

Reliability and Trust Based Workflows' Job Mapping on the Grid

*Ani Anciaux--Sedrakian^{1,2}, Rosa M. Badia¹, Thilo Kielmann², Andre Merzky²,
Josep M. Pérez¹, Raül Sirvent¹*

*1Polytechnic University of Catalonia
Barcelona, Spain*

`ani.anciaux@bsc.es`
`rosa.m.badia@bsc.es`
`josep.m.perez@bsc.es`
`raul.sirvent@bsc.es`

*2Vrije Universiteit
Amsterdam, The Netherlands*
`anciaux@few.vu.nl`
`kielmann@cs.vu.nl`
`andre@merzky.net`



CoreGRID Technical Report
Number TR-0069

January 30, 2007

Institute on Grid Systems, Tools and
Environments

CoreGRID - Network of Excellence
URL: <http://www.coregrid.net>

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-00426

Reliability and Trust Based Workflows' Job Mapping on the Grid

*Ani Anciaux Sedrakian^{1,2}, Rosa M. Badia¹, Thilo Kielmann², Andre Merzky²,
Josep M. Pérez¹, Raül Sirvent¹*

1Polytechnic University of Catalonia

Barcelona, Spain

ani.anciaux@bsc.es
rosa.m.badia@bsc.es
raul.sirvent@bsc.es
josep.m.perez@bsc.es

2Vrije Universiteit

Amsterdam, The Netherlands

anciaux@few.vu.nl
kielmann@cs.vu.nl
andre@merzky.net

CoreGRID TR-0069

January 30, 2007

Abstract

The dynamic nature of grid computing environment requires maintaining a level to predict the resource reliability and application performance. In such environment, avoiding resource failures is as important as the performance criteria. The aim of this work is to identify and to acquire the most available, the least-loaded and the fastest resources in order to choose them before running the application. In this work, we describe a strategy for mapping the application jobs on the grid resources using GRID superscalar [7] and GAT [6] systems.

Introduction

This work presents an approach, which permits to predict the reliable resources by assuming application performance -computation and data transfer capacity-. This approach has been integrated on the GRID superscalar programming environment, where the applications are automatically structured as a workflow or task graph.

Many important grid applications fall into the category of workflow applications. In this case, the workflow application consists of a collection of several interacting jobs that need to be executed in a

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

certain partial order for successful execution of the application as a whole instead of a single large component doing all the jobs.

Workflow system is defined as "system that specifies, executes, monitors, and coordinates the flow of work cases, job, in a distributed environment". The system contains two basic components: first component is the build time system, which is related to define, and model workflow jobs and their dependencies. The second component is the workflow execution component, sometimes called the run-time system. The run-time system most often consists of managing workflow executions and interactions with grid resources for processing workflow applications [1].

A grid environment offers a large numbers of similar or equivalent resources that grid users can select and use them for their workflow applications. These resources may provide the same functionality, but offer different QoS measures. A workflow QoS constraint includes five dimensions: time, cost, fidelity, reliability and security. The basic performance measure is the time (the first dimension) which presents the total time required for completing the execution of a workflow. The second dimension represents the cost, which is associated with the execution of workflows. This dimension includes the cost for managing workflow systems and usage charge of grid resources for processing workflow jobs. Fidelity refers to the measurement related to the quality of the output of workflow execution. Some study in this direction is presented in [3]. Reliability is related to the number of failures for execution of workflows. Finally, security refers to confidentiality of the execution of workflow jobs and trustworthiness of resources (in [2] some studies are done in this direction). Schedulers for resource allocation at run-time may require these QoS constraints.

This work is focused on the reliability dimension, considering the time dimension, in order to minimize failures. Mapping the applications jobs on the most reliable resources is as important as mapping them on the most powerful ones. Since, the gain of time, obtained by choosing the most powerful resources, can be lost by restarting the job again, caused by any failure reason. Resources chosen by grid users may be the most powerful ones, but not always the most reliable ones. The problem is how to map the jobs onto suitable resources, in order to minimize the number of failure for execution of workflows. The workflow execution failure can occur for the following reasons [1]: the variation in the execution environment configuration, non-availability of required services or software components, overloaded resource conditions, system running out of memory², and errors in computational and network fabric components. Therefore, the natural way to maximize the reliability will be verifying these reasons.

The rest of the article is organized as follows: in section 2 GRID superscalar as a workflow system is described. Section 3 describes how to select the reliable and performance resources. Section 4 reviews the implementation of the proposed strategy in section 3 by using GAT, the infrastructure of GAT and the integration of GAT in GRID superscalar. Section 5 will be the conclusion.

GRID superscalar: Grid workflow system

The core of the grid-computing systems matures quickly, but the applications, which use these technologies, stay limited. One of the blocking reasons of the applications development on a grid-computing system is the difficulty of their programming. The objective of GRID superscalar [7] is to reduce the complexity of writing or porting applications to run them in a grid environment. GRID superscalar is a grid programming environment that enables to parallelize the execution of sequential applications in computational grids.

The GRID superscalar framework is mainly composed of the programming interface, the deployment center and the run-time system. GRID superscalar programming environment requires the following functions in the main program. GS_On() and GS_Off() functions are provided for initialization and

² Insufficient disk space for staging-in input files or for writing output or temporary files [23][24].

finalization of the run-time. `GS_Open()`, `GS_Close()`, `GS_FOpen()` and `GS_FClose()` for handling files. `GS_Barrier()` function has been defined to allow the programmers to wait till all grid jobs finish. There exist also other functions, for example `GS_Speculative_End()` which allows to catch exceptions from remote machines.

