Project no. FP6-034442

GridCOMP

Grid programming with COMPonents: an advanced component platform for an effective invisible grid

STREP Project

Advanced Grid Technologies, Systems and Services

D.UC.05.A – Use cases: final documentation

(Manual and detailed architectural design)

Due date of deliverable: 1 December 2008

Actual submission date: 19 January 2009

Start date of project: 1 June 2006 **Duration**: 33 months

Organisation name of lead contractor for this deliverable: GS

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	PUBLIC	PU

Keyword List: use case, prototype, component, GCM
Responsible Partner: Gastón Freire, GS

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
1.0	01-12-2008	Draft	Fabio Tumiatti, Irati R. Sáez de Urabain, Thomas Weigold, Gastón Freire
1.1	17-12-2008	Draft	Fabio Tumiatti, Irati R. Sáez de Urabain, Thomas Weigold, Gastón Freire
1.2	12-01-2009	Final	Gastón Freire

Deliverable manager

- Gastón Freire, GS

List of Contributors

- Thomas Weigold, IBM
- Fabio Tumiatti, ATOS
- Irati R. Sáez de Urabain, ATOS
- Gastón Freire, GS

List of Evaluators

- Magdalena Escalas, GS
- Marco Danelutto, UNIPI

Summary

- This document describes the final prototypes of the use case applications. For each use case, its architectural design is explained in depth, offering complete details about the components (both primitive and composite) and their interfaces. Also, a comprehensive user manual is offered, including installation and running instructions, along with screenshots of the final prototype applications. Finally, the impact of GridCOMP and the Grid Component Model (GCM) on the different use cases is summarized in the conclusions.

Table of Content

1 Introduction

This document describes the final prototypes of the use cases for the GridCOMP project. These use cases are not only a presentation of the technical functionalities developed within the project, but they have also been selected because of their exploitation potential. They will be used to showcase the project results in front of both the industrial and the academic worlds. The selection of use-cases pursued the following criteria in order to obtain a comprehensible, varied and truly representative list of examples and their potential:

- Use-cases selected had to cover multiple industrial sectors and scientific research areas, such as management, security, engineering, or telecommunications, in order to appeal to a large audience, and proving that the framework developed can be applied to different industrial environments.
- Use-cases selected had to prove the flexibility and interoperability of GridCOMP, interacting with services generally deployed in these areas, such as database servers, workflow systems, etc.
- Use-cases selected had to demonstrate legacy application integration. The possibility of seamlessly integrating these applications to build more advanced ones with optimized performance (thanks in part to the distribution of the processing effort).
- Use-cases selected had to show the effectiveness of the solution for both computing and data intensive processes.
- Use-cases selected had to address and tackle issues with both embarrassingly parallel processes and stateful distributed applications, in such a way that the solutions can be redeployed to solve more complex situations
- Use-cases selected had to attest the autonomic resource management features, in order to achieve different objectives: better performance (throughput) or a reduced response time.

Taken as a whole, the use-cases selected in GridCOMP meet the above criteria and the partners responsible for each selected use-case proved their reliability and ensured high-quality and performance in the project. The applications selected for the use cases are the following:

1. Biometric Identification System, by IBM Zurich Research Labs.
2. Computing of DSO Value, by Atos Origin.
3. EDR Processor, by GridSystems.
4. Wing Design, by GridSystems.

Each one is covered in a separated section of this document, all of which have a common structure:

1. A detailed description of the architectural design and the infrastructure needed to run the application (data bases, application servers, workflow systems, third-party software components, etc.). All the components (both primitive and composite) and their interfaces are explained in detail.
2. Manual of the final prototype, including instructions on how to install or deploy the application. Several screenshots of the applications in action are included, and a set of examples is provided.

Along with this document, a set of 4 compressed files (D.UC.05.B*.zip) is provided, containing the code (binaries and/or source) of the four use case final prototypes. In order to run them, a common set of tools is required: Java [5], Apache Ant [6] and ProActive 3.90 [7].

2 Biometric Identification System

The IBM use case application is a biometric identification system (BIS) based on fingerprint biometrics, which works on a large user population. The core problem is to identify a given person solely on his biometric information by comparing its fingerprints against a large database of enrolled (known) identities. This requires massive computing power because biometric matching algorithms are non trivial and must be applied may times. Therefore, the identification system takes advantage of a Grid infrastructure and an appropriate GCM component system. The identification problem is distributed across the nodes in the Grid and this way real-time identification performance can be achieved even when working on a very large user population. Additionally, the system scales independently in accordance with a given quality of service (QoS) contract thanks to the autonomic reconfiguration functionality provided by the GridCOMP GCM framework.

2.1 Detailed architectural design

2.1.1 High-level architecture of the application

The high-level architectural design of the BIS as outlined in D.UC.03/04 ([1], [2]) and shown in Figure 1 has been retained for the final prototype described in this document. However, under the covers, there have been many changes in the way the system is implemented. The parts of the system that have undergone significant changes are the GCM component architecture and the GCM adapter, the business processes (workflow scripts) interacting with the GCM adapter, and the demo application. The main reason for this is the fact that, during the second and third year of the project, we have considered the use of various WP3 results, namely, we have implemented and improved the BIS by using autonomic behavioural skeletons. In a first iteration we used the task-parallel farm skeleton as documented in D.UC.04 [2]. For the final prototype we moved to the data-parallel skeleton, which has been developed by the WP3 partners to specifically support data-parallel applications such as the BIS use case. The details of the final BIS prototype based on the data-parallel skeleton are described in the following subsections.

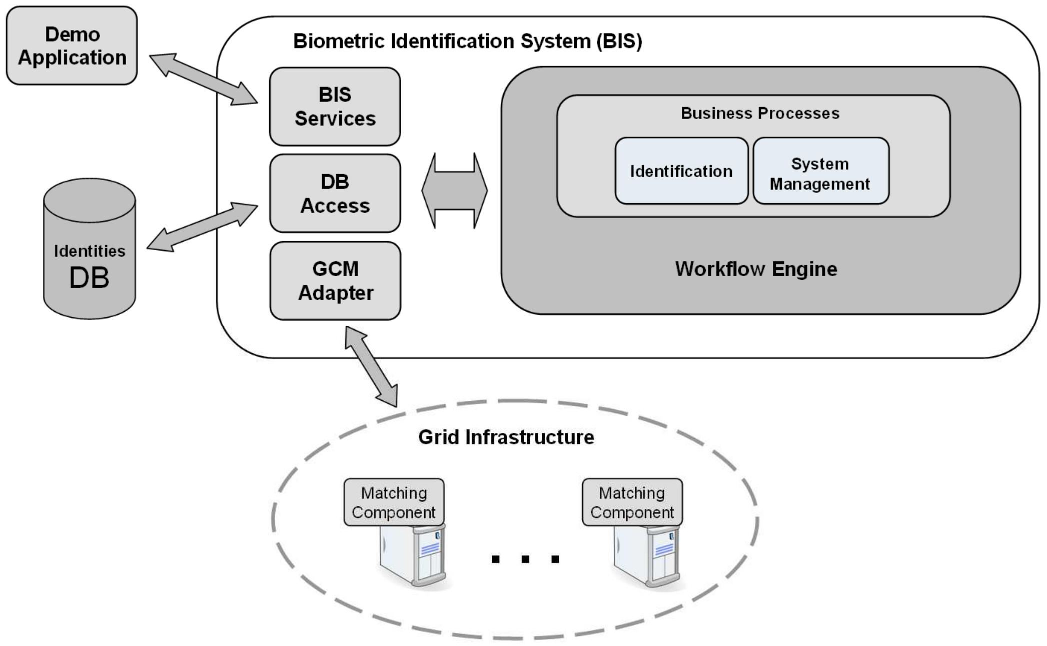


Figure1: Biometric identification system high-level overview

Figure 1 outlines the high-level system design of the BIS. It is built around a workflow execution engine acting as the central control unit of the system. A number of business processes are implemented as workflow scripts running within the engine. The processes comprise functionality accessible from the demo application via the BIS services (e.g. identification, QoS definition) as well as internal system management logic required to control the distributed biometric matching. Furthermore, the BIS provides a number of adapters to the workflow engine such that the business processes can interact with other external entities, namely, the identity database storing information about enrolled identities, and the interface to the Grid infrastructure.

2.1.2 Business Processes

The business processes for BIS management and for the actual identification functionality are interacting with the Grid via the GCM adapter. Consequently, the change in the component architecture, where we now make use of the data-parallel skeleton, also affects the logic implemented in the corresponding workflow scripts. The “startup” process, as illustrated in Figure 2, no longer allocates the desired initial number of workers. This has to do with the way the skeleton works. To be able to define the initial number of workers the ADL definition of the skeleton must be adapted before it is deployed. This functionality has been implemented as part of activity 2. Another modification is that, in contrast to the task parallel approach used in D.UC.04 [2], now the database must be initially distributed to all workers. This is done in activity 4.

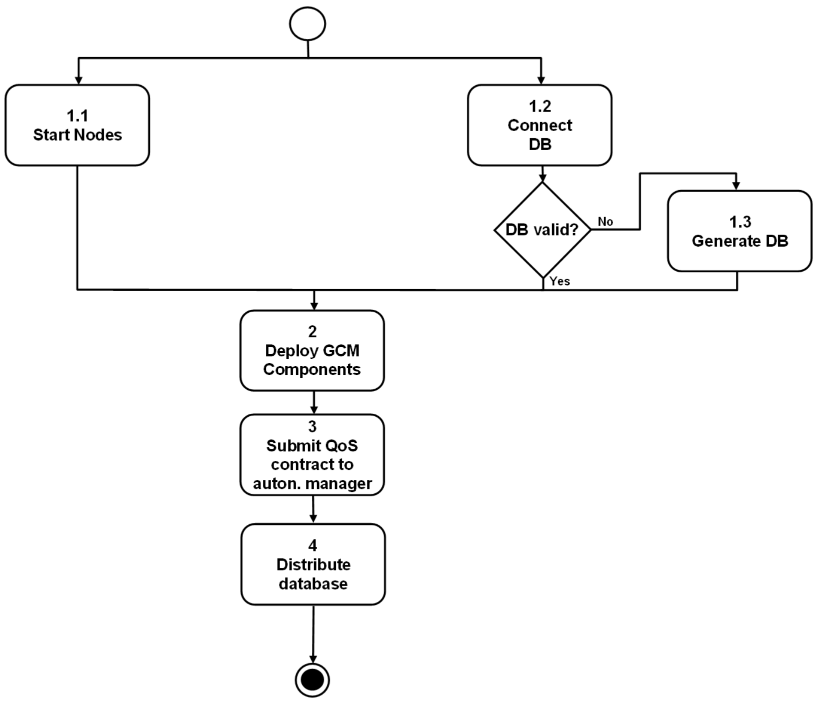
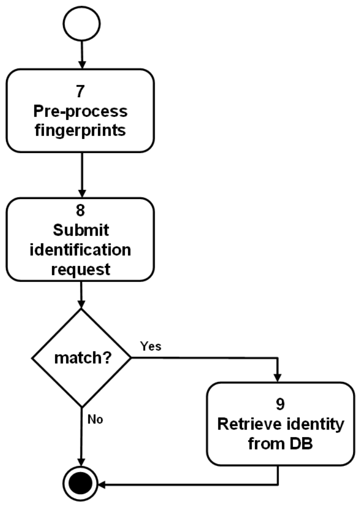


Figure 2: Business process “startup” activity-flow diagram

The identification process, named “identify”, has changed as well. As the data-parallel skeleton fits much better to the problem than the task farm, the identification process became much simpler. Instead of generating tasks for the farm and collecting asynchronous results it now just submits the identification request including the fingerprints of the person to be identified to the skeleton and receives the result synchronously in activity 8. Figure 3 illustrates the control flow within the identification process.



7 Pre-process fingerprints
8 Submit identification request
match?
Yes
No
9 Retrieve identity from DB

Figure3: Business process “identify” activity-flow diagram

2.1.3 Workflow adapters

The workflow adapters, as indicated in Figure 1, have already been defined in D.UC.03 [1]. As the services they provide to the workflow scripts have not changed substantially, not all of them are described again at this point. In particular, the implementation of the DB adapter and the BIS services API remained unchanged. However, the implementation of the GCM adapter has changed significantly as the architecture of the GCM component system changed through the use of skeletons. The details on how the GCM adapter of the final prototype implements the distributed biometric matching via GCM components is presented in the following Section.

2.1.4 GCM Components

This section describes the GCM component architecture, component description, and interface description of the final BIS prototype.

2.1.4.1 Components diagram

The final BIS prototype, in contrast to the previous version described in D.UC.04 [2], is now based on the data-parallel autonomic farm skeleton developed within WP3. Figure 4 illustrates the GCM component architecture used in the prototype and indicates how it interacts with the non-componentized part of the application.

At the heart of the component system, there is the data-parallel behavioural skeleton (BeSke), as defined in D.NFCF.04 [3], which consists of the composite component highlighted in yellow. It includes a custom controller, the autonomic behaviour controller (*ABC*), which implements the mechanisms supporting autonomic functionality, for instance, increasing or decreasing the number of worker components and data redistribution. Furthermore, the farm includes a default implementation of an autonomic manager (*AM*) component, the component actually implementing autonomic control of the BeSke. These three components represent the data-parallel skeleton. To apply the skeleton, we need to parameterise it by adding a worker component, here named *IDMatcher*. This is the component which the ABC clones and adds to the farm as many times as required to increase parallelism. By default, the skeleton starts with the number of matcher components defined in the skeletons ADL. Therefore, the ADL is adapted accordingly before it is deployed. Since the data-parallel skeleton fits very well to the biometric matching problem we have to solve, there is no need to develop any additional custom components. We only wrap the skeleton into an *Application* component to hide it from the non-componentized part of the BIS application. The dashed lines in Figure 4 illustrate the interaction between the non-componentized part of the BIS application, namely the GCM adapter, and the GCM component system. Furthermore, Figure 4 indicates that all *IDMatcher* components have access to a shared database storing the identities known by the BIS. As far as deployment is concerned, the idea is that only the *IDMatcher* components, which represent the workers, are to be distributed.

When the BIS application is started, it submits a QoS contract to the AM which defines the conditions under which the AM should increase or decrease the number of workers. In the prototype, the AM enforces this QoS contract every two seconds. Furthermore, the prototype allows updating the QoS contract at runtime to be able to trigger reconfiguration during the demos.

Before identification requests can be processed, the identity database must be initially distributed across the worker components. The identity DB holds information such as name, address, and fingerprints of all enrolled (known) people. For distributing the DB, the skeleton offers a multicast port, also called the scatter (c.f. Figure 4) or initialization port, which takes a list of tasks as input parameter. In case of the BIS, each task defines a part (defined by index and length) of the identity DB. The skeleton distributes these tasks equally among the workers and the workers load the parts from the shared DB. As a result, each worker can be considered to have one partition of the DB loaded into transient memory as indicated in Figure 4. In other words, a partition is the fragment of the DB (a number of records) each worker has loaded.
Once the skeleton has been initialized, identification requests can be submitted to the second multicast port provided by the skeleton, the so-called broadcast port. Fingerprints of the person to be identified are broadcasted via this port to all worker components and each worker matches them against its partition of the DB. Results are, in contrast to the task parallel farm used in D.UC.04, returned synchronously via method return values.
If the AM triggers reconfiguration via the ABC, for example, to increase the number of worker components, the ABC retrieves all tasks from all workers, modifies the number of workers, and finally redistributes the tasks. This way the DB is redistributed during each reconfiguration operation.

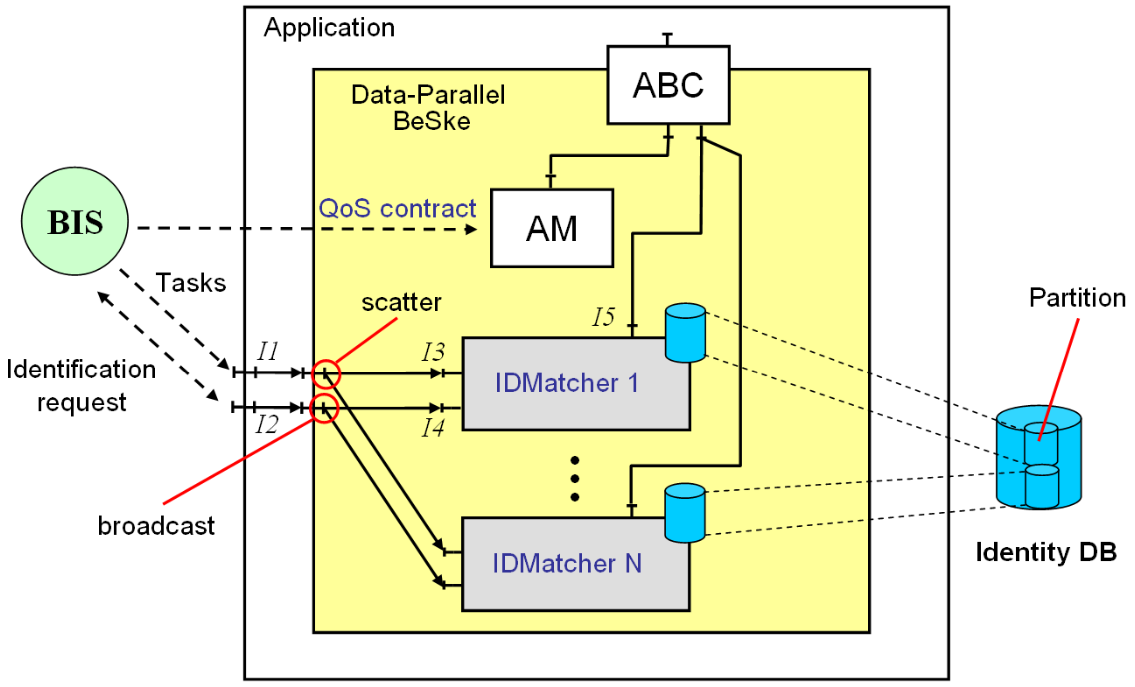


Figure 4: GCM component architecture

Figure 5 shows how the component system has been graphically composed within the Grid Integrated Development Environment (GIDE) [14].

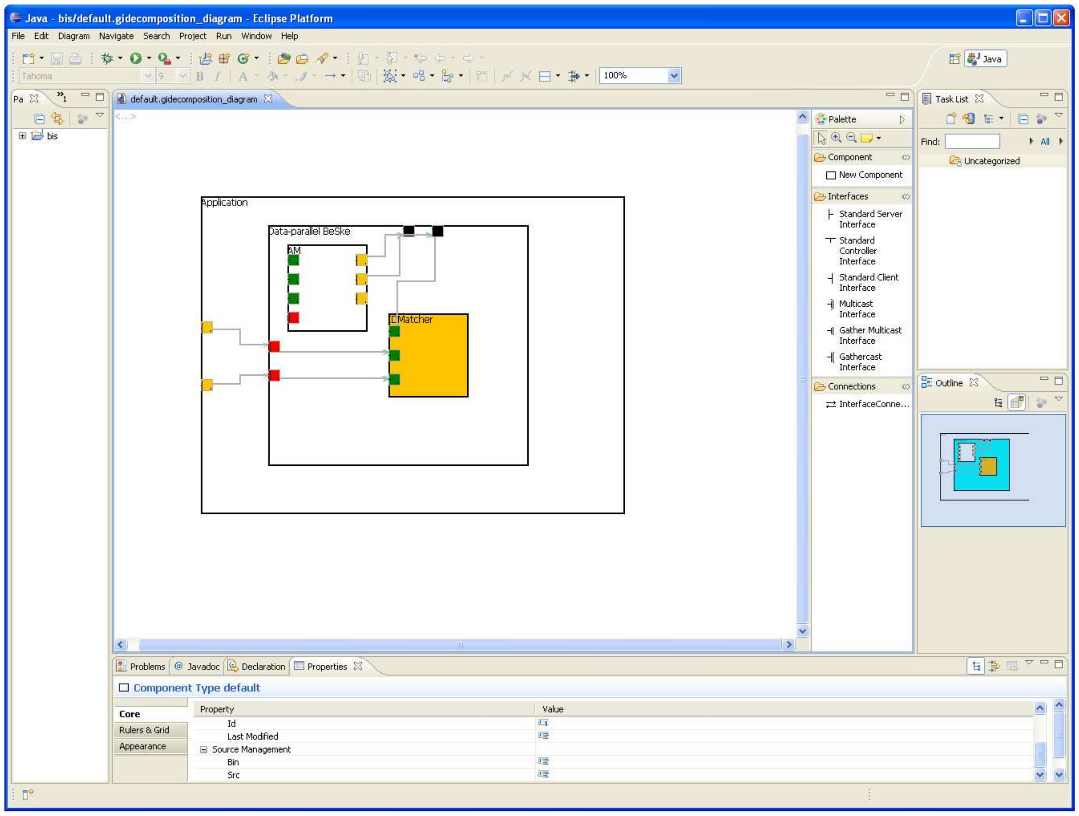


Figure 5: BIS component composition in GIDE

2.1.4.2 Components description

The availability of the data-parallel skeleton has significantly reduced the number of custom components to be developed compared to previous versions of the BIS use case. Only the primitive component *IDMatcher* and the *Application* composite have been designed from scratch. Compared to earlier use case versions, not making use of behavioural skeletons, the code size has been reduced by approximately 35%. All components used, including the skeleton components, are briefly introduced in this sub-section.

