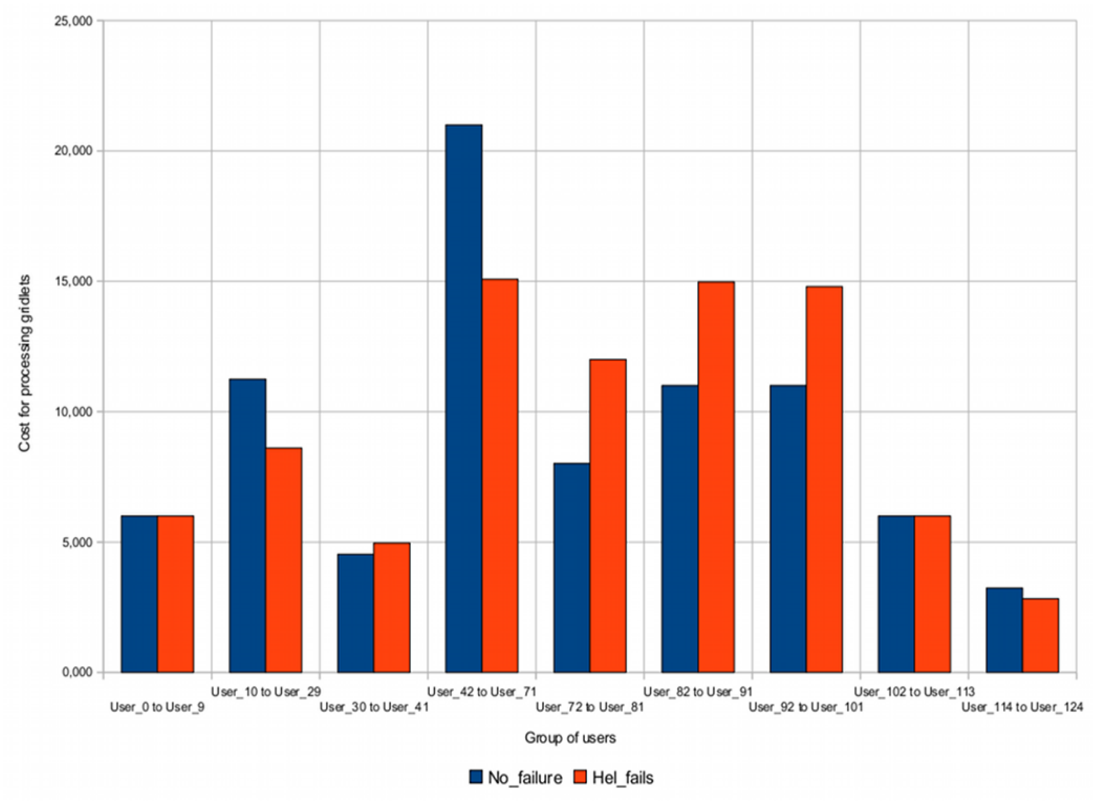

Figure 6.1.: Graph: Comparison cost without and with failure of Res_Helsinki in "single point of failure" scenario

Infact, in this group, these users have to submit gridlets to Res_Budapest and to Res_Athens that have small rating (700 MIPS) and a small number of machines so the average cost is equal to 21s. In the "single point of failure" scenario, due to Res_Helsinki failure the gridlets are sent to Res_Athens and Res_Milan. This last has a high rating (49000 MIPS) and a high number of machines, so the cost of the gridlets processing by this resource (for every user the last five) is low. This causes a decrease of the average cost for this group of users.

For the group of users between #72 and #81 the cost bar of "single point of failure" scenario is greater. When there isn't failure the gridlets are submit to Res_Milan and Res_Pisa. When Res_Helsinki is out of order, the gridlets are sent to Res_Pisa and Res_Madrid, so the cost in this case increases because ten of fifteen gridlets are sent to Res_Pisa that has a low rating (1000 MIPS), while in the scenario without failure, only the last five gridlets are submit to this resource, so the average cost is less.
Note that Res_Milan and Res_Madrid have almost the same features.

Users between #82 and #101, in the scenario without failure, submit the gridlets to Res_Madrid and to Res_Lisbon. With the failure of Res_Helsinki the gridlets are submit to Res_Lisbon and Res_Paris, like the previous group of users, the cost in this case increases, because ten gridlets on fifteen are sent to Res_Lisbon that has a low rating (700 MIPS), while in the scenario without failure, only five gridlets are sent to Res_Lisbon. For this reason the average cost is greater in the "single point of failure" scenario (refers to 6.1).

The graph 6.1 shows that the cost for processing gridlets in the "Artificial EU Grid", in general, is equals or a little bit greater in the scenario with a single point of failure. Only in two group of users this trend is reversed.

Table 6.2 shows the comparison of average latency without failure and with a single point of failure, so Res_Helsinki is down.

From *User* to *User*	**Latency without failure**	**Latency with Res_Helsinki fails**
from User_0 to User_9	6277,639	5971,241
from User_10 to User_29	1227,045	1411,775
from User_30 to User_41	905,271	848,288
from User_42 to User_71	1353,058	1298,832
from User_72 to User_81	1362,220	4799,388
from User_82 to User_91	12492,746	13528,455
from User_92 to User_101	12367,32500	12860,996
from User_102 to User_113	6433,074	6251,193
from User_114 to User_124	11908,29900	12466,366

Table 6.2.: Comparison latency without and with failure of Res_Helsinki in "single point of failure" scenario

Like for the cost, also here we have calculated the average latency for each group of users. We can note that the average latency for the users is more or less the same, we have a significant difference only for the users between #72 and #81.

Analyzing the output, we can note that without failure, the gridlets are submit to Res_Milano and to Res_Pisa, but if Res_Helsinki fails the gridlets are submit to Res_Pisa and Res_Madrid. But we have to remember that Res_Madrid is connected to Router10 that is connected with a low baud rate (0,00001 GB/s) with Router9 and Router11, so when the gridlets are submit to this resource we have an increase of the latency. Graph 6.2 shows the results in a graphic way of Table 6.2. We can see that all bars have approximately the same height, except, as mentioned before, for the group of users between #72 and #81 that have different values of average latency in the two scenario.