The user via an IDL file specifies the functions (jobs), which are desired to be executed in a remote server in the grid. For each of these functions, the type and nature (input, output or input/output) of the parameters must be specified. The deployment center is a Java-based Graphical User Interface, which implements the grid resource and application configuration. It verifies early failure detection, transfers to the remote machines the source code, and generates some additional source code files required for the master and the worker parts (using the `gsstubgen` tool). It compiles the main program in the localhost and the worker programs in the remote hosts and finally generates the configuration files needed for the run-time. The run-time library is able to detect the job dependencies, build a job graph, which enables to discover the inherent parallelism of the sequential application and performs concurrent job submission. Techniques such as file renaming, file locality, disk sharing, checkpointing constraints specification are applied to increase the application performance.

Of course, there exist several other systems like *Satin* [8], *ProActive* [9], *Triana* [10], which make easy the parallel programming in the grid. *Satin* [8] is a simple and efficient Java based grid-programming model for divide-and-conquer applications. *Satin* uses marker interfaces to indicate that certain method invocations need to be considered for potentially parallel (spawned) execution, rather than being executed synchronously like a normal method. Furthermore, *Satin* disposes a synchronisation mechanism. This mechanism is required explicitly, when the application necessitates waiting for the results of parallel method invocations. *ProActive* [9] is a library based on “active object” pattern for parallel, distributed and concurrent computing. It offers asynchronous communications with transparent “future objects”, “wait by necessity” mechanism and featuring mobility and security in a uniform framework. With a reduced set of simple primitives, *ProActive* provides a comprehensive API masking the used specific underlying tools and protocols. It simplifies the programming of distributed on a LAN, on a cluster of PCs, or on Internet grids. Another approach based on the job workflow description is *Triana* [10]. *Triana* is a problem-solving environment, which combines an intuitive visual interface with powerful data analysis tools. *Triana* includes a large library of pre-written analysis tools and the ability for users to integrate easily their own tools. Therefore, the users can describe their applications by dragging and dropping the components, connecting them together to build a workflow graph. Since *GRID* superscalar system offers a very widely applicable and efficient programming model, this paper focuses on it and uses this system as a starting point for design and specification of the proposed toolkit.

Resource selection

It is difficult to assume resources reliability. This is due to the existence of variability³ in the grid-computing environment. As the applications can have different characteristics, there is no single best solution for mapping workflows onto found reliable resources for all workflow applications. This section describes a strategy to find the most reliable resources, which may minimize also the overall application completion time: maximizing jobs performance and/or minimizing communications time regarding the application characteristic.

Information to retrieve

Our approach, for solving this problem, is to retrieve the information about resources. This information might be requested before workflow execution in order to help users to take the right decision

³ The load and the availability of grid heterogeneous resources vary dynamically.

in the moment of choosing the appropriate resources. The proposed strategy, composed of two steps, is based on the ranking of the resources. In the first step, called trust-driven step, the mentioned failure reasons are checked. Then the resources are ranked regarding to their capacity of guarantying the maximum level of reliability. This part will permit to increase the overall reliability of workflow execution. In the second step, called performance-driven step, the reliable resources are rearranged by taking into account the performance criteria. This step will allow an optimal execution performance, which means minimizing overall execution time. These two steps, which are done before execution, are described more in detail in the following part.

The trust-driven step is composed of three parts. In this step, some hardware requirements and historical information regarding the resources are retrieved. Like the availability of the resources, the trustfulness⁴ of them and their available amount of memory. Some of this information, as the availability of resources and the trustfulness of them are stored in a database, in order to have a trace of resources state, so they are historical. The assigned database can be located somewhere in the grid. The collected information is used for helping the users to find an appropriate resource where mostly the jobs will not fail. All this information could be gathered and stored either independently and/or before running the application. Obviously, the database could become more affluent if more information is retrieved about the hardware requirements.

During the trust-driven step, first the availability of resources is computed. Let r_i denote the i th grid resource, na_i^{j+1} the number of times when the resource r_i was not available at the moment⁵ $j+1$ and ta_i^{j+1} the total number of attempt to verify the resource availability at the same moment ($j+1$). The availability of resources is defined as:

$$\text{availability}(r_i) = na_i^{j+1} / ta_i^{j+1}$$

$$na_i^{j+1} = na_i^j + dbna_i \text{ and } ta_i^{j+1} = ta_i^j + dbta_i$$

Where $dbna_i$ and $dbta_i$ are the information stored previously on the database. After the availability calculation, the database is updated; the new value of na_i^{j+1} and ta_i^{j+1} will be stored in it.

In the second part of this step, a value is assigned to each resource, by taking the trustfulness of them into account. This value trv_i^{j+1} (the trustfulness value of i th resource at the $j+1$ moment), increases when the assigned resource does not meet the computation requirement. For example, when a job has crashed during the execution, the trustfulness value of that resource will be increased. Consequently, the resources with the lower trustfulness value are better matched for the components (workflow jobs) of the application. The trustfulness of resources is defined as bellow:

$$\text{trustfulness}(r_i) = trv_i^{j+1} = trv_i^j + dbtrv_i$$

Where $dbtrv_i$ is the trustfulness value of the resource r_i stored previously on the database. At this point of work, the resources could be sorted using the availability and the trustfulness metrics.