2.1.4.2.1 Application

The *Application* component is a composite component enclosing the complete component system of the BIS application. As such it hides internal components and only offers two server ports, one to initially distribute the DB via interface I1 (c.f. Figure 4), and one to broadcast identification requests via interface I2.

2.1.4.2.2 IDMatcher

The *IDMatcher* component is a primitive component, which includes the actual fingerprint matching functionality. When a matcher component is initialized it receives a number of tasks which together define a partition of the identity DB. The matcher then accesses the shared database and loads the partition into RAM for faster access. Initialization happens either via interface I3 while the DB is initially distributed or during a reconfiguration operation via interface I5. Afterwards, it receives identification requests via interface I4 and returns results via method return values. While processing an identification request, it matches the fingerprints of a given person against its part (partition) of the database.

2.1.4.2.3 Data-Parallel BeSke

The *Data-Parallel BeSke* component represents the autonomic data-parallel skeleton as provided by WP3 (c.f. D.NFCF.04) [3].

2.1.4.2.4 Autonomic Behaviour Controller (ABC)

To implement the behavioural skeleton the default component controller has been replaced by an autonomic behaviour controller (*ABC*). The *ABC* is part of the data-parallel skeleton. It offers autonomic operations such as increasing or decreasing the parallel degree which are triggered by the autonomic manager if required by the QoS contract.

2.1.4.2.5 Autonomic Manager (AM)

The *AM* component regularly enforces the QoS contract and triggers an autonomic operation via the *ABC* if required. The QoS contract consists of a JBoss Drools rule file as listed below. The BIS application allows defining the desired average target partition size as a performance contract, for example, say 1700 identities per *IDMatcher* component. Based on this value, defined by the BIS administrator, a rule file is generated which defines an upper bound and a lower bound for the partition size. The upper bound is defined as the target partition size plus 10% and the lower bound is defined as the target partition size minus 10%. If the upper bound is reached, the rule triggers the AM operation *ADD_EXECUTOR* to instruct the AM to trigger increasing the parallel degree via the ABC. Similarly, the operation *REMOVE_EXECUTOR* is triggered if the lower bound is reached as can be seen in the listing. Whenever the target partition size is modified by the BIS administrator, a new rule file is generated and submitted to the AM to update the QoS contract.

```
[methodMonitor="searchMatch"]
rule "CheckHigherBound"
        when
            $arrivalBean : PartitionSizeBean(value > 1870)
        then
            $arrivalBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
 end
[methodMonitor="searchMatch"]
rule "CheckLowerBound"
        when
            $arrivalBean : PartitionSizeBean(value < 1530)
        then
            $arrivalBean.fireOperation(ManagerOperation.REMOVE_EXECUTOR);
 end
```

2.1.4.3 Interfaces

This section describes the interfaces I1-I5, as denoted in Figure 4 in more detail.

2.1.4.3.1 Interface I1

The multicast interface I1 is used to transfer tasks, generated by the GCM adapter, from the A*pplication* component to the *Data-Parallel BeSke* and from there to the *IDMatcher* components. It includes the *getService()* method as shown in the listing below. The purpose of this interface is to initially distribute the identity DB across the *IDMatcher* components. It takes a list of tasks as a parameter whereas each task represents a part of the DB described by an index and the number of records. In the prototype each task represents up to 100 identities in the DB. When the list of tasks arrives at the multicast interface of the *Data-Parallel BeSke* component, the tasks are distributed via a custom dispatch mode as indicated in the method annotation. This dispatch mode distributes the tasks equally across the bound *IDMatcher* components.

```
public interface MulticastTestItf1  {

/** Initially distribute the identity DB.
 *
 * @param list List of tasks to be distributed across matcher components.
 * @return RFU.
 */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode =ParamDispatchMode.CUSTOM,
  customMode = MapDispatch.class))
  public List<Task> getService(List<Task> list);
}
```

2.1.4.3.2 *Interface I2*

The multicast interface I2 is used to broadcast identification requests to all bound *IDMatcher* components. Thus, the distribution mode *BROADCAST* is indicated in the method annotation of the *searchMatch()* method as shown in the listing below. An identification request is represented by a task object which holds the fingerprints. The resulting list of tasks represents the matching results from all *IDMatcher* components. Each result task includes the ID of the matching DB record, if any, and the name of the node which found the match.

```
public interface MulticastTestItf2 extends MulticastTestItf1  {

/** Broadcast an identification request.
 *
 * @param list List including one task which holds the fingerprints of the person to be
 *        identified.
 * @return List of tasks holding the matching results from all IDMatcher components.
 */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode =ParamDispatchMode.BROADCAST))
  public List<Task> searchMatch(List<Task> list);
}
```

2.1.4.3.3 *Interface I3*

Interface I3 allows *IDMatcher* components to be bound to the multicast interface I1. The matcher components receive their tasks via this interface and automatically load the corresponding partition of the DB into RAM upon arrival of the tasks. The task parameter can represent one or more tasks (the Task object is a generic container).

```
public interface IDDistribute {

/** Receive tasks representing the DB partition to match against.
 *
 * @param t One or more tasks representing a part of the DB.
 * @return RFU.
 */
  public Task getService(Task t);
}
```

2.1.4.3.4 *Interface I4*

Interface I4 allows *IDMatcher* components to be bound to the multicast interface I2. The matcher components receive an identification request via this interface. In response, each matcher component matches the given fingerprints against its DB partition and returns a task object including the ID of the matching DB record, if any, and the name of the node the component is running on.

```
public interface IDMatch extends IDDistribute {
```

```
/** Receive an identification request.
 *
 * @param list List including one task which holds the fingerprints of the person to be
 *        identified.
 * @return Tasks holding the matching result.
 */
  public Task searchMatch(List<Task> list);
}
```

2.1.4.3.5 *Interface I5*

Interface I5 is used by the ABC during reconfiguration operations. If the ABC modifies the parallel degree within the skeleton (by adding or removing an *IDMatcher* component), then it firstly retrieves all tasks from all current components via the method *provideStatus()*, secondly it adds or removes a component, and lastly it redistributes the tasks via the *setStatus()* method. This way the DB is redistributed during every reconfiguration operation.

```
public interface ReconfigSupport {
       public void setStatus(List<Task> tsl);
       public List<Task> provideStatus();
}
```

2.1.5 Demo Application

The previous version of the BIS demo application only provided a command line interface to interactively work with the application. This was the case because the command line interface is easier to adapt to the changing base functionality during development. For the final prototype, a graphical user interface (GUI) has been developed which allows configuring the application and interactively triggering identification operations. Furthermore, the GUI not only visualizes the identification process but also allows monitoring and manipulating the autonomic functionality of the data-parallel skeleton at runtime. More details about the configuration and usage including some screen shots can be found in the following section.

2.2 Manual

2.2.1 Final prototype description

The final prototype as described in the previous sections includes a number of improvements over the previous versions delivered in D.UC.03/04 ([1], [2]), mostly because it uses the autonomic data-parallel skeleton developed in WP3. Its overall functionality can be summarized as follows:

- It represents a biometric identification system which can work on a large user population in real-time.
- The system scales independently thanks to the autonomic reconfiguration functionality provided by the behavioural skeleton.
- The QoS contract which controls the autonomic functionality can be conveniently defined in a rule file. Furthermore, the QoS contract can be updated at runtime.
- The graphical user interface supports triggering identification requests and updating the QoS contract interactively. Furthermore, it visualizes the autonomic management functionality to monitor the state of the component system.

- The BIS application can be deployed on arbitrary hardware infrastructures without code changes.
- As the data-parallel skeleton supports the distribution and redistribution of the DB, the system can now, in contrast to the version described in D.UC.04 [2], be scaled to any number of nodes.
- The code size and thus the development time are reduced due to the availability of advanced features such as behavioural skeletons and the GIDE.

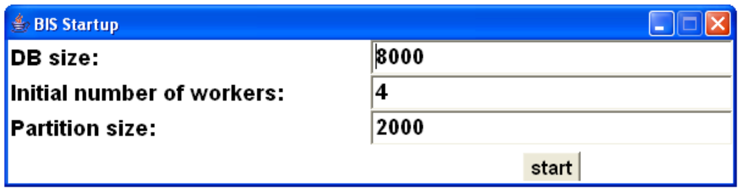
More details about the functionality provided by the GUI and its usage are provided in the next Section.

2.2.2 Configuration and usage

The final prototype is available in the file D.UC.05B-IBM.zip. The prototype is configured to run on Grid5000. It makes use of the deployment descriptor file descriptor/BIS-Grid.xml, which defines all nodes reserved in Grid5000. The skeleton uses these nodes for allocating *IDMatcher* components. All other components are running on the default node (the local JVM of the application). Since all nodes in the descriptor should ideally be available solely for IDMatcher components, the application should be started from a dedicated node not listed in the descriptor.

The application can be started via the included *run* script. The application takes command line arguments with the following syntax: *<target-partition-size> <db-size> <number-workers>*. *Target-partition-size* denotes the desired number of biometric matches per *IDMatcher* component and thus per node, *db-size* defines the desired database size, and *number-workers* defines the initial number of workers the skeleton should start with. The command line parameters can also be changed in the GUI later on as described below.

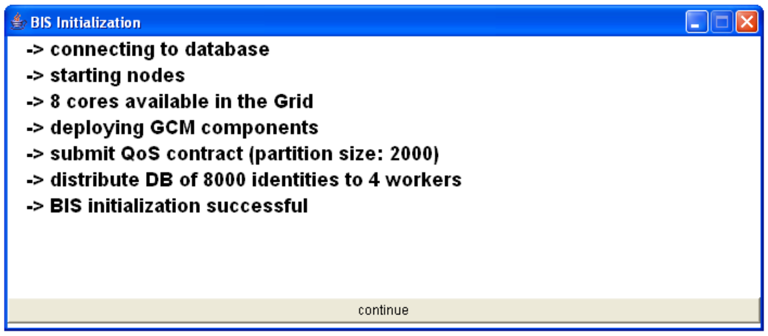
When the application is started, it displays the initial parameters defined at the command line in the startup dialog as shown in Figure 6.



DB size:	8000
Initial number of workers:	4
Partition size:	2000

Figure 6: BIS startup dialog

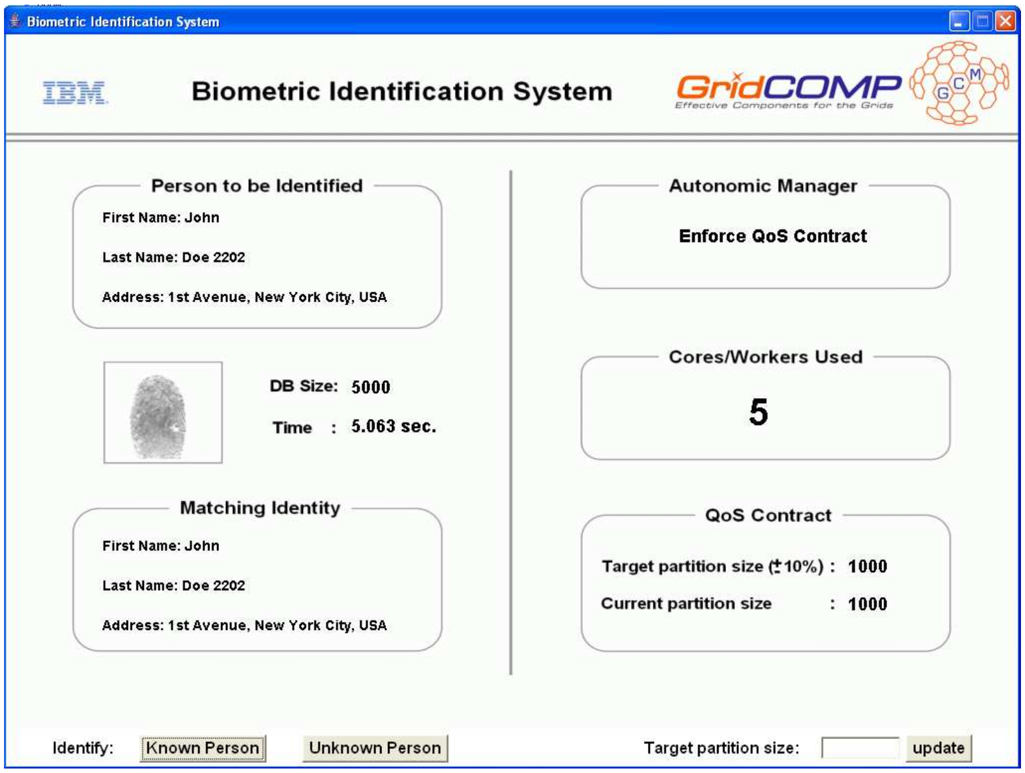
The user can further modify the parameters and finally press the start button to trigger the BIS application initialization phase.


BIS Initialization

```
-> connecting to database
-> starting nodes
-> 8 cores available in the Grid
-> deploying GCM components
-> submit QoS contract (partition size: 2000)
-> distribute DB of 8000 identities to 4 workers
-> BIS initialization successful
```

Figure 7: BIS initialization window

During the BIS initialization phase, the information window as shown Figure 7 is visible. It allows the user to monitor the initialization steps such as GCM component deployment or DB distribution carried out during the initialization phase. Once all steps have been completed successfully, as can be seen in Figure 7, the *continue* button becomes active and the user arrives in the main GUI upon pressing it.


Biometric Identification System

Biometric Identification System

Person to be Identified
First Name: John
Last Name: Doe 2202
Address: 1st Avenue, New York City, USA

DB Size: 5000
Time : 5.063 sec.

Matching Identity
First Name: John
Last Name: Doe 2202
Address: 1st Avenue, New York City, USA

Autonomic Manager
Enforce QoS Contract

Cores/Workers Used
5

QoS Contract
Target partition size (±10%) : 1000
Current partition size : 1000

Figure 8: BIS main GUI

Once successfully started, the BIS application can be used interactively via the main GUI as shown in Figure 8. The GUI is split into two main parts, the biometric identification part on the left side, and the autonomic management part on the right side.

The biometric identification part allows initiating identification of either an unknown person or a known person. In the latter case, a person to be identified is randomly chosen from the database of known identities and her details are shown in the upper left area of the GUI. Then, the distributed identification is initiated, and the fingerprint image starts changing to visually indicate progress as long as the nodes are searching for a matching identity. Once a match is found, the identification time is shown beside the fingerprint image and the details of the matching identity are printed below. In the case, where an unknown person is to be identified, fingerprints, not in the identity DB and thus an unknown identity, are submitted to the distributed identification process.

The autonomic management part of the GUI serves two main purposes. Firstly, it visualizes the autonomic functionality such that it can be monitored in real time. Secondly, it allows updating the QoS contract at runtime to trigger reconfiguration operations during the demo. In the top right area of the GUI the activity of the autonomic manager is indicated. Here, it can be seen that the autonomic manager leaves idle state every two seconds to enforce the QoS contract. Below, the current number of CPU cores used, which corresponds to the number of *IDMatcher* components, can be monitored. The lower right area of the GUI shows the current QoS contract represented by the target partition size. The input field below allows updating the target partition size while the component system is running. When the target partition size is updated, the user can monitor the resulting reconfigurations operations. For example, the autonomic manager indicates that the parallel degree must be increased, the number of cores changes, and the current partition size is updated as soon as the DB has been redistributed.

3 Computing of DSO Value

"Computing of DSO Value" is the use case selected by Atos for the development of this project. The DSO (Days Sales Outstanding) is an application used by Atos Origin to calculate the mean time that their clients delay to pay an invoice to Atos.

The information is needed by several internal departments, so it must be calculated from several points of view (by client, by group, by month, etc.).

The amount of data needed to compute the payments information is over 6.000 clients, and the process lasts about 4 hours to complete. The objective is to achieve the utilization of spare resources via Grid computing to reduce the computing time.

3.1 Detailed architectural design

The previous deliverable documents from WP5 show how the architectural design of the use case called "Computing of DSO value" has changed over the project. This section includes the final architectural design of the application.

3.1.1 Architecture of the application

The DSO application is based on a client / server infrastructure with heavy processes implemented in PL/SQL procedures to make all the calculations needed to get the final results.

The architecture of the application is divided into two different parts, or it could be also said that there are two different types of nodes: The master node and worker nodes.

- The *master node* is the one which controls the running of the application. It divides the process into different tasks, and after that it, sends them to the different worker nodes. Here is where the application starts the workflow, and the main database is installed. The master database normally is the same database used by the initial application, the one that we want to grid-enable. With that, the company doesn't need to change or recreate their existing application database infrastructure. Usually the database management system used in the companies to execute their PL-SQL code is Oracle Enterprise Edition or Oracle Standard Edition.

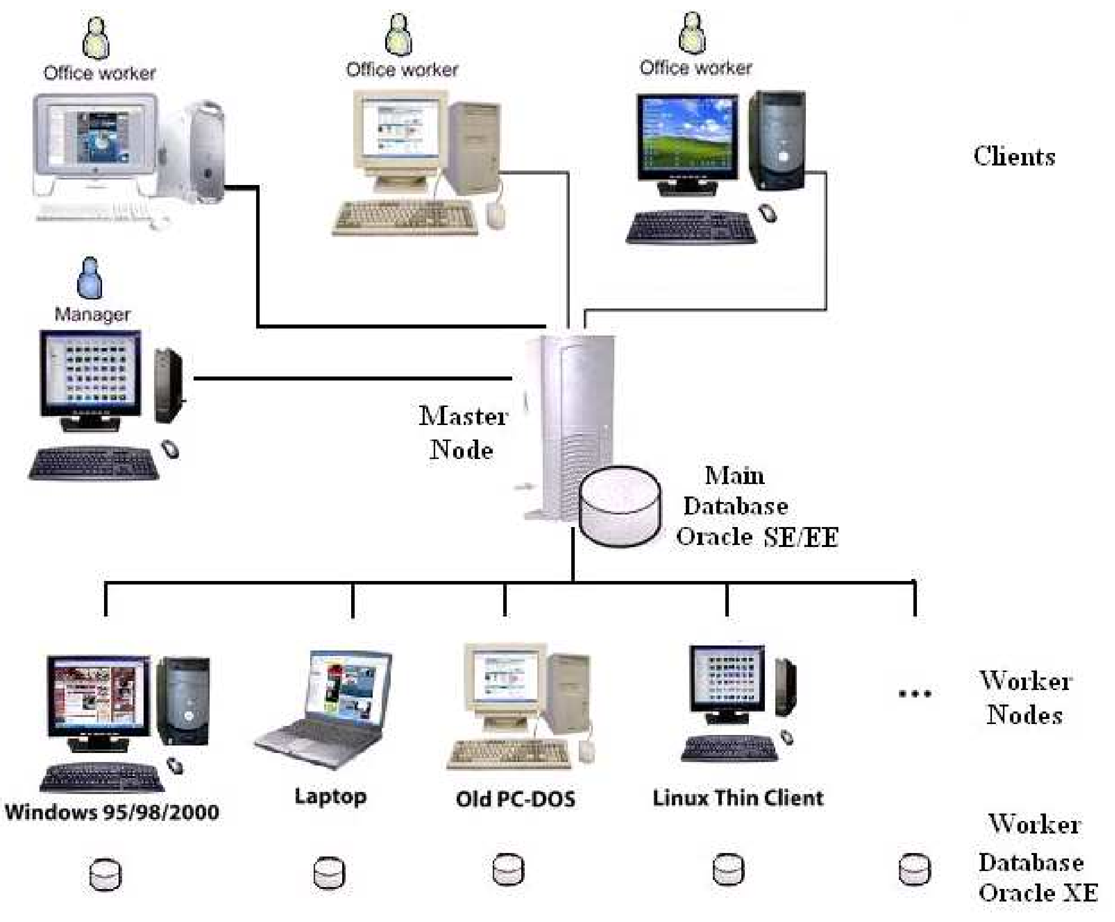
- The *worker nodes* receive the tasks to process them. When this work is done, the results of the operation are sent back to the master node. So, worker nodes are the ones doing the application calculations. Then, the results of those calculations are sent back to the master node. It is important to know that any computer (server, desktops, and laptops) could be used as a worker node, as long as it has Oracle management system installed. If the worker node doesn't have Oracle installed, we recommend installing Oracle Express Edition because it is free of charge.