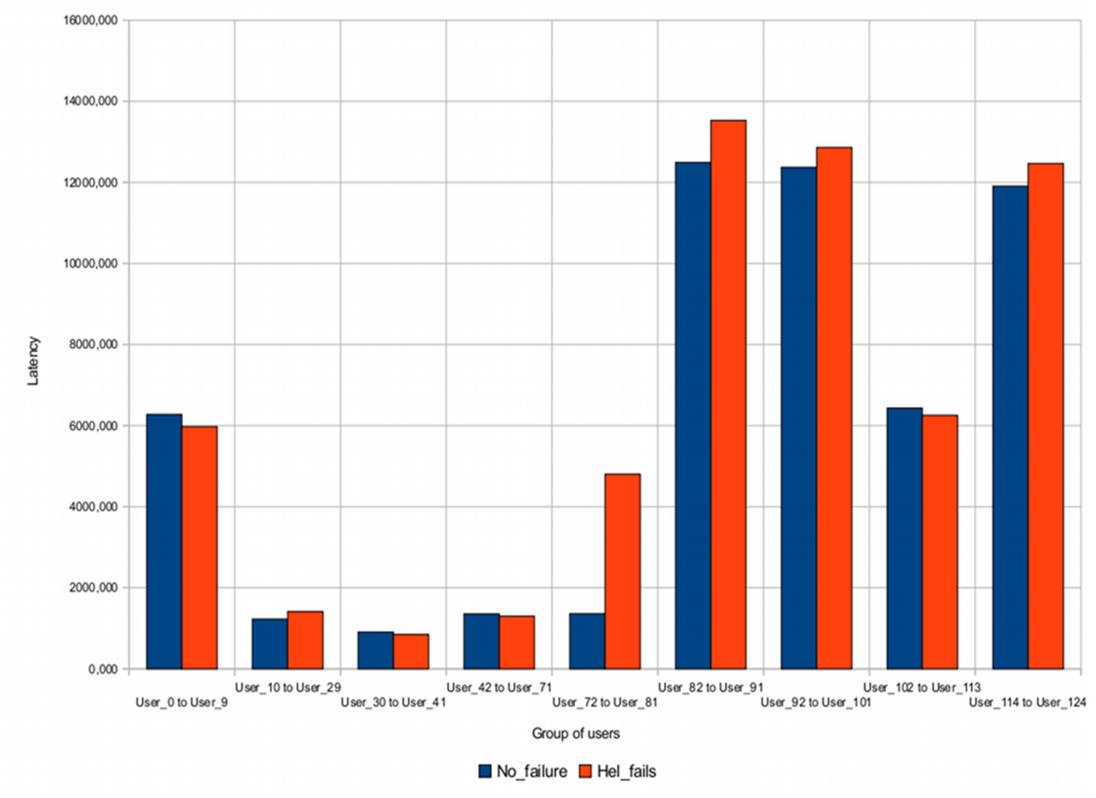

Figure 6.2.: Graph: Comparison latency without and with failure of Res_Helsinki in "single point of failure" scenario

6.1.2. Set of failure simulation

In this simulation six resources fail: Res_Dublin, Res_Helsinki, Res_Warsaw, Res_Munich, Res_Pisa, Res_Brussels. Some gridlets are submit while the resources are down, only Res_Pisa is recovered before the end of simulation. Refers to Table 5.4 (Chapter 5.2) to know at which resources, the group of users submits their gridlets.
Table 6.3 shows the different costs for the scenario without failure and for the scenario with a set of failures.

Graph 6.3 reports the table so it is more easier to see the trend of average cost without failure or with the failure of Res_Helsinki.

In the group of users between #0 and #9, if there are no failures, the gridlets are sub-

From *User* to *User*	**Cost without failure**	**Cost with set of failures**
from User_0 to User_9	6	6
from User_10 to User_29	4,320	1,594
from User_30 to User_41	3,038	10,528
from User_42 to User_71	10,786	21
from User_72 to User_101	4,320	3,286
from User_102 to User_124	16	14

Table 6.3.: Comparison cost without and with a set of failures

mit to Res_Dublin, if we consider the scenario "set of failures" the gridlets are submit to Res_Glasgow. These resources have the same features, so the cost in the two different scenarios is the same and is equals 6s.

In the group of users between #10 and #29 the gridlets are submit to Res_Glasgow and to Res_Helsinki. In the case of "set of failures" scenario the gridlets are submit to Res_Oslo and to Res_Stocklom due to the fact that Res_Dublin and Res_Helsinki are out of order. These last resources have more machines than the first two, for this reason the cost in the scenario without failures is greater than in the scenario with failures.

In the graph we can see that, for users between #30 and #41 the cost for processing gridlets in the scenario with failures is greater than the scenario without. This is due to the fact that in this last scenario the gridlets are sent to Res_Oslo and to Res_Stocklom that have a good rating (49000 MIPS) and a good number of machines, while in the scenario with failures the gridlets are submit to Res_Moscow and to Res_Warsaw that has a low rating (1000 MIPS) and few machines. This fact causes an increment of average cost in the "set of failure" scenario. Note that the average cost without failure is 3,038s that is very similar to the average cost of the previous group of users (4,320s). Infact if we check the resources where the gridlets are submit we can note that they more or less have the same computational power.

Also the users between #42 and #71 have an average cost greater in the scenario with failure than in that without. The cause, like in the previous group of users, is the computational power of the resources. In the testbed without failure the gridlets are submit to Res_Warsaw and to Res_Vienna (medium computational power) while in the testbed with a set of failures the gridlets are sent to Res_Budapest and to Res_Athens that each one has a rating of 700 MIPS, so the average cost increases to 21s.

The users between #72 and #101 have more or less the same average cost in the two different testbeds. This is because in the testbed without failure the gridlets are sent to Res_Berlin and to Res_Munich, while in the testbed with failure are sent to Res_Milano and to Res_Madrid. These resources have approximatly the same features, so the average cost in the two testbeds are very similar. The costs are low because these four resources have an high rating and number of machines.

In this last group of users (from #120 to #124) the average cost is a little bit low in the case of testbed with failures. In the test without failure the gridlets are submit to Res_Budapest (low rating) and to Res_Dublin (high rating). In the testbed with failures, the gridlets are submit to Res_Lisbon and to Res_Glasgow due to the failure of six resources, but in this case Res_Pisa is recovered so some gridlets are also submit to Res_Madrid. Without the recovery of Res_Pisa the average cost would be the same because Res_Lisbon has approximatly the same features of Res_Budapest, and Res_Dublin the same of Res_Glasgow, but with recovery some gridlets are sent to Res_Pisa that has an higher rating than Res_Lisbon. This causes a small decrease of average cost for "set of failure" scenario.

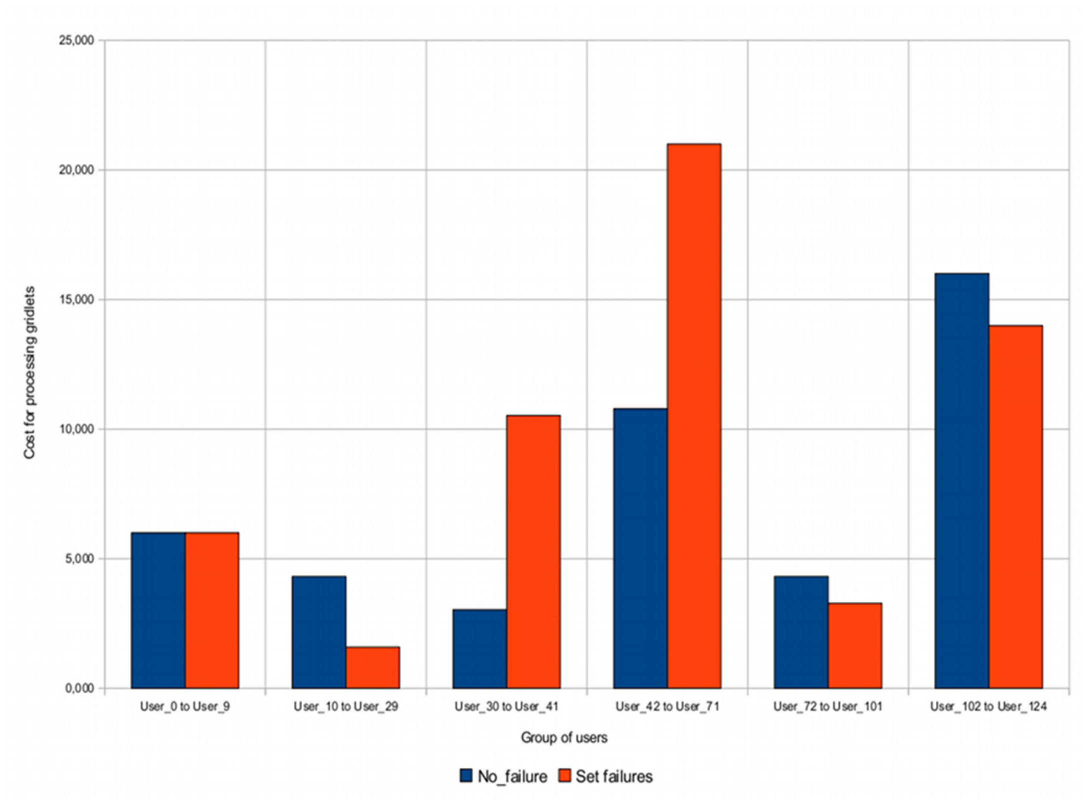

Figure 6.3.: Graph: Comparison cost without and with a set of failures

Table 6.4 shows the comparison of average latency without failure and with a set of failures.