Afterwards, during the third part of the first step, the sorted resources will be qualified, by taking into consideration the amount of memory, which the application requires for the execution. The necessary amount of physical memory is computed using estimation on the number and the size of input, output and temporary files. Let “ m ” be the amount of disk memory, which the applications’ job requires; Figure 1 shows the functionality of this part. In order to avoid restarting the application, we first authenticated the most available and trustfulness resources, then we identify the resources with the available amount of disk memory. Once the reliability of resources is assumed, a rank value is evaluated to each eligible resource: $\text{rank}(r_i)$. The resources with the least rank value are the less reliable resources.

$$\text{if } (\text{rank}(r_i) < \text{rank}(r_j)) \rightarrow r_i \text{ is less reliable than } r_j$$

⁴ This parameter contains the information concerning previously produced fault reason due to the variation in the execution environment configuration or computational and network fabric components.

⁵ Considering that $j+1$ is the moment of verification.

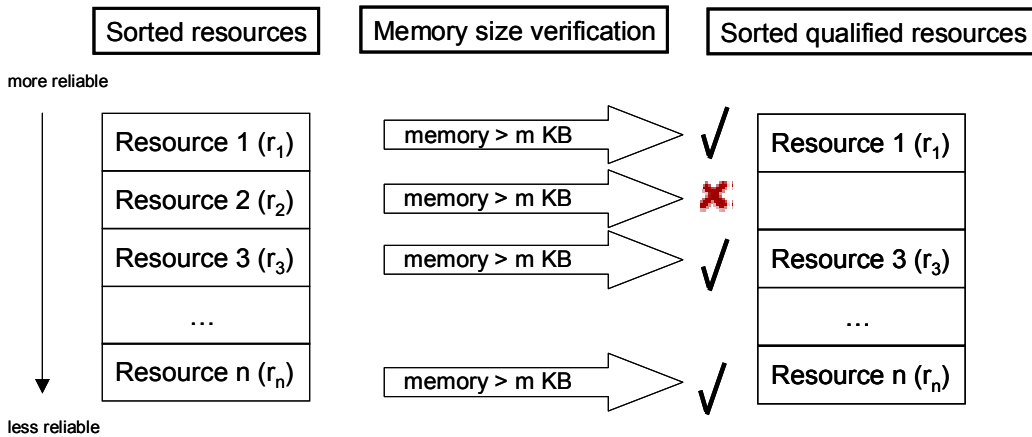


Figure 1- Sorted reliable resources

In the second step (the performance-driven step), our challenge consists to select the fastest and the least-loaded resources among the eligible reliable ones. Therefore, some appropriate reliable resources must be found for the applications' jobs, where their execution time will be the least.

The spent time for the round trip of packets on the network and the processor speed are among the parameters, which can give us an estimation regarding the execution time. This step as trust-driven step is based on the static information and is done before execution. Let $R = \{r_1, r_2, \dots, r_n\}$ denote the set of qualified grid resources. The following function give us an estimation of execution time on resource r_i :

$$\text{Time}(r_i) = \gamma \text{file_transfer_time}(r_i) + \mu \text{execution_time}(r_i) + \delta(r_i)$$

In this equation, $\text{file_transfer_time}(r_i)$ presents the spent time for the round trip of packets on the network from the local host to the resource r_i . The $\text{execution_time}(r_i)$ presents the time required for completing a job on the resource r_i . This time take into consideration the processor speed and memory access pattern. Finally $\delta(r_i)$ describes queue waiting time. Two γ and μ weighted parameters are specified to give more importance to $\text{file_transfer_time}(r_i)$ or to $\text{execution_time}(r_i)$ regarding the application requirement.

The queuing waiting time, $\delta(r_i)$, is among the parameters, which can increase the overall execution time of an application⁶. Therefore, a deeper study is required on it. In order to retrieve the information regarding the applications' waiting time in a queue, we inspire Delphoi [30] to implement an appropriate strategy to predict this time. In general the applications' waiting time may depend on the both applications' size (i.e., the number of hosts required to run it) and the queues' load⁷. For this reason, the proposed strategy forecasts two categories (fully used and normally used queue), where each of them uses four classes of application sizes, single (1 host), small (2 to 4 hosts), medium (5 to 16 hosts), and large (17 or more hosts). By taking into account the applications' size and the queues' load, an average waiting time could be predicted before running the application in order to find the less loaded one at the runtime. This parameter helps users to claim a resource, where the response time will be little.

Once the execution time is estimated, another rank value is assigned to each resource called $\text{prank}(r_i)$ which expresses the power of each resource. In this case the resource r_i is more powerful when its prank value is bigger.

$$\text{if } (\text{prank}(r_i) < \text{prank}(r_j)) \rightarrow r_i \text{ is less powerful than } r_j$$

⁶ In the case of a cluster of workstations, $\delta(r_i)$ is replaced by the resources' load. This information permits a user to choose the less loaded workstations for the components (workflow jobs) of their application and assumes by then a better performance. Once the resource load is acquired, it is compared with the amount of load, which is retrieved previously and is stored in the database. Let $\text{load}(r_i)$ describes the load of resource r_i and $\text{dbload}(r_i)$ is the amount of load computed previously and kept in the database. Then the load of resources is defined as follows: $\text{If } (\text{load}(r_i) > \text{dbload}(r_i)) \rightarrow \text{dbload}(r_i) = \text{load}(r_i)$

⁷ The jobs must often wait longer for the required resources to become available.

