Project No. FP6-004265

CoreGRID

European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies

Network of Excellence

GRID-based Systems for solving complex problems

D.STE.05 – Design of the Integrated Toolkit with Supporting Mediator Components

Due date of deliverable: November 30, 2006
Actual submission date: February 7, 2007

Start date of project: 1 September 2004 Duration: 48 months

Organisation name of lead contractor for this deliverable:
Universitat Politècnica de Catalunya

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: Integrated Toolkit, Component Model, Tools/System Components

Contents

1 **Executive Summary** 2

2 **Introduction** 6

3 **Mediator Components Overview** 8

4 **Grid Component Model Overview** 10

5 **Overview of GRID superscalar** 12

6 **A GCM-based Design of the Integrated Toolkit** 13
- 6.1 Task Analyser 15
- 6.2 Task Scheduler 15
- 6.3 Job Manager 16
- 6.4 File Manager 17
  - 6.4.1 File Information Provider 18
  - 6.4.2 File Transfer Manager 18

7 **Support for Grid-unaware and Grid-aware Applications** 18
- 7.1 Grid-unaware Applications 19
- 7.2 Grid-aware Applications 19

8 **Interaction with External Entities** 20
- 8.1 Service and Resource Abstraction Layer 20
- 8.2 Application Controllers 21

9 **Efficient Component Composition** 21

10 **Design, Deployment and Implementation Details** 23

11 **Conclusions and Future Work** 24

1 Executive Summary

This deliverable presents the design of an Integrated Toolkit for Grid-enabling applications based on the Grid Component Model (GCM) [1]. The objective of the Integrated Toolkit is to provide a framework to enable the easy development of Grid-unaware applications. This Integrated Toolkit will rely on several functionalities offered by the mediator components, reducing the complexity of the application development. The current design is inspired by the behaviour and functionalities of the GRID superscalar framework [2], although other designs of Integrated Toolkits can be considered in the future, inside or outside the CoreGRID NoE. The objective of this work is to offer an environment which can automatically Grid-enable Java sequential applications.

While Task 7.3 of the Institute on Grid Systems, Tools and Environments has as its main objective the specification and design of an Integrated Toolkit that enables the development of Grid-unaware applications, Task 7.2 of the same Institute focuses on the design of a set of mediator components. By Grid-unware applications it is meant applications where the Grid is transparent to them but that are able to exploit its resources. The final objective of the Integrated Toolkit is to be able to run on the Grid any sequential application written for a generic platform. Furthermore, the application will be optimized by means of automatically exploiting the inherent application concurrency and efficient utilization of the available Grid resources.

The Integrated Toolkit is composed of a runtime part and bindings to different programming languages, graphical tools and portals.

The interface of the Integrated Toolkit has the following features:

- Supports Grid-unaware and Grid-aware applications
- Offers support for any programming language or graphical language
- Offers bindings for several interface languages

The Integrated Toolkit runtime offers the following features:

- The Grid remains transparent to the application: it performs all necessary actions to make this possible. The user is only required to select the tasks to be executed on the Grid and to use very few API methods (maximum 6-8).
- Automatic decision of what should be run on Grid resources.
- Performance optimization of the application (exploiting possible concurrency). Since the concurrency exploitation is at task level, the best suitable applications for the Integrated Toolkit are those with large granularity tasks.

- Task scheduling.
- Resource selection taking into account requirements and performance.
- Communication with mediator components for resource selection, file location,... through service and resource abstraction layer (SAGA[19] /GAT[20],...).

The Integrated Toolkit will rely on the Mediator Components which will offer system-component capabilities, achieving both steering and performance adaptation. This set of components will allow to integrate resources and services in the Grid into one overall system with homogeneous component interfaces, and the advantage that this abstracts from the many software architectures and technologies used underneath. The mediator components considered since now are:

- Application-level information cache, which provides a unified interface to deliver all kinds of meta-data
- Application steering and tuning components, which will enable the users to control and steer the applications
- Application manager controller, which will enable to track an application from submission till completion.

Furthermore, the Integrated Toolkit will also rely on a Service and Resource abstraction Layer that abstracts from the underlying Grid middleware.

The design of the Integrated Toolkit presented in this deliverable is a GCM-based one. GCM is the component model adapted for the Grid and based on the Fractal specification that the CoreGRID Programming Model Institute has proposed.

In the design presented in this report the Integrated Toolkit runtime is defined as a set of *Fractal Components*. Each of these components is in charge of a given functionality among those intended for the Integrated Toolkit. Communication between components is obtained through *bound client and server interfaces*, which emit operation invocations and accept them, respectively.

To Grid-enable Java applications, the Integrated Toolkit runtime will need a specific Java binding which will be responsible for automatically inserting the corresponding calls to the API offered by the Integrated Toolkit.

The API will offer a small number of methods for initialization, task execution, file synchronization and finalization (no more are needed).

The overall Integrated Toolkit runtime is a composite mega-component that serves as a unique point of deployment and invocation. It also acts as container of several other subcomponents which implement the main functionalities of the system. These subcomponents are:

- *Task Analyser*: receives incoming tasks and detects their precedence. This component manages a task dependency graph. It implements the Integrated Toolkit interface used by the application to submit tasks. The component, when receiving a task execution request, looks for data dependencies between the new task and previous tasks. This data-dependency analysis is done with the help of the File Manager component. This component is also responsible for submiting tasks for execution to the Task Scheduler component when tasks have their dependencies solved.

- *Task Scheduler*: decides where to execute tasks whose dependencies are already solved. It receives the requests from the Task Analyser component, and taking into account information about the Grid resources and information, information about where the data required for the task is located and the user constraints, it decides where and when to execute a task. Through the dynamic and reconfigurable features of GCM the scheduling algorithm would be sensible of being dynamically changed.

- *Job Manager*: submits and monitors the remote execution of jobs, and also requests the necessary file transfers. This is the component which actually submits the tasks for execution to the Grid resources. It delegates the file transfers to the File Manager and submits the job for execution through the Service and Resource Abstraction Layer. It monitors the job execution and completion, and offer fault-tolerance features.

- *File Manager*: gathers all information related with files (accesses, location, versions) and performs file transfers on demand. The File Manager is a composite component, composed of the File Information Provider and the File Transfer Manager. The File Information Provider is able to provide information about the location of the files, both for logical and physical files. This information can be stored in the same File Information Provider or obtained by this component through external information services. Besides, it is able to provide information about the last write access to files, which is used by the Task Analyser for the dependency detection. The File Transfer Manager is the component that actually transfers the files from one host to another and is also responsible for giving information about the new file locations to the File Information Provider.