From *User* to *User*	**Latency without failure**	**Latency with set of failures**
from User_0 to User_9	8151,673	9104,547
from User_10 to User_29	11448,932	467,071
from User_30 to User_41	261,939	507,844
from User_42 to User_71	1094,686	1668,480
from User_72 to User_101	618,165	2791,910
from User_102 to User_124	11544,424	6021,168

Table 6.4.: Comparison latency without and with a set of failures

We can note from Table 6.4 and graph 6.4 that for the group of users from #10 to

#29 there is a big difference of latency between the two testbeds. This, is due to the fact that in the scenario without failures the gridlets are submit to Res_Glasgow and to Res_Helsinki (Res_Oslo and Res_Stocklom respectively in the scenario with set of failure), but the connection between Router0 and Router1 has a low baud rate (0.00001 GB/s), and Res_Glasgow is connected to Router1. The connection with a low baud rate causes an increment of the average latency value. For this reason the average latency for the testbed without failures is greater than the testbed with failures.

Also in the group of users from #102 to #104 there is a difference in the value of average latency. It is always the baud rate value between connection that modifies the final average latency. In the testbed without failures the gridlets are submit to Res_Budapest and to Res_Dublin, both are connected to router with low baud rate, while in the testbed with failures the gridlets are sent to Res_Madrid, Res_Lisbon and to Res_Glasgow. So these last two are connected to routers with a low baud rate, but Res_Madrid is connected with an high baud rate, these differences of connection cause a greater average latency (for this users) in the testbed without failure.

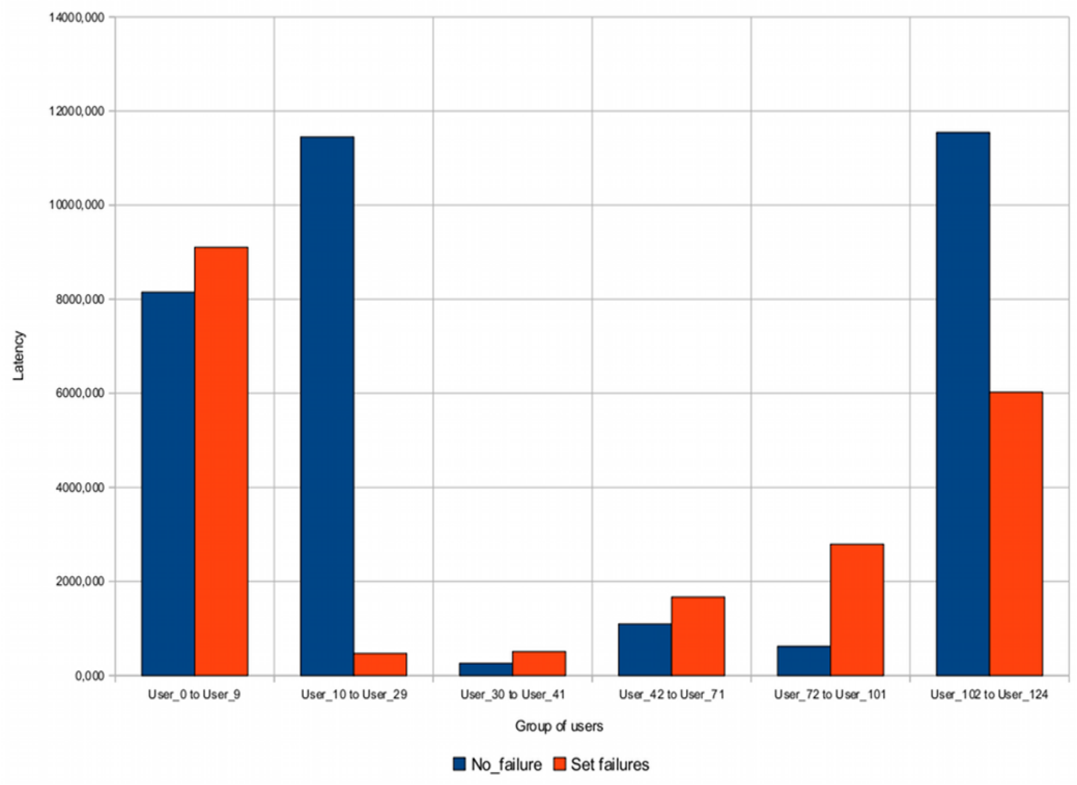

Figure 6.4.: Graph: Comparison latency without and with a set of failures

6.1.3. Set of failure with custom time of failure

In this section we do not compare the results of the simulation without failures with the results of the simulation with failures, but instead examine the fifth testbed to see the behavior of the "Artificial EU Network" when the failure time parameters are changed.

Group of users	**Low duration**	**Medium duration**	**Medium duration high init_time**	**1 hour duration high init_time**	**high duration**
User_0 to User_9	6	6	6	6	6
User_10 to User_29	4,486	4,738	4,862	5,014	3,276
User_30 to User_41	5,452	5,390	2,908	3,298	10,474
User_42 to User_71	2,774	3,294	11,380	11,380	21
User_72 to User_101	15,169	13,321	3,632	3,446	4,216
User_102 to User_124	16	16	16	16	16

Table 6.5.: Comparison cost for the 5 testbed

Table 6.5 shows the average costs for processing gridlets for the six groups of users. As we already mentioned in Chapter 5.3.5, analyzing the output files we can note that the users sent the gridlets to the same resources in the "testbed with medium duration of failure" and in testbed with low duration of failure". For this reason the second and third columns have more or less the same values.

From User_42 to User_101 the average cost is completly different for the "medium duration testbed" and the "medium duration testbed with high init_time", this is because in this last testbed the gridlets are sent when the resources are already recovery. For users form #42 to #71 (in "medium duration testbed with high init_time") the gridlets are submit to resources with a lower computational power and this causes an increase of the average cost. For example, User_45 in medium duration testbed submits the gridlets to Res_Vienna and to Res_Berlin (high computational power), in the medium duration testbed with high init_time, the user submits the gridlets to Res_Warsaw and to Res_Vienna, so the computational power is lower and the average cost increases.

Note that, in "medium duration testbed with high init_time", all gridlets are sent to the resources established at the beginning, except gridlets #0 for group of users between #30 and #41 that are sent to a different resource respect that we had decided. In the following testbed the duration of the failure is set to 1 hour, so all gridlets are submit to the correct resources. The average costs for the fourth and fifth columns are more or less the same because except for the gridlet #0, all gridlets are submit to the same resources in the two different testbeds.

From *User* to *User*	**cost without failure**	**1 hour duration high init_time**
from User_0 to User_9	6	6
from User_10 to User_29	4,320	5,014
from User_30 to User_41	3,038	3,298
from User_42 to User_71	10,786	11,380
from User_72 to User_101	4,320	3,446
from User_102 to User_124	16	16

Table 6.6.: Comparison cost without failures and with a medium duration and high init_time

In Table 6.6, it is possible to compare the average cost without failures of resources and the average cost of "1 hour duration, high init_time". These values are similar because in this last testbed all gridlets are submit when the resources are recovered, so the grid network is completly available. Note that the average costs are similar but the processing of the gridlets finishes later in the "1 hour duration, high init_time" because the gridlets are submit later.