The major issue in this part concerns how to find the most powerful resource among the most reliable ones. Since the most reliable resources are not certainly the most powerful ones. The proposed policy must find a compromise between these two metrics. For this purpose, we give another rank value to the resources called grank (r_i). This value is a weighted linear combination of rrank (r_i) and prank (r_i) computed as follows, where α and β are the weights, which can be customized by the users (regarding the application request) to give more importance to one over the other:

$$\text{grank}(r_i) = \alpha.\text{rrank}(r_i) + \beta.\text{prank}(r_i)$$

As a result, one resource with highest grank will be relatively the most powerful and the most reliable resource regarding the user and application request. Hence, the performance-derived step allows identifying the reliable resources, which will be able to finish the job in less time.

Related work

There exist several research groups working on the scheduling algorithms for the grid environment. The proposed algorithms take into consideration the time and the performance metric. Some of them are presented in following part. Some other works are realized also by taking into account the reliability notion like the works presented in [25][26][27][28]. Nevertheless, these works are based on the failure detection and their handling, but in our case we try to predict the failure in advance.

The Grid Application Development Software (GrADS) project [21] aims to provide programming tools and execution environments for ordinary scientific users to develop, execute, and tune applications on the grid. GrADS provides application-level scheduling to map workflow application jobs to a set of resources. New grid scheduling and rescheduling methods [17] are introduced in GrADS. These scheduling methods are guided by an objective function to minimize the overall job completion time of the workflow application. The scheduler obtains resource information by using services such as MDS [18] and NWS [19] and locates necessary software on the scheduled node by query GrADS Information Service (GIS). The workflow scheduler ranks each qualified resource for each application component. A rank value is calculated by using “a weighted sum of the expected execution time on the resource and the expected cost of data movement for the component”. After ranking, a performance matrix is constructed and used by the scheduling heuristics to obtain a mapping of components onto resources. Three heuristics have been applied in GrADS; those are Min-Min, Max-Min, and Sufferage heuristics [20].

Casanova et al. in [22] propose an adaptive scheduling algorithm for parameter sweep applications on the grid. They modify standard heuristic for job/host assignment in perfectly predictable environments (Min-Min, Max-Min, and Sufferage), and they propose an extension of Sufferage called XSufferage. XSufferage can take advantage of file sharing to achieve better performance than the other heuristics. GRID superscalar possesses such scheduling based on file sharing [16].

The next section explains how we use and extend GRID superscalar infrastructure in order to handle mapping the workflow jobs on the grid resources.

GAT and GRID superscalar systems

Nowadays, GRID superscalar runs on top of Globus Toolkit [12], Ninf-G [13] and ssh/scp. Our challenge in this part is to implement the GRID superscalar’s runtime system using GAT, in order to allow both high-level and platform-independent Grid programming environment. Grid environments are dynamically changing environments: Resources and services may dynamically join or leave the grid. Various versions of services may co-exist in a single grid, and various services providing similar capability may be available. Grid Application Toolkit (GAT) [6] provides a glue, which maps the API function calls executed by an application to the corresponding adaptor-provided functionality. GAT is developed by the EC-funded GridLab project. It provides a simple and stable API to various grid environments (like Globus, Unicore, Gridlab services). GAT handles both the complexity and the variety of existing grid middleware

services via so-called adaptors. Complementing existing grid middleware, GridLab also provides high-level services to implement the GAT functionality. Therefore, the same interface could be used for the different grid environment assuming the same service. Consequently, if there are any changes on the grid environment service, the GAT-API and the application code do not change. The changes will be included in the appropriate GAT adaptor. Moreover, GAT offers the reliability; therefore, if one grid service is not available, another grid service will be used. The next session describes GAT architecture.

GAT architecture

GAT design is split in two parts: the GAT engine and the GAT adaptors. The API exposed to the application is independent of the used grid middleware service. All GAT applications link against the GAT engine, which provides proxy calls for all GAT API calls. The GAT API is designed to be simple and stable, and to provide the application with calls for essential grid operations. The GAT adaptors are lightweight, modular software elements, which provide access to these specific capabilities. Adaptors are used to bind the GAT engine to the actual middleware service providing the capabilities. The interfaces between the GAT engine and the GAT adaptors mirror the GAT API. When called via the GAT API, the GAT engine dynamically selects from the currently available adaptors implementing the specific capabilities, and forwards the API request. The mentioned characteristics of GAT lead us to implement a new version of GRID superscalar using GAT. This new version will form a high-level and platform independent programming environment for the grid.

Implementing GRID superscalar using GAT

Implementing GRID superscalar using GAT consist of replacing the Globus codes⁸ by the GAT ones. The following sections describe the Globus and the GAT functionalities.

Initialization and Session Management

The Globus module initialization (shown in Figure 2) is performed once per session, and does not require more than a single call for each used Globus module.

```
globus_module_activate (GLOBUS_GASS_COPY_MODULE);  
globus_module_activate ( GLOBUS_IO_MODULE);  
globus_module_activate (GLOBUS_GRAM_CLIENT_MODULE);  
globus_module_activate ( GLOBUS_COMMON_MODULE);
```

Figure 2- Globus initialization

GAT initializes a default session handle, which is transparently used. GAT requests GATContext creation (see Figure 3) just once. An instance of a GATContext is used to store various state information related to GAT calls, such as security information or the errors in the current call stack.

```
GATContext context;  
context = GATContext_Create();
```

Figure 3- GAT initialization

⁸ Suppose that we use the GRID superscalar version running on top of Globus.
CoreGRID TR-0069

File management (copying, moving and deleting file instances)