An important requirement is to have Java runtime environment 1.6 installed in all the nodes, as Java is the language used to develop the application. It is essential to take into account the operating system of each node. It is necessary to have a ssh server installed, so if the node works with Linux operating system, there is no problem because Linux has a ssh server incorporated. However, if the node uses Windows, it is necessary to install Cygwin [11](creating a Linux-like environment for Windows) and SSH server for Cygwin [11].

In summary, the infrastructure requirements needed to run the application are:

- A Master Node with:
  - Main database : Oracle Enterprise Edition or Oracle Standard Edition
  - Java runtime environment 1.6 [5]
- Several Worker Nodes, each with.
  - Node database: Oracle Express Edition
  - Java runtime environment 1.6 [5]
  - If Windows operating system is used: Cygwin [11] and SSH server for Cygwin [11].

The next image shows the application architecture implementation:

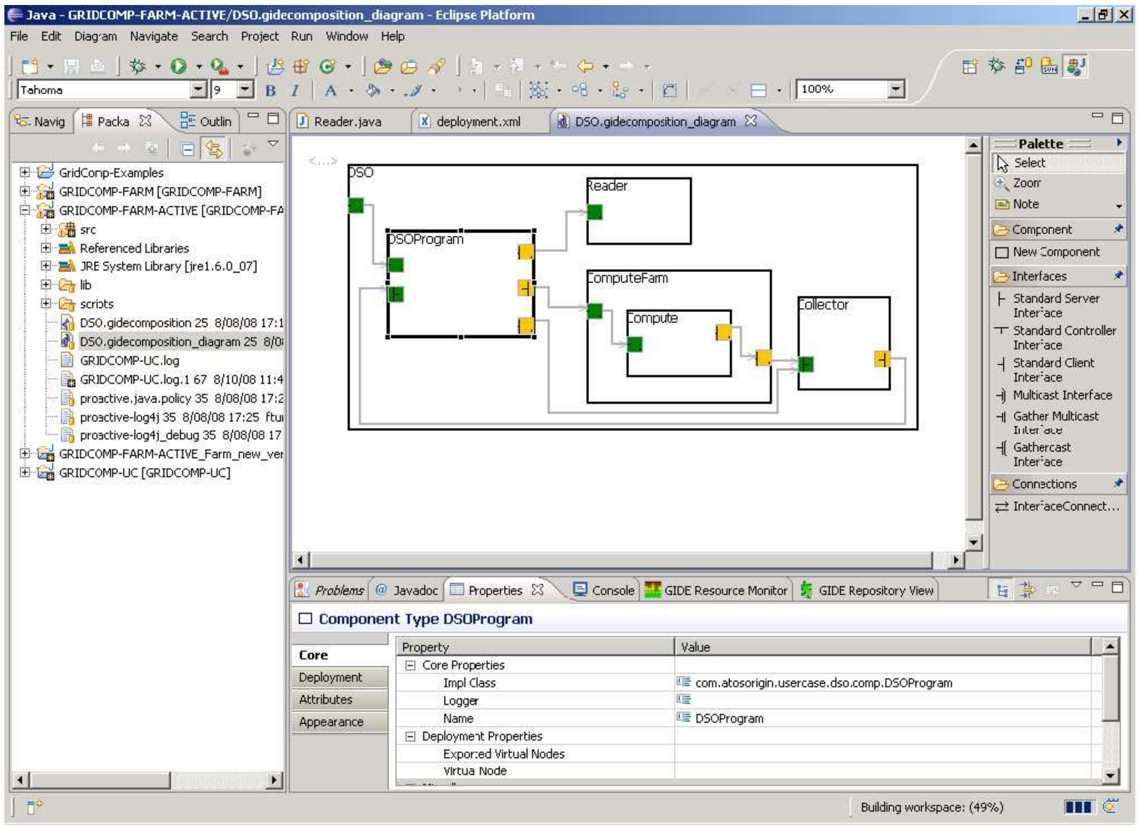


3.1.2 GCM Components

The following sections explain the GCM components architecture used in this use case.

3.1.2.1 Components diagram

The following screenshot from the GIDE [14] illustrates the components diagram used in the final version of the Computing of DSO Value use case:


Java - GRIDCOMP-FARM-ACTIVE/DSO.gidecomposition_diagram - Eclipse Platform
File Edit Diagram Navigate Search Project Run Window Help
Navig
Packa
Outlin
GridComp-Examples
GRIDCOMP-FARM [GRIDCOMP-FARM]
GRIDCOMP-FARM-ACTIVE [GRIDCOMP-FA
src
Referenced Libraries
JRE System Library [jre1.6.0_07]
lib
scripts
DSO.gidecomposition 25 8/08/08 17:1
DSO.gidecomposition_diagram 25 8/0
GRIDCOMP-UC.log
GRIDCOMP-UC.log.1 67 8/10/08 11:4
proactive.java.policy 35 8/08/08 17:2
proactive-log4j 35 8/08/08 17:25 ftu
proactive-log4j_debug 35 8/08/08 17
GRIDCOMP-FARM-ACTIVE_Farm_new_ver
GRIDCOMP-UC [GRIDCOMP-UC]
Reader.java
deployment.xml
DSO.gidecomposition_diagram
DSO
Reader
DSOProgram
ComputeFarm
Compute
Collector
Palette
Select
Zoom
Note
Component
New Component
Interfaces
Standard Server Interface
Standard Controller Interface
Standard Client Interface
Multicast Interface
Gather Multicast Interface
Gathercast Interface
Connections
InterfaceConnect...
Problems
Javadoc
Properties
Console
GIDE Resource Monitor
GIDE Repository View
Core
Deployment
Attributes
Appearance
Building workspace: (49%)

Component Type DSOProgram

Property	Value
Core Properties	
Impl Class	com.atosorigin.usercase.dso.comp.DSOProgram
Logger	
Name	DSOProgram
Deployment Properties	
Exported Virtual Nodes	
Virtual Node	

This components diagram is different from the last one of the previous document version (D.UC.04 [2]) because the final UC version is using the Farm component developed by WP3.

In the lines below there is an explanation of the application workflow, which describes step by step the UC diagram:

- The client user interface makes a request to the DSOProgram component to start the process.

- The DSOProgram component obtains the list of clients' IDs to be processed from the Reader component.

- The DSOProgram component breaks the list of clients' IDs into chunks (tasks). The number of tasks that the application generates is specified in the file called DSOProgram.fractal.

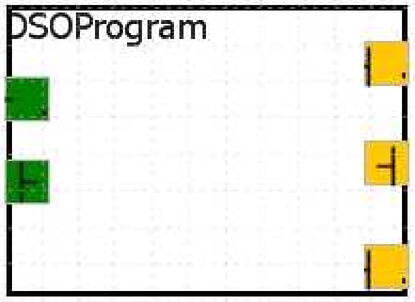
- These tasks are sent to the ComputeFarm component, which distribute the tasks between the remote nodes to be processed. At the same time the number of tasks is sent to the Collector component. This way it knows the total number of tasks that will be processed by the nodes.

- The Compute component receives the tasks from the ComputeFarm and inserts the information on the slave database. After that, the Compute component calls the stored procedure situated inside the slave database to execute the PL/SQL code. When the Compute component finishes each task, it sends a notification to the Collector component informing that the task is done.

- When the Collector receives all notifications from the Compute components, it sends a notification to the DSOProgram component telling that the execution of the application is done.

3.1.2.2 Components description

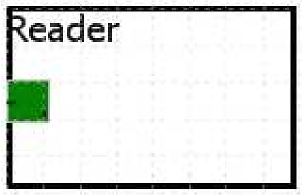
3.1.2.2.1 DSOProgram Component



The DSOProgram is the master component of the application, and it is responsible of the program workflow. It offers some server and client interfaces:

- *RunDSO*: It is a server interface used to start the execution of the application using the parameters needed.
- *Result*: It is a server interface used to receive the notification from the Collector when the application ends the execution.
- *Reader*: It is a client interface used to get the information of the clients from the Reader component.
- *OurTask*: It is a client interface used to start the process in the remote nodes.
- *Collector*: It is a client interface used to initiate the collector component with the number of tasks that will be sent to the workers.

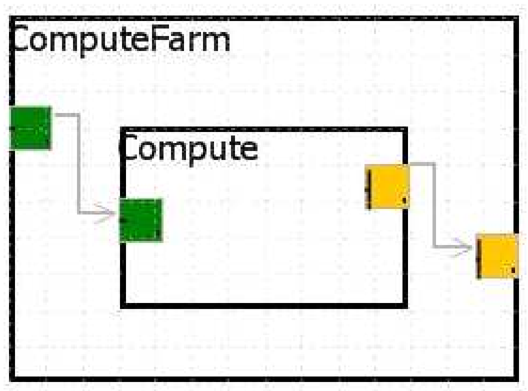
3.1.2.2.2 Reader Component



The Reader component offers the functionality to connect to the master database and gets the list of clients' IDs that will be processed by the application. It offers a server interface:

- *Reader:* It is a server interface to get the information of the clients from the database.

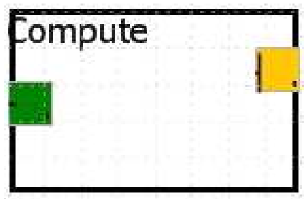
3.1.2.2.3 ComputeFarm Component



The ComputeFarm component is responsible for distributing and controlling the worker nodes. Using Farm component we can manage the workers of the application, adding or removing them when it is necessary. The interfaces used in ComputeFarm component are explained in the lines below:

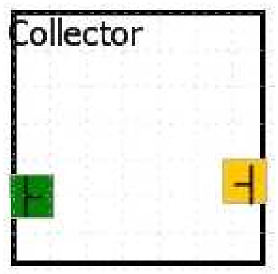
- *OurTask:* It is a server interface that starts the process in the node.
- *Collector:* It is a client interface to notify the Collector component that the task is finished

3.1.2.2.4 Compute Component



The Compute component offers the functionality to execute the tasks received by the ComputeFarm and execute them. The component receives the task with the list of clients to be inserted in the node database. After that, the component calls an Oracle stored procedure stored in the node database to execute the PL/SQL code. The Compute component offers the same server and client interfaces as the ComputeFarm component: *OurTask* and *Collector*.

3.1.2.2.5 Collector Component



When a worker node ends a task it sends a notification to the Collector component saying that the task is finished. This way, this component can know how many tasks are still pending to process, how many tasks are already finished and the moment when all tasks are finished. The interfaces used to develop this component are explained in the lines below:

- *Collector:* It is a server interface used to receive the notifications from the worker nodes.
- *Result:* It is a client interface used to notify the DSOProgram that the process has finished.

3.1.2.3 Interfaces

As is mentioned above, there are some interfaces to build the components. Those interfaces are explained in the following lines:

3.1.2.3.1 RunDSO

When the user starts the application execution, the first interface called is *RunDSO*. This interface takes the parameters imputed by the user (client id, group id or initial and final dates) and begins the execution.

```
public interface RunDSO {

        public final static String  ITF_NAME        = "runnable";

        /**
         * Starts the execution of the application. It gets some of the
         * parameters needed for the application
         *
         * @param startDate   The initial date of the period to be processed.
         * @param endDate     The final date of the period to be processed.
         * @param clientID    Id from the specific client to be processed
         * @param groupId     Id from the group of clients to be processed
         *
         * @return void
         */
        public void run(String startDate, String endDate, String clientId,
                        String groupId);
}
```

3.1.2.3.2 Reader

As it could be guessed, the Reader interface is the one who connects to the master database and gets the list of the clients from the master database.

```
public interface Reader {

        public final static String  ITF_NAME        = "read";

        /**
         * Gets the list of the client's Ids from the master database.
         *
         * @param clientId     Id from the specific client to be processed
         * @param groupId      Id from the group of clients to be processed
         *
         * @return list of client's Ids
         */
        String[] getClients(String clientId, String groupId);
}
```

3.1.2.3.3 *OurTask*

The *OurTask* interface is responsible for starting the process in the remote nodes. The information needed for this operation is the list of the clients and the list of dates. The list of dates is composed by the initial and final dates that the user enters at the beginning of the execution. Those parameters are used to specify the interval of dates, in short, to specify the time spot we need to evaluate by the PL/SQL code.

```
public interface OurTask {

        public final static String  ITF_NAME        = "work";

        /**
         * A server interface to receive the tasks on the worker nodes
         *
         * @param clients     The list of client's ids
         * @param dates       The period to be processed
         *
         * @return void
         */
        public void compute(List<String> clients, List<String> dates);
}
```

3.1.2.3.4 *Collector*

The *Collector* interface is an important new one included in the final version of this use case. It is used to control when the tasks sent to the different worker nodes are finished. This interface implements four different methods. The first one is called *init* and it is used to initiate the Collector with the total number of tasks. The second one, called *collect*, is used by the worker nodes to notify the Collector when a task is finished. There are also other two methods to control the number of tasks which are finished and the number of task which are not finished. Those methods are called *getResultFilesCount* and *getResultFilesToCollect*.

```
public interface Collector {

        public final static String  ITF_NAME        = "collect";
        /**
         * This method is to initiate the Collector storing the total number of tasks.
         *
         * @param numOfResultsToCollect The total number of tasks.
         *
         * @return void
         */
        void init(int numOfResultsToCollect);


        /**
         * When a worker ends its task this is the method which is executed
         * @return void
         */
         void collect();


        /**
         * This method indicates the number of tasks finished.
         *
         * @return IntWrapper
         */
        IntWrapper getResultFilesCount();

        /**
         * This method indicates the number of tasks unfinished.
         *
         * @return IntWrapper
         */
        IntWrapper getResultFilesToCollect();
}
```

3.1.2.3.5 Result

The last interface is called *Result* and it is used to send the notification of the end of the execution to the DSOProgram.

```
public interface Result {

      public final static String   ITF_NAME   = "result";

       /**
        * This method is used to send the results of the execution to the DSOProgram.
        *
        * @return void
        */
      public void result();
}
```

3.2 Manual

This section describes how to configure and execute the use case application.

3.2.1 Final prototype description

The main differences between the final prototype implementation and the primitive one are:

- The application progress information is displayed through the user interface, as a progress bar.
- The number of workers used and the time that the application will finish depending on the number of workers are graphically displayed.
- The task farm behavioral skeleton was integrated with the DSO program, providing a way to add or remove workers at execution time.
- A time controller was implemented to manage automatically the time that the application should finish.
- The computation result is shown at the graphical user interface.

During the development of this project, we analyzed different ways to distribute the PL/SQL code between the master node and worker nodes with a Grid solution. In the last document, four different distributions were explained. Now, after the performance tests, we came to the conclusion that only two of them were really suitable to get good, fast and satisfactory results. In the following lines, we give a short explanation of those two possibilities.

The first possibility is to store a part of the PL/SQL code in the databases installed in the worker nodes, and the other part of the PL/SQL code and all the tables in the master database located in the master node. This method is used when the information is distributed between many tables and it is necessary to use all of them when executing the PL/SQL code.

The second possibility is to store all the code and empty main tables in the worker nodes databases, and the main tables in the master database. The information stored in the worker database only belongs to the specific clients whom are going to be processed in each node. Then, this method is suitable when the database is big. This option can be used when the PL/SQL code does a lot of calculations with specific data access. This database structure was selected to be used with the Computing of DSO Values use case because the application PL/SQL code use specific information stored in specific tables to do the calculations. When we did the performance tests, we could see that in this case the process time decreased when this kind of distribution was used. The PL/SQL code will use the information stored inside the node tables to do the calculation, and if it needs more data, it will take it from the master database.

It is possible to find more information about the distribution of the databases in the previous document D.UC.04 [2].

The final version of the application is using the active Farm skeleton. This component is used to manage the worker nodes by adding or removing them when necessary. Furthermore, it provides a manager module that automatically adds or removes worker nodes when necessary. To use this module some parameters must be set to complete the criterion needed to determinate when the nodes should be added or removed. Those specifications are written in the `rules.drl` file. In this use case the manager module is used to start the initial worker nodes of the application.

3.2.2 Configuration and usage

The first thing to do before executing this prototype is to install the required software listed in section 3.1.1. After installing, testing and running all required software, you can start configuring the prototype. To configure the application is essential to add the following libraries to the lib directory:

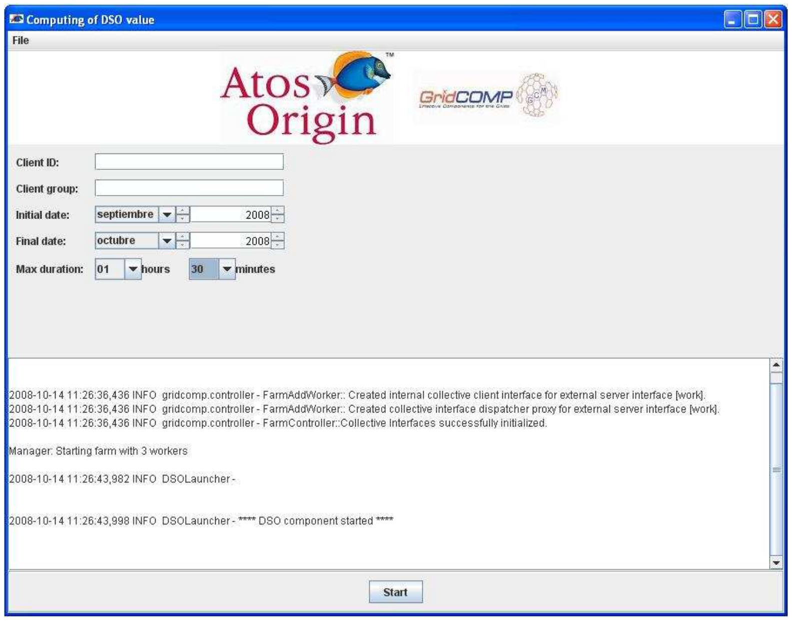
- ProActive 3.9 (ProActive binaries and related libraries)
- classes12.jar (JDBC library)
- Task farm behavioural skeleton binaries and related libraries

The following files need to be changed to configure and run the prototype on your environment:

- *\classes\com\atosorigin\usercase\dso\deployment.xml* - open the deployment file and rewrite it with the nodes information.
- *\classes\com\atosorigin\usercase\dso\adl\DSOProgram*.fractal - open the fractal file and change the attribute numTasks value, specifying the number of tasks that are going to be executed in the worker nodes. To specify this number we have to take into account that we need enough tasks to distribute the process between the nodes, but this number should not be too large to avoid having too many accesses in the worker nodes databases.
- *\classes\com\atosorigin\usercase\dso\adl\ReaderImp.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the master database information
- *\classes\com\atosorigin\usercase\dso\adl\Compute.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the master node database information. It is a worker node located inside the master node, specifically the first worker node the application initializes.
- *\classes\com\atosorigin\usercase\dso\adl\ComputeSlaves.fractal* - open the fractal file and change the attributes *url*, *user* and *pwd* with the worker database information.
- *\classes\com\atosorigin\usercase\dso\adl\rules.drl* - open the FARM rules file to set the number of workers that the application needs to start when the application begins the execution.

