A SEARCH ARCHITECTURE FOR GRID SOFTWARE COMPONENTS

Diego Puppin, Fabrizio Silvestri and Domenico Laforenza
HPC-Lab, ISTI-CNR
Via Moruzzi 1, 56100, Pisa
Italy
{diego.puppin, fabrizio.silvestri, domenico.laforenza}@isti.cnr.it

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

Abstract

Today, the development of Grid applications is a very difficult task, due to the lack of Grid programming environments, standards, off-the-shelf software components, and so on.

Nonetheless, we can observe an emerging trend: more and more services are available as Web Services, and can be linked to form an application. This is why we envision a market where developers can pick up the software components they need for their application. A natural process of evolution in this market will reward components that are faster, cheaper, more reliable or simply more popular.

In this work, we present our vision of GRIDLE, a search engine for software components. It will rank components on the basis of their popularity, their cost and performance, and other users' preferences. We built a prototype of GRIDLE, which works on Java classes. It is able to give them a rank based on the social structure of Java classes.

Keywords: Software components, service ecosystem, search engine, ranking.

1. Introduction

Present-day Grid programming is considered a very hard task, due to the lack of mature and easily available software components, of a standard for workflow description, of easy-to-use development environments, and so on.

Many authors envision the existence of a marketplace for software components where developers can gather components for their applications [5]. 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, which satisfy some kind of cost constraints. 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 will simplify the use of components for building component-based Grid Applications[1]. We can expect that in a very near future, there will be thousands of components providers available on the Grid. Within this market of components, the same kind of service will be sold by different vendors at different prices, and with different quality.

It is clear that when this way of developing applications will become fully operational, the most challenging task will be to find the best components suitable for each user. As far as we know, there has been limited effort in the Grid research community towards this goal. In this paper, we 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*.

This idea can be described with an analogy to the Web. In our vision, a software component can be compared to a 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 locally available component (i.e. a *local* web page), or a remote one (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 available the relationships between the different components involved in the application.

As a matter of fact, a very popular workflow language, BPEL4WS, is designed to expose links among Web Services. It allows a two-level application description: an executable description, with the specification of the processes, and an abstract description, which specifies the invocations of and the interactions with external web services [15].

It could be argued that a developer would not publicize how s/he has realized an application. We do not think so. Today, there are many examples of popular Internet services that publicize their use of other important and effective services: AOL, for instance, claims that it uses the Open Directory Project as its backbone for offering its search service. Translated to the Grid, the importance of an application could be raised by using another popular component.

As already said, our system tackles the concept of workflow graphs for modeling Grid Application to compute a *static importance value*. This 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. This concept is very close to the well known PageRank [14] measure used by Google [7] to rank the pages it stores.

This contribution is structured as follows. In the next section, we introduce the concept of the ecosystem of components, and we discuss our initial findings. Then, we introduce our vision of GRIDLE, a tool for searching components on the Grid. In particular, we discuss its architecture, the ranking metrics, and we show our initial results with our first prototype. After an overview of related work, we conclude.

This paper extends our previous work, presented at Web Intelligence 2004 [17].

2. The Ecosystem of Component

In the first phase of our experiment, we collected and analyzed about 7700 components in the form of Java classes. Clearly, Java classes are only a very simplified model of software components, because they are supported by only one language, they cannot cooperate with software developed with other languages, but they also support some very important features: their interface can be easily manipulated by introspection; they are easily distributed in form of single JAR files; they have a very consistent documentation in the form of JavaDocs; they can be manipulated visually in some IDEs (BeanBox, BeanBuilder etc.).

We were also able to retrieve very high-quality Java Docs for the several projects, including Java 1.4.2 API; HTML Parser 1.5; Apache Struts; Globus 3.9.3 MDS; Globus 3.9.3 Core and Tools; Tomcat Catalina; JavaDesktop 0.5, JXTA; Apache Lucene; Apache Tomcat 4.0; Apache Jasper; Java 2 HTML; DBXML; ANT; Nutch.