File copying in the grid from resource A to resource B is probably the most common use case for grid applications, particularly for GRID superscalar. Globus provides various means to perform such operations, with different complexity, performance, and behaviour. The Global Access to Secondary Storage (GASS) module, as shown in Figure 4, provides one of the simplest versions.

```
globus_gass_copy_handle_init (& handle , GLOBUS_NULL);

globus_io_file_open ("/home/workingdir/file.txt",
                    GLOBUS_IO_FILE_WRONLY | GLOBUS_IO_FILE_CREAT
                    | GLOBUS_IO_FILE_TRUNC ,
                    GLOBUS_IO_FILE_IRUSR | GLOBUS_IO_FILE_IWUSR
                    | GLOBUS_IO_FILE_IRGRP ,
                    GLOBUS_NULL , & fileHandle);

globus_gass_copy_url_to_handle (& handle , "gsiftp://B/path/file.txt", GLOBUS_NULL , & fileHandle);

globus_io_close (& fileHandle);
```

Figure 4- File coping using Globus

One of the major problems for file copying in grids is that the application programmer or even the end user needs to be aware of the transport protocols available in the code. For example, the user needs to know if gsiftp is actually available for that resource. GAT API provides a more abstract notion of that transport, by allowing the protocol “any” (see Figure 5). Then the GAT implementation selects an available protocol for the data transfer. Moreover, contrary to Globus, GAT allows to move and delete files easily. The following two functions `GATResult GATFile_Delete (GATFile_const file)` and `GATResult GATFile_Move(GATFile_const file, GATLocation_const targetlocation, GATFileMode mode)` permit the realisation of mentioned functionalities.

```
/* Create GATLocation sourceLocation */
sourceLocation = GATLocation_Create( "any://A//home/workingdir/test.in");

/* Create GATLocation destinationLocation */
destinationLocation = GATLocation_Create("any://B//home/workingdir/test.copy");

/* Create GATFile sourceFile */
sourceFile = GATFile_Create( context, sourceLocation, 0);

/* Create GATFile sourceFile */
sourceFile = GATFile_Create( context, sourceLocation, 0);

/* Copy sourceFile to destinationLocation */
result = GATFile_Copy(sourceFile, destinationLocation, GATFileMode_Overwrite );

/* Destroy GATFile sourceFile */
GATFile_Destroy( &sourceFile );

/* Destroy GATLocation destinationLocation */
GATLocation_Destroy(&destinationLocation );

/* Destroy GATLocation sourceLocation */
GATLocation_Destroy(&sourceLocation );
```

Figure 5- File coping using GAT

Remote job submission

The simplest way to submit a remote job via Globus is to provide a RSL description of the job to the GRAM module (see Figure. 6). This module will forward the job submission request to the Globus gatekeeper on the remote resource B, which will then run the job. A job handle is returned for later reference, e.g., for checking the jobs' status.

```
/* The callback_contact is passed in order to receive notifications*/
sprintf(RSL , "%(executable=/path/exec)(directory=/path)" "(arguments=-1)(queue=short)"
"( file_stage_in = gsiftp://B/path/file1 /path/file1)" "(file_stage_out= /path/file2 gsiftp://B/path/file2)"
"(file_clean_up = /path/file3 )" "(environment=(NAME1 val1)(NAME2 val2))");

globus_gram_client_job_request (B , RSL ,
                                GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE |
                                GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING |
                                GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE |
                                GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED,
                                callback_contact , & job_contact);
```

Figure 6- Remote job submission using Globus

In GAT before submitting a job in resource B, GATSoftwareDescription and GATHardwareResourceDescription must be created. Each instance of GATSoftwareDescription and GATHardwareResourceDescription stores the software (the executable file (job) location, job arguments, pre-staged files, post-staged files) and the hardware (compute resources) requirements for running the job on the grid resources.

```
GATResourceBroker broker = NULL;
GATJob job = NULL;
GATJobDescription jd = NULL;
GATSoftwareDescription sd = NULL;
GATResourceDescription hrd = NULL;
GATJobID_const jobid = NULL;
GATJobDescription jd = NULL;

/* submit the job to the resource broker */
broker = GATResourceBroker_Create (context, 0, 0);

/* create the software description of the job to start */
sd = create_software_description (context, host, exe, nargs, (char const **)args);

/* create a hardware resource description describing the required job environment */
hrd = create_hardware_resource_description (host);

/* make a job description out of the software and hardware resource description */
jd = GATJobDescription_Create_Description (context, sd, hrd);

/* submit a new job */
GATResourceBroker_SubmitJob (broker, jd, &job);

/* retrieve the GAT job id of the newly created job */
GATJob_GetJobID (job, &jobid);
```

Figure 7- Remote job submission using GAT

Job state notification

Via callbacks and blocking polls, Globus provides a very convenient way for GRID superscalar to wait for the completion of submitted Jobs (Figure 8).

```

/* JobEnds treats the state change*/
globus_gram_client_callback_allow ( JobEnds , NULL ,& callback_contact);

/* Blocked waiting for notifications*/
globus_poll_blocking ();

```

Figure 8- Job state notification using Globus

Unfortunately, there is no special and adequate way envisaged in GAT for this purpose apart from polling (described in Figure 9).

```

GATJobState state = GATJobState_Unknown;
while ( state != GATJobState_Stopped )
{
    GATJob_GetState (job, &state);
    sleep (1);
}