After changing all the files listed above, you can start the application running the command:
…\GRIDCOMP-FARM-ACTIVE\scripts\unix\dso.sh

When the main application called "DSOProgram" begins running the graphical user interface appears.



Client ID:	
Client group:	
Initial date:	septiembre 2008
Final date:	octubre 2008
Max duration:	01 hours 30 minutes

```
2008-10-14 11:26:36,436 INFO  gridcomp.controller - FarmAddWorker:: Created internal collective client interface for external server interface [work].
2008-10-14 11:26:36,436 INFO  gridcomp.controller - FarmAddWorker:: Created collective interface dispatcher proxy for external server interface [work].
2008-10-14 11:26:36,436 INFO  gridcomp.controller - FarmController::Collective Interfaces successfully initialized.

Manager: Starting farm with 3 workers

2008-10-14 11:26:43,982 INFO  DSOLauncher -


2008-10-14 11:26:43,998 INFO  DSOLauncher - **** DSO component started ****
```

At this time, the farm manager starts running and initializes the number of nodes specified in the rules file. As explained before, this file specifies the rules that the farm manager will use to manage the application, adding or removing workers automatically depending on the rules that you define. The final prototype uses the manager only to start the initial workers that the application will use. The new feature called time controller is being used to manage the number of workers needed (adding and removing them). This new feature was implemented because the farm manager doesn't work based on time, only with the task throughput.

The application log is located in the box on the bottom of the first window. There, it is also possible to see what is happening with the worker nodes.

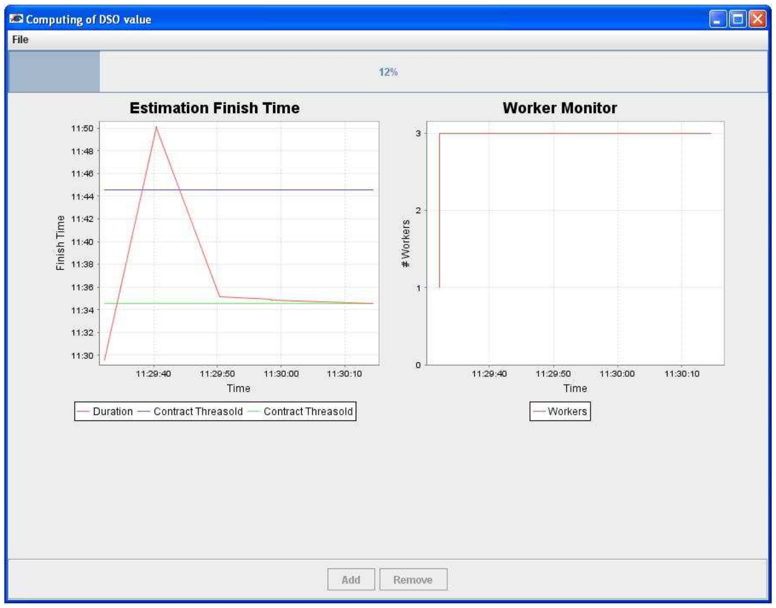
When the initial worker nodes are initiated, the *Start* button will be enabled. At this moment it is possible for the user to enter some parameters in order to filter the output information. These parameters are: *Client ID, Group ID*, *initial* and *final dates,* or the *maximum duration* of the process. Only two of those parameters are obligatory: the *initial* and *final dates.*

The *maximum duration* parameter is used to start the time controller feature developed by ATOS. This feature is used in the final version of the use case to give to the user the possibility to set the time that the application must finish. This controller will add or remove workers automatically depending on the time specified in the *maximum duration* parameters.

When the *Start* button is pushed, all the tasks are distributed between the different workers, and the window with the graphics appears (two possibilities).

1- The user sets the *maximum duration* parameters

If the user has set the *maximum duration* parameters, the time controller starts running. The window the user is going see is the one showed in the picture bellow:



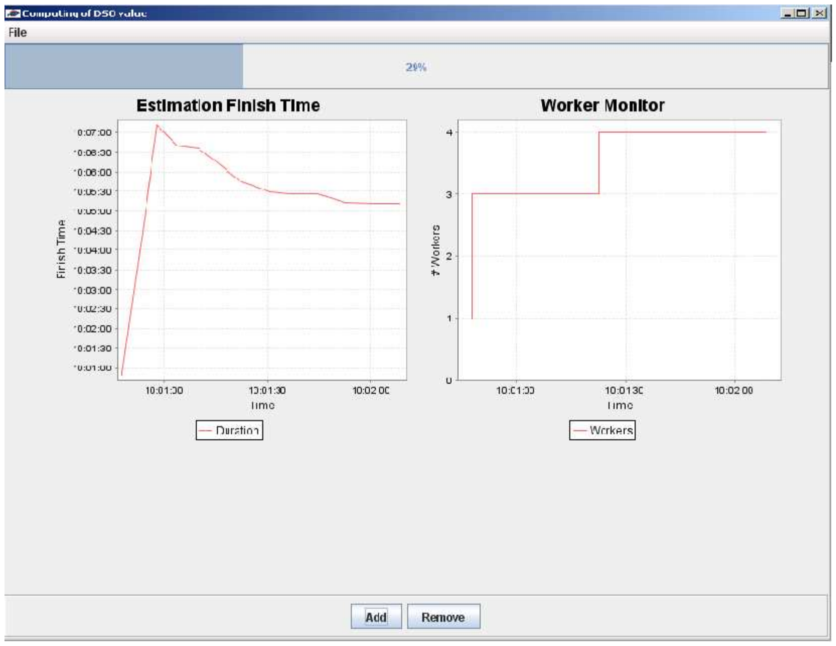
In this window you can see some graphics and a progress bar that shows the execution state belong the time. The progress bar on the top is used to show the percentage of tasks that was already executed.

The graphic on the left shows the time that the application is going to take to finish (based on the task executions throughput), and it is updated every 5 seconds. If the time controller is running, the graphic will show two lines: the blue line represents the maximum time that the application should finish and the green line represents the minimum time. If the red line is under the green line, the time controller is going to remove a worker to increase the execution time, making the red line return inside the two lines. However, if it is over the blue line, the time controller is going to add a worker to decrease the execution time. This is the way this tool works to meet the time specified by the user.

The graphic on the right shows the number of workers the application is using during the execution time. In this case, it shows that the application starts the process with three workers. At the beginning, there is a transition period when the average throughput is calculated and this makes an initial peak showed in the graphic on the left. The time controller doesn't take into account this initial event.

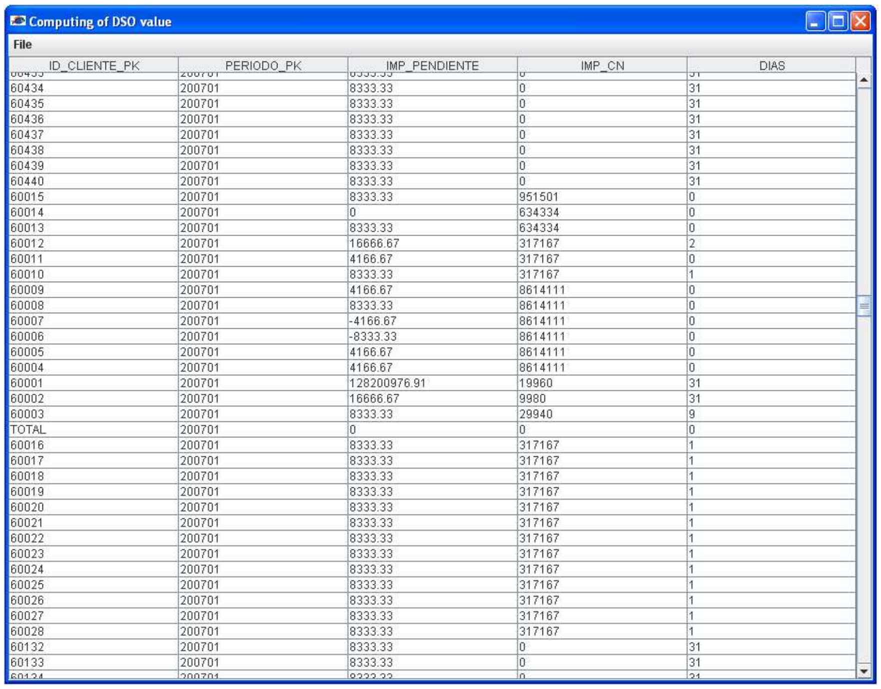
2- The user doesn't set the *maximum duration* parameters

If the user has not set the *maximum duration* parameter, the time controller is not started. The window the user is going see is the one showed in the picture bellow:



Using this way the *Add* and *Remove* buttons will be enabled to execute the corresponding operations manually when the user considers properly. If the user press the *add* button, the application will add a new worker, and the execution will be faster. If the user press *remove* button the result will be the opposite, the application will remove a worker and the execution will be slower.

When the application finishes the execution of all tasks, a new window will appear showing the calculation results. The following picture shows the final window:



ID_CLIENTE_PK	PERIODO_PK	IMP_PENDIENTE	IMP_CN	DIAS
60434	200701	8333.33	0	31
60435	200701	8333.33	0	31
60436	200701	8333.33	0	31
60437	200701	8333.33	0	31
60438	200701	8333.33	0	31
60439	200701	8333.33	0	31
60440	200701	8333.33	0	31
60015	200701	8333.33	951501	0
60014	200701	0	634334	0
60013	200701	8333.33	634334	0
60012	200701	16666.67	317167	2
60011	200701	4166.67	317167	0
60010	200701	8333.33	317167	1
60009	200701	4166.67	8614111	0
60008	200701	8333.33	8614111	0
60007	200701	-4166.67	8614111	0
60006	200701	-8333.33	8614111	0
60005	200701	4166.67	8614111	0
60004	200701	4166.67	8614111	0
60001	200701	128200976.91	19960	31
60002	200701	16666.67	9980	31
60003	200701	8333.33	29940	9
TOTAL	200701	0	0	0
60016	200701	8333.33	317167	1
60017	200701	8333.33	317167	1
60018	200701	8333.33	317167	1
60019	200701	8333.33	317167	1
60020	200701	8333.33	317167	1
60021	200701	8333.33	317167	1
60022	200701	8333.33	317167	1
60023	200701	8333.33	317167	1
60024	200701	8333.33	317167	1
60025	200701	8333.33	317167	1
60026	200701	8333.33	317167	1
60027	200701	8333.33	317167	1
60028	200701	8333.33	317167	1
60132	200701	8333.33	0	31
60133	200701	8333.33	0	31

3.2.3 Examples

To test the application you can use the following parameters:

- Client ID: <leave empty>
- Client group: <leave empty>
- Initial date: September 2008
- Final date: October 2008
- Maximum Time of process: <don't change>

After that, insert the parameters you need and press the *start* button to start the application. The second window will appear and them you can see the execution process. When all tasks are done, the result window will appear with the calculation results.

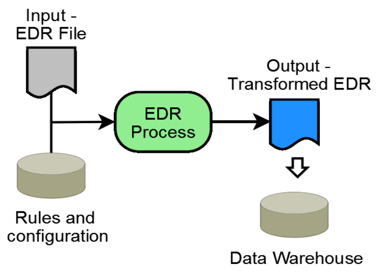
4 EDR Processor

This use case, taken from the Telecommunications world, is an embarrassingly parallel[1] application that deals with huge amounts of data. This special feature will serve to assess and test whether the GCM is prepared to deal with real world applications.
Record processing is a common computing problem that enterprises have to deal with, especially in the Telco world. Basically, the objective is to transform a big set of data records into another, which is in the end an essential part for the business. This transformation usually takes a lot of time to be performed and requires a considerable amount of computing resources.
Because real-world EDR processing is time constrained and critical to the company's objectives, the objective in this use case is to provide a high performance computing solution, based on a Grid Component version of the EDR Processor application, thus improving the quality of the processing by offering redundancy, fault-tolerance, scalability, load balancing, and reduced computing times.

4.1 Detailed architectural design

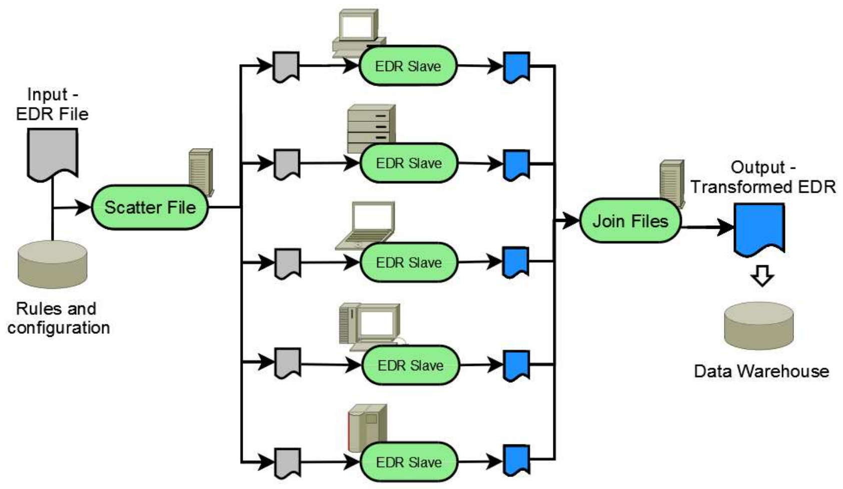
4.1.1 Architecture of the application

An actual EDR Processor application will work unattended, inside a nightly batch process, taking information from a sequential file (previously generated from some source database) and storing the results into another sequential file (eventually imported to a target database).
For the purposes of this project, the source and target databases will be ignored.

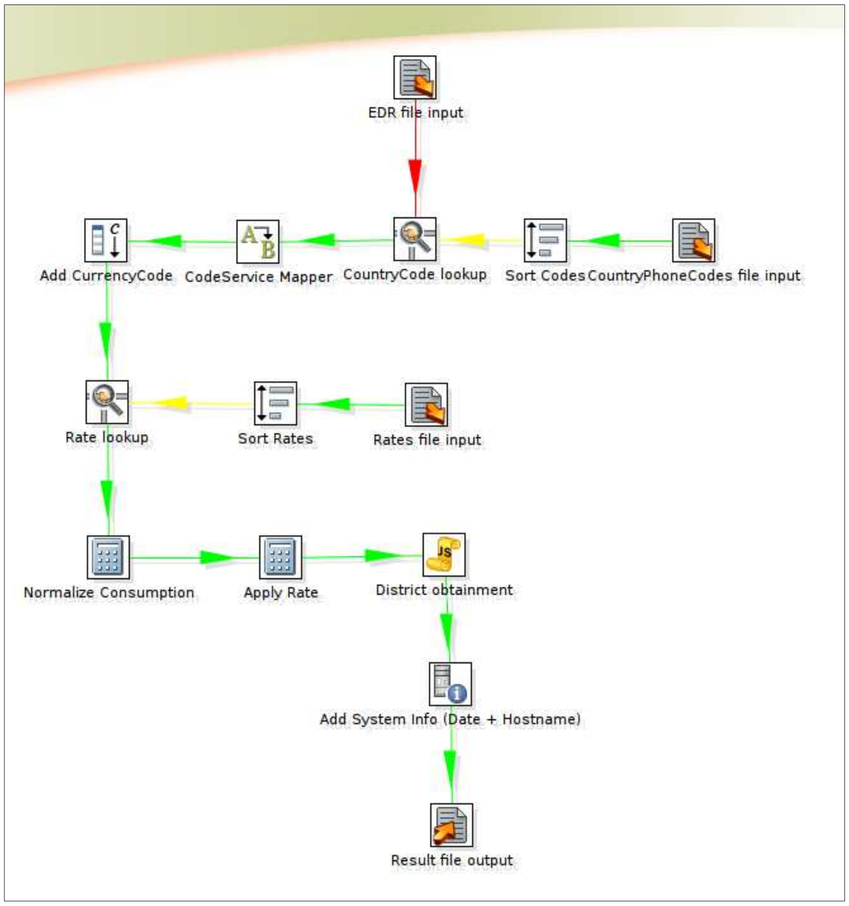


Being an embarrassingly parallel process, the EDR processing can be easily distributed among a set of (likely heterogeneous) computing resources. In order to do that, the input EDR file must be split into fragments. Each fragment will be processed by a grid resource, and the results will later be joined.
The scattering and joining of the files is performed by a "master" resource (the one running the application). The processing of the fragments is done by the "EDR slaves", which transfer the result files back to the "master". As we will see later, this fits perfectly one of the behavioural skeletons from WP 3 (the task farm).
The following picture shows the conceptual behaviour of the application:

[1] http://en.wikipedia.org/wiki/Embarrassingly_parallel



As explained in previous deliverable documents and presentations ([1], [2]), the processing of an EDR is a rather simple case of an Extract, Transform and Load (ETL) process.
The following picture shows the design of the ETL corresponding to the processing of an EDR file:



The Extract Transform and Load processing is done using Pentaho Data Integration, also known as Kettle Project [4]. Kettle is an open source ETL library that includes a very user-friendly integrated development environment. Using that IDE the user can easily design the

ETL process and save it to a metafile. That metafile can later be used to execute the ETL process through the Java API of the Kettle libraries.
Each one of the steps of the transformation is described in depth in D.UC.04.A [2].

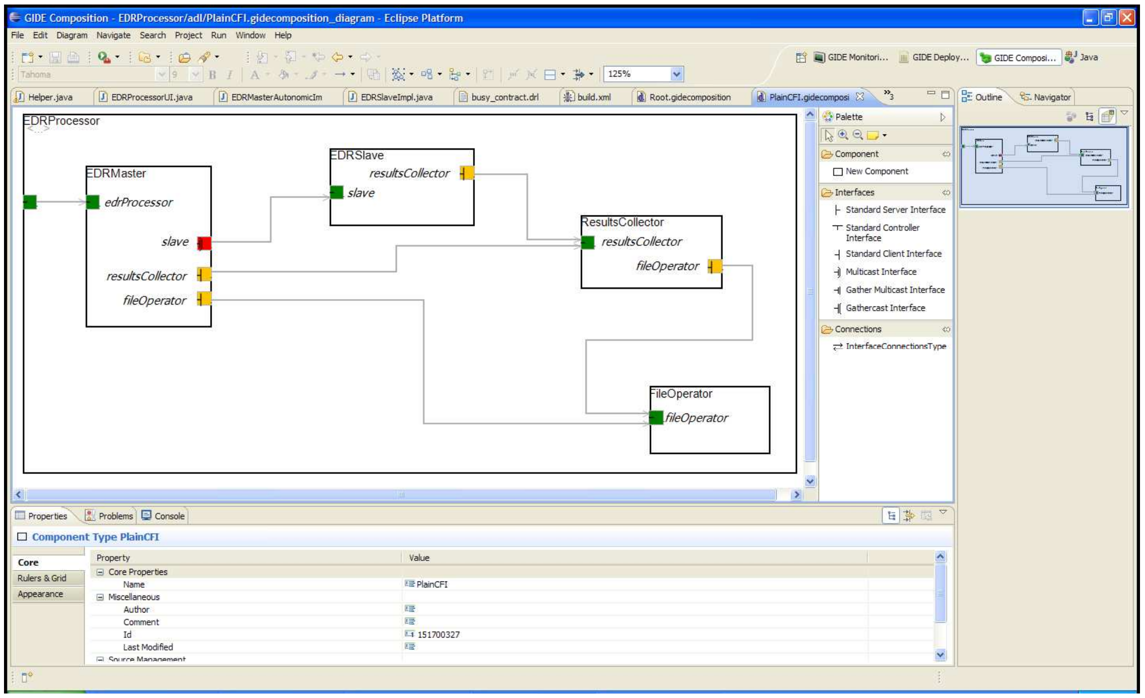
4.1.2 GCM Components

Both this and the Wing Design use-cases have two distinct architectural designs: one using autonomic features, and the other not. The latter was developed first, as autonomic features were made available later during the project. The non-autonomic design is based on collective interfaces that take care of the distribution of the computations. The autonomic design, based on Behavioural Skeletons, adds quality of service (QoS) features along with the distribution of the computations, offering a higher level of integration with the application. Although not based on the same features, these two versions can be seen as a "before and after", showing the evolution of features within the project.