The testbed "high failure duration" for users between #30 and #101 has different values for average cost compared to testbeds with medium and low failure duration. More precisely:

- users from #30 to #71, in "high duration failure" have a huge average cost because the gridlets are submit to Res_Warsaw, Res_Vienna, Res_Budapest and Res_Athens that have low rating, this causes an increase of the cost for processing gridlets.
- users from #72 to #101 have a small average cost for processing the gridlets because them are submit to Res_Milan and Res_Madrid that have a good rating, so the average cost is lower than the other two testbeds.

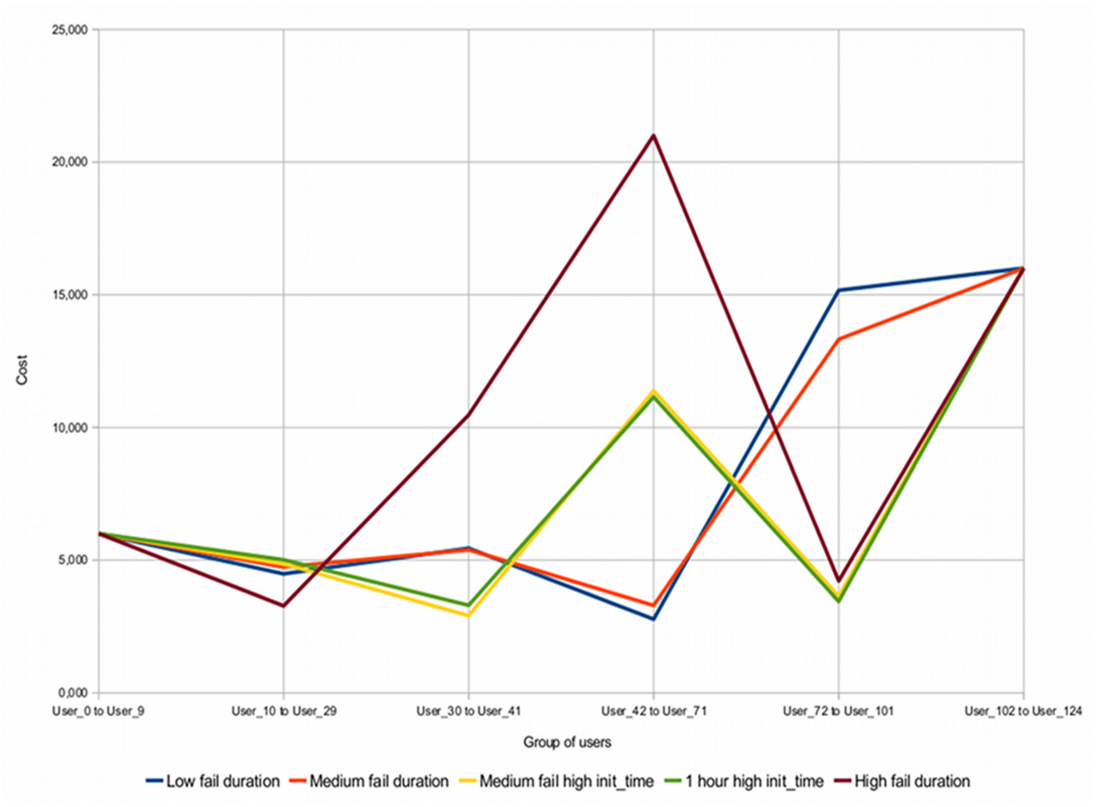

Figure 6.5.: Graph: Comparison cost for the 5 testbed

In the graph 6.5 the purple line shows the trend of the average cost for the testbed "high failure duration" . Note the high peak with respect to the other lines for the group of users between #30 and #41 and between #42 and #71. For the users between #72 and #101 note the low peak with respect to the other tests due to the high rating of the machines that process these gridlets.

Note also that in all testbeds the average cost for the first group of users is equal 6s and for the last group is equals 16s. It's obvious because, for the first group, in all five testbeds, the gridlets are submit to Res_Dublin or Res_Glasgow (depends if Res_Dublin fails or not), these two resources have the same features so the average cost is the same for all testbeds. For the last group of users, in all five testbeds, some gridlets are submit to Res_Budapest, Res_Athens and Res_Lisbon, the other gridlets to Res_Dublin and Res_Glasgow, depending on the resources that are out of order. The first three resources have the same performance as the last two, consequently the average cost for this group of users is the same in the five testbeds.

6.2. Considerations on simulations

After several simulations, with gradual changes of the grid network parameters, we can extrapolate some conclusions. Some of these confirm what we expect.

1. To modified the baud rate between 2 routers we have to change the value in the "network_thesis.txt" file. If we decrease the baud rate between two routers and gridlets travel this link, the value of latency increases. Obviously, if we set an high baud rate, the latency values for those gridlets that travel the link decreases.

2. In the GridSim toolkit it is possible to change the value of baud rate between router and resource. We note that if we decrease this baud rate the values of latency for the gridlets increase, but we can also notice that if a user has to submit 15 gridlets, the latency decreases from the first to the last gridlet.

3. It is possible to change the rating value for the machines of the resource. Table 4.1 shows the rating for each resource, some resources have a high rating (49000 MIPS) some others have low rating (700 MIPS). The rating is the number of instructions per second that the processor of the machine can execute. Obviously, the higher it is, the lower the cost for processing gridlets, and vice versa..

4. After some tests, we see that, if the rating of machine is set very low, then the latency has a big value, and if we have 15 gridlets the latency increases from the first to the last gridlets[1]

5. It is possible to set the number of processors for the machines of the resource; the greater the number of processors, the lower the value of latency. If we have a very small grid network, with 1 user, 1 resource and 1 machine (with 3 processors), and the user has to submit 3 gridlets, then these gridlets have the same latency (if low or high depends on the rating of processors). But if the user has 5 gridlets to submit, the latency for the first three is the same, for the fourth and fifth it increases.[2]

[1]The tests, were done keeping a low baud rate between router and resource (like in point 2), and a very low rating for the machines of the resources. So, if with low baud rate, the latency should decrease from the first to the last gridlet, mixed with a very low rating happens the contrary, it increases from the first to the last gridlet submit.

[2]This usually what happens, but the values of latency depends also from rating of processor, baud rate of the connection and size of the gridlet

6. If the user submits some gridlets to a resource and this resource fails while these gridlets are processed, then the user resubmits the gridlets to the same resource until it is recovered.

Scalability is one of the main characteristics of the grids and indicates the system's ability to grow and decrease according to the needs. Through GridSim toolkit, it is easy to scale by adding to the network "Artificial EU Gid" resources in places where we want and increase the computing power of the network. In our simulations the network "Artificial EU Grid" can expand simply by connecting routers, resources and users, in other words, we can pass from a European grid network, to a network that covers the entire world. However, the architecture of the "Artificial Eu Grid" is not changed for the simulations carried out in this project, so we studied the behavior with the original network without failures and the network with failures (ie with fewer resources available).