```

Figure 9- Job state notification using GAT

Deactivation

The deactivation of all the Globus modules (described in Figure 10) must be called once at the end of the program. This later permit to deactivate GRAM, GASS and I/O clients which were activated using the Globus `globus_module_activate()` function.

```

res = globus_module_deactivate(GLOBUS_COMMON_MODULE);
if(res != GLOBUS_SUCCESS)
    printf("ERROR: deactivating Globus common module\n");

res = globus_module_deactivate(GLOBUS_GRAM_CLIENT_MODULE);
if(res != GLOBUS_SUCCESS)
    printf("ERROR: deactivating Globus client module\n");

res = globus_module_deactivate(GLOBUS_GASS_COPY_MODULE);
if(res != GLOBUS_SUCCESS)
    printf("ERROR: deactivating Globus gass copy module\n");

res = globus_module_deactivate(GLOBUS_IO_MODULE);
if(res != GLOBUS_SUCCESS)
    printf("ERROR: deactivating Globus I/O module\n");

```

Figure 10- Globus modules deactivation

In GAT in order to destroy the passed `GATContext` instance and to free up any resources, held by the passed `GATContext` we use `GATContext_Destroy(&context)` function.

Resource selection on GRID superscalar

The proposed strategy described previously, which tries to find the reliable and powerful resources, can be realised thanks to the GAT APIs and GRID superscalar system. This section describes briefly the implementation of this strategy using these two systems.

At the beginning, the required information like the availability and the trustfulness of resources, the resources' load and their amount of disk space, the consistency of retrieved information, etc are computed. Some of this information like resources' amount of disk memory is computed thanks to the mercury⁹ monitoring system [15]. The trustfulness of resource is computed thanks to GRID superscalar. GRID superscalar can detect if any of the workers' jobs fails due to an internal error, or because it has been killed for any reason. The information regarding the remote resources is computed using GAT remote job submission (see Figure 12). Figure 11 describes how the amount of disk memory of remote hosts could be retrieved using GAT job submission.

```

broker = GATResourceBroker_Create (context, 0, 0);
/*mercury: to find the size of the host's free physical memory, in kilobytes*/
sprintf(exe, "monclient -n host.mem.free -p host=%s monp://%", ResourceName, ResourceName);

/* create the software description of the job to start */
sd = create_software_description (context, exe);

/* create a hardware resource description describing the required job environment */
hrd = create_hardware_resource_description (host);

/* make a job description out of the software and hardware resource description */
jd = GATJobDescription_Create_Description (context, sd, hrd);

/* submit a new job */
GATResourceBroker_SubmitJob (broker, jd, &job);

```

Figure 11 - code snippet: free amount of memory computation of a remote resource

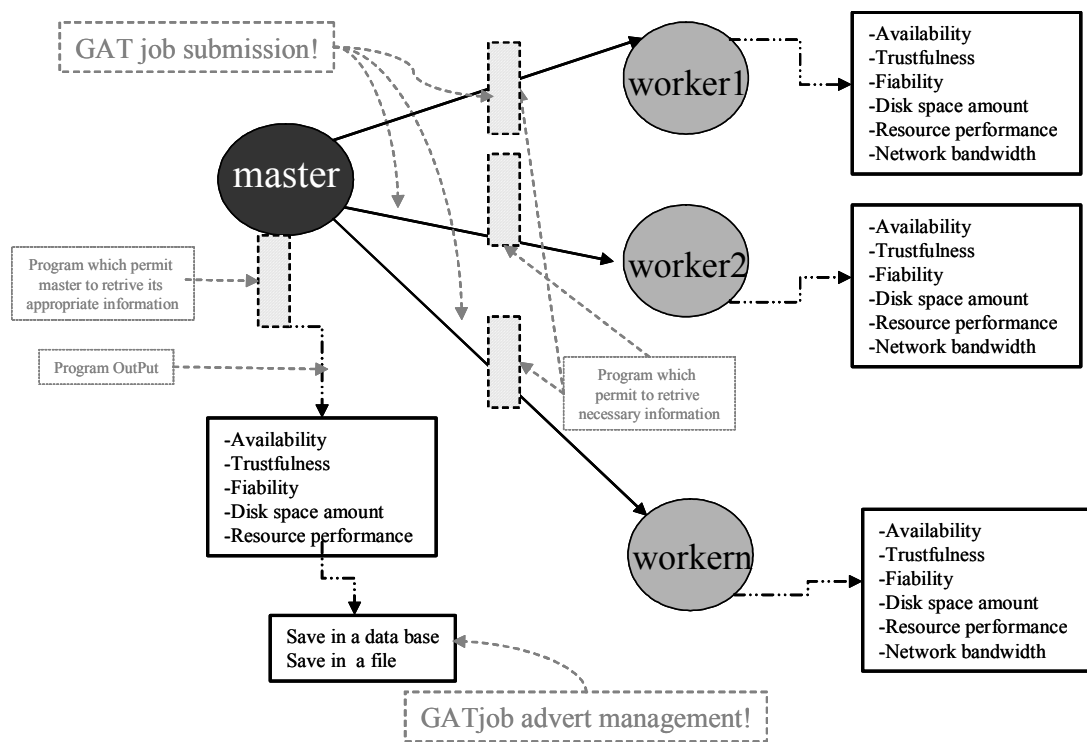


Figure 12 - How to compute the desired information

Once the required informations are computed, the local host collects them, using GAT file copy (see Figure 13). Thereafter, some of these informations are stored in a database situated somewhere. This later

⁹ The Mercury Monitoring System is a general-purpose grid monitoring system developed by the GridLab project. It supports the monitoring of machines, grid services and running applications. Mercury features a flexible, modular design that makes it easy to adapt Mercury for various monitoring requirements.

permits us to dispose the historic trace of previously computed information, which will help us to take a good decision to choose the most appropriate resources. In this purpose the “Advert Management” service in GAT offers us the possibility of saving and interrogation of information easily throughout several and independent execution (see Figure 13). This service permits to each resource to possess its own advertisement and more clearly its own storage space.