4.1.2.1 Components diagram

4.1.2.1.1 Plain CFI version

This version of the architectural design uses only features from the Component Framework Infrastructure (developed in Work Package 2). In this case, a multicast interface is used in order to distribute the processing effort among the nodes of the grid.



Component Type PlainCFI

Property	Value
Core Properties	
Name	PlainCFI
Miscellaneous	
Author	
Comment	
Id	151700327
Last Modified	

Summarizing, the architectural design is as follows:

- The EDRProcessor, the composite component containing everything else, receives the request to process an EDR file, and redirects it to the enclosed EDRMaster component.
- Using the FileOperator, the EDR file is split into fragments.
- Using a **multicast interface**, the fragments are processed by the EDRSlave components

- The partial results are sent to the ResultsCollector.
- When all fragments have been processed, the ResultsCollector, using the FileOperator, merges the partial results, obtaining the final result.

The above composition diagram, made using the Grid IDE [14], includes a single EDRSlave. The ADL cannot express the dynamic deployment of a given component; it is a known limitation, which can be circumvented using Behavioural Skeletons (as in the other version of the architectural design) or a programmatic solution.
Making use of the programmatic solution allows us to use the multicast interface (the main objective of the Plain CFI version of the use case). In order to adapt better to the underlying deployment, a helper class will instantiate and bind as many EDRSlave components as nodes available. The following fragment of code is responsible for doing that:

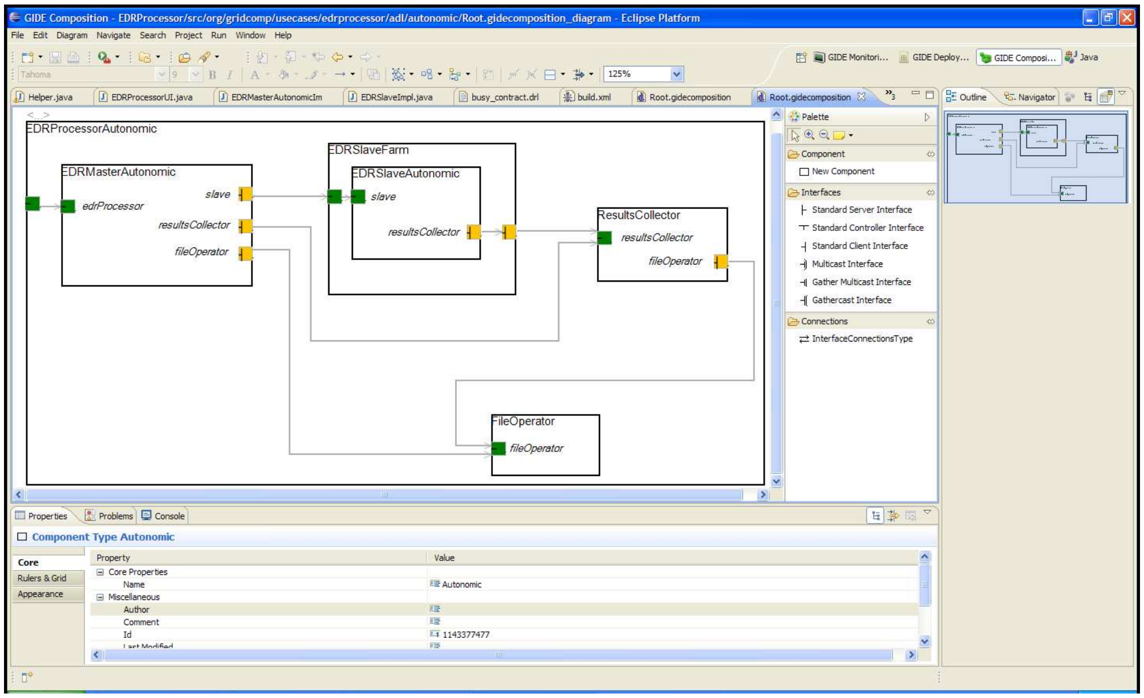
```
Factory factory = FactoryFactory.getFactory();
edrProcessor = (Component) factory.newComponent("EDRProcessor", context);

// Looks for well-known subcomponents
ContentController cc = (ContentController) edrProcessor
    .getFcInterface(Constants.CONTENT_CONTROLLER);
Component comps[] = cc.getFcSubComponents();
for (Component c : comps) {
  if (Fractal.getNameController(c).getFcName().equals("EDRMaster")) {
    edrMaster = c;
  } else if (Fractal.getNameController(c).getFcName().equals("ResultsCollector")) {
    resultsCollector = c;
  }
}

// create EDRSlave components for all nodes
// First node should already have an EDRSlave deployed in
slaves = new Component[nodes.length - 1];
for (int i = 0; i < slaves.length; i++) {
  slaves[i] = (Component) factory.newComponent("EDRSlave", context);
  Fractal.getBindingController(slaves[i]).bindFc("resultsCollector",
      resultsCollector.getFcInterface("resultsCollector"));
  Fractal.getBindingController(edrMaster).bindFc("slave",
      slaves[i].getFcInterface("slave"));
  // start slave component
  Fractal.getLifeCycleController(slaves[i]).startFc();
  logger.info("EDRSlave component " + i
      + " created, and bound to EDRMaster and ResultsCollector");
}
// start EDRProcessor component
Fractal.getLifeCycleController(edrProcessor).startFc();
```

4.1.2.1.2 *Autonomic version*

The autonomic version of the architectural design is focused in testing and assessing the Behavioural Skeleton developed in Work Package 3. This use case features a Task Farm, as it is the Behavioural Skeleton which fits better its needs.



Component Type Autonomic

Property	Value
Core Properties	
Name	Autonomic
Miscellaneous	
Author	
Comment	
Id	1143377477
Last Modified	

In this version, made using the Grid IDE [14], the multicast interface between the EDRMaster and the EDRSlave components has been replaced by a *Task Farm* component, the EDRSlaveFarm. This component will take care of deploying as many EDRSlave components as needed, controlling the parallelism degree. This degree can vary during the execution of the application, in order to adapt the performance to the requirements of the user.

4.1.2.2 Components description

4.1.2.2.1 EDRProcessor and EDRProcessorAutonomic

These are the higher level components from its corresponding architectural designs, the Plain CFI and the autonomic one. They contain the rest of the components from the architecture and offer a single *edrProcessor* server interface. The GUI of the application makes use of this interface to submit EDR processing requests.

4.1.2.2.2 EDRMaster and EDRMasterAutonomic

The EDRMaster and the EDRMasterAutonomic act as the master component of their corresponding architectures, offering an *edrProcessor* server interface. The main difference between them is their *slave* client interface: the former uses a *multicast* interface, while the latter uses a *singlecast* one, but bound to a Task Farm BeSke. Otherwise, the behavior of these components is identical:

- Scatter the file using the *fileOperator* client interface.
- Initialize the ResultsCollector component through its interface, telling it how many fragments must be processed.
- Process the fragments using the *slave* client interface.

4.1.2.2.3 EDRSlaveFarm

This component is only present in the autonomic version of the application. This composite component extends the *gridcomp.manager.farm.adl.farm* from the NFCF, offering a *farm* of

EDRSlaveAutonomic components. Using the non functional interfaces of this component, the user can modify the parallelism degree of the application.

4.1.2.2.4 EDRSlave and EDRSlaveAutonomic

These components only differ slightly in their ADL specification, in order to adapt them to their respective architectures. Otherwise, they are identical and have the same behavior.
These are the components in charge of applying the ETL process implemented using Kettle and described in a previous section of this document. When receiving the first request, the Kettle library will be initialized with all the needed configuration files. Subsequent requests will be processed faster, as no initialization has to be performed again.
When Kettle initialization is done, the component transfers the corresponding fragment of the EDR file from the node where the EDRMaster component is deployed. Then, the transformation is applied to the file using the Kettle library. The result is transferred back and the ResultsCollector is invoked to notify another fragment has been processed.

4.1.2.2.5 ResultsCollector

This component collects the intermediate results, sent from the EDRSlave or EDRSlaveFarm components. When all results are collected, they are joined, using the FileOperator.
Also, this component offers information about the progress of the processing (how many fragments have already been processed).

4.1.2.2.6 FileOperator

The FileOperator component offers the functionality to scatter and join files. Those files must reside in the local file system.

4.1.2.3 Interfaces

In this section, the interfaces from the different components are presented.

4.1.2.3.1 EDRProcessor

This is the server interface offered by the EDRProcessor, EDRProcessorAutonomic, EDRMaster and EDRMasterAutonomic components:

```
public interface EDRProcessor {

  /**
   * Processes the given EDR input file.
   *
   * @param inputFilePath path to the EDR input file
   * @param outputFilePath path to store the results file to
   * @param recordsPerFragment number of EDR to include on each fragment of the input file
   * @param gzipFiles whether to compress fragment files when transferring them
   */
  void process(String inputFilePath, String outputFilePath, int recordsPerFragment,
      boolean gzipFiles);
}
```

It contains a single, straight-forward method.

4.1.2.3.2 EDRSlave

This is the server interface offered by the EDRSlave and EDRSlaveAutonomic components:

```
public interface EDRSlave {

  /**
   * Processes an EDRRequest.
   * <p>
   * This means:
   * <UL>
   * <LI>Downloading the given fragment of the EDR file from source node</LI>
   * <LI>Processing all the contained EDRs, generating a partial results file</LI>
   * <LI>Uploading the partial results file to the source node</LI>
   * <LI>Calling the ResultsCollector component to let it know processing is finished</LI>
   * </UL>
   *
   * @param request the EDRRequest to be processed
   */
  void process(EDRRequest request);
}
```

An `EDRRequest` contains the path to one of the fragments of the EDR input file, the node where it is stored, the path to store the partial results, and a flag that states whether to compress those files when transferring them.

4.1.2.3.3 EDRSlaveMulticast

This is the multicast client interface used by the plain CFI version of the EDRMaster component to invoke the EDRSlave components.

```
public interface EDRSlaveMulticast {

  /**
   * Processes a list of EDRRequests, using Round Robin parameter dispatch mode.
   * <p>
   * This means:
   * <UL>
   * <LI>Downloading EDR fragment files from source node</LI>
   * <LI>Processing all the contained EDRs, generating partial results files</LI>
   * <LI>Uploading the partial results files to the source node</LI>
   * <LI>Calling the ResultsCollector component to let it know processing is finished</LI>
   * </UL>
   *
   * @param requests the list of EDRRequest to be processed
   */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN))
  void process(List<EDRRequest> requests);
}
```

4.1.2.3.4 FileOperator

This interface contains the needed operations with files: `scatter()` and `join()`.

```
public interface FileOperator {

  /**
   * Scatters the given file into fragments of the given size (expressed in number of
   * lines/Extended Data Records).
   * <p>
   *
   * @param source the file to be scattered
   * @param linesPerFile size of the fragments
   * @param gzipFiles whether to gzip fragments
   * @return List of the resulting Files
   */
  List<File> scatter(File source, int linesPerFile, boolean gzipFiles);

  /**
   * Joins the given source files into the given output one.
   *
   * @param sources list of files to be joined
   * @param outputFile where to save the resulting file
   * @param gzipFiles whether the source files are gzipped
   */
  void join(List<File> sources, File outputFile, boolean gzipFiles);
}
```

4.1.2.3.5 ResultsCollector

The server interface offered by the ResultsCollector component is the following:

```
public interface ResultsCollector {

  /**
   * Initializes the ResultsCollector component.
   *
   * @param numOfResultFilesToCollect number of partial results to be collected
   * @param resultsFile path to the results file
   * @param gzipFiles whether the files are compressed
   */
  void init(int numOfResultFilesToCollect, File resultsFile, boolean gzipFiles);

  /**
   * Informs the collector a new partial result is available, providing its location.
   * <p>
   * If the number of results files to be collected has been reached, this triggers the
   * joining of the partial results files into the final result, using the FileOperator.
   *
   * @param remoteFile path to the new partial result.
   */
  void collect(File remoteFile);

  /**
   * Gets the number of partial result files already collected.
   *
   * @return the number of partial result files already collected.
   */
  IntWrapper getResultFilesCount();

  /**
   * Gets the number of partial result files to be collected.
   *
   * @return the number of partial result files to be collected.
   */
  IntWrapper getResultFilesToCollect();
}
```

The `init()` method is invoked from the EDRMaster and EDRMasterAutonomic components, the `collect()` method from the EDRSlave and EDRSlaveAutonomic components, and the `getResultFilesCount()` and `getResultFilesToCollect()` methods from the Graphical User Interface, in order to provide feedback about the progress of the processing.

4.2 Manual

4.2.1 Final prototype description

The final prototype, as well as revised component architecture, includes several enhancements over the previous prototypes, specially the autonomic version:

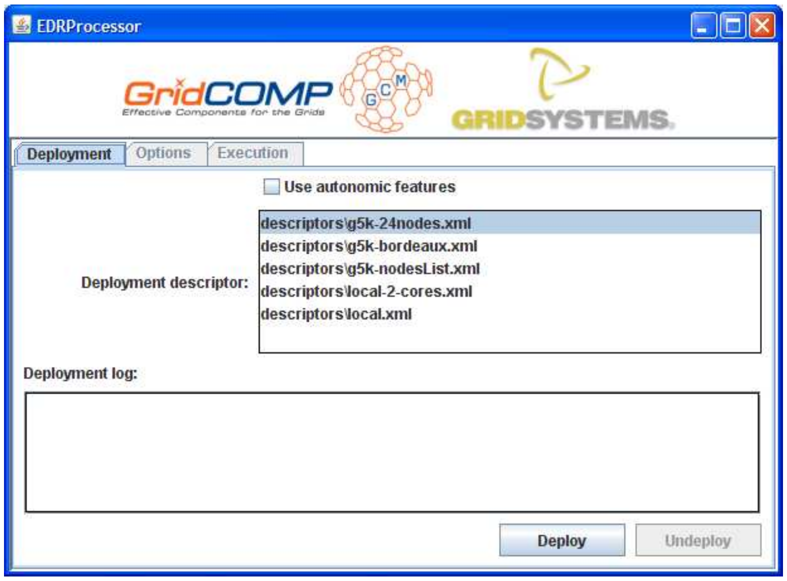
- Integration with final NFCF prototype.
- Dynamic quality of service (QoS) contract: desired throughput and maximum parallelism degree can be changed during the execution.
- “Idle” and “busy” QoS: when the application is idle the target is to reduce the deployment to its minimum (only one worker component). When the application is busy, the target is to reach the quality of service the user expects.
- Monitoring of the Task Farm: showing the achieved and desired throughput and the current parallelism degree.
- Full integration of the source code and ADL files with the GIDE: the programmer can navigate the source code using the composition diagrams.

4.2.2 Configuration and usage

Both the source code and the binaries of the final prototype are included in the file "D.UC.05.B – EDR Processor final prototype.zip". The latest version of this prototype is also publicly available at INRIA's GForge `gridcompwp5gs` project [12].
In addition to the common system requirements (listed in the introduction of this document (Section 1)), the following tools are needed in order to run the application:

- Gnuplot [10] standalone, or
- Gnuplot bundled in cygwin distribution [11] (windows only).

After uncompressing the aforementioned zip file, and assuming that both java and ant are in the path, just type `ant processor` to invoke the EDR Processor. The application will request to enter the path to the distribution folder of ProActive 3.90. After that, the user interface will appear:

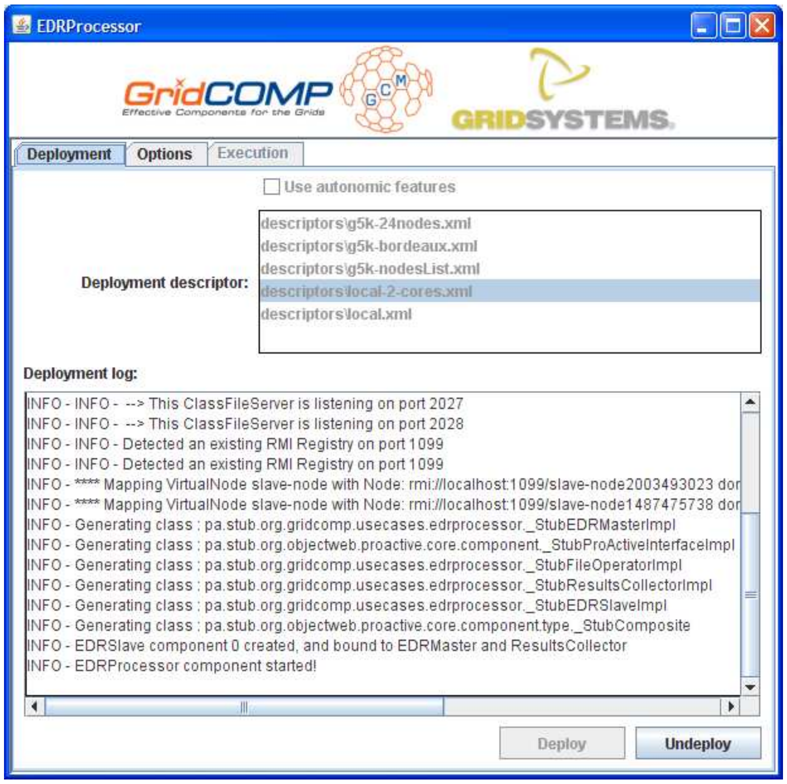


Use autonomic features

Deployment descriptor:
descriptors\g5k-24nodes.xml
descriptors\g5k-bordeaux.xml
descriptors\g5k-nodesList.xml
descriptors\local-2-cores.xml
descriptors\local.xml

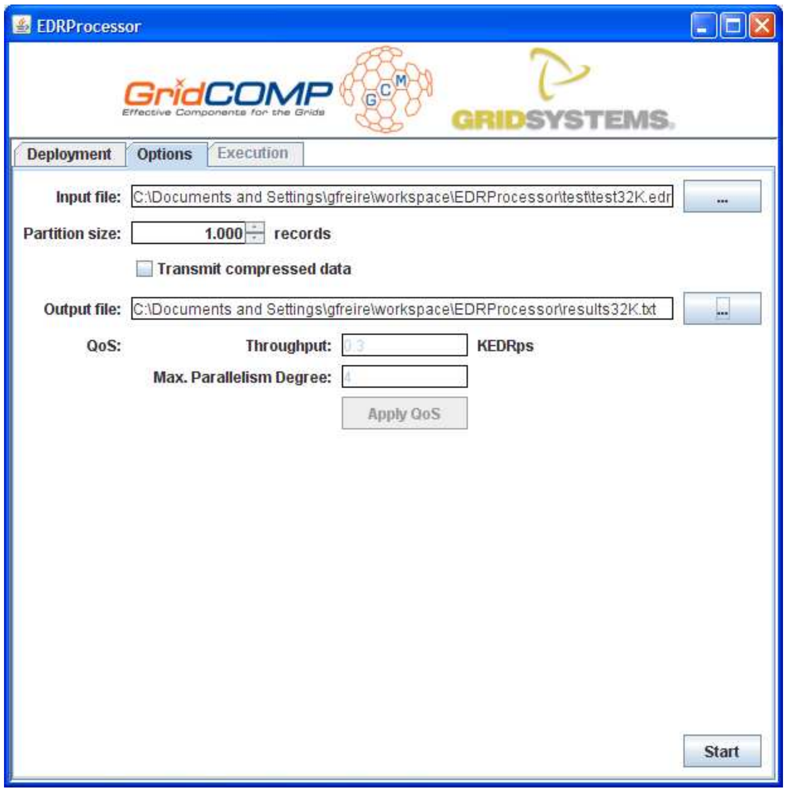
Deployment log:

This first "tab" contains the deployment details. Depending on your infrastructure, select one of the included deployment descriptors and press the "Deploy" button.
The "Deployment log" text box will show the log trace output during the deployment:



```
INFO - INFO -  --> This ClassFileServer is listening on port 2027
INFO - INFO -  --> This ClassFileServer is listening on port 2028
INFO - INFO - Detected an existing RMI Registry on port 1099
INFO - INFO - Detected an existing RMI Registry on port 1099
INFO - **** Mapping VirtualNode slave-node with Node: rmi://localhost:1099/slave-node2003493023 dor
INFO - **** Mapping VirtualNode slave-node with Node: rmi://localhost:1099/slave-node1487475738 dor
INFO - Generating class : pa.stub.org.gridcomp.usecases.edrprocessor._StubEDRMasterImpl
INFO - Generating class : pa.stub.org.objectweb.proactive.core.component._StubProActiveInterfaceImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.edrprocessor._StubFileOperatorImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.edrprocessor._StubResultsCollectorImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.edrprocessor._StubEDRSlaveImpl
INFO - Generating class : pa.stub.org.objectweb.proactive.core.component.type._StubComposite
INFO - EDRSlave component 0 created, and bound to EDRMaster and ResultsCollector
INFO - EDRProcessor component started!
```

After the deployment is done, the “Options” tab is enabled:



Input file: C:\Documents and Settings\gfreire\workspace\EDRProcessor\test\test32K.edr
Partition size: 1.000 records
☐ Transmit compressed data
Output file: C:\Documents and Settings\gfreire\workspace\EDRProcessor\results32K.txt
QoS: Throughput: 0.3 KEDRps
Max. Parallelism Degree: 4

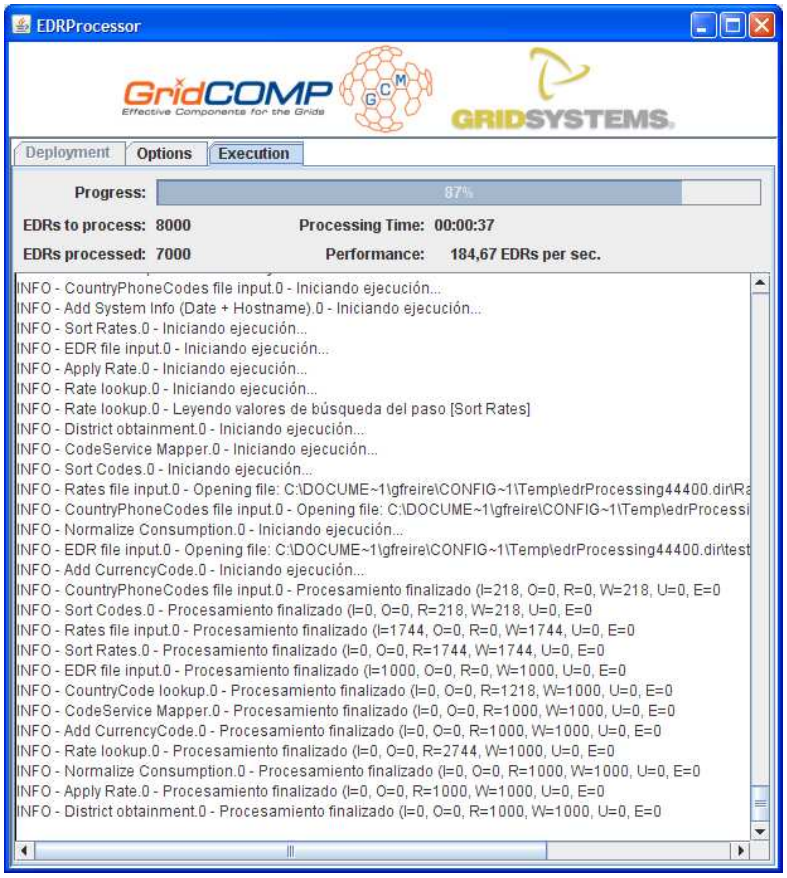
The “options” tab contains the controls to select the desired input parameters:

- Input file: path to the file containing the EDRs to be processed.

- Partition size: number of EDRs each fragment file will contain.
- Transmit compressed data: whether to compress the fragments of the input file before transferring them. This may reduce the time needed to transfer the data through the network.
- Output file: path to the file where the results of the processing will be stored.

The controls related to the autonomic behaviour are, obviously, disabled when running the plain CFI (non-autonomic) version of the application.

When all of the above fields have been complimented, the “Start” button can be pushed. The request will be submitted to the components, and the “Execution” tab will be enabled, showing the log trace of the execution and a progress bar.



Progress: 87%

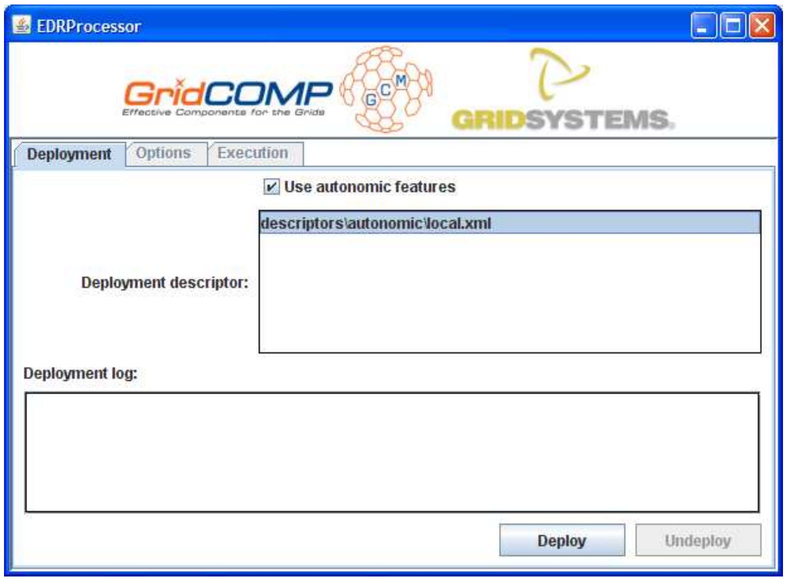
EDRs to process: 8000
Processing Time: 00:00:37
EDRs processed: 7000
Performance: 184,67 EDRs per sec.

```
INFO - CountryPhoneCodes file input.0 - Iniciando ejecución...
INFO - Add System Info (Date + Hostname).0 - Iniciando ejecución...
INFO - Sort Rates.0 - Iniciando ejecución...
INFO - EDR file input.0 - Iniciando ejecución...
INFO - Apply Rate.0 - Iniciando ejecución...
INFO - Rate lookup.0 - Iniciando ejecución...
INFO - Rate lookup.0 - Leyendo valores de búsqueda del paso [Sort Rates]
INFO - District obtainment.0 - Iniciando ejecución...
INFO - CodeService Mapper.0 - Iniciando ejecución...
INFO - Sort Codes.0 - Iniciando ejecución...
INFO - Rates file input.0 - Opening file: C:\DOCUME~1\gfreire\CONFIG~1\Temp\edrProcessing44400.dir\Ra
INFO - CountryPhoneCodes file input.0 - Opening file: C:\DOCUME~1\gfreire\CONFIG~1\Temp\edrProcessi
INFO - Normalize Consumption.0 - Iniciando ejecución...
INFO - EDR file input.0 - Opening file: C:\DOCUME~1\gfreire\CONFIG~1\Temp\edrProcessing44400.dir\test
INFO - Add CurrencyCode.0 - Iniciando ejecución...
INFO - CountryPhoneCodes file input.0 - Procesamiento finalizado (I=218, O=0, R=0, W=218, U=0, E=0
INFO - Sort Codes.0 - Procesamiento finalizado (I=0, O=0, R=218, W=218, U=0, E=0
INFO - Rates file input.0 - Procesamiento finalizado (I=1744, O=0, R=0, W=1744, U=0, E=0
INFO - Sort Rates.0 - Procesamiento finalizado (I=0, O=0, R=1744, W=1744, U=0, E=0
INFO - EDR file input.0 - Procesamiento finalizado (I=1000, O=0, R=0, W=1000, U=0, E=0
INFO - CountryCode lookup.0 - Procesamiento finalizado (I=0, O=0, R=1218, W=1000, U=0, E=0
INFO - CodeService Mapper.0 - Procesamiento finalizado (I=0, O=0, R=1000, W=1000, U=0, E=0
INFO - Add CurrencyCode.0 - Procesamiento finalizado (I=0, O=0, R=1000, W=1000, U=0, E=0
INFO - Rate lookup.0 - Procesamiento finalizado (I=0, O=0, R=2744, W=1000, U=0, E=0
INFO - Normalize Consumption.0 - Procesamiento finalizado (I=0, O=0, R=1000, W=1000, U=0, E=0
INFO - Apply Rate.0 - Procesamiento finalizado (I=0, O=0, R=1000, W=1000, U=0, E=0
INFO - District obtainment.0 - Procesamiento finalizado (I=0, O=0, R=1000, W=1000, U=0, E=0
```

When the progress bar reaches 100%, the execution is done (all fragments from the EDR file have been processed and their results joined), and new requests can be submitted.

4.2.2.1.1 *Autonomic version*

In order to test the autonomic version of the application, the “Use autonomic features” check box must be checked in the “Deployment” tab, and one of the specific deployment descriptors must be selected.


EDRProcessor

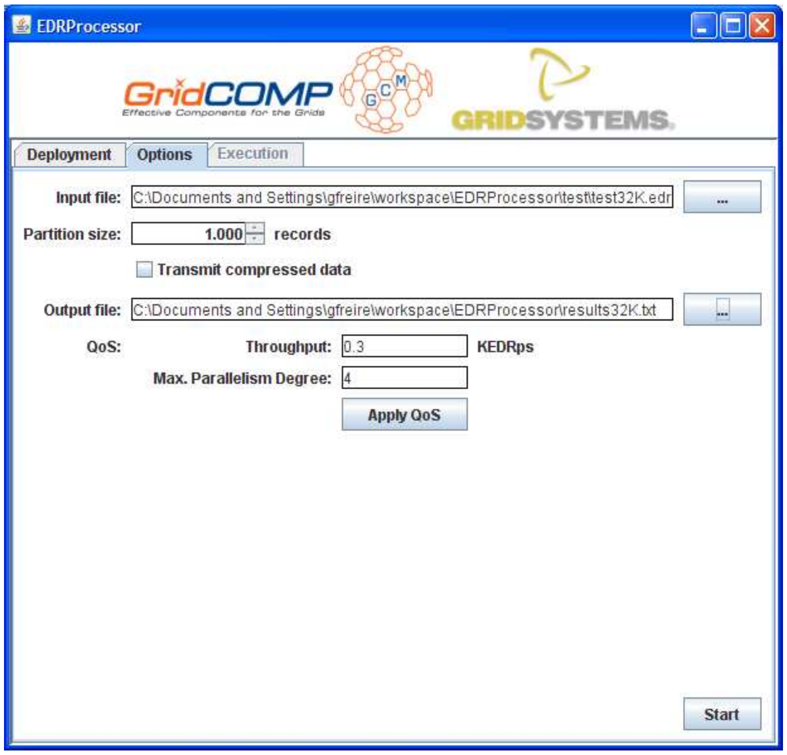
Deployment | Options | Execution

✓ Use autonomic features

Deployment descriptor: descriptors\autonomic\local.xml

Deployment log:

When in autonomic mode, the QoS controls are enabled, letting the user decide the target throughput (measured in thousands of EDR processed per second) and the maximum allowed parallelism degree (the maximum number of workers). After changing any of those values, the user must press the “Apply QoS” button to make effective that change.


EDRProcessor

Deployment | Options | Execution

Input file: C:\Documents and Settings\gfreire\workspace\EDRProcessor\test\test32K.edr

Partition size: 1.000 records

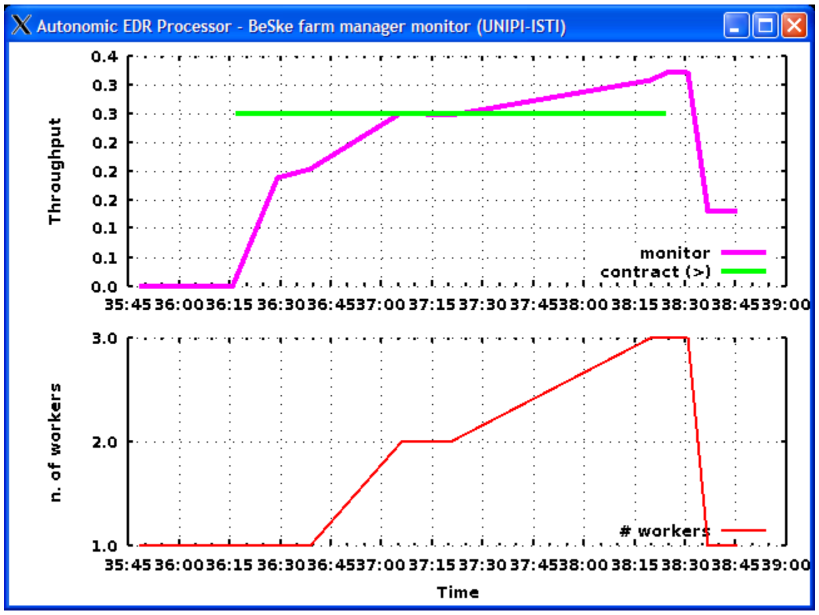
☐ Transmit compressed data

Output file: C:\Documents and Settings\gfreire\workspace\EDRProcessor\results32K.txt

QoS: Throughput: 0.3 KEDRps

Max. Parallelism Degree: 4

During the execution of the request, the user can have access to the monitoring of the EDRSlaveFarm, issuing the following command in a shell: `gnuplot farm_monitor.gp` (or `gnuplot farm_monitor_windows.gp` in windows). This will open a window displaying two different charts made with the monitoring information coming from the BeSke:



The green line represents the throughput requirement as entered in the QoS section of the GUI (and will change accordingly). The magenta line is the current throughput, as measured by the monitor. That measure will differ slightly from the one being offered by the GUI, as they are not computed the same way: the manager computes a running average (based on time windows, and more sensitive to changes in the parallelism degree) while the GUI displays the complete average (from start to finish). Finally, the red line represents the current number of workers (EDRSlaveAutonomic components) bound to the EDRSlaveFarm.
In the above figure, we can see how the autonomic manager adds more workers after realizing the required throughput could not be achieved with the initial single worker. When the processing is done, the extra workers are removed.

4.2.3 Examples

The /test folder contains several sample input files (generated using the script provided on the same folder), ranging from one thousand EDRs to one million EDRs. If needed, more files can be generated, invoking the EDRGenerator tool (`ant generator` in the main folder). Generating random EDR files is a time consuming task so, in order to create a new file, it is advised to use one of the included ones to repeatedly append it to the new one.

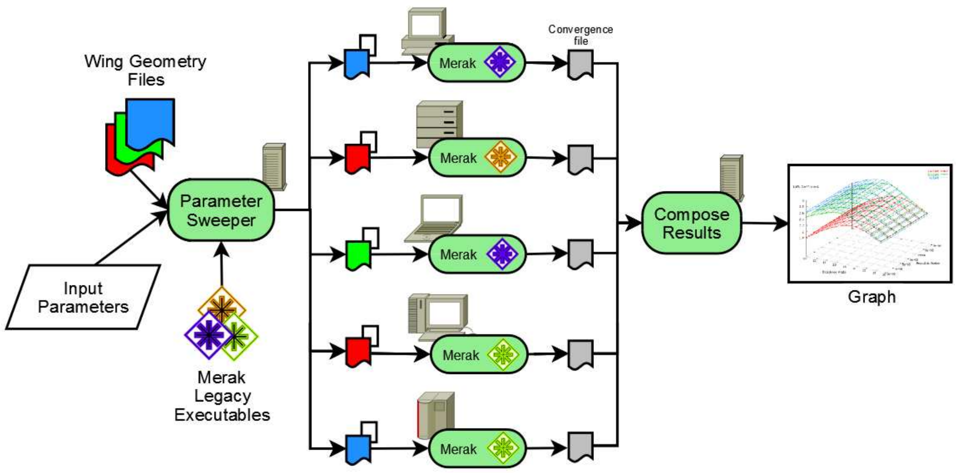
5 Wing Design

This use case, in contrast to the Telco one (Section 4), manages small amounts of information, but needs lots of computing power. The application is also an embarrassingly parallel one, but based on pre-existing legacy code. The ability to wrap and grid-enable existing legacy code is crucial for the adoption of the GCM by the industry.
In the aerospace sector, the software that computes the aerodynamic wing performance for a given configuration is used to test different configurations of the wing features and to eventually find an acceptable design. Merak is a legacy application, written in FORTRAN 77 by Dr. Mariano Vázquez, with binaries for Windows, Linux and Solaris. This application permits to analyze parameters variation in incompressible turbulent flow around a triple element airfoil, to evaluate the stalling angle of different wing geometries. Turbulence is simulated using different k-epsilon models[2], including law-of-the-wall and two-layer low Re[3]. Finally, the application can extract the desired information and create a graph using gnuplot [10].

5.1 Detailed architectural design

5.1.1 Architecture of the application

Merak, manages small amounts of information, but needs lots of computing power. Our objective is to use GridComp solution to wrap and grid-enable this existing legacy code and also to prove the integration of data staging for the input files and output files into this sweeping process.
A depiction of the operation of the Wing Design application is offered next:



1. The user provides a set of wing geometry files and input parameters for the experiment.
2. The legacy application binaries (Merak) are provisioned to the resources on the grid, as new components.

[2] http://en.wikipedia.org/wiki/Turbulent_Kinetic_Energy
[3] http://en.wikipedia.org/wiki/Reynolds_number

3. The complete set of parameter combinations is obtained by the Parameter Sweeper component.
4. Each parameter combination is sent to a Merak component, which performs its simulation.
5. Results are collected, composed and a graph is generated.

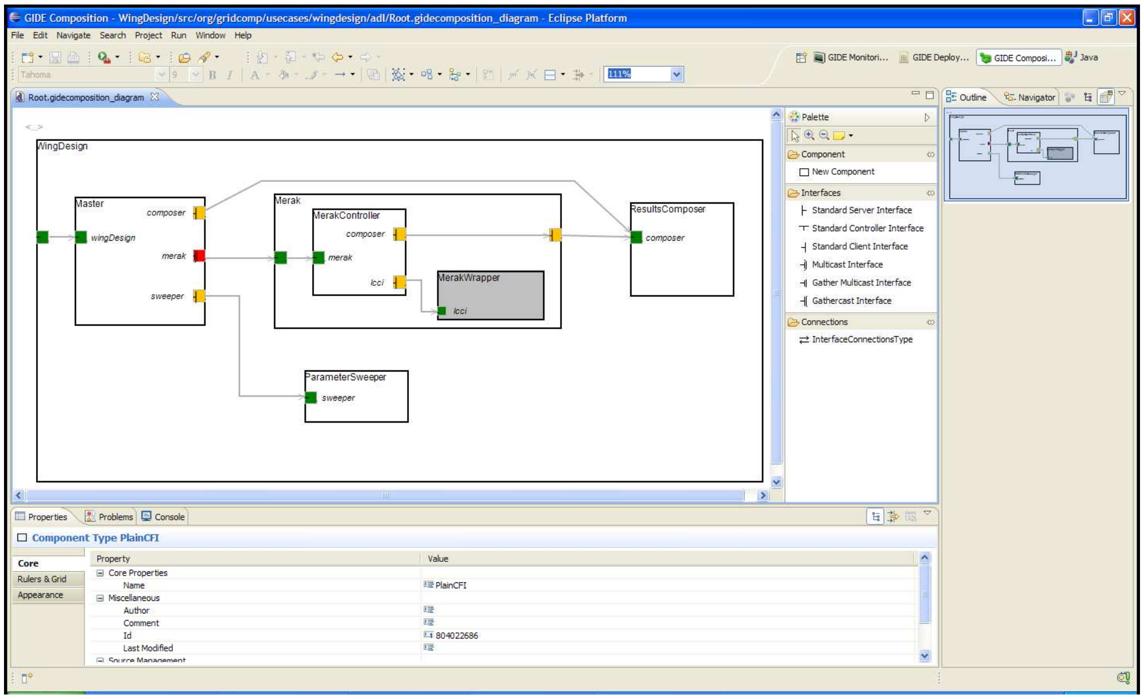
Legacy executable files are only available for Windows, Linux and Solaris, so resources running these operating systems are needed. Also, result graph is generated using gnuplot [10], which must be installed in the computer running the application.
In order to provide a better user experience, an interactive graph is built during the execution (per wing geometry involved). The user can change the point of view; zoom in and out, etc. These interactive graphs are generated using Visad [8], which is based on Java3D [9] that must be present in the computer running the application.