With the classes of GridSim toolkit that support the failures, the dependability of the system is always supported, but obviously, there are changes in system performance and user satisfaction. Remember that we have set the GridSim, so that when a failure occurs, the works assigned to the resource that fails are sent to the next available resource. Section 6.1.1 shows the difference between the scenario without failure and with a single failure. The dependability is guaranteed also when Res_Helsinki fails, because the system continues to work normaly, but the gridlets are submits to different resources respect those we had established. If we check the latency we note that there isn't difference between the two scenario, only the cost for processing gridlets changes for some users but not in a excessive way. In section 6.1.2 a set of failure scenario is comapred with the scenario without failures. Also here, the failures are isolated and the correct functioning of the grid network is guaranteed, but in this case gridlets are submit in some case to resources with high computational power and some time with low computational power, so the cost and the latency are different in the two scenario (refers to graphs 6.3 and Table 6.4). Finally, in the section 6.1.3 the Table 6.5 and the graph 6.5 show the comparison between the five tests executed with custom time of failure duration. As we already mentioned the cost with a low and high failure duration are more or less the same. It is interesting to note that we can customise the simulation as we want: if a set of failures occurs but we want to submit the gridlets to the established resources, we must wait for recovery of all resources that failed. This is the test with 1 hour duration of the failure and high init_time (the time when the users begin to submit gridlets). Refers to Chapter 5.3.4. The Table 6.6 proves the equality of the costs when we submit the gridlets to the resources established, but while waiting for all of the resources to be recovery, the gridlets will be sent to the resources and executed later (hence, the time of senting and receiving gridlets is higher). With an high duration of failure, the dependability is guaranteed, and as we expected the cost for processing gridlets is higher for some users because resources with high computational power fail and the gridlets are submit to resources with low computational power, this causes the increase of the average cost.
Applying "Artificial Eu Grid" network we can recommend some things about the large-scale networks. Simulating a large-scale network, the number of resources and machines is very high, then, the network is heterogeneous, so we haven't only machines with good power. Obviously, the possibility of failure of many resources is very high, so to maintain similar performance of the ideal network with no failures, we should have always in each Regional-GIS, resources with the same computational power. This because if one or more resources

fail, users can submit the gridlets to resources that have the same features of the resources that are out of order and so we haven't a decrease of performance.
Configuring the large grid network with the GridSim toolkit, we can manage resources, machines and their power as we want (thanks to reconfigurability provided by the modified toolkit). This allows us to carry out many experiments and configure the network grid to reach the best performances. For example, after several simulation we can know where place resources with high computational power and where place resources with low power for have an increase of performances also when a failure happens.
Is always important to try to deploy the workload on all resources, and if we have the possibility, is better to submit big jobs on the most powerful resources, in this way we can have a less cost for the elaboration and a less latency. This can be done manually through the GridSim toolkit with the changes made, because we can decide which jobs sending to certain resources.
In a large grid network, ensuring reliability is fundamental for prevent coordinated and distributed attacks. So in the construction of the network, a router that are connected many resources must be reach by multiple connections, this because if a failure interrupts a connection, we can reach this by using other connections, without latency or diversions of work on other resources.

7. Conclusion and future works

The primary goal of this project is to simulate a large grid network, "Artifcial EU Grid" with many resources, machines and users with varied requirements and study scalability of systems, efficiency of resources, satisfaction of users and the dependability of the system.

GridSim is the toolkit used to do the several testbeds that simulate the grid network. As mentioned in Chapter 3, the toolkit was modified to adapt to our needs. With calibration of some parameters we can customize the network as we want, for example, if a user actually submits the gridlets to Res_Budapest, but he needs more computational power so he submits them to Res_Munich, it is sufficent to just change the index of the resource in the java code. The toolkit is very flexible and it allows to set heterogeneous types of resources. In "Artificial EU Grid" all resources have a different rating, different number of machines[1] and different baud rate connection (Refers to Table 4.1).

The resources can be located in any time zone, in "Artificial Eu Grid" they are located throughout Europe. Very important for this project is the possibility to specify network speed between router and between resource, in Chapter 6.2 we saw the behavior of latency when there are low connections and where there are fast connections.

There is no limit on the number of application jobs that can be submitted to a resource. However we use a big number of gridlets, but without overdo, because there have been many testbeds, and each testbed take a lot of time for the simulation. A large number of gridlets and large numbers of users and resources cause a high simulation time. If we have a thousand gridlets we need a machine with a good computational power. Simulations performed on the network "Artificial EU Grid", where each of the 125 users submit 15 gridlet (so 1875 gridlet total), last an average of 10 minutes[2]

GridSim Toolkit provides the trace function, that is very useful to know in the simulation made the behavior of the resources and the users. For the users that we want, we can enable the trace function, that after the simulation builds a file for each user, where is possible to see when the gridlets are created, when they are sent, to which resource they are sendt and when they are received.

With the GridSim toolkit, it is possible to perform scheduler performance evaluation in a repeatable and controllable manner as resources and users are distributed across multiple organizations with their own policies. The results confirm the initial expectations, the dependability is ensured by the classes of GridSim dedicated to managing failures. For example, it's clear that if we increase the performance of resources, the cost to process gridlets decreases, the simulations confirm this. The considerations in section 6.2 confirm what we

[1]For simplicity, the number of processors for each machine is the same. It was changed only in some testbeds to check the behavior of the grid network when a machine has many processors and when it has only one.
[2]Simulation performed on a machine with memory equals 1 GB and Intel Core 2 Duo 2.16 GHz.

expected with the exception of point two. But after several test (also on small scenario, that are not reported in this thesis) we can confirm this behavior: when the baud rate between resource and router is set low, the values of latency for the gridlets increase, and if a user has to submit 15 gridlet, the latency decreases from the first to the last gridlet.

Numerous tests have been performed before reaching the simulations presented in this thesis. These tests were performed on smaller grid networks, or on the "Artificial EU Grid" to evaluate the behavior with some parameters set, the results are not presented in this thesis because they have not been fully analyzed and don't cover the objective of the project. Surely, however, are a good start for future development. We have seen that in the "Artificial EU Grid", dependability is guaranteed, but for example we haven't studied the behavior when all resources fail[3] in it. This test was performed, on a small grid network, the result is that we must wait for the first resource recovery to begin to submit gridlets, this, depending on the duration of failure, significantly increases the receving time of the gridlets for the users. So a possible future development is the study of behavior of a large scale grid network with all failed resources.
The tests in this thesis are based on the "Artificial EU Grid" that is a large grid and its "size"[4] was increased to the maximum that the machine on which we perform the simulations could support. Next step is to study the dependability on an even larger network with thousands of users, machines and hundreds of resources. Our simulation has 1875 gridlets; it would be useful to increase the number of these to see the behavior of the network and the increases of the various execution times.
In "Artificial EU Grid" we set to all resources a space shared allocation policy, so when a gridlet fails it will be resubmitted to another machine, as future development we can make the network even more heterogeneous, setting to some resources a time shared allocation policy (in this way if some machines are available on the resource that has failed, the gridlet is not resubmitted but continues its execution on these available machines).
Finally, the last future development, is to take advantage of the capabilities of GridSim toolkit to create a network traffic. In fact, the toolkit contains a class named TrafficGenerator that generates a background traffic, that which has not been explored in the Artificial EU Grid.

[3]Pheraps in reality, it is improbable, but it would be useful to see what happens in this case
[4]With "size" we intend the number of resources, machines, users and gridlets that belong to the grid network

8. Acknowledgment

Thanks are probably the most complex part of the thesis for the simple reason that it is impossible to write in a few lines all the people who have contributed to achieving this important goal of my life (but they know it).

The warmest thanks are certainly to Prof. Dieter Kranzlmüller and Dr. Michael Schiffers who gave me the great opportunity to develop an innovative and compelling thesis at the LMU University of Munich. Special acknowledgment also to Prof. Alberto Trombetta for having followed my thesis project from Italy and for the precious teachings given to me through the course of "Distributed Systems" that have proved useful for the preparation of the project.
Thanks also to Stephan, for the advice he gave me about Grid computing and the Globus toolkit, and a special thanks to Jaime, who corrected the English of this thesis.

A proper thanks to Prof. R. Buyya and the Dept. of Computer Science and Software Engineering at the University of Melbourne, that made available GridSim toolkit, the heart of my project.
They have made available an excellent open source software, with numerous examples and guides that help the understanding of the toolkit and with these I solved many problems. It was very important to me that I could use the GridSim mailing list where could ask some questions, and the people answered to me in few days.

