A Search Architecture for Grid Software Components

Fabrizio Silvestri
HPC-Lab
ISTI-CNR, Italy
fabrizio.silvestri@isti.cnr.it

Diego Puppin
HPC-Lab
ISTI-CNR, Italy
diego.puppin@isti.cnr.it

Domenico Laforenza
HPC-Lab
ISTI-CNR, Italy
domenico.laforenza@isti.cnr.it

Salvatore Orlando
Dipartimento di Informatica
Università di Venezia - Mestre
orlando@unive.it

Introduction. Today, the development of Grid applications is considered a nightmare, due to lack of grid programming environments, standards, off-the-shelf software components, and so on.

Many authors envision the existence of a marketplace for software components where developers can gather the components for their applications [3]. This model, if globally accepted, would find its natural end in the Grid platform. The main obstacles to this goal seem to be the: (a) the lack of a standard for describing components and their interactions, and (b) the need for a service able to locate relevant components satisfying some kind of cost constraints. Regarding standards, recent advances in Component and Grid technology, such as JavaBeans, ActiveX, and Grid Services, are providing a basis for interchangeable parts. The Grid and the Internet, instead, provide a means for consumers (i.e. programmers) to locate available components.

Moreover, standardization efforts on component models, integration platforms, and business domain concepts will accelerate the usage and the spreading of components for building component-based Grid Applications[1]. Hence, it can be expected that in a very near future, there will be thousands of components providers available on the Grid. It is worth noting that within the above mentioned market of components, the same kind of service would be sold by different vendors at different prices and with different quality.

From the considerations made so far, it is clear that when this way of developing applications starting from basic and separately bought building blocks will become fully operational, the most challenging goal to pursue will be finding the best components suitable for each user's needs. As far as we know, there has been limited effort in the Grid research community towards this goal. In this paper, we are going to discuss the challenges we have to face in designing a search service for locating software components on the Grid. Indeed, the specifications of our search engine rely heavily on the concept of *Ecosystem of Components*. Basically it consists in an extension of the marketplace concept to the Grid. The ecosystem, in fact, provides applications a virtual environment where they can *live* (i.e. are executed), and find other components to which they can refer (i.e. *cooperate*) to build larger (and more complex) systems.

In other words, the idea of *Ecosystem of Components* can be easily compared to the well known concept of Web. Under this vision a software component is a sort of Web page and an application built by composing different blocks can be seen as a Web site (i.e. a composition of different Web pages). From the application perspective, each part can be either a component available locally (i.e. a *local* web page), or a remote component (i.e. a *remote* web page). In addition, the links interconnecting Web pages can be compared to the links indicating interactions among components of the same application. The most interesting characteristic of this model, anyway, is that a user can, possibly, make publicly available the relationships between the different components involved in the application. Regarding this last point, one could argue that a developer would not publicize how s/he has realized an application. We do not think so: there are many reasons why s/he would do so. First of all, as in the Web there are portals like SourceForge[1], or FreshMeat[2] that allow the publishing of open source application, we think that in the Grid will exists public repository of applications. Moreover, in the Web there are many examples of popular services that publicize their use of other important and effective services: AOL, for instance, claims that it uses the Open Directory Project (ODP [9]) as its backbone for offering its search service. Ask Jeeves [7] shows results from its own editors and it asserts that it gets spidered listings from *Teoma.com* [5] and paid listings from

[1] http://www.sourceforge.net
[2] http://www.freshmeat.net

Google [4]. Translated to the Grid, the importance of an application could be raised by using another, and more popular, component. In other words, we would inherit the importance of the used components.

As already said, our system tackles the concept of Workflow graphs for modeling Grid Application to compute a sort of *static importance value* that will be used as a measure of the "*quality*" of each application. The idea is rather simple: the more an application is referred by other applications the more important this application is considered. Note that this concept is very close to the well known PageRank[8] measure used by Google[4] to rank the pages it stores.

Grid Applications and Workflows. To date, we can observe that there has been very limited work on Grid-specific programming languages. This is not too surprising since automatic generation of parallel applications from high-level languages usually works only within the context of well defined execution models [11], and Grid applications have a more complex execution space than parallel programs. Some interesting results on Grid programming tools have been reached by scripting languages such as Python-G [6], and workflow languages such as DAGMAN [10]. These approaches have the additional benefit that they focus on coordination of components written in traditional programming languages, thus facilitating the transition of legacy applications to the Grid environment.