In addition to providing support to Grid-unaware applications, the subcomponents of the Integrated Toolkit will offer an alternative way for developing Grid-aware applications. In this case, componentised applications will be able to deploy the Integrated Toolkit components and bind their

application components to the interfaces offered by them. For example, a given application can deploy the Task Scheduler subcomponent that will offer scheduling functionality to the application. This deliverable also arguments how efficient component composition can be used to balance efficiency and componentisation.

Currently, the work done for the Integrated Toolkit project comprises the design of the tool internal structure, the definition of each subcomponent role and the specification of the bindings between different subcomponents. Forthcoming phases of this project will refine the presented design and implement it using ProActive, in order to offer a first functional prototype of the Integrated Toolkit.

2 Introduction

This document presents a design of the Integrated Toolkit based on GCM. According to the CoreGRID JPA, the integrated toolkit will:

- Provide means for simplifying the development of Grid applications
- Allow to execute the applications on the Grid in a transparent way
- Optimize the performance of the application

This Integrated Toolkit is integrated in the STE generic component platform, as it is shown in Figure 1, and builds on top of the mediator components (application manager, tuning component, ...) and on top of the service and resource abstraction layer.

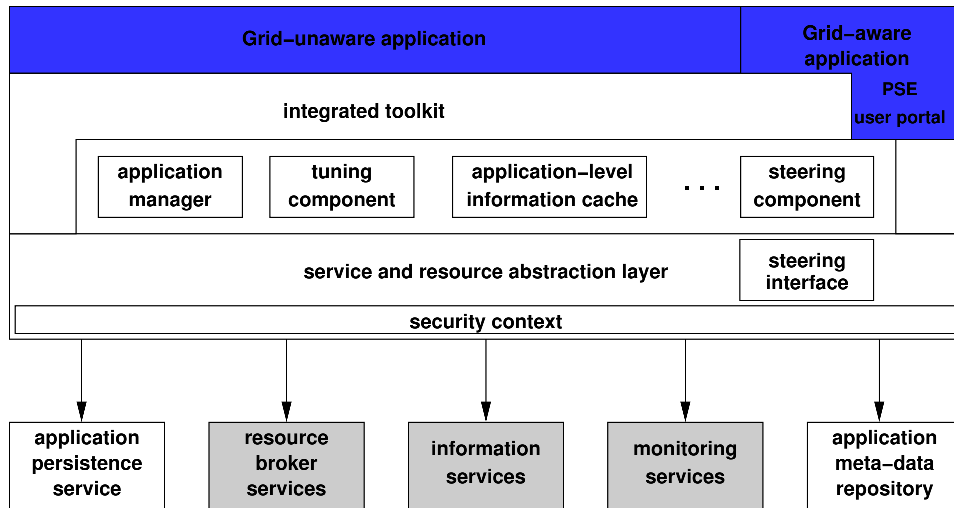

Figure 1: Envisioned generic component platform

Task 7.3 of CoreGRID STE Institute (WP7) has as objective the specification and development of such an Integrated Toolkit. This document drafts a specification of the Integrated Toolkit, the impact of GCM on the integrated toolkit and the definition of the interface with the mediator components.

Figure 2 shows an updated diagram of the Integrated Toolkit as has been envisaged in the 2nd STE roadmap [3]. In this vision, the Integrated Toolkit is composed of a run-time and some bindings to different programming languages, graphical tools and portals.

The interface of the Integrated Toolkit has the following features:

- Supports Grid-unaware and Grid-aware applications
- Offers support for any programming language or graphical language

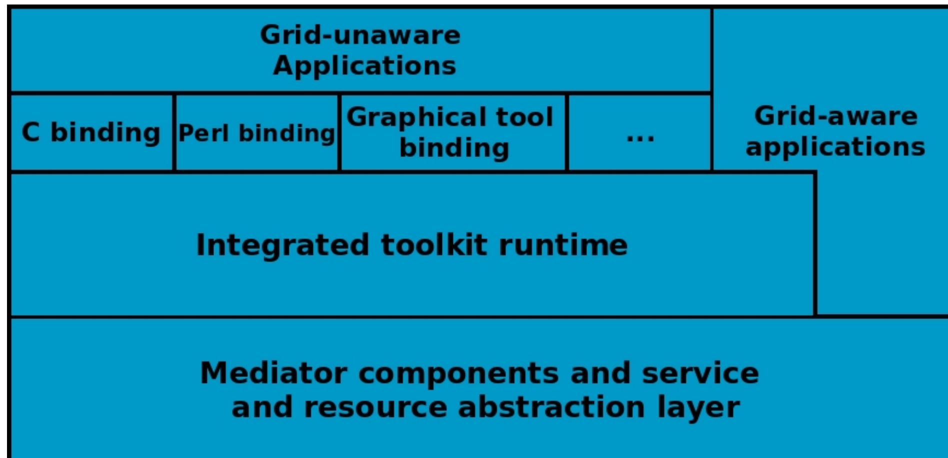

Figure 2: Integrated Toolkit

- Offers bindings for several interface languages

By Grid-unaware it is meant applications where the Grid is transparent to them but that are able to exploit its resources. The final objective of the Integrated Toolkit is to be able to run on the Grid any sequential application written for a generic (non-Grid) platform. The bindings are interfaces provided for the different input languages that the Integrated Toolkit can support. In this way, a single implementation of the Integrated Toolkit can support applications in Java, C/C++, Perl, etc. We include here graphical languages, considering those as the ones resulting from programming frameworks where the user can *program* its applications with a Graphical Interface. Examples of such type of environments can be the P-GRADE Portal [4], Triana [5] or SEGL [6].

Besides, the runtime offers the following features:

- The Grid remains transparent to the application: it performs all necessary actions to make this possible. The user is only required to select the tasks to be executed on the Grid and to use very few API methods (maximum 6-8).
- Automatic decision of what should be run on Grid resources.
- Performance optimization of the application (exploiting possible concurrency). Since the concurrency exploitation is at task level, the best suitable applications for the Integrated Toolkit are those with large granularity tasks.
- Task scheduling.
- Resource selection taking into account requirements and performance.

- Communication with mediator components for resource selection, file location,... through service and resource abstraction layer (SAGA[19] /GAT[20],...).