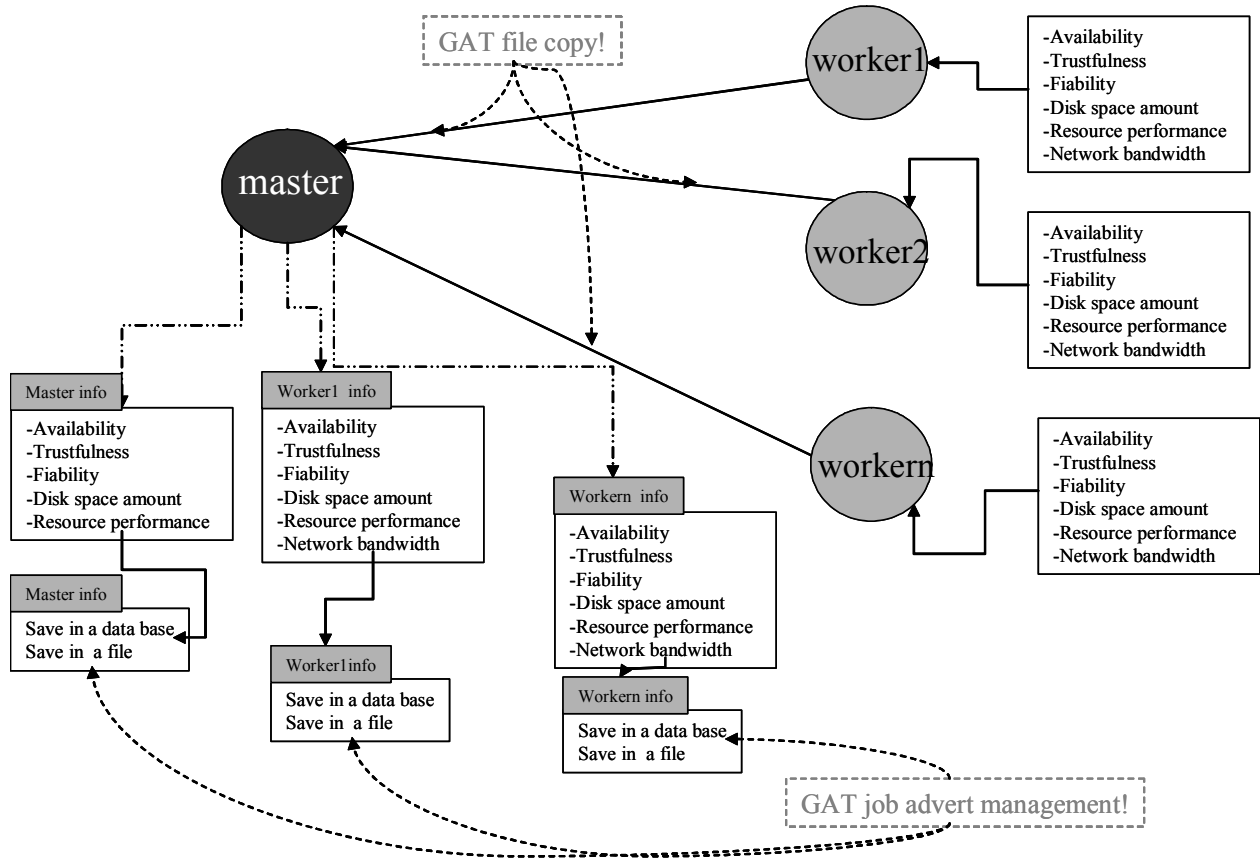


Figure 13 - How retrieve the desired information

GRID superscalar provides also a service, which allows users to specify the cost function for the estimation of the execution time of each job in a given resource. Figure 14 describes an example of such function, where GS_Filesize and GS_GFlops are runtime primitives. The cost function is dynamically evaluated by the runtime library.

```
double simulation_cost(file cfgFile, file traceFile)
{
double time;
time = (GS filesize(traceFile)/GS GFlops()*1000000);
return(time);
}
```

Figure 14 - Sample user modified time estimations function

The pervious mentioned points confirm that GAT and GRID superscalar allow us to implement the prediction of the reliable resources by assuming application performance (computation and data transfer capacity).

Future Work

As mentioned previously, Satin is a simple and efficient Java based grid-programming model for divide-and-conquer applications. This programming model seems to be very similar to the GRID superscalar one. As in GRID superscalar, the jobs which will be run on the grid are identified before the execution by the users. Moreover, in Satin each specified jobs can be spawned itself on the several grid resources. This functionality¹⁰ makes Satin more interesting. Nevertheless, contrary to GRID superscalar, Satin does not deal with data dependency. It is up to the users to add the necessary synchronisation points on the program. Solving this problem will make Satin more powerful than its current version and compatible with the theory of GRID superscalar.

Introduction of “*Future*” concept in Satin (as it exists in ProActive system) can enrich this programming model by solving the data dependency problem. *Futures* represent the result or return value of a user identified function, executing in a separate thread. In this concept, the “*wait by necessity*” mechanism is the only synchronization which waits the results or return values only if they are not available. In this way there will be no need to add explicitly the synchronization points in the program.