I thank all the guys who were on erasmus in Munich, from my compatriots to the most distant American and Indian students. They were fantastic months, when in addition to the study I expanded my language skills, I visited fabulous places and I had the good fortune to observe many different cultures. It was an experience that changed my way of dealing with life. Thanks also to the German people who have taught me much, except for punctuality, a defect that I carry with me for all my life.

The friends list to thank would be endless but I can nominate Alberto, Luca, Stefano who have always supported me in both good times and those that were difficult, but to tell the truth, the latter were few. A special thanks goes to my university friends Barba, Nella, Danz and the two crazy guys Tia and Tito, who are now not only classmates, but true friends. Thanks to them, these 5 years have flown by and were really pleasant and unforgettable. I will probably now begin a new phase of my life away from the academic world, but I certainly won't begin to forget you.

Finally a special thanks to my family: I'm sorry if the study took so much of my time and that I couldn't always be around, but at the same time, I'm happy to have reached this goal in my life that I would not have achieved without all of your moral and financial support along the way.

Thanks also to those who haven't believed in me about the study, you gave me an incentive to continue and thanks to all those people I have not mentioned but who have helped me to grow and to complete my studies.

A. Appendix: modified code

A.1. Changes in User-Router connection

A.1.1. Original code

Listing A.1: Original Code: connect User-Router

```
[....]
trace_flag = true;
  for (int i = 0; i < num_user; i++)
    {
      String userName = NAME + "User_" + i;

      // a network link attached to this entity
      Link link2 = new SimpleLink(userName + "_link", baud_rate,
         propDelay, mtu);

      // only keeps track activities from User_0
      if (i != 0)
       {
         trace_flag = false;
       }
      GridUserFailureEx03 user = new GridUserFailureEx03(userName,
         link2, pollTime, glLength, glSize, glSize, trace_flag);
      user.setGridletNumber(totalGridlet);

      // link this user to a router
      String routerName = null;
      if (random.nextBoolean() == true)
       {
        linkNetwork(router0, user);
        routerName = router0.get_name();
       }
      else
       {
       linkNetwork(router1, user);
       routerName = router1.get_name();
      }

      // randomly select which GIS to choose
      int index = random.nextInt( gisList.size() );
    RegionalGISWithFailure gis = (RegionalGISWithFailure) gisList
        .get(index);
    user.setRegionalGIS(gis); // set the regional GIS entity
[....]
```

A.1.2. Modified code

Listing A.2: Modified Code: connect User-Router

```
[....]
trace_flag = true;
  for (int i = 0; i < num_user; i++)
    {
     String userName = NAME + "User_" + i;

     // a network link attached to this entity
     Link link2 = new SimpleLink(userName + "_link", baud_rate
                              propDelay, mtu);

     // only keeps track activities from User_0
     if (i != 0)
      {
       trace_flag = false;
      }

     GridUserFailureEx03 user = new GridUserFailureEx03(userName,
         link2, pollTime, glLength, glSize, glSize, trace_flag);
     user.setGridletNumber(totalGridlet);

     // link this user to a router
     String routerName = null;
     RegionalGISWithFailure1 gis = null;
     int statement = 0; // variable for link User-Router in the
         switch-case below
     if (i<48)
                {//for Users from 0 to 47 the variable statement
                    =0;
                 statement=0;
                }
     if (48<=i && i<=79)
                { //for Users from 48 to 79 the variable
                    statement=1;
                 statement=1;
                }
     if (80<=i && i<=87)
                { //for Users from 80 to 87 the variable statement
                    =2;
                    statement=2;
                  }
     if (88<=i && i<100)
                  { //for Users from 88 to 100 the variable
                     statement=3;
                    statement=3;
                  }
     switch (statement)
        { //Check the variable "statement" and link User to a
            Router
                  case 0:
                         linkNetwork(router2, user);
                         routerName = router2.get_name();
                         gis = (RegionalGISWithFailure1) gisList.
                            get(0);
                         user.setRegionalGIS(gis);
                         break;
                  case 1:
                         linkNetwork(router3, user);
                         routerName = router3.get_name();
                         gis = (RegionalGISWithFailure1) gisList.
                            get(1);
                         user.setRegionalGIS(gis);
                         break;
                  case 2:
                         linkNetwork(router4, user);
                         routerName = router4.get_name();
                         gis = (RegionalGISWithFailure1) gisList.
                            get(1);
                         user.setRegionalGIS(gis);
                         break;
                  case 3:
                         linkNetwork(router7, user);
                         routerName = router7.get_name();
                         gis = (RegionalGISWithFailure1) gisList.
                            get(2);
                         user.setRegionalGIS(gis);
                         break;
                  default:
                        System.out.println("Not a recognized test
                           case.");
                        break;
        }

 System.out.println();
[....]
```

A.2. Changes in Resource-Router connection

A.2.1. Original code

Listing A.3: Original Code: connect Resource-Router

```
[....]
for (int i = 0; i < totalResource; i++)
  {
   // creates a new grid resource
   String resName = NAME + "Res_" + i;
   GridResourceWithFailure res = createGridResource(resName,
   baud_rate, propDelay, mtu, totalPE, totalMachines,
      rating, sched_alg);

   if (i \% 2 == 0)
    {
     trace_flag = true;
    }
   else
    {
     trace_flag = false;
    }
   resList.add(res);           // add a resource into a list
   res.setTrace(trace_flag);   // record this resource activity

   // link these GIS to a router
   String routerName = null;
   if (random.nextBoolean() == true)
    {
     linkNetwork(router0, res);
     routerName = router0.get_name();
    }
   else
    {
     linkNetwork(router1, res);
     routerName = router1.get_name();
    }

   // randomly select which GIS to choose
   int index = random.nextInt( gisList.size() );
   RegionalGISWithFailure gis = (RegionalGISWithFailure) gisList.
      get(index);
   res.setRegionalGIS(gis); // set the regional GIS entity
[....]
```

A.2.2. Modified code

Listing A.4: Modified Code: connect Resource-Router

```
[....]
for (int i = 0; i < totalResource; i++)
  {
   String resName = null;
   String routerName = null;
   RegionalGISWithFailure2 gis = null;

   //2 baud_rate for link Resource-Router
   //1 high and 1 low. Is important. when a resource fail,
      probably
   //I sent the gridlet to another resource with a low connection
   double baud_rate_high=        1 * 1000000000;
   double baud_rat_low= 0.0001 * 1000000000;
   GridResourceWithFailure res = null;
   // creates a new grid resource
   switch(i)
     {
       case (0):
          resName = NAME + "Res_Dublin";
          res = createGridResource(resName,
                       baud_rate_high, propDelay, mtu, totalPE,
                          totalMachines,
                       rating, sched_alg);
          linkNetwork(router0, res);
          routerName = router0.get_name();
          gis = (RegionalGISWithFailure2) gisList.get(0);
          res.setRegionalGIS(gis);
            break;

       case (1):
         resName = NAME + "Res_Glasgow";
         res = createGridResource(resName,
                     baud_rate_high, propDelay, mtu, totalPE,
                        totalMachines,
                     rating, sched_alg);
         linkNetwork(router0, res);
         routerName = router0.get_name();
         gis = (RegionalGISWithFailure2) gisList.get(0);
         res.setRegionalGIS(gis);
           break;

       [....]
```

```
            case (17):
                resName = NAME + "Res_Brussels";
                res = createGridResource(resName,
                                baud_rate_high, propDelay, mtu, totalPE,
                                    totalMachines,
                                rating, sched_alg);
                linkNetwork(router11, res);
                routerName = router11.get_name();
                gis = (RegionalGISWithFailure2) gisList.get(4);
                res.setRegionalGIS(gis);
                break;

            default:
                System.out.println("Not a recognized test case.");
                break;
        }
        if (i \% 2 == 0)
          {
           trace_flag = true;
          }
        else
         {
          trace_flag = false;
         }
        resList.add(res);    // add a resource into a list
        res.setTrace(trace_flag);    // record this resource activity
[....]
```