This document presents the specification and design for such an Integrated Toolkit. These design and implementation are based on the ProActive [16] implementation of GCM and on the GRID superscalar framework ideas.

The structure of the document is as follows: section 3 presents an overview of the mediator components and how they interact with the Integrated Toolkit and with the service and resource abstraction layer. Section 4 presents an overview of the CoreGRID Grid Component Model and section 5 shows an overview of GRID superscalar and which functionalities are taken into account for the design described in this document. Section 6 presents the design of the Integrated Toolkit. Section 7 presents how Grid-unaware and Grid-aware applications will be supported by the Integrated Toolkit and section 8 describes the relation between the Integrated Toolkit and the underlying layers. Section 9 presents how component composition can be handled. Section 10 gives some design and deployment details and finally section 11 presents some conclusions and future work.

3 Mediator Components Overview

The goal of the mediator component toolkit is to integrate system-component capabilities into application code, achieving both steering of the application and performance adaptation by the application to achieve the most efficient execution on the available resources offered by the Grid.

By introducing such a set of components, resources and services in the Grid get integrated into one overall system with homogeneous component interfaces. The advantage of such a component system is that it abstracts from the many software architectures and technologies used underneath.

The strength of such a component-based approach is that it provides a homogeneous set of well-defined (component-level) interfaces to and between all software systems in a Grid platform, ranging from portals and applications, via mediator components to the underlying system software. The set of envisioned mediator components, with their embedding in the generic component platform, can be seen in Figure 1; a detailed description can be found in [9]. We briefly summarize the mediator components in the following.

Application-level information cache

This component is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a GIS, a monitoring system, from application-level meta data) to the application. Its purpose is twofold.

First, it is supposed to provide a unifying component interface to all data (independent of its actual storage), including mechanisms for service and information discovery. Second, this application-level cache is supposed to deliver the information really fast, cutting down access times of current implementations like Globus GIS (up to multiple seconds) to the order of a method invocation. For the latter purpose, this component may have to prefetch (poll) information from the various sources to provide them to the application in time. An implementation of such a component, albeit without a "real," e.g. GCM, component interface, has been described in [10].

Application steering and tuning components
Controlling and steering of applications by the user, e.g. via application managers, user portals, and PSE's, requires a component-level interface to give external components access to the application. Besides the steering interface, also dedicated steering components will be necessary, both for mediating between application and system components, but also for implementing pro-active steering systems, carrying their own threads of activity. These components will have to interface with Grid monitoring services and Grid resource brokers. This way, the steering components will provide a framework for performance tuning, which can be used to improve the execution time of applications automatically as well as for improving services and tools which are involved in the environment. Steering and tuning components are supposed to utilize both the application-level meta data repository and the application-level information cache components.

Application manager component
The envisioned application manager component establishes a pro-active user interface, in charge of tracking an application from submission to successful completion. Such an application manager will be in charge of guaranteeing such successful completion in spite of temporary error conditions or performance limitations. (In other settings, such an active component might be referred to as an agent.) For performing its task, the application manager will need to interoperate with most of the other mediator components, like the application itself (via the steering interface), the application meta data repository and cache, as well as an application persistence service, like the one published in [14].

The set of mediator components as described so far denotes the current research scope. In the future, this set may be refined and enriched as new experience will be gained, and as practical experience with implemented components will be gained.

4 Grid Component Model Overview

GCM is the component model adapted for the Grid that the Programming Model Virtual Institute proposes, taking Fractal [18] as the reference specification. As defined in the Institute, GCM can be summarized as follows.

First, GCM is a **hierarchical** component model. This means users of GCM (programmers) have the possibility of programming GCM components as compositions of existing GCM components. The new, composite components programmed in this way are first class components, in that they can be used in every context where non-composite, elementary components can be used, and programmers need not necessarily perceive these components as composite, unless they explicitly want to consider this feature. This property is already present in existing component models. In particular, the Fractal component model assumes components can be hierarchically composed, and this is one of the reasons that led to the Fractal component model being chosen as the reference model upon which GCM would be based.

GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RCP-like" use/provide (or client/server) ports, GCM allows **data**, **stream** and **event ports** to be used in component interaction. Data ports allow data sharing mechanisms to be implemented. Using data ports, components can express data sharing between components while preserving the ability to properly perform ad hoc optimization of the interaction among components sharing data. Stream ports allow one way data flow communications among components to be implemented. While stream ports can be easily emulated by classical use/provide ports, their explicit inclusion allows much more effective optimizations to be performed in the component run-time support (framework). Event ports may be used to provide asynchronous interaction capabilities to the component framework. Events can be subscribed and generated. Furthermore, events can be used just to synchronize components as well as to synchronize *and* to exchange data while the synchronization takes place.

As regards collective interaction patterns, GCM supports several kinds of collective ports, including those supporting implementation of structured interaction between a single use port and multiple provide ports (multicast collective) and between multiple use ports and a single provide port (gathercast collective). The two parametric (and therefore customizable) interaction mechanisms allow implementation of most (hopefully all) of the interesting collective interaction patterns deriving from the usage of composite (parallel) components. The current definition of GCM does not exclude the possibility of having further collective interaction patterns in the future, should the ones included in the current definition turn out to be insufficient to support commonly used grid component patterns.

GCM is intended to be used in grid contexts, that is in highly dynamic,

heterogeneous and networked target architectures. GCM therefore provides several levels of **autonomic managers** in components, that take care of the **non-functional** features of the component programs. GCM components have thus two kind of interfaces: a functional one and a non-functional one. The functional interface includes all those ports contributing to the implementation of the functional features of the component, i.e. those feature directly contributing to the computation of the result expected of the component. The non-functional interface comprises all those ports needed to support the component manager activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in the achievement of the expected (functional) results but not directly involved in actual result computation. Each GCM component therefore contains one or more managers, interacting with other managers in other components via the component's non-functional interfaces and with the managers of the internal components of the same component using the mechanism provided by the GCM component implementation. Each component has a manager whose job it is to ensure efficient execution of the component on the target grid architecture.

The GCM component program architecture is described using a proper **ADL** (Architecture Description Language) that decouples functional program development from the underlying tasks needed to deploy, run and control the components on the component framework. In GCM, the ADL is mostly inherited from the Fractal ADL.