For each class, we determined which other classes it used and were used by. With usage, we mean the fact that a class A has methods returning, or using as an argument, objects of another class B: in this case, we recorded a link from A to B. This way, we generated a directed graph describing the social network of the Java library.

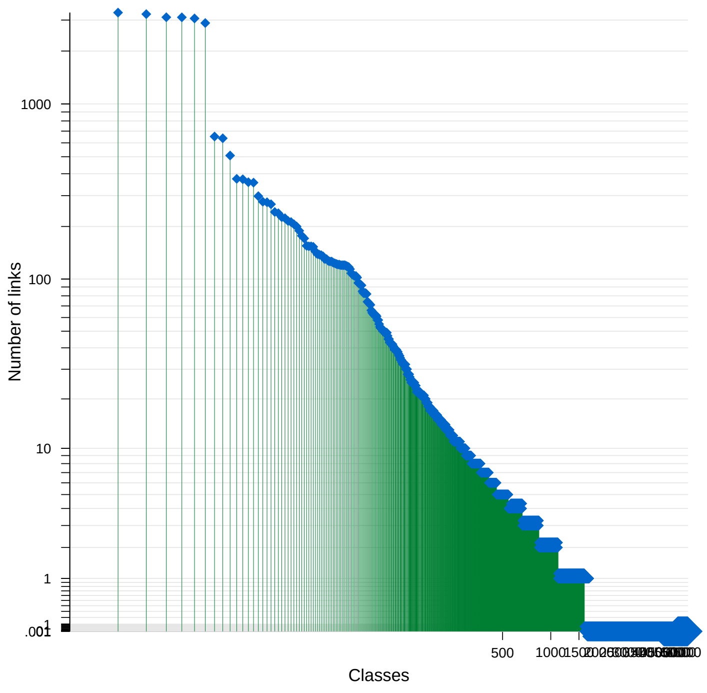


Figure 1. Social behavior of Java classes: Power-Law distribution

We could observe some very interesting phenomena. First of all, the number of links to each class follows a typical power-law rule: very few classes are linked by very many others, while several classes are linked by only a few other classes.

In Figure 1, the reader can see a graph representing the number of incoming links to each class, in log-log scale. Classes are sorted by the number of incoming links. The distribution follows closely a power-law pattern, a small exception given by the first few classes (Object, Class etc.) which are used by almost all other derived classes to provide basic services, including introspection and serialization.

This is a very interesting result: within Java, the popularity of a class among programmers seems to follow the pattern of popularity shown by the Web, blogs and so on. This is very promising: hopefully, we will be able to build a very effective ranking for components out of this.

3. GRIDLE

3.1 Grid Applications and Workflows

To date, 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 [22], 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 [11], and workflow languages such as DAGMAN [18]. 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 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 2. Even if this graph is flat, it has been obtained by composing together different metacomponents, in particular “flight reservation” and “hotel reservation” components. 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 present some preliminary ideas on this issue.

3.2 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:

- 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:

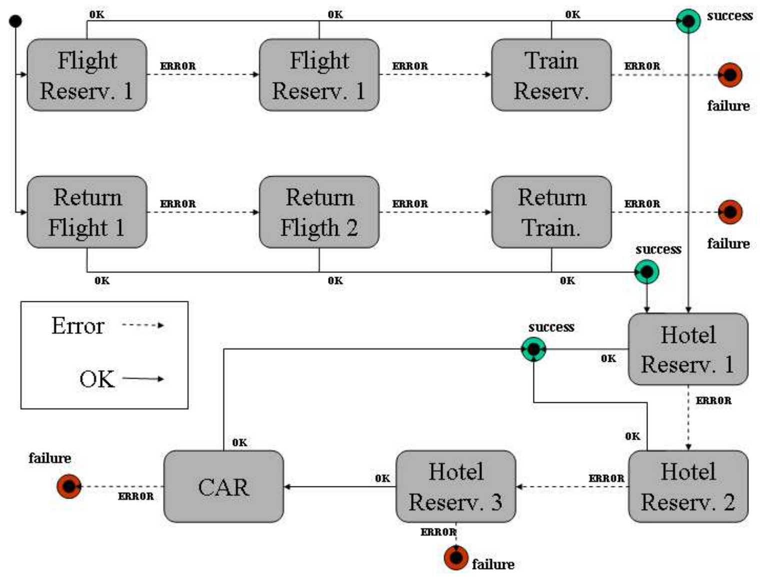


Figure 2. An example of a workflow-based application for arranging a conference trip. The user must reserve two flights (outward and return) before reserving the hotel for the conference. Note that, in the case that only the third hotel has available rooms, a car is needed and must be booked too.

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 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.

3.3 The Component Search Module

In the last years, the study of Web Search Engines has become a very important research topic. In particular, a large effort has been spent on Web models suitable for ranking query results [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 3 shows the overall architecture of our Component Search Engine, GRIDLE, a *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*. Its main modules are the *Component Crawler*, the *Indexer* and the *Query Answering*.

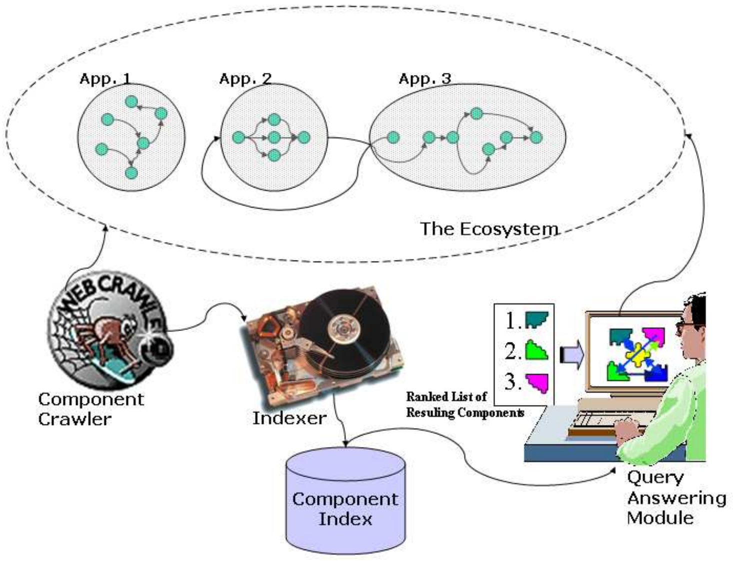


Figure 3. The architecture of GRIDLE.

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 is the *Query Answering* module, which actually resolves the components queries on the basis of the index. As other search engines, our 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 further 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 steps above in the development process of the example depicted in Figure 2. In a Grid software development environment a programmer could have sketched the abstract workflow plan graph depicted in Figure 4.

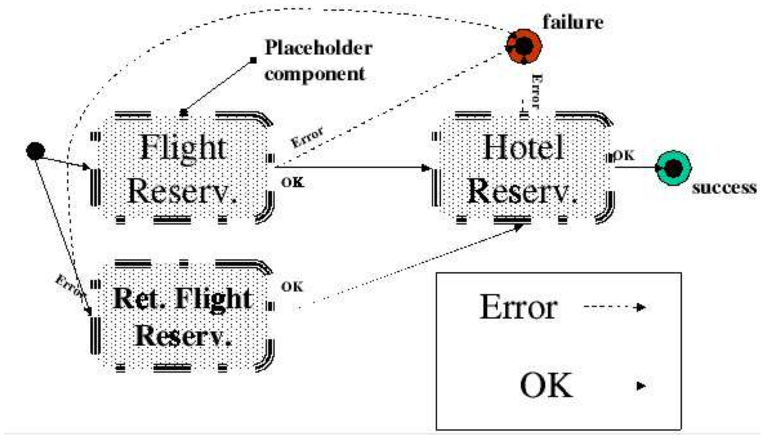


Figure 4. A partially specified workfbw graph, describing the application of Figure 2 at the highest level possible.