A.3. Changes failure parameters

A.3.1. Original code

Listing A.5: Original Code: setting failure parameters

```
[....]
for (int i = 0; i < num_GIS; i++)
  {
    String gisName = NAME + "Regional_GIS_" + i;   // GIS name

    // a network link attached to this regional GIS entity
    Link link = new SimpleLink(gisName + "_link", baud_rate,
                                                propDelay, mtu);

    // HyperExponential: mean, standard deviation, stream
    // how many resources will fail
    HyperExponential failureNumResPattern =
            new HyperExponential(totalMachines / 2, totalMachines, 4);

        // when they will fail
        HyperExponential failureTimePattern =
            new HyperExponential(25, 100, 4);

        // how long the failure will be
        HyperExponential failureLengthPattern =
            new HyperExponential(20, 25, 4); // big test: (20, 100, 4);


        // creates a new Regional GIS entity that generates a resource
        // failure message according to these patterns.
        RegionalGISWithFailure gis = new RegionalGISWithFailure(gisName
            , link, failureNumResPattern, failureTimePattern,
                            failureLengthPattern);
        gis.setTrace(trace_flag);    // record this GIS activity
        gisList.add(gis);    // add into the list

        // link these GIS to a router
        String routerName = null;
        if (random.nextBoolean() == true)
         {
          linkNetwork(router0, gis);
          routerName = router0.get_name();
         }
        else
         {
          linkNetwork(router1, gis);
          routerName = router1.get_name();
         }
[....]
```

A.3.2. Modified code

Listing A.6: Modified Code: setting failure parameters

```
[....]
for (int i = 0; i < num_GIS; i++)
 {
   String gisName = NAME + "Regional_GIS_" + i;   // GIS name

   // a network link attached to this regional GIS entity
   Link link = new SimpleLink(gisName + "_link", baud_rate,
                              propDelay, mtu);

   // how many resources for each RegionaGIS fail
```

```
int failureNumResPattern= 2;

// how many machine for each resource fail
int failureNumMacPattern= 12;

// when the resources will fail
int failureTimePattern = 0;

// how long the failure will be
int failureLengthPattern = 0;

 switch (i){ // we set the 4 variables as we want for each
    RegionalGIS

 case 0: failureNumResPattern=1;
         failureNumMacPattern=1;
         failureTimePattern = 360;
         failureLengthPattern = 1*1000;
                 break;

 case 1: failureNumResPattern=0;
         failureNumMacPattern=0;
         failureTimePattern = 390;
         failureLengthPattern = 1*1000;
                 break;
 }


// creates a new Regional GIS entity that generates a resource
// failure message according to these patterns.
RegionalGISWithFailure3 gis = new RegionalGISWithFailure3(
   gisName,
           link, failureNumResPattern, failureTimePattern,
           failureLengthPattern, failureNumMacPattern);

gis.setTrace(trace_flag);   // record this GIS activity
gisList.add(gis);   // add into the list

 //  link these GIS to a established router
 String routerName = null;
 switch (i){

      case (0):
              linkNetwork(router0, gis);
              routerName = router0.get_name();
               break;
```

```
        case (1):
                linkNetwork(router1, gis);
                routerName = router1.get_name();
                 break;

        default:
               System.out.println("Not a recognized test case.");
                 break;
                }
[....]
```

A.4. Change init_time

A.4.1. Original code

Listing A.7: Original Code: setting initTime

```
[....]
int PAUSE = 10*60;  // 10 mins. Wait to allow GIS to receive
    registrations from resources.
       Random random = new Random();
       int init_time = PAUSE + random.nextInt(5*60);
       // sends a reminder to itself
       super.send(super.get_id(), init_time, GridUserFailureEx03.
          SUBMIT_GRIDLET);
  [....]
```

A.4.2. Modified code

Listing A.8: Modified Code: setting initTime

```
[....]
int PAUSE = 10*60;  // 10 mins.  Wait to allow GIS to receive
    registrations from resources.
        int init_time = PAUSE + 5*60;
        // sends a reminder to itself
        super.send(super.get_id(), init_time, GridUserFailureEx03.
           SUBMIT_GRIDLET);
[....]
```

A.5. Set which resources fail

A.5.1. Original code

Listing A.9: Original Code: resources that fail are chosen at random

```
[....]
boolean isWorking;
 do
  {
   res_num = random.nextInt(resList_.size());
   res_id_Integer = (Integer) resList_.get(res_num);
   res_id = res_id_Integer.intValue();
   resChar = getResourceCharacteristics(res_id);

   if (resChar == null)
     {
       System.out.println(super.get_name() + " resChar == null");
       isWorking = false;
      }
   else
      {
        isWorking = resChar.isWorking();
      }
  } while (isWorking == false);

       FailureMsg resFailure = new FailureMsg(failureLength,
          res_id);
       resFailure.setNumMachines(numMachFailed);

       // Sends the recovery time for this resource. Sends
       // a deferred event to itself for that.
       super.send(super.get_id(),
          GridSimTags.SCHEDULE_NOW + failureLength,
          GridSimTags.GRIDRESOURCE_RECOVERY, resFailure);

       /****************/
       if (record_ == true) {
          System.out.println(super.get_name() +
          ": sends a GRIDRESOURCE_FAILURE event to the resource " +
          GridSim.getEntityName(res_id) + ". numMachFailed: " +
             numMachFailed + ". Clock: " + GridSim.clock() +
             ". Fail duration: " + (failureLength / 3600) +
             " hours. Some machines may still work or may not.");
        }

       /****************/
       // Send the GRIDRESOURCE_FAILURE event to the resource.
```

```
        super.send(super.output, 0.0, ev.get_tag(),
          new IO_data(resFailure, Link.DEFAULT_MTU, res_id));
      }
  } // else (if begining)
[....]
```

A.5.2. Modified code

Listing A.10: Modified Code: resources that fail are chosen with a criterion

```
[....]
int res_num=0;
boolean isWorking;
do
 {
  for (int j=0; j<resList_.size() -1; j++){
   for (int k=j+1; k < resList_.size(); k++){
      res_id_Integer = (Integer) resList_.get(j);
      res_id = res_id_Integer.intValue();
      System.out.println("first comparison value: " + res_id);

      res_id_Integer_1 = (Integer) resList_.get(k);
      res_id_1 = res_id_Integer_1.intValue();
      System.out.println("second comparison value: " + res_id_1);

      if (res_id > res_id_1)
       {
        int memory = res_id;
        resList_.set(j, res_id_1);
        resList_.set(k, memory);
        System.out.println("change made");
       }
      else {System.out.println("not changed");}
      }
    }
  for (int j=0; j<resList_.size(); j++){
    res_id_Integer = (Integer) resList_.get(j);
    res_id = res_id_Integer.intValue();
    System.out.println("Values: " + res_id );
   }

  if (gisName.equals("ArtificialEU_Regional_GIS_0")) {res_num=0;}
          //In RegionalGIS0 fail the first resource
  if (gisName.equals("ArtificialEU_Regional_GIS_1")) {cont = cont
      + 1;//In RegionalGIS1 fail the first(0) and fourth(3)
    resources
        if (cont== 1)
        {
        res_num=0;
        }
        else {res_num=3;}}
if (gisName.equals("ArtificialEU_Regional_GIS_2")) {res_num=1;}
        //In RegionalGIS0 fail the second resource
if (gisName.equals("ArtificialEU_Regional_GIS_3")) {res_num=2;}
        //In RegionalGIS0 fail the third resource
if (gisName.equals("ArtificialEU_Regional_GIS_4")) {res_num=1;}
        //In RegionalGIS0 fail the second resource

System.out.println("il resnum " + res_num);
res_id_Integer = (Integer) resList_.get(res_num);
res_id = res_id_Integer.intValue();
System.out.println(res_id);
resChar = getResourceCharacteristics(res_id);

if (resChar == null)
 {
  System.out.println(super.get_name() + " resChar == null");
  isWorking = false;
 }
else
 {
  isWorking = resChar.isWorking();
 }
 } while (isWorking == false);

 FailureMsg resFailure = new FailureMsg(failureLength, res_id);
 resFailure.setNumMachines(numMachFailed);

 // Sends the recovery time for this resource. Sends
 // a deferred event to itself for that.
 super.send(super.get_id(),
     GridSimTags.SCHEDULE_NOW + failureLength,
     GridSimTags.GRIDRESOURCE_RECOVERY, resFailure);

 /****************/
 if (record_ == true) {
   System.out.println(super.get_name() +
   ": sends a GRIDRESOURCE_FAILURE event to the resource " +
   GridSim.getEntityName(res_id) + ". numMachFailed: " +
   numMachFailed + ". Clock: " + GridSim.clock() +
   ". Fail duration: " + (failureLength / 3600) +
   " hours. Some machines may still work or may not.");
 }
```

```
  /*****************/
  // Send the GRIDRESOURCE_FAILURE event to the resource.
  super.send(super.output, 0.0, ev.get_tag(),
      new IO_data(resFailure, Link.DEFAULT_MTU, res_id));
  }
} // else (if begining)
[....]
```