Last but not least, the GCM component model supports **interoperability** at several levels. First, interoperability is guaranteed in terms of the ability to support several grid middle-ware environments as possible platforms on which to implement GCM and, in particular, to host the GCM framework. Second, interoperability is guaranteed by the possibility of wrapping GCM components into standard Web Services, in such a way that the WS framework can benefit from the "services" provided by the GCM framework (passive WS-GCM interoperability). Third, GCM components are allowed to invoke standard Web Service services during their execution (active WS-GCM interoperability). The current definition of GCM does not prevent the extension of the interoperability features to other frameworks in the future.

GCM supports the features mentioned above according to several compliance levels, in order to allow easy transition to GCM from other existing component frameworks. Lower compliance levels accommodate components that do not support all the features required by the GCM model, but at least can be identified as GCM components with limited support for the GCM features. High compliance levels host full-featured GCM components.

The above-mentioned features allow GCM to be viewed in the light of other component models currently available. For example, GCM can be characterized as CCA plus hierarchical composition, advanced communication patterns and autonomic control; or again, it can be regarded as Fractal

plus autonomic control together with advanced communication patterns.

5 Overview of GRID superscalar

GRID superscalar is a programming framework for Grid-enabling applications, composed of an interface, a run-time, an automatic deployment graphical tool (the *Deployment Center*) and a runtime monitor. With GRID superscalar a sequential application composed of tasks of a certain granularity is automatically converted into a parallel application where the tasks are executed in different servers of a computational GRID.

In the current version, the user should manually select which parts of the application (functions or routines of the application, from now on called tasks) are going to be executed on the Grid. This is done by the declaration of the task interface in an Corba-like IDL file or with pragma annotations in the code. This depends on the version of GRID superscalar that is used: for the Corba-like IDL case is the generation-code based version, or version 1; and for the pragma annotations based version is the source-to-source compiler version, or version 2. We foresee that in the future this selection of which application parts should run on the Grid and which not, will be done automatically.

In the case of version 1, the application developer have to add a few (very few) calls to the GRID superscalar API for initialization, file management and finalization, but the program remains basically unchanged. In any case, these calls are Grid-unaware calls, therefore the code remains "unaware" of the underlying Grid. For version 2, the code remains unchanged.

In addition, several tools are provided (code-generation, source-to-source compiler, Deployment Center and scripts) that allow to very easily generate the binaries of the application in each of the Grid resources. Again, the Grid remains very transparent to the application developer.

The behaviour of the application when run with GRID superscalar is the following: for each task candidate to be run on the Grid (listed in the IDL or annotated with a pragma), the GRID superscalar run-time inserts a node in a task graph when it is invoked. Then, the GRID superscalar run-time system seeks for data dependencies between the new task and all previous ones. If a task does not have any dependency with previous tasks which have not been finished, it can be submitted for execution to the Grid. In that case, the GRID superscalar run-time requests a Grid server to the broker and if a server is provided, it submits the task. Those tasks that do not have any data dependency between them can be run on parallel on the Grid. This process is automatically controlled by the GRID superscalar runtime, without any additional effort for the user. GRID superscalar is notified when a task finishes. After that, the data structures are updated and any task which now has its data dependencies resolved, can be submitted for

execution.

Besides, GRID superscalar takes care of all the data transfers required between the Grid resources. It is able to do data renaming, which is a technique that enables to further increase the application parallelism. Other features of the GRID superscalar runtime are: data-locality exploitation, to reduce the data transfers; checkpointing at task level; exception handling; fault-tolerance; support for NFS-like file systems; threaded execution in the local node, for version 2 only; and automatic resource selection based on user-constraints.

Additionally, the evolution of the execution at task level can be monitored with a graphical interface that depicts the task dependency graph and the state of the tasks at each moment: waiting, ready for execution, running and finished (and in which resource was executed).

6 A GCM-based Design of the Integrated Toolkit

The main purpose of the Integrated Toolkit is offering an environment to automatically Grid-enable Java applications. We assume as input a Java application where the methods to be executed on the Grid are either selected manually by the user or automatically through a Java binding. In any case, a Java binding should exist to automatically insert the corresponding calls to the API offered by the Integrated Toolkit.

The API will offer methods for initialization, execution, file synchronization and finalization (no more than 6-8 methods). The Integrated Toolkit will provide the features listed in section 2. For reasons of simplicity, we have limited the data dependency analysis to those parameters that are files. However, this limitation will be overcome in forthcoming versions by also controlling the accesses to scalars, arrays, etc.

Since the design the Integrated Toolkit is being performed simultaneously with the one of GCM, only some innovative characteristics of this model have been so far considered, concretely the following ones:

- Hierarchical composition.
- Separation between functional interfaces, used to access the implementation of the functionalities that the component offers, and non-functional interfaces, provided by default or custom controllers (see section 8).
- Synchronous and asynchronous communications.
- Collective interactions between components: multicast and gathercast communications will be available in the next version of ProActive, and the Integrated Toolkit will benefit of them.

- ADL-based description of the Integrated Toolkit component architecture (see section 10).

Concerning its structure, the Integrated Toolkit is formed by a set of *Fractal components*. Each of these components is in charge of a given functionality among those intended for the Integrated Toolkit, aiming to follow the separation of concerns requirement. Communication between components is obtained through *bound client and server interfaces*, which emit operation invocations and accept them, respectively.

Figure 3 shows an overall vision of the Integrated Toolkit, a composite mega-component that serves as a unique point of deployment and invocation. It also acts as a container of several other subcomponents which implement the main functionalities of the system. These subcomponents are:

- *Task Analyser*: receives incoming tasks and detects their precedence.
- *Task Scheduler*: decides where to execute tasks whose dependencies are already solved.
- *Job Manager*: submits and monitors the remote execution of jobs, and also requests the necessary file transfers.
- *File Manager*: gathers all information related with files (accesses, location, versions) and performs file transfers on demand.

Next subsections explain these subcomponents in a more detailed way.