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 5 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.

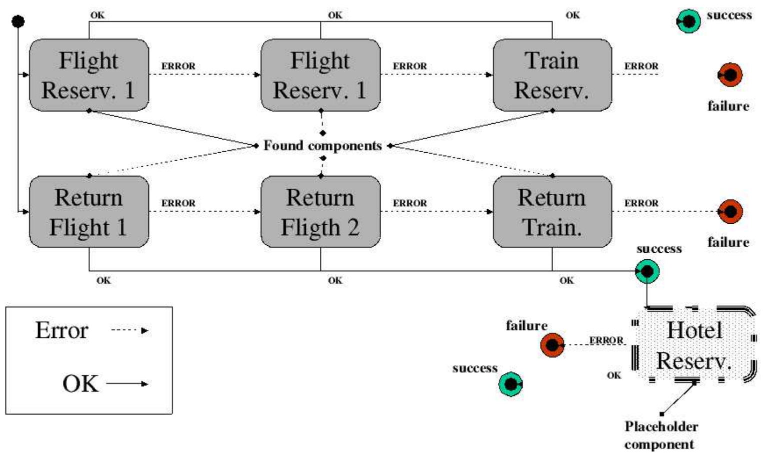


Figure 5. The abstract workflow graph as it appears after the "flight reservation component" has been found.

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, more relevant, list of components is obtained. Once the search phase is concluded, the user will pick up the most suitable component (*binding*). Finally, when all the components are fully specified, the developer will continue refining the application 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. 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.

3.4 Ranking Metrics

Valid ranking metrics are fundamental in order to have relevant search results out of the pool of known components. As said above (Section 2), the social structure of the component ecosystem is clearly the main source of information about component relevance: components that are used by several many other trusted components are clearly more relevant. Below, we illustrate two more metrics: reputation and XML similarity.

3.4.1 Building a Market: Trust and Reputation.

In this market of components, trust and reputation will play a very important role. Vendors that are known for their dependability and for offering a reliable service, will be preferred to newcomers, obscure providers or small vendors. Consider for instance buying a car: a big dealer will have a reputation built out of the comments of the buyers, and can charge a higher price in exchange for a reputable service. On the other side, one buyer can choose a less known seller, if the price is advantageous.

A tool called DBIN [19], developed at Università Politecnica delle Marche, is able to spread comments about resource across a big community of users. In other words, any user of a specific resource can add a comment to it, and this will reach all the other user in a very scalable pattern. Filters can be used to accept only comments from trusted peers.

This idea could be used to build the reputation of software components: each user will receive comments and suggestions about the software they are using or planning to use.

All this information can be considered in the ranking of components. A similar approach is taken by eBay[1], where vendors are ranked by the number of negative and positive feedback they received.