A.6. Submit the gridlet to a specific resource

A.6.1. Original code

Listing A.11: Original Code: gridlets are submit to a random resource

```
[....]
// If we have resources in the list
if ((resourceID != null) && (resourceID.length != 0))
 {
  index = random.nextInt(resourceID.length);

  // make sure the gridlet will be executed from the begining
  resetGridlet(gl);

  // submits this gridlet to a resource
  super.gridletSubmit(gl, resourceID[index]);
  gridletSubmissionTime[gl.getGridletID()] = GridSim.clock();

  // set this gridlet as submitted
  ((GridletSubmission) GridletSubmittedList_.get(i)).setSubmitted(
      true);
[....]
```

A.6.2. Modified code

Listing A.12: Modified Code: gridlets are submit to a specific resource

```
[....]
// If we have resources in the list
if ((resourceID != null) && (resourceID.length != 0))
 {
  index = 0;
  String name_actual_user = super.get_name();
  System.out.println("ACTUAL_USER:" + name_actual_user);

  switch (Stato.valueOf(name_actual_user)){
    // User_0 submit the first 10 gridlets to the first resource (
        index=0)
    // and the others gridlets to the second resource (index=1)

    case ArtificialEU_User_0: if (0<= i && i <= 9)
                                                {index = 0;}
                                                    else {index = 1;}//
                                                  break;

    // User_1 submit the first 10 gridlets to the first resource (
        index=0)
    // and the others gridlets to the second resource (index=1)
    case ArtificialEU_User_1: if (0<= i && i <= 9)
                                                {index = 0;}
                                                    else {index = 1;}
                                                    break;

    // User_2 submit his gridlets to the second resource (index=1)
    case ArtificialEU_User_2: index = 1;
                                                   break;

    // User_3 submit his gridlets to the second resource (index=1)
    case ArtificialEU_User_3: index = 2;
                                                   break;
            }

    System.out.println("ACTUAL_INDEX:" + index);

    // make sure the gridlet will be executed from the begining
    resetGridlet(gl);

    // submits this gridlet to a resource
    super.gridletSubmit(gl, resourceID[index]);
    gridletSubmissionTime[gl.getGridletID()] = GridSim.clock();

    // set this gridlet as submitted
    ((GridletSubmission) GridletSubmittedList_.get(i)).setSubmitted
        (true);
[....]
```

A.6.3. Original code for function getResList()

Listing A.13: Original Code: function getResList()

```
[....]
private int[] getResList()
 {
   Object[] resList = super.getLocalResourceList();
   int resourceID[] = null;

   // if we have any resource
   if ((resList != null) && (resList.length != 0))
    {
     resourceID = new int[resList.length];
      for (int x = 0; x < resList.length; x++)
        {
         // Resource list contains list of resource IDs
         resourceID[x] = ((Integer) resList[x]).intValue();

         if (trace_flag == true)
           {
             System.out.println(super.get_name() +
             ": resource[" + x + "] = " + resourceID[x]);
           }
        }
 }
 return resourceID;
 }
[....]
```

A.6.4. Modified code for function getResList()

Listing A.14: Modified Code: function getResList()

```
[....]
private int[] getResList()
  {
   Object[] resList = super.getLocalResourceList();
   Object[] resList1= super.getGlobalResourceList();
    String d=super.get_name();
    System.out.println("User name considered ': " + d);
     int resourceID[] = null;
     int Array[]=null;
     int lengthtotal= resList.length + resList1.length;

     System.out.println("Total number of local resources ': " +
         resList.length);
     System.out.println("Total number of global resources ': " +
        resList1.length);
     System.out.println("Total number of resources ': " +
        lengthtotal);
       Array = new int[lengthtotal];
       int mem=0; // is a variable that save the position of the
          array index
        if (resList.length !=0)
         {
          for (int y=0; y<resList.length; y++){
            Array[y] = ((Integer) resList[y]).intValue();
             mem = y;
           };
         }
        if (resList.length ==0){mem=-1;}
         for (int k=0; k<resList1.length; k++){
           mem++;
          Array[mem] = ((Integer) resList1[k]).intValue();
          }

     //we order the resources avaialable by id
     for (int j=0; j<Array.length -1; j++){
      for (int k=j+1; k < Array.length; k++){
       int res_id = Array[j];
        System.out.println("first value compared: " + res_id);
        int res_id_1 = Array[k];
            System.out.println("second value compared: " +
               res_id_1);
              if (res_id > res_id_1){
                 int memory = res_id;
                 Array[j] = res_id_1;
                Array[k] = memory;
                System.out.println("change made");
              }
             else {System.out.println("not changed");}
       }
      }

// if we have any resource
if ((resList != null) || (resList1 != null) && (lengthtotal != 0))
 {
  resourceID = new int[lengthtotal];
  for (int x = 0; x < lengthtotal; x++)
   {
     // Resource list contains list of resource IDs
     resourceID[x] = Array[x];
```

```
        if (trace_flag == true)
         {
          System.out.println(super.get_name() +
               ": resource[" + x + "] = " + resourceID[x]);
         }
   }
  }
return resourceID;
}
[....]
```

List of Figures

List of Tables

Listings

Bibliography

[1] Gridsim - a grid simulation toolkit for resource modelling and application scheduling for parallel and distributed computing. `http://www.cloudbus.org/gridsim/`.

[2] How grid computing works. `http://www.nm.ifi.lmu.de/`.

[3] Image of europe. `http://www.immaginieuropa.com/`.

[4] Oracle solaris. `http://www.oracle.com/us/products/servers-storage/solaris/index.html`.

[5] Wikipedia - maximum transmissio unit. `http://en.wikipedia.org/wiki/Maximum_transmission_unit`.

[6] Wikipedia - taxonomy describing relationship between dependability security, attributes, threats and means. `http://en.wikipedia.org/wiki/File:Dep-1.jpg`.

[7] Anthony Sulistio et al. A toolkit for modelling and simulating data grids: An extension to gridsim. *Concurrency and Computation: Practice and Experience (CCPE)*, 20, 2008.

[8] International Conference on Parallel and Distributed Systems. *Extending GridSim with an Architecture for Failure Detection*, 2007.

[9] Barboni Stefano. *Assessment of dependability scenarios in large-scale grids using GridSim toolkit.* Thesis, 2011.