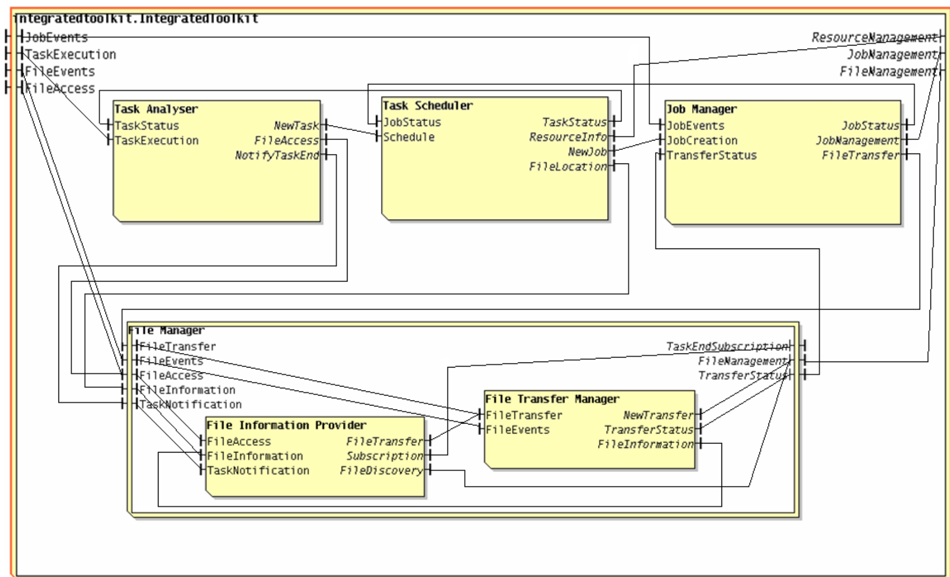

Figure 3: Global structure of the Integrated Toolkit

6.1 Task Analyser

The Task Analyser component manages the *task dependency graph*, which represents all dependencies between tasks. It implements the Integrated Toolkit interface used by the application to submit tasks.

The Task Analyser is contacted when one of these two events occurs:

- The application requests, through the Integrated Toolkit API, the execution of a new task. In this case, the Task Analyser searches for dependencies between the current task and all previous ones. Concretely, for each file that the current task accesses, the Task Analyser asks the File Manager which is the last task that has written the file. If any dependency is discovered, it adds the corresponding edge/s to the task dependency graph; otherwise, it informs the Task Scheduler that the task can be executed.

- The Task Scheduler notifies that a task has been successfully finished. When this happens, the Task Analyser removes the finished task from the task dependency graph and checks whether any data dependency has been solved. If this is the case, it communicates to the Task Scheduler the new group of dependency-free tasks.

In addition, the Task Analyser can implement a *checkpointing* mechanism: it can keep track of which tasks have successfully finished and, if the application fails, the Task Analyser can resume it from the last checkpoint.

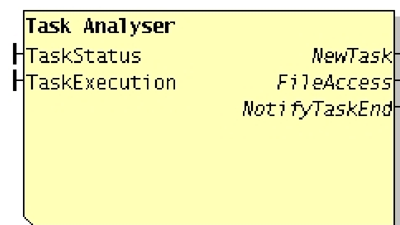

Figure 4: Task Analyser component

6.2 Task Scheduler

The Task Scheduler component is responsible for deciding where to execute tasks, according to a certain *scheduling algorithm*. Moreover, it should be able to switch between several scheduling algorithms on demand. A simple scheduling strategy, based in a FIFO matching, could be employed in a first version of this component. However, more complex algorithms could be added later on; for instance, the Scheduler could try to minimize the following function for each task:

$$f(t, r) = \alpha FT(r) + \beta ET(t, r)$$

where t represents a task and r a computational resource, FT is an estimation of the transfer time of all files needed by t to r and ET is an estimation of the execution time of t on r (ET estimations are calculated using a user-provided cost function).

This algorithm favours the exploitation of the locality of the files: as a trivial example, for a given task that uses a large input file, a slower computational resource can be selected if the file is already located in this resource.

In order to make a decision, the Task Scheduler uses four information sources:

- The Task Analyser: tells which tasks have no dependencies and the files they access.
- An external Resource Broker/Locator: allows to discover available computing and storage resources.
- The File Manager: provides the location of each file used by a task.
- User constraints: users should be able to impose requirements about the execution environment, like the desired operating system or the software which must be available in the target machine.

With all this information and following its scheduling strategy, the Task Scheduler decides which tasks to execute, the resource where each task will be executed and the locations that needed files should be brought from.

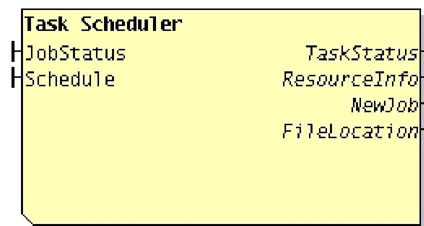

Figure 5: Task Scheduler component

6.3 Job Manager

The Job Manager component is in charge of *job submission and monitoring*. Once the Task Scheduler has mapped tasks to resources, it requests the Job Manager to prepare the associated jobs for execution.

One task (an application method and its parameters) is usually transformed into one job, which represents the submission of the task to the Grid;

however, several tasks could be grouped to form one job and submitted as a whole.

In particular, the Job Manager follows these steps for each job:

- It checks which files need to be sent, delegates this work to the File Manager and waits for the latter to notify the end of the transfers.
- It submits the job to its assigned resource, and controls its state transitions and its proper completion.
- It orders the File Manager to transfer back the output files generated if necessary, and informs the Job Scheduler about the end of the job.

Besides, the Job Manager could also implement some kind of fault tolerance mechanism. For example, when there is a given time since a job has been submitted and no response has yet been received for it, the Job Manager could assume that it has failed and resubmit it, either to the same host as the first time or to another one. If the Job Manager decided to change the destination of the job, it could delegate the selection of this new host to the Task Scheduler.

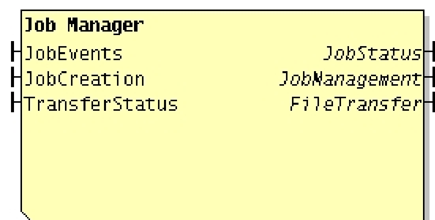

Figure 6: Job Manager component

6.4 File Manager

In this design of the Integrated Toolkit, we have included a component that takes care of all the operations where files are involved, called File Manager. One more general conception of the Integrated Toolkit would probably propose a Data Manager instead, in order to support different data types (such as scalars, arrays, etc). Although we will take into account this idea in future versions of the Integrated Toolkit, we have decided to begin with a simpler one.

Our File Manager is a composite component which encompasses two inner primitive components, each one having its own concern: the *File Information Provider* and the *File Transfer Manager*. Next we present their features.