This workflow-centric vision of the Grid is the one we are going to investigate in this work. We envision a Grid programming environment where different components can be adapted and coordinated through workflows, also allowing hierarchical composition. According to this approach, we thus may compose *metacomponents*, in turn obtained as a workflow that uses other components. An example of workflow graph is shown in Figure 1. Even if this graph is flat, it has been obtained through composing different metacomponents, in particular "flight reservation" and "hotel reservation" components. As you can note, we have not chosen a typical *scientific* Grid application, but rather a *business-oriented* one. This is because we are at the moment of convergence of the two worlds, and because we would like to show that such Grid programming technologies could also be used in this case.

The strength of the Grid should be the possibility of picking up components from different sources. The question is now: where are the components located? In the following we are going to present some preliminary ideas on this issue.

Application Development Cycle. In our vision, the application development should be a three-staged process, which can be driven not only by an expert programmer, but also by an experienced end-user, possibly under the supervision of a problem solving environment (PSE). In particular, when a PSE is used, we would give the developer the capability of using components coming from:

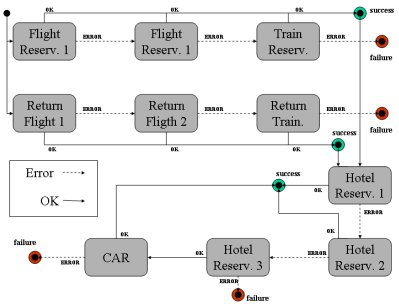

Figure 1. An example of a workflow-based application for arranging a conference trip. The user must reserve two flights (outward and return) before reserving the hotel for the conference. Note that, in the case that only the third hotel has available rooms, a car is needed and must be booked too.

- a *local repository*, containing components already used in past applications, as well as others we may have previously installed;
- a *search engine*, which is able to discovery the components that fit users' specifications.

Hence, the three stages which drive the application development process are:

1. application sketching;
2. components discovering;
3. application assembling/composition.

Starting from stage 1 (i.e. *sketching*), developers may specify an *abstract* workflow graph or *plan*. The abstract graph would contain what we call *place-holder* components and flow links indicating the way information passes through the application's parts.

A place-holder component represents a partially specified object that just contains a brief description of the operations to be carried out and a possibly inaccurate description of its functions. The place-holder component, under this model, can be thought as a *query* submitted to the component search module, in order to obtain a list of (possibly) relevant component for its specifications.

Obviously, the place-holder specifications can be as simple as specifying only some keywords related to *non functional* characteristics of the component (e.g. its description

in natural language), but it can soon become complex if we include also *functional* information. For example, the "Flight Reservation" component can be searched through a place-holder query based on the keywords: *airplane, reserve, flight, trip, take-off*, but we can also ask for a specific method signature to specify the desired destination and take-off time.

The Component Search Module. In the last years, the Web Search Engine study has became a new and important research topic. In particular, several researchers' efforts have been spent on Web models suitable for ranking results of a query to a Search Engine [2].

We would like to approach the problem of searching software components using this mature technology. We would like to exploit the concept of *ecosystem* of components to design a solution able to *discover* and *index* applications' building blocks, and allows the search of the most relevant components for a given query. Furthermore, the most important characteristic is the exploitation of the *interlinked structure* of metacomponents (workflows) in the designing of smart Ranking algorithms. These workflows ranking schemas, in fact, will be aware of the context where the components themselves are placed.

To summarize, Figure 2 shows the overall architecture of our Component Search Engine called *GRIDLE*: *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*.

The main modules of GRIDLE are the *Component Crawler*, the *Indexer* and the *Query Answering*.

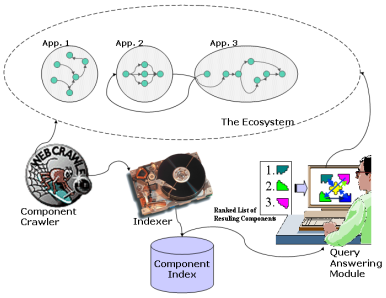

Figure 2. The architecture of GRIDLE.

The *Component Crawler* module is responsible for automatically retrieving new components. The *Indexer* has to build the *index* data structure of GRIDLE. This step is very important, because some information about the relevance of the components within the ecosystem must be discovered and stored in the index. The last module of GRIDLE is the *Query Answering* one, which actually resolves the components queries on the basis of the index. As other, traditional, search engine, the GRIDLE searching algorithm is made up of two steps. First, GRIDLE tries to resolve the place-holder by using the components contained in the local repository. If a suitable component is found locally, then this is promptly returned to the user without searching on remote sites. On the other hand, if it cannot be found locally, a *Query Session* is started. The goal is to retrieve a *ranked list* of components that are *relevant* to the specification given in the place-holder plan graph.

After the searching phase, we have to put together all the chosen modules in order to: (1) fill in all the place-holders, and (2) *materialize* the connections among the components.