Conclusion

This paper has presented a mechanism, which tries to run the application jobs on the most reliable and most powerful resources regarding the application requirements. One part of this system is based on solving the problems by evaluating the past experience. In the beginning, users furnish a list of hosts, which they want to use and the requirement regarding the application jobs. The proposed system gathers all the necessary characteristics about resources in a database. Thanks to the collected information, our system finds and chooses the eligible and the adequate resource for each of the jobs.

References

- [1] J. Yu and R. Buyya. Taxonomy of Scientific Workflow Systems for Grid Computing. *Sigmod Record*, Vol. 34, No. 3, Sept. 2005
- [2] S. Zhao and V. Lo. Result Verification and Trust-based Scheduling in Open Peer-to-Peer Cycle Sharing Systems.
- [3] F. Vraalsen, R. A. Aydt, C. L. Mendes and D.I A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior.
- [4] V. Hamscher et al. Evaluation of Job-Scheduling Strategies for Grid Computing. In 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), Springer-Verlag, Heidelberg, Germany, 2000; 191-202.
- [5] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, L. Johnsson. Scheduling Strategies for Mapping Application Workflows onto the Grid. In IEEE International Symposium on High Performance Distributed Computing (HPDC 2005).
- [6] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. In the proceeding of the IEEE.
- [7] R. Sirvent, J. M. Pérez, R. M. Badia, J. Labarta, Automatic Grid workflow based on imperative programming languages, *Concurrency and Computation: Practice and Experience*, John Wiley & Sons, Published Online: Dec 2005

¹⁰ Disposing two level of parallelism: first level consist to distribute and run the jobs on the grid and second level consist to spawn each job on the several resource of grid.

- [8] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19-32, September 2005.
- [9] F. Huet, D. Caromel and H. E. Bal. A High Performance Java Middleware with a Real Application. In *SuperComputing Conference*, November 2004.
- [10] Taylor I, Shields M, Wang I, Philp R. Distributed P2P computing within Triana: A Galaxy visualization test case. *Proceedings of IPDPS 2003*, 22–26 April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.
- [11] R. Raman, Miron Livny and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.
- [12] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of Supercomputer Applications*, 1997.
- [13] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41-51, 2003.
- [14] R. M. Badia, R. Sirvent, J. Labarta and J. M. Pérez. Programming The Grid: An Imperative Language Based Approach. *Engineering the Grid: Status and Perspective*, American Scientific Publishers, 2006
- [15] G. Gombás, C. Attila Marosi and Z. Balaton Grid Application Monitoring and Debugging Using the Mercury Monitoring System. In *Advances in Grid Computing book*
- [16] R. M. Badia, Jesús Labarta, Raúl Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima. Programming Grid Applications with GRID superscalar. In *Journal of GRID Computing*, Vol. 1 Issue 2. Pages: 151-170 , June 2003.
- [17] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, J. Dongarra. New Grid Scheduling and Rescheduling Methods in the GrADS Project. *NSF Next Generation Software Workshop, International Parallel Distributed Processing Symposium*, Santa Fe, IEEE CS Press, Los Alamitos, CA, USA, April 2004.
- [18] S. Fitzgerald, I. Foster, C. Kesselman, G. Von Laszewski, W. Smith and S. Tuecke. A Directory Configuring High-Performance Distributed Computations. In *6th IEEE Symposium on High-Distributed Computing*, Portland, OR, IEEE CS Press, Los Alamitos, August 1997; 365-375.
- [19] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. In *Future Generation Computer Systems*, 15(5-6):757-768, 1999.
- [20] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic Matching and Scheduling of a Class of Independent Jobs onto Heterogeneous Computing Systems. In *8th Heterogeneous Computing Workshop (HCW'99)*, Juan, Puerto Rico, IEEE Computer Society, Los Alamitos, April 12, 1999.
- [21] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. In *International Journal of High Performance Computing Applications(JHPCA)*, 15(4):327-344, SAGE Publications Inc., London, UK, Winter 2001.
- [22] H. Casanova, A. Legrand, D. Zagorodnov and F. Berman. Heuristic For Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW)*, pages 349--363, Cancun, may 2000.
- [23] G. Kola, T. Kosar and M. Livny Phoenix: Making Data-intensive Grid Applications Fault-tolerant. In *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, 2004

- [24] S. Hwang and C. Kesselman. Grid Workflow: A Flexible Failure Handling Framework for the Grid. In High Performance Distributed Computing, 2003.
- [25] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, R. D. Schlichting. Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance. In Proceedings of the 2004 IEEE International Conference on Cluster Computing.
- [26] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In Proceedings of the 1999 IEEE International Conference on Cluster Computing.
- [27] G. Kola, T. Kosar and M. Livny. Phoenix: Making Data-intensive Grid Applications Fault-tolerant
- [28] S. Hwang; C. Kesselman,, Grid workflow: a flexible failure handling framework for the grid. In High Performance Distributed Computing, 2003
- [29] E. Krepska, T. Kielmann, R. Sirvent, R. M. Badia. A Service for Reliable Execution of Grid Applications. In CoreGRID Integration Workshop 2006, Krakow, Poland, October 2006
- [30] J. Maassen, R. V. Van Nieuwpoort, T. Kielmann, K. Verstoep. Middleware Adaptation with the Delphoi Service. In Concurrency and Computation: Practice & Experience, 2006