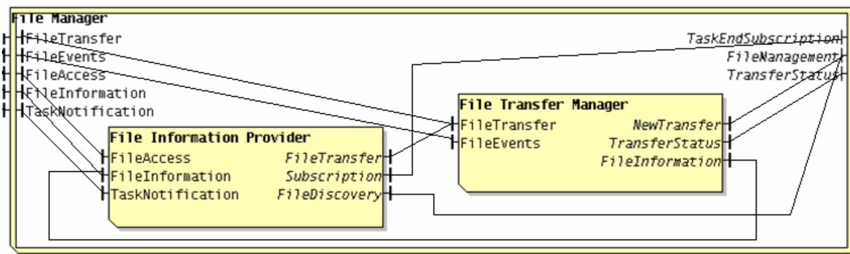

Figure 7: File Manager component

6.4.1 File Information Provider

The File Information Provider subcomponent stores all the information related to files, more precisely these two kinds:

- *File Location*: the File Information Provider performs file location and file version management: it controls the initial input files of the application, the temporary intermediate versions that are generated by tasks and also the final result files. The application can access local or remote files, and use physical or logical names to refer to them. The File Information Provider could use an external source (like a Replica Manager) to discover the unknown physical location of a file, and since that moment it could manage the location of this file itself.

- *Registry of the last write access* that each file suffers and which task does it. This registry will be useful for the Task Analyser to detect new data dependencies between tasks. Optionally, file renaming could be implemented in order to avoid false dependencies, and therefore to increase parallelism.

6.4.2 File Transfer Manager

The File Transfer Manager component has two main purposes: first, sending to the execution host the input files required by a job; second, transferring back the output files of a job at the end of execution when necessary. Besides, it informs the File Information Provider about the new location of a file when it performs a transfer.

7 Support for Grid-unaware and Grid-aware Applications

The main purpose of the Integrated Toolkit is to ease the development of Grid-unaware applications, where the Grid (resources, services, middleware, etc.) is transparent to the programmer. However, we also target Grid-aware

applications, since we think they could benefit from our tool, too. The way these different types of applications could interact with the Integrated Toolkit is explained in the following subsections.

7.1 Grid-unaware Applications

There is a large experience from the work developed at UPC with GRID superscalar in enabling the development of Grid-unaware applications. This experience will be taken into account when developing the Integrated Toolkit.

As said, a Grid-unaware application has no prior knowledge about the Grid, and it is usually programmed in a sequential fashion. Nevertheless, by means of the Integrated Toolkit it can be run on the Grid and exploit its resources, while increasing the performance if possible; to achieve that, our tool should be able to identify the tasks that compose the application, detect task precedence, decide which tasks should be submitted to the Grid and control their remote execution.

In order to use the Integrated Toolkit, the application programmer is only required to specify the interface of the tasks which should be run on the Grid, for instance through an XML document which includes all method names, their parameters and the type and direction of the latter. A source-to-source compiler could use this XML document and the original main program to generate a new code, responsible of making the deployment and the subsequent invocation of the Integrated Toolkit through an interface, thus hiding the manipulation of components from the programmer.

7.2 Grid-aware Applications

Nowadays, there exist several middleware tools which cover some basic services a developer must deal with when programming Grid-aware applications. Programmers may use such tools to gather the available resources, submit jobs and stage data to them, monitor computations and so on. Some examples of these middleware systems are Globus [21], UNICORE [22] and gLite [23]. Besides, in order to link all these middleware packages and grid applications through a simple standard API, the SAGA project [19] is in progress.

Despite all these already existing solutions, we believe that the Integrated Toolkit can also be an alternative to develop Grid-aware applications. It is the case of a componentised application, which will be able to deploy the Integrated Toolkit mega-component and bind its application components to the interfaces our tool implements.

Furthermore, if the programmer is only interested in the functionalities that certain subcomponents of the Integrated Toolkit offer, he/she will be free to deploy solely specific subcomponents and perform the proper bindings to his/her own application components. Think, for example, in an

application which uses an API like SAGA to perform task submission, file staging and resource location, but lacks a scheduling functionality; in such a situation, the programmer could choose to deploy only the Task Scheduler subcomponent. Similarly, if an application needs a component which keeps track of file location and performs file transfers on demand, the File Manager subcomponent could attend it.

All these previously described features could also be applied to non-componentised applications, since they could also instantiate either the Integrated Toolkit or its subcomponents separately, the only requirement for the programmer to carry out is knowing how to deal with component deployment and invocation.

8 Interaction with External Entities

Besides the application, the Integrated Toolkit also interacts with other entities situated in lower layers, which provide an interface for it to invoke some basic Grid functionalities and a steering interface for users to control and modify it. Figure 8 shows this overall generic component platform.

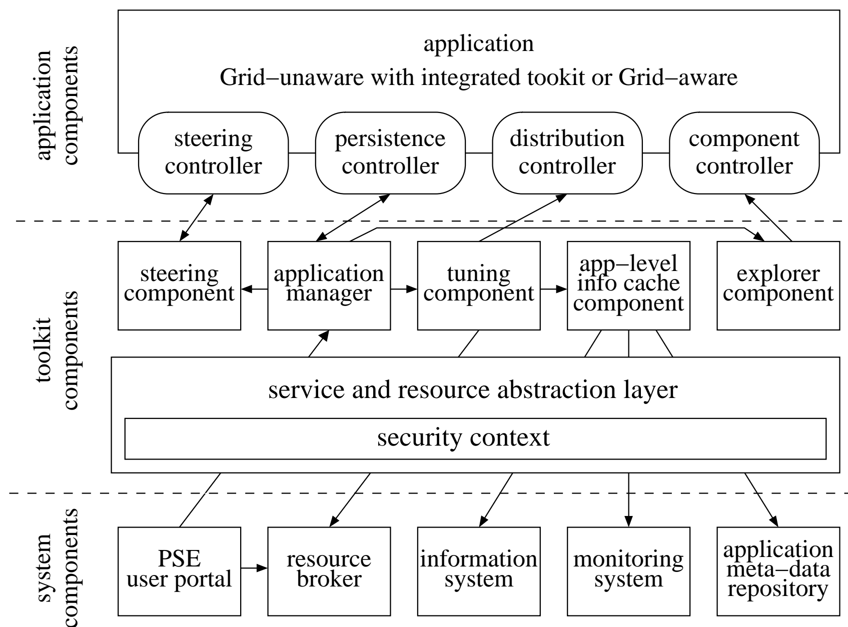

Figure 8: Generic component platform

8.1 Service and Resource Abstraction Layer

The Integrated Toolkit distributes the jobs over the Grid through the *Service and Resource Abstraction Layer* API, which offers methods to:

- Submit jobs to target hosts for execution.
- Monitor the status of submitted jobs.
- Move files between different storage devices.
- Discover the available computing and storage resources, as well as the location of files.