5.1.2 GCM Components

5.1.2.1 Components diagram

5.1.2.1.1 Plain CFI version

As in the previous use case, this version of the architectural design uses only features from the Component Framework Infrastructure (developed in Work Package 2): a multicast interface is used in order to distribute the computation effort among the nodes of the grid.


GIDE Composition - WingDesign/src/org/gridcomp/usecases/wingdesign/adl/Root.gidecomposition_diagram - Eclipse Platform
File Edit Navigate Search Project Run Window Help
GIDE Monitori... GIDE Deploy... GIDE Composi... Java
Root.gidecomposition_diagram
WingDesign
Master
composer
wingDesign
merak
sweeper
Merak
MerakController
composer
merak
lcci
MerakWrapper
lcci
ResultsComposer
composer
ParameterSweeper
sweeper
Palette
Component
New Component
Interfaces
Standard Server Interface
Standard Controller Interface
Standard Client Interface
Multicast Interface
Gather Multicast Interface
Gathercast Interface
Connections
InterfaceConnectionsType
Outline
Navigator
Properties
Problems
Console
Core
Rulers & Grid
Appearance

Component Type PlainCFI

Property	Value
Core Properties	
Name	PlainCFI
Miscellaneous	
Author	
Comment	
Id	804022686
Last Modified	
Source Management	

A brief explanation of this architectural design, made using the Grid IDE [14], is the following:

- The *WingDesign* receives the request to perform a simulation, given a set of wing configurations and input parameters. This request is redirected to the Master component.

- Using the *ParameterSweeper*, the complete list of parameter combinations to evaluate is obtained.

- The above information is also passed to the *ResultsComposer*. This is done in order to be able to keep track of the progress of the computations.

- Each one of the parameter combinations is delivered to a *Merak* component, using the multicast interface.

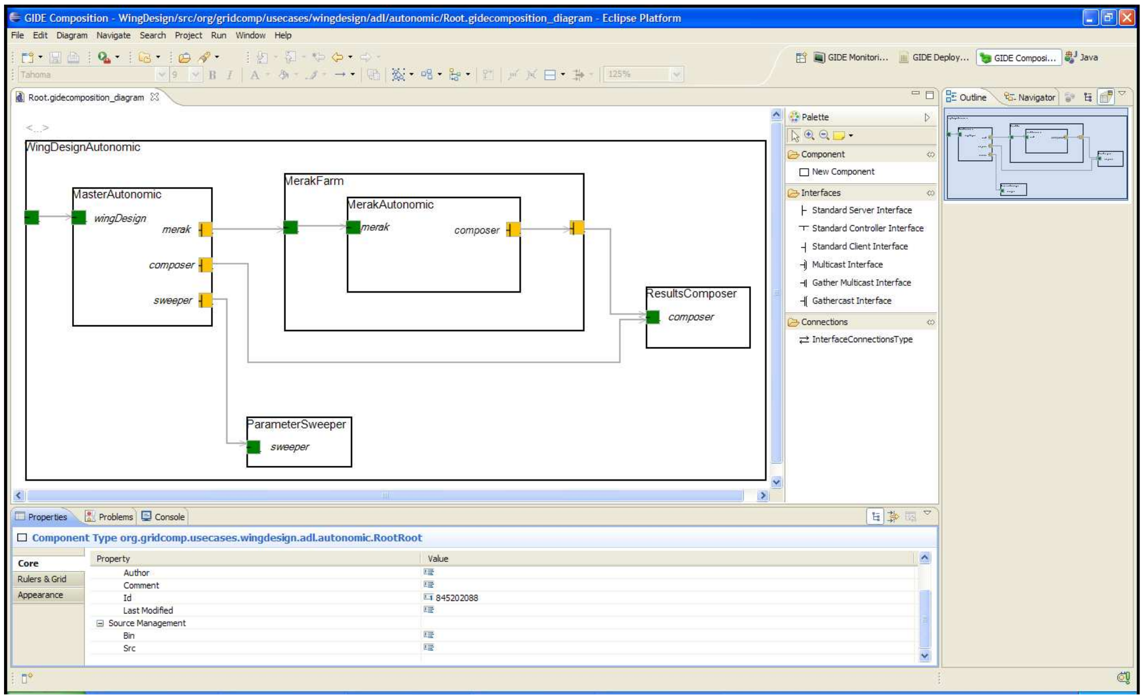
- Merak is a composite component, consisting in a *MerakController* and a *MerakWrapper*: the latter wraps the legacy code (using the interfaces and techniques contributed by Tsinghua University [13]), while the former prepares and transfers both the parameter and result files. Combining these components, Merak performs the simulation using the given parameters.

- Results are delivered to the *ResultsComposer.* When all results are received, the graph showing the comparison of the results is made.

As in the EDR Processor, for this version of the architectural design the same programmatic mechanism is used to deploy as many Merak composite components as nodes available.

5.1.2.1.2 *Autonomic version*

The autonomic version of the components diagram, made using the Grid IDE [14], is the following:



Component Type org.gridcomp.usecases.wingdesign.adl.autonomic.RootRoot

Property	Value
Author	
Comment	
Id	845202088
Last Modified	
Source Management	
Bin	
Src	

This autonomic architectural design makes use of a Farm controller (the *MerakFarm),* replacing the multicast interface between the *Master* and the *Merak* components from the plain CFI version. The *MerakFarm* controls the parallelism degree of the application, deploying as many *MerakAutonomic* components as needed. This degree can vary during the execution of the application, in order to adapt to the performance requirements of the user.

As the task farm behavioural skeleton does not offer complete support for composite workers (monitoring and rebalancing of tasks have issues), we had to turn the MerakAutonomic component into a primitive one, removing the legacy code wrapper component, and using the ad hoc approach as in previous deliverables ([1], [2]).

5.1.2.2 Components description

5.1.2.2.1 WingDesign and WingDesignAutonomic

These are the higher level components of their respective architectural designs. They are composite components, encapsulating all the others, and offering a server interface, name *wingDesign*, which is used by the graphical user interface. Through this interface the user can submit simulation requests, providing one or more wing geometry files and the needed input parameters.

5.1.2.2.2 Master and MasterAutonomic

The previous components redirect their wingDesign server interface to these ones. They control the execution, invoking the appropriate interfaces from the different components they interact with:

- Initialize the ResultComposer component with the input parameters.
- Initialize the Merak components, with the appropriate legacy application binaries for their platform.
- Obtain the list of all the parameter combinations that must be processed, calling the ParameterSweeper component.
- Processes the list of parameter combinations, invoking the Merak components (whether through the multicast interface or the farm controller).

5.1.2.2.3 ParameterSweeper

This component computes the complete list of parameter combinations to be processed. This is simply the Cartesian product of:

- The range of incidence angles
- The range of Reynolds numbers
- The range of wing configurations

This component must be co-allocated with the *Master* (or *MasterAutonomic*) one, as it needs local access to the wing geometry files.

5.1.2.2.4 MerakFarm

As in the previous use case, this component is only present in the autonomic version of the application. This composite component extends the *gridcomp.manager.farm.adl.farm* from the NFCF, offering a *farm* of *MerakAutonomic* components. Using the non functional interfaces of this component, the user can modify the parallelism degree of the application, both directly or setting a QoS contract.

5.1.2.2.5 MerakAutonomic

This is the worker component of the MerakFarm. It is a primitive component that manages the execution of the legacy code. It is not based on the legacy code wrapping techniques from Tsinghua University [13], as this will require the use of a composite component, and the task farm behavioural skeleton has issues with this kind of workers.

5.1.2.2.6 *Merak*

In the plain CFI version, this component takes care of everything related to the execution of the legacy application, encapsulating the two primitive components described next: MerakController and MerakWrapper.

5.1.2.2.7 *MerakController*

The *MerakController* component makes sure the execution of the legacy code can be achieved:

- Downloads the proper executable files from the master node on initialization (this is only done once).
- Processes each received request for execution and prepares the input parameters
- Invokes the legacy code control interface of the *MerakWrapper* component, thus executing the legacy application.
- Transfers the result file after finishing the execution.
- Deletes temporary file when done.

5.1.2.2.8 *MerakWrapper*

This component, deriving from the templates and interfaces provided by Tsinghua University [13], wraps the legacy code, and keeps track of its execution.

5.1.2.2.9 *ResultsComposer*

The *ResultsComposer* gathers the result files from the simulations, generating a graph where the different wing geometries are compared. It also offers information about the progress of the process and temporary results, allowing the graphical user interface to display that results *live*.

5.1.2.3 *Interfaces*

In this section, the interfaces from the different components are presented.

5.1.2.3.1 *WingDesign*

This is the interface offered by the *WingDesign*, *WingDesignAutonomic*, *Master* and *MasterAutonomic* components. Its only method is the starting point of the whole process:

```
public interface WingDesign {
  /**
   * Performs a simulation, processing the given parameter specification.
   *
   * @param spec contains the wing geometries, incidence angles, reynolds and iteration
   *        numbers to perform the simulation
   * @return a graph comparing the different wing geometries.
   */
  File process(SweepSpecification spec);
}
```

A `SweepSpecification` is a java class that contains a list of geometry files, the initial and final values for the incidence angle and the Reynolds number ranges, the number of samples to take from each range, and a number of iterations to perform the simulation.

5.1.2.3.2 *ParameterSweeper*

This interface contains the method to generate the list of parameter combinations to be used when invoking the legacy application (merak):

```
public interface ParameterSweeper {
  /**
   * Generates the combination of parameters to be submitted to merak.
   * <p>
   * This is the Cartesian product of the ranges contained in the sweep specification
   *
   * @param spec the specification of the ranges of values to be swept
   * @return a list of Merak Parameters
   */
  List<MerakParameters> sweep(SweepSpecification spec);
}
```

5.1.2.3.3 *Merak*

The interface of the Merak, MerakAutonomic and MerakController components, offering access to the legacy application is the following:

```
public interface Merak {

  /**
   * Prepares the component for the execution of the Merak legacy application.
   *
   * @param supplier the node to download the executable files from
   * @param platformFiles map with the executable files per platform.
   * @param resultsDir path on the supplier node where result files must be stored
   */
  void prepare(Node supplier, Map platformFiles, File resultsDir);

  /**
   * Runs the Merak legacy application, using the given parameters.
   *
   * @param params parameters to pass to the legacy application
   */
  void run(MerakParameters params);

  /**
   * Deletes all temporary files stored in the local file system during preparation or
   * execution.
   */
  void cleanUp();
}
```

5.1.2.3.4 *MerakMulticast*

This is the multi-cast client interface used by the plain CFI version of the *Master* component to invoke the *Merak* components:

```
public interface MerakMulticast {

  /**
   * Prepares the component for the execution of the Merak legacy application.
   *
   * @param supplier the node to download the executable files from
   * @param platformFiles map with the executable files per platform.
   * @param resultsDir path on the supplier node where result files must be stored
   */
  void prepare(Node supplier, Map platformFiles, File resultsDir);

  /**
   * Runs the Merak legacy application, using the given list of parameters, using Round
   * Robin parameter dispatch mode.
   *
   * @param params the list of parameters to pass to the legacy application
   */
  @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode = ParamDispatchMode.ROUND_ROBIN))
  void run(List<MerakParameters> params);
  /**
   * Deletes all temporary files stored in the local file system during preparation or
   * execution.
   */
  void cleanUp();
}
```

The only method offering a multicast behaviour is `run`. This is dispatched in a round robin fashion among the *Merak* components.

5.1.2.3.5 ResultsComposer

The server interface offered by the *ResultsComposer* component is the following:

```
public interface ResultsComposer {

  /**
   * Initializes the component.
   *
   * @param spec the input parameters to be processed
   * @param resultsDir the path to the directory where temporary results will be stored
   */
  void init(SweepSpecification spec, File resultsDir);

  /**
   * Adds a new result (a point in the graph for a certain wing geometry) to the composer.
   * When all results are added, the graph comparing the different wing geometries is
   * made.
   *
   * @param geoPoint a point in the graph for a certain wing geometry
   */
  void addResult(GeometryPoint geoPoint);

  /**
   * Gets the percentage of results received since the component was initialized.
   *
   * @return the percentage of results received since the component was initialized
   */
  Integer getResultsPercent();

  /**
   * Gets a list with all the results added since the last call to this method.
   *
   * @return a list with all the results added since the last call to this method
   */
  List<GeometryPoint> getPendingResults();
}
```

The `init` method is invoked by the *WingDesign* component at the start of the process. The `addResult` method is invoked by the *Merak* components each time a new result is obtained. Last, `getResultsPercent` and `getPendingResults` are invoked from the GUI in order to implement a progress bar and the interactive *live* graphs, respectively.

5.2 Manual

5.2.1 Final prototype description

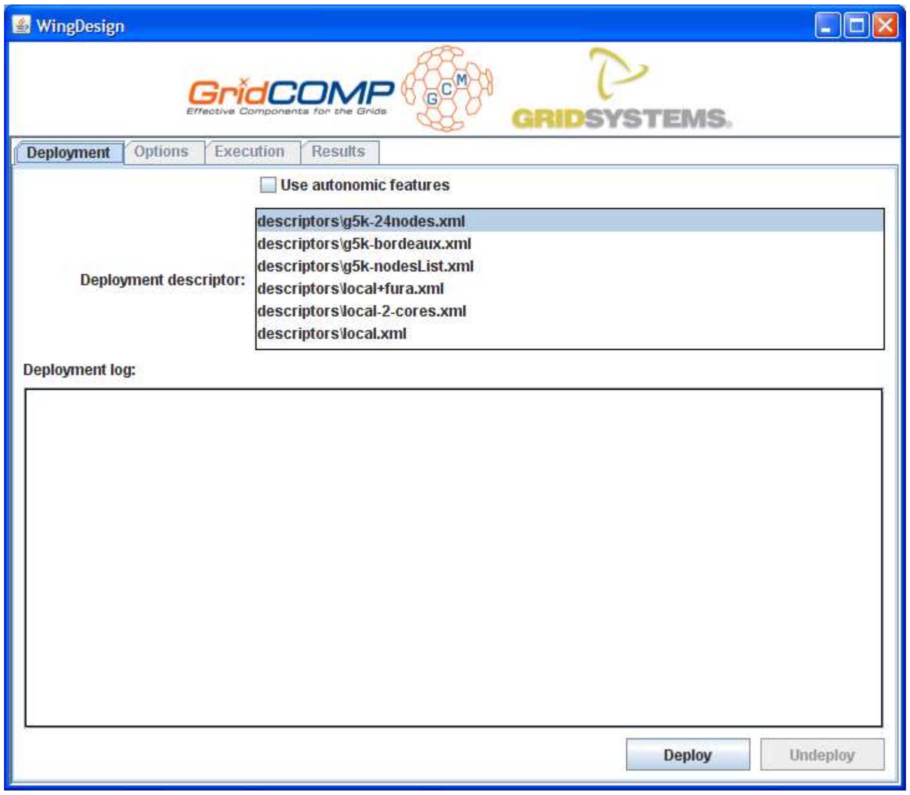
The final prototype of the Wing Design use case features the same enhancements as the EDR Processor, described in 4.2.1, and also makes use of the legacy code wrapping techniques contributed by the Tsinghua University [13] (only the non-autonomic version).

5.2.2 Configuration and usage

The file “D.UC.05.A – Wing Design final prototype.zip” contains both the source and the binaries of the final prototype. The latest version of this prototype is also publicly available at INRIA's GForge `gridcompwp5gs` project [12].
In addition to the common system requirements (listed in the introduction of this document (Section 1)), the following tools are needed in order to run the application:

- Java3D [9]
- Gnuplot standalone [10] or
- Gnuplot bundled in cygwin distribution [11] (windows only).

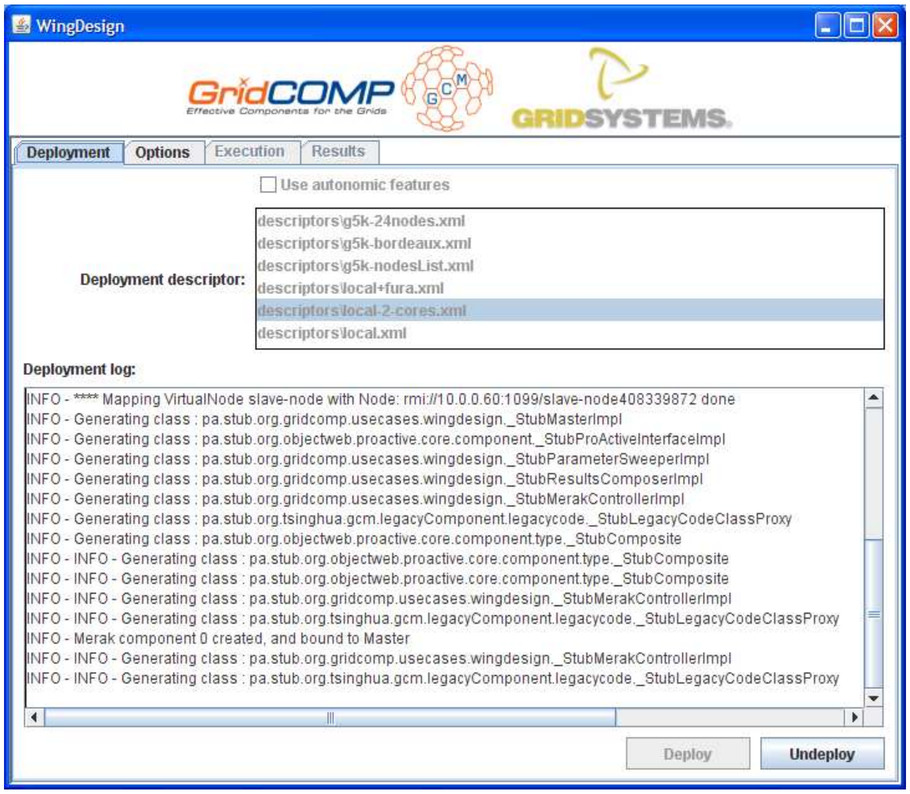
Run `ant WingDesign` to invoke the Wing Design. The application will request you to enter the path to the distribution folder of ProActive 3.90. After that, the user interface will appear:



descriptors\g5k-24nodes.xml
descriptors\g5k-bordeaux.xml
descriptors\g5k-nodesList.xml
descriptors\local+fura.xml
descriptors\local-2-cores.xml
descriptors\local.xml

This first “tab” contains the deployment details. Depending on your infrastructure, select one of the included deployment descriptors and press the “Deploy” button.