As an example, let us consider the above steps in the development process of the example depicted in Figure 1. In a Grid software development environment a programmer could have sketched the abstract workflow plan graph depicted in Figure 3.

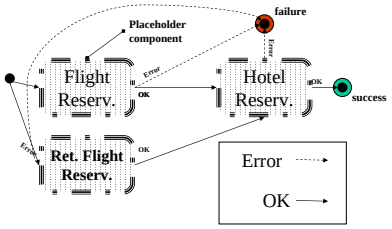

Figure 3. A partially specified workflow graph, describing the application of Figure 1 at the highest level possible.

Starting from here, s/he would proceed as follow. First, s/he would look for a flight reservation component matching the place-holder. Let us suppose that such a component is available locally. GRIDLE will automatically return a pointer to it and expand the place-holder with the found component (*binding*). Figure 4 shows the workflow graph as it appears at this point of development. In the picture we can see that the found component has been instantiated twice, for both the outward and return flight reservations. Moreover, note that the matching component is a metacomponent, i.e. it is composed of several (interconnected) components.

Then, the user selects the "Hotel Reservation" place-holder. Since this is not available in the local repository, a query session is initiated. GRIDLE starts looking for a component. The search process is two-staged. In its first part, GRIDLE tries to find an initial (possibly inaccurate) list of components. Then, the user has to refine it until a shorter, and more relevant, list of components is obtained. Once the search phase is ended the user would pick up the most suitable component to replace the corresponding place-holder (*binding*). Finally, when all the components are fully specified, the developer will continue refining the application

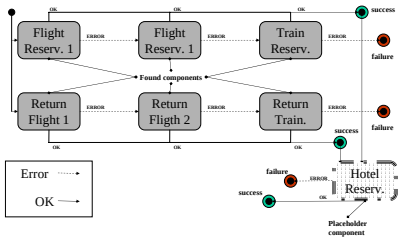

Figure 4. The abstract workflow graph as it appears after the "flight reservation component" has been found.

until it meets his/her original requirements.

The binding phase may be as simple as forwarding the output channel of a component to the input of the next (as for Unix pipes), but it may be more complex if data and/or protocol conversions are needed. In this latter case, a user-driven, framework-assisted procedure is needed. The framework should try to determine the type and the semantics of components' input/output ports, using any available header, XML and textual descriptions, Web ontologies, pattern matching and naming conventions. With this information, the programmer should choose the best chain of conversions, and ask the framework to instantiate an ad-hoc filter, performing the transformation needed (for instance, the output of a components needs to be converted from a chain of strings into a Java array of double, and sent over HTTP/SOAP).

Conclusions. In this contribution, we presented our vision of a new tool allowing the design of workflow-based Grid applications where a composition of different workflows can be seen as a single autonomous meta-component. The main issue presented in the work is the *component search service*, which allows users to locate the components they need. We believe that in the near future there will be a growing demand for ready-made software services, and current Web Search technologies will help in the deployment of effective solutions. The search engine, based on information retrieval techniques, in our opinion should be able to *rank* components on the basis of: their similarity with the place-holder description, their popularity among developers (something similar to the hit count), their use within other services (similarly to PageRank) etc.

Clearly, it is of primary importance the existence of a quick, efficient, automatic way to deploy software components out of existing code. In our opinion, there is the need for automatic tools able to extract signature information from legacy code, and able to create the bridging code needed to make the component communicate with other entities, designed with different languages or running on different platforms.

When all these services become available, building a Grid application will become a straight-forward process. A non-expert user, aided by a graphical environment, will give a high-level description of the desired operations, which will be found, and possibly paid for, out of a quickly evolving market of services. At that point, the whole Grid will become as a virtual machine, tapping the power of a vast numbers of resources [3].

References

[1] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 13. IEEE Computer Society, 1999.

[2] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] Rajkumar Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, April 2002.

[4] The Google Search Engine. http://www.google.com.

[5] The Teoma Search Engine. http://www.teoma.com.

[6] N. Jackson. pyglobus: a python interface to the globus toolkit. *Concurrency and Computation: Practice and Experience*, 14(13-15):1075–1084, 2002.

[7] Ask Jeeves. http://www.askjeeves.com.

[8] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[9] The Open Directory Project. http://www.dmoz.org.

[10] D. Thain, T. Tannenbaum, and M. Livny. *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11 - Condor and the Grid, pages 299–335. John Wiley, 2003.

[11] Cheer-Sun D. Yang and Lori L. Pollock. All-uses testing of shared memory parallel programs. *Software Testing, Verification, and Reliability Journal*, (13):3–24, 2003.