The Service and Resource Abstraction Layer should provide the Integrated Toolkit with a uniform interface to numerous types of Grid middleware, like the already mentioned Globus [21], UNICORE [22] and gLite [23].

8.2 Application Controllers

The behaviour of the Integrated Toolkit can be modified using several *Application Controllers*, which interact with Mediator Components. These controllers could be either a part of the Integrated Toolkit membrane or usual components. Their proposed possible roles are:

- *Steering*: to modify certain parameters which affect the behaviour of the Integrated Toolkit. The Steering Controller could, for example, change the scheduling algorithm used by the Task Scheduler to match tasks with resources, or to specify whether the Job Manager should implement a fault tolerance feature when a job is likely to have failed.
- *Persistence*: to manage a checkpointing and fault recovery mechanism. In a given moment, the Persistence Controller could order the Task Analyser to perform a snapshot of the current application, in other words, to store which tasks have successfully finished. This information could be used later, in case of an application or system failure, to resume the execution from the snapshot.
- *Distribution*: to inform of changes in resource state. For instance, the Distribution Controller could tell the Task Scheduler that a specific resource is no longer available.
- *Component*: to change the overall component structure of the Integrated Toolkit. One example could be replacing the File Information Provider by another component which implements file renaming.

9 Efficient Component Composition

Component-based programming aims at producing higher quality software, increasing the re-use of components and permitting late composition. In the context of component-based programming, applications are treated as

compositions of components. Given an application composition, some of the components might have also been developed outside the context of the application or its domain. For example, a generic mathematical component primarily developed for a financial application may be used inside a computational fluid dynamics application. Such a composition of non-domain-specific components challenges the overall efficiency of the application development process. As a result, the overall efficiency of the composition, in terms of cost and performance, becomes non-deterministic — may not be guaranteed to be efficient enough, even if the individual components have been proven to be efficient. In other words, in such cases two primary goals of software practice, efficiency and quality, may conflict with each other. Partly, this is due to the fact that components are developed to be separately deployable and little or no attention is paid about the context of use or possible future compositions, which is impractical.

By the introduction of efficient component composition [13] one could argue that, this problem can partly be overcome by paying more attention to component-specific information, for example component metadata, during composition. More specifically, efficient component composition involves a set of collective properties: optimal performance, optimal cost and optimal resource utilisation. The main approach of efficient component composition relies on the description of possible means for extracting and organising the metadata and formats for specifying the metadata. This scheme is independent of component- and programming-models and extensible. It can be seen as a precursor to a possible runtime scheme, where we intend to facilitate extraction, maintenance and usage of component metadata at runtime.

Efficient component composition aims to seek a balance between these two conflicting goals of software engineering: efficiency and componentisation. In order to do that we need two different types of information, firstly detailed information about components and secondly the current context information. If each component is augmented with additional data, which is called metadata in the literature [12, 16, 11], describing the characteristics and functional behaviours of the component such that they are accessible outside the component's boundaries, then the composition can be adjusted accordingly. However, behaviour of some of the components may be subject to the current context of use, for instance the underlying architecture. This forces us to consider context information during composition, which at some point may become part of the metadata. If components in a composition or the application developer do have access to the metadata, efficient component composition is about appropriately using, maintaining and staging the metadata and context information of components.

10 Design, Deployment and Implementation Details

The component design proposed in this paper derives from a previous work presented in [15], which shows the overall structure of GRID superscalar. Taking this architectural description as a starting point, so far we have performed the two first steps of the componentisation process described in [8], which corresponds to the definition and refinement of the Integrated Toolkit components and their interfaces. Finally, we have also considered the componentisation example shown in [7] for the SEGL Runtime Architecture.

Moreover, in order to ease the design and later deployment of the Integrated Toolkit, we have used the Fractal GUI provided by the IC2D ProActive tool. This graphical interface permits to:

- Create primitive and composite components.
- Define their client and server interfaces and the bindings between them.
- Specify the Java classes associated with each component implementation and interfaces.

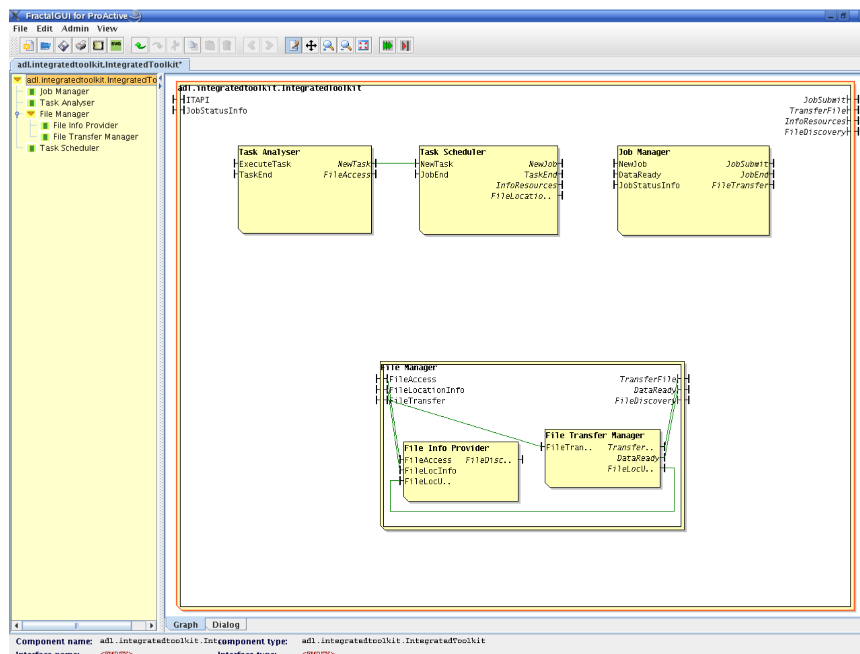

Figure 9: ProActive Fractal GUI

In addition, the Fractal GUI allows to save the overall graphical structure into XML files, which correspond to the ADL component definition.

This simplifies notably the later deployment of the Integrated Toolkit, because the ProActive library provides operations to instantiate components using their ADL definition. It is also possible to define components programmatically with ProActive, but such method is more tedious and implies reprogramming each time the overall component structure is changed.