The “Deployment log” text box will show the log trace output during the deployment:

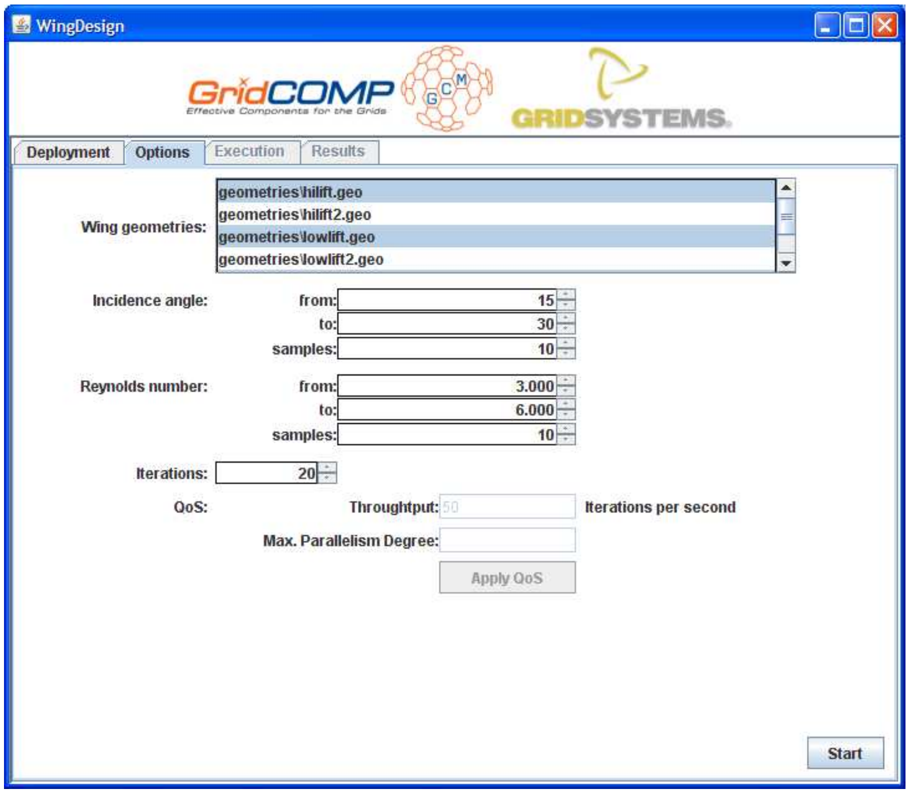


descriptors\g5k-24nodes.xml
descriptors\g5k-bordeaux.xml
descriptors\g5k-nodesList.xml
descriptors\local+fura.xml
descriptors\local-2-cores.xml
descriptors\local.xml

```
INFO - **** Mapping VirtualNode slave-node with Node: rmi://10.0.0.60:1099/slave-node408339872 done
INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubMasterImpl
INFO - Generating class : pa.stub.org.objectweb.proactive.core.component._StubProActiveInterfaceImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubParameterSweeperImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubResultsComposerImpl
INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubMerakControllerImpl
INFO - Generating class : pa.stub.org.tsinghua.gcm.legacyComponent.legacycode._StubLegacyCodeClassProxy
INFO - Generating class : pa.stub.org.objectweb.proactive.core.component.type._StubComposite
INFO - INFO - Generating class : pa.stub.org.objectweb.proactive.core.component.type._StubComposite
INFO - INFO - Generating class : pa.stub.org.objectweb.proactive.core.component.type._StubComposite
INFO - INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubMerakControllerImpl
INFO - INFO - Generating class : pa.stub.org.tsinghua.gcm.legacyComponent.legacycode._StubLegacyCodeClassProxy
INFO - Merak component 0 created, and bound to Master
INFO - INFO - Generating class : pa.stub.org.gridcomp.usecases.wingdesign._StubMerakControllerImpl
INFO - INFO - Generating class : pa.stub.org.tsinghua.gcm.legacyComponent.legacycode._StubLegacyCodeClassProxy
```

After the deployment is done, the “Options” tab is enabled. This tab includes controls to select the input parameters:

- Wing geometries: all .geo files found in the geometries folder are listed; one or more can be selected (Ctrl + click).
- Range of incidence angle: from, to, and number of samples
- Range of Reynolds number: from, to, and number of samples
- Number of iterations



Wing geometries: geometries\hilift.geo, geometries\hilift2.geo, geometries\lowlift.geo, geometries\lowlift2.geo

Incidence angle: from: 15 to: 30 samples: 10

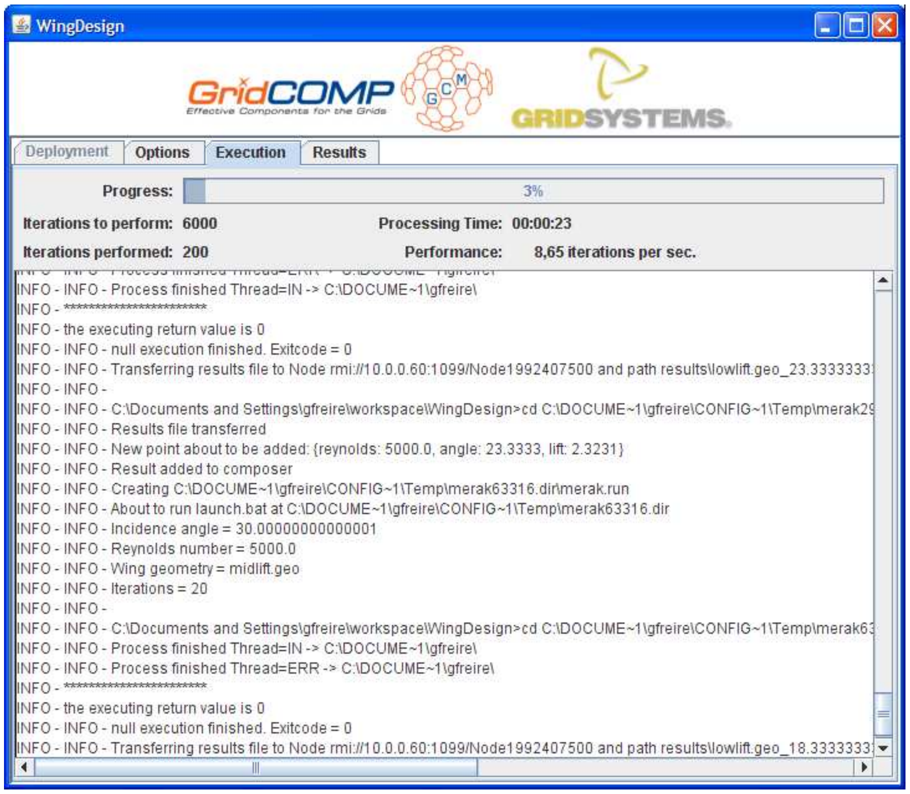
Reynolds number: from: 3.000 to: 6.000 samples: 10

Iterations: 20

QoS: Throughtput: 50 Iterations per second

Max. Parallelism Degree:

The “start” button submits the request, when pressed. The “Execution” and “Results” tabs are enabled. The former shows log messages and a progress bar:



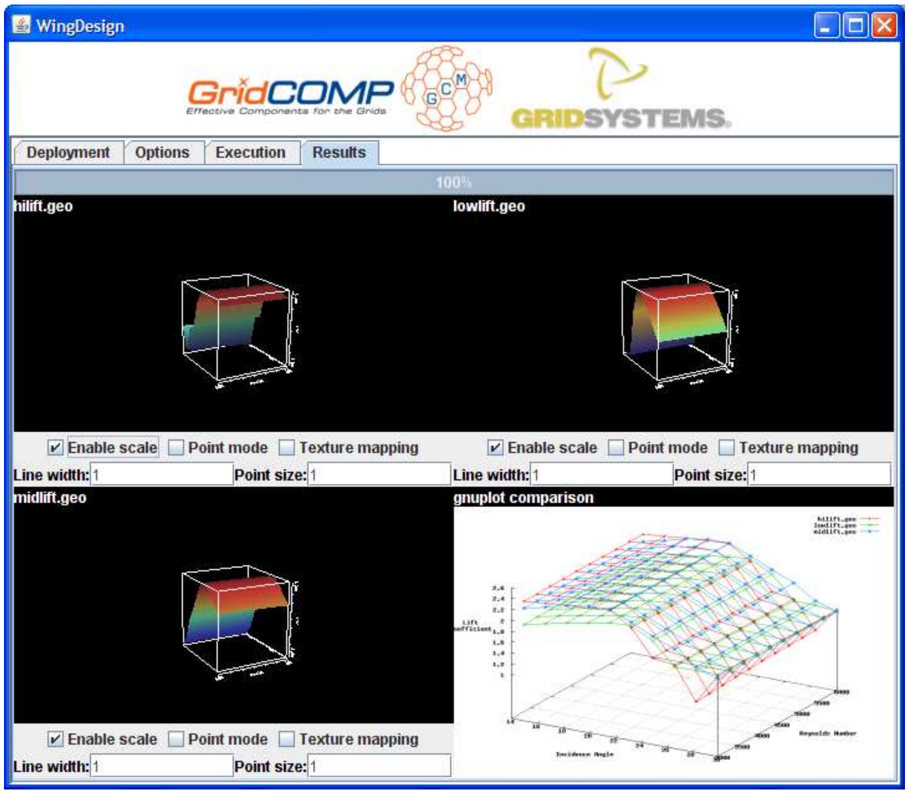
Progress: 3%

Iterations to perform: 6000 Processing Time: 00:00:23

Iterations performed: 200 Performance: 8,65 iterations per sec.

```
INFO - INFO - Process finished Thread=IN -> C:\DOCUME~1\gfreire\
INFO - ************************
INFO - the executing return value is 0
INFO - INFO - null execution finished. Exitcode = 0
INFO - INFO - Transferring results file to Node rmi://10.0.0.60:1099/Node1992407500 and path results\lowlift.geo_23.3333333
INFO - INFO -
INFO - INFO - C:\Documents and Settings\gfreire\workspace\WingDesign>cd C:\DOCUME~1\gfreire\CONFIG~1\Temp\merak29
INFO - INFO - Results file transferred
INFO - INFO - New point about to be added: {reynolds: 5000.0, angle: 23.3333, lift: 2.3231}
INFO - INFO - Result added to composer
INFO - INFO - Creating C:\DOCUME~1\gfreire\CONFIG~1\Temp\merak63316.dir\merak.run
INFO - INFO - About to run launch.bat at C:\DOCUME~1\gfreire\CONFIG~1\Temp\merak63316.dir
INFO - INFO - Incidence angle = 30.00000000000001
INFO - INFO - Reynolds number = 5000.0
INFO - INFO - Wing geometry = midlift.geo
INFO - INFO - Iterations = 20
INFO - INFO -
INFO - INFO - C:\Documents and Settings\gfreire\workspace\WingDesign>cd C:\DOCUME~1\gfreire\CONFIG~1\Temp\merak63
INFO - INFO - Process finished Thread=IN -> C:\DOCUME~1\gfreire\
INFO - INFO - Process finished Thread=ERR -> C:\DOCUME~1\gfreire\
INFO - ************************
INFO - the executing return value is 0
INFO - INFO - null execution finished. Exitcode = 0
INFO - INFO - Transferring results file to Node rmi://10.0.0.60:1099/Node1992407500 and path results\lowlift.geo_18.3333333
```

The “Results” tab displays the interactive graphs, a progress bar and the gnuplot comparison graph (only when finished):



In order to control the interactive graphs:

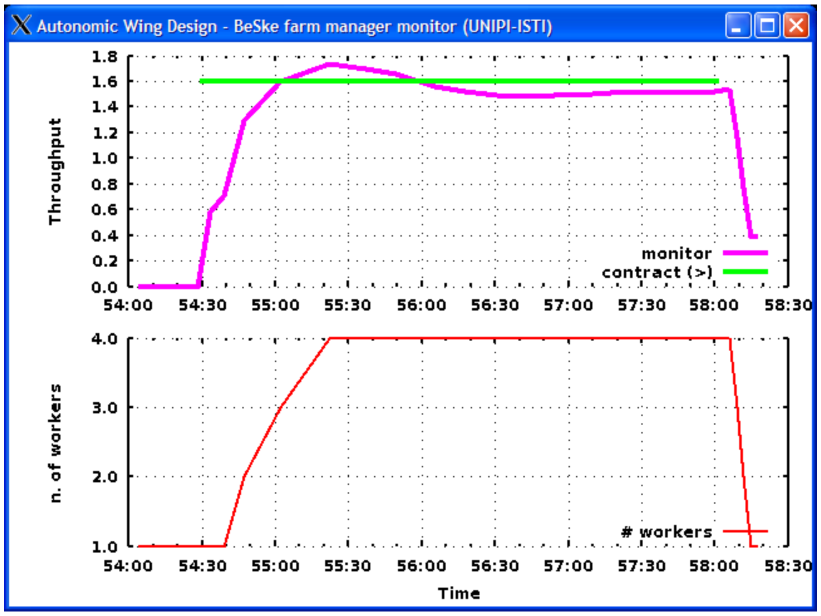
- Dragging the mouse will rotate the point of view
- Pressing caps while dragging the mouse up or down will zoom in or out.
- Pressing ctrl while dragging the mouse will move the point of view
- The controls at the bottom of each window will change the appearance of the corresponding graph.

5.2.2.1.1 Autonomic version

In order to test the autonomic version of the application, the “Use autonomic features” check box must be checked in the “Deployment” tab, and one of the specific deployment descriptors must be selected (at the time of this writing only a local deployment descriptor is offered). While running a request, a set of controls will be enabled in the “Options” tab:

- Throughput: the desired number of iterations per second. The autonomic manager will try to reach this throughput.
- Maximum parallelism degree: the maximum number of workers the autonomic manager can use for the distribution of the computations.

During the execution of the request, the user can have access to the monitoring of the MerakFarm, issuing the following command in a shell: `gnuplot farm_monitor.gp` (or `gnuplot farm_monitor_windows.gp` in windows). This will open a window displaying two different charts made with the monitoring information coming from the BeSke:



The meaning of the lines is the same than in the previous use case: the green line represents the throughput requirement; the magenta line is the current throughput; and the red line represents the current number of workers (MerakAutonomic components) bound to the MerakFarm.

In the image above, we can see how the manager tries to reach and keep up with the requested throughput but, as the limit of the parallelism degree is 4 (in this case), it fails.

5.2.3 Examples

The early prototype includes a few wing geometry files to be used for testing (under the geometries folder). The more files that are included or the more samples that are selected for the incidence angle or Reynolds number, the higher the amount of invocations to the legacy application. Changing the amount of iterations will also increase or decrease the time needed to accomplish each invocation (the higher, the longer). Default values should take a few minutes to complete for a local deployment, when one wing geometry is selected.

6 Conclusions

This section summarizes the impact of GridCOMP and the Grid Component Model (GCM) on the different use cases.

Biometric identification is known to be computational intensive and thus time consuming if applied to a large user population. With the BIS use case, a scalable distributed identification system has been built which is able to process identification requests in real-time, for instance, in a few seconds, while working on a very large user population.
Traditionally, such systems have been specifically developed to satisfy pre-defined QoS requirements. Consequently, system modifications were required whenever the QoS contract changed. The BIS prototype eliminates this implication, since it scales independently in accordance with the current QoS contract thanks to the autonomic reconfiguration functionality provided through the component framework.
Furthermore, traditional identification systems were usually built for a specific dedicated hardware infrastructure. The GridCOMP framework, however, with its strict separation of concerns and its advanced deployment infrastructure, allows building distributed applications independent of the target hardware infrastructure. As a result, the BIS prototype can be deployed on arbitrary and possibly heterogeneous hardware without changing a single line of application source code.
Finally, the BIS use case demonstrates that, with the help of the GridCOMP framework, development of Grid applications is no longer complex and time consuming. On the contrary, development time could be significantly reduced due to advanced features such as behavioural skeletons and the GIDE provided as part of the GridCOMP framework.

“Computing of DSO value” was an application which needed a long time to process and obtain the expected results. It is used to work out the mean time that the clients delay to pay an invoice to Atos.

Those are the main improvements after use Gridcomp Framework:

- The reduction of the execution time.
- The reduction of the complexity throughout the development phase.
- The development of a cheaper infrastructure and easily scalable application.

The first improvement that should be mentioned is the reduction of the execution time. It is important to know that for carrying through the operation a heavy PL/SQL process is executed which has a high calculation level. Moreover, if we take into account that the results the application calculates are processed based on the information stored in a central and very large database, and that those results should be accessed by several departments inside the company, we are going to realize that the data processing is quite high and will only increase over time. The old DSO application actually takes about four hours to compute around 6.600 clients. Some tests were performed on top of Grid5000 platform to measure the benefits using a grid solution. For this test, the master database was replicated to all nodes making each node independent to do their calculation. The calculation time using 25 nodes was almost 96% less than using only one node.

The second objective achieved is to reduce the complexity throughout the development phase. Therefore, we stress the importance of the components of the GridCOMP framework during this phase: GridCOMP provides some features, like the GIDE, that reduce the complexity when implementing the components of a Grid application. It is also important to highlight the significance of other advanced features such as behavioral skeletons, which saves the application the trouble of manage the nodes. Those features fulfill our objective.

The third aim of this use case consists of developing a cheaper infrastructure and easily scalable application. To maintain the old application, a new powerful server was needed. Using Grid technology, existing machines can be used reducing the infrastructure costs. What is more, using the deployment file, worker nodes can be added easily.

The Extended Data Record Processing application was implemented for very specific, and expensive, hardware. The first objective, when porting the application to GridCOMP, was to write a portable, multiplatform, application. Being a 100% Java solution, GridCOMP based applications can be run on almost any platform, opening the path to an affordable and easy scalability. Also, thanks to the XML-based deployment, the new application can be scaled up without changing its source code.
Using the Grid IDE, the new application benefited from a top-down design, taking advantage of the composition of components. Moreover, the NFCF provided the means to achieve autonomically the desired QoS using Behavioural Skeletons and JBoss rules based contracts.
While the original application was commercial and closed source, the new one is free and open.

The Wing Design use case consisted in bringing new life to an obsolete and hardly reusable legacy code application. This is a common situation in research environments, where there is multitude of programs developed using ancient programming languages, no longer maintained and poorly documented. Instead of rewriting them using a new language, which will need a big investment both in time and money, those programs can be wrapped inside components and reused by new (most probably distributed) applications, taking advantage of what it is already implemented and worked well for years.
In this case, an aerodynamic simulation program, written in FORTRAN 77, was used to show the benefits of this philosophy. Using the methods and techniques for wrapping the legacy code into a component (developed by the Tsinghua University), the resulting component was integrated into a new, GridCOMP-based demonstrator application, taking advantage of the distributed execution and the autonomic management of its performance. This resulted in reduced response times, thanks to the distribution of the computations, enabling also the scalability of the application, adding more (low cost) resources without the need to do any change in the code.

7 References

[1] T. Weigold, F. Tumiatti, E. Prunés, J. Santacatalina, G Freire. D.UC.03 Use cases description: preliminary architectural design and primitive prototypes. https://bscw.ercim.org/bscw/bscw.cgi/d315688/D.UC.03-final.pdf

[2] T. Weigold, F. Tumiatti, G Freire. D.UC.04.A Use cases: early documentation. https://bscw.ercim.org/bscw/bscw.cgi/d510892/D.UC.04.A_Final.pdf

[3] M. Aldinucci, S. Campa, P. Dazzi, N. Tonellotto, G. Zoppi. D.NFCF.04: NFCF prototype and early documentation. https://bscw.ercim.org/bscw/bscw.cgi/d510923/D.NFCF.04_final.pdf

[4] Pentaho Data Integration: http://kettle.pentaho.org/

[5] Java: http://java.sun.com/

[6] Apache Ant: http://ant.apache.org/

[7] ProActive 3.90: http://proactive.inria.fr/

[8] Visad: http://www.ssec.wisc.edu/~billh/visad.html

[9] Java3D: https://java3d.dev.java.net/

[10] Gnuplot: http://gnuplot.info/

[11] Cygwin: http://www.cygwin.com/

[12] gridcompwp5 project at INRIA's GForge: http://gforge.inria.fr/projects/gridcompwp5gs/

[13] D. Caromel, L. Du, Y. Wu, X. Wu, C. Dalmasso, G. Peretti Pezzi. D.CFI.04: Methods and techniques for legacy code wrapping as components. https://bscw.ercim.org/bscw/bscw.cgi/d510898/D.CFI.04_Final.pdf

[14] V. Getov, S. Isaiadis, A. Basukoski, J. Thiyagalingam. D.GIDE.03: Grid IDE Prototype and Early Documentation, EU GridCOMP Project, June, 2008. https://bscw.ercim.org/bscw/bscw.cgi/d510932/D.GIDE.03_Final.pdf