11 Conclusions and Future Work

Since the Integrated Toolkit project is still in an early stage, so far our work comprises the design of the tool internal structure, the definition of each subcomponent role and the specification of the bindings between different subcomponents. The next steps to follow are:

- Refine, if necessary, the server interfaces and their interaction with the client ones.
- Define more precisely the interaction between the Integrated Toolkit, the Service and Resource Abstraction Layer and the Application Controllers.
- Develop the Integrated Toolkit in a progressive way, implementing a component and then testing its correct behaviour at each step.

References

[1] Proposals for a Grid Component Model, CoreGRID Deliverable D.PM.02, 2006.

[2] R. M. Badia, Jesús Labarta, Raúl Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima. *Programming Grid Applications with GRID superscalar* . Journal of GRID Computing, Vol. 1 Issue 2. Pages: 151-170 , June 2003.

[3] Roadmap version 2 on Grid Systems, Tools, and Environments, CoreGRID Deliverable D.STE.04, 2006.

[4] G. Sipos and P. Kacsuk. *Classification and Implementations of Workflow-Oriented Grid Portals*, Proc. of HPCC-2005 Conference, 2005.

[5] I. Taylor, M. Shields, I. Wang, R. Philp, *Distributed P2P Computing within Triana: A Galaxy Visualization Test Case*, in proceedings of IPDPS 2003, 22-26 April 2003.

[6] Natalia Currle-Linde, Uwe Kuester , Michael M. Resch , Benedetto Risio, *Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems*, In proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005), pp 49-56, Malaga, Spain.

[7] H. L. Bouziane, C. Pérez, N. Currle-Linde, M. Resch. *A Software Component-based Description of the SEGL Runtime Architecture*, CoreGRID Technical Report, Number TR-0054, July 2006

[8] F. Baude, D. Caromel, F. Huet, V. Getov, M. Morel, N. Parlavantzas *Componentising a Scientific Application for the Grid*, CoreGRID Tehcnical Report TR-0031, April 2006

[9] Proposal for mediator component toolkit, CoreGRID Deliverable D.ETS.02, 2005.

[10] G. Aloisio, Z. Balaton, P. Boon, M. Cafaro, I. Epicoco, G. Gombas, P. Kacsuk, T. Kielmann, and D. Lezzi. *Integrating Resource and Service Discovery in the CoreGrid Information Cache Mediator Component.* CoreGRID Integration Workshop 2005, Pisa, Italy, December 2005.

[11] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002)*, 2002.

[12] P. H. J. Kelly, O. Beckmann, T. Field, and S. B. Baden. THEMIS: Component Dependence Metadata in Adaptive Parallel Applications. *Parallel Processing Letters*, 11(4):455–470, Dec. 2001.

[13] J. Thiyagalingam and V. Getov. A Metadata Extracting Tool for Software Components in Grid Applications. *IEEE JVA 2006 Symposium on Moder Computing*, 189-196, IEEE CS Press, 2006.

[14] E. Krepska, T. Kielmann, R. Sirvent, R. M. Badia. *A Service for Reliable Execution of Grid Applications.* CoreGRID Integration Workshop 2006, Krakow, Poland, October 2006.

[15] J. M. Perez, R. M. Badia, J. Labarta. *Scalar-aware GRID superscalar.* Technical Report UPC-DAC-RR-2006-12, Universitat Politècnica de Catalunya, April 2006

[16] ProActive, http://www-sop.inria.fr/oasis/proactive/

[17] F. Baude, D. Caromel, M. Morel, *From Distributed Objects to Hierarchical Grid Components*, International Symposium on Distributed Objects and Applications (DOA), November 2003 http://www-sop.inria.fr/oasis/ProActive/doc/HierarchicalGridComponents.pdf

[18] Fractal specification, http://fractal.objectweb.org/specification/index.html

[19] *A Simple API for Grid Applications*
https://forge.gridforum.org/projects/saga-rg/

[20] *Grid Application Toolkit*
http://www.gridlab.org/gat

[21] The Globus Project, http://www.globus.org

[22] UNICORE, http://www.unicore.org

[23] gLite, http://glite.web.cern.ch

Appendix 1. Integrated Tookit Interfaces

This appendix details the server interfaces provided by each subcomponent of the Integrated Toolkit.

Task Analyser

```
// To initiate the execution of a task
// To inform that the application will request no more tasks
public interface TaskExecution {
    void Execute(String methodName, int parameterCount,
                 Object[] parameters);
    void finishExecution();
}

// To inform about the end of a task
// To request the notification of the end of a task
public interface TaskStatus {
    void notifyTaskEnd(int taskId, int endStatus, String message);
    void subscribeToTaskEnd(int taskId);
}
```

Task Scheduler

```
// To request the scheduling of a set of tasks
public interface Schedule {
    void scheduleTasks(Set<Task> tasks);
}

// To inform about the end of a job
public interface JobStatus {
    void notifyJobEnd(int jobId, int endStatus, String message);
}
```

Job Manager

```
// To request the creation of a job
public interface JobCreation {
    void newJob (String methodName, List<Parameter> parameters,
                 String targetHost, Set<Transfer> neededTransfers);
}

// To inform about the status of a transfer
public interface TransferStatus {
    void fileTransferInfo(int transferId, int transferStatus);
}

// To notify events about the state of a job
public interface JobEvents {
    void jobStatusNotification(int jobId, int jobStatus, int errorCode);
}
```

File Information Provider

```
// To register a new file access, either from a task
// or from the main code of the application
public interface FileAccess {
    // Returns an identifier for the file
    int registerFileAccess(String fileName, String path, String host, int mode);
    // Returns the identifier of the last task which has written the file
    int accessFromTask(int fileId, int currentTaskId, int mode);
    void accessFromApplication(int fileId, int mode);
}

// To request or update information about a file
public interface FileInformation {
    // Returns a set of locations (host,path) which have a copy of the file
    Set<Location> getLocation(int fileId);
    String getName(int fileId, String host, String path);
    void addLocation(int fileId, String newHost, String newPath);
}
```

```
    // To inform about the end of tasks
public interface TaskNotification {
    void taskFinished(int taskId);
    void allTasksFinished();
}
```

File Transfer Manager

```
// To request the transfer of a local or remote file to a destination host
public interface FileTransfer {
    // Returns an identifier for the transfer
    int transferFile(int fileId, String sourceHost, String sourcePath,
                     String destHost, String destPath);
}

// To notify events about the state of a file transfer
public interface FileEvents {
    void transferStatusNotification(int transferId, int transferStatus,
                                    int errorCode);
}
```