3.4.2 XML Interface Matching.

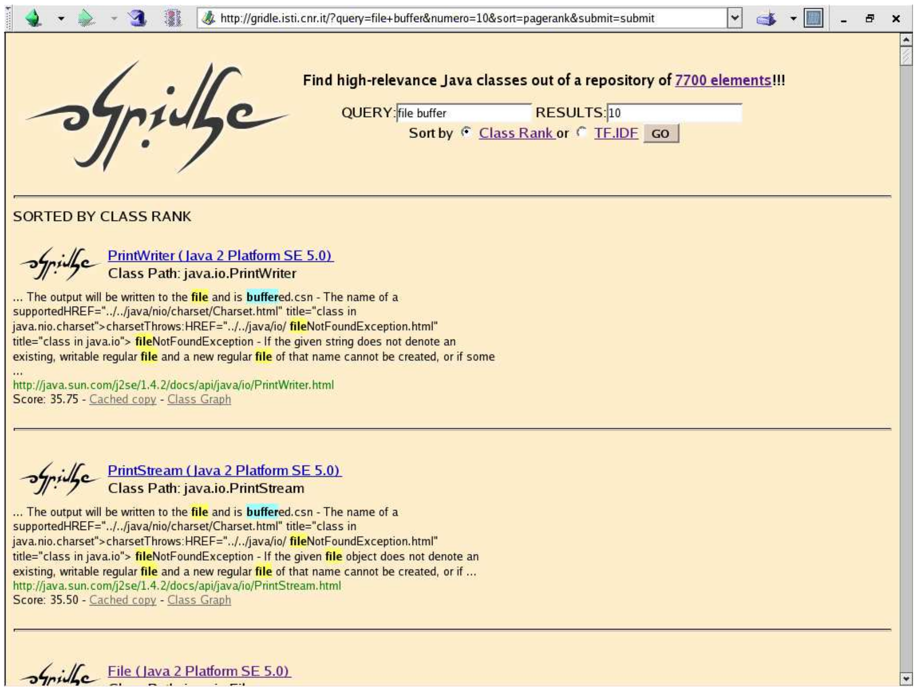
In order to offer an effective search tool, it is very important to offer a way to browse and analyze the known components. There are very important results about XML classification and clustering [8, 13]. We can build over these results the following way: component interfaces can be analyzed and transformed into XML documents, which can be clustered and organized using tools for computing XML similarity. A similar approach is taken in [21]to cluster computational resources out of a dynamic network.

Ranking can include the degree of similarity between each component and the place-holder designed by the developer.

3.5 Searching Java Classes

The Java Documentation is a very rich body of documentation about the Java API. It is publicly available and published with a very consistent format, which

[1]http://www.ebay.com/


http://gridle.isti.cnr.it/?query=file+buffer&numero=10&sort=pagerank&submit=submit

Find high-relevance Java classes out of a repository of 7700 elements!!!

QUERY: file buffer RESULTS: 10

Sort by [✓ Class Rank] or TF.IDF GO

SORTED BY CLASS RANK

PrintWriter (Java 2 Platform SE 5.0)
Class Path: java.io.PrintWriter

... The output will be written to the **file** and is **buffer**ed.csn - The name of a supportedHREF="../../java/nio/charset/Charset.html" title="class in java.nio.charset">charsetThrows:HREF="../../java/io/ **file**NotFoundException.html" title="class in java.io"> **file**NotFoundException - If the given string does not denote an existing, writable regular **file** and a new regular **file** of that name cannot be created, or if some ...
http://java.sun.com/j2se/1.4.2/docs/api/java/io/PrintWriter.html
Score: 35.75 - Cached copy - Class Graph

PrintStream (Java 2 Platform SE 5.0)
Class Path: java.io.PrintStream

... The output will be written to the **file** and is **buffer**ed.csn - The name of a supportedHREF="../../java/nio/charset/Charset.html" title="class in java.nio.charset">charsetThrows:HREF="../../java/io/ **file**NotFoundException.html" title="class in java.io"> **file**NotFoundException - If the given **file** object does not denote an existing, writable regular **file** and a new regular **file** of that name cannot be created, or if ...
http://java.sun.com/j2se/1.4.2/docs/api/java/io/PrintStream.html
Score: 35.50 - Cached copy - Class Graph

File (Java 2 Platform SE 5.0)

Figure 6. Web-interface of GRIDLE.

can be automatically processed by a machine. Out of this, we were able to identify social patterns among Java classes (see Section 2).

Out of our preliminary results, we developed a very simple search engine, able to find high-relevance classes out of our repository. Classes can be ranked by TF.IDF (a common information retrieval method, based on a metric that keeps into account both the number of occurencies of a term within each document and the number of documents in which the term itself appears) or by Class Rank, our version of PageRank for Java classes, based on the class usage links (see Section 2). Figure 6 shows the first web interface of our tool.

3.6 Component Search Engines: Related Work

In the last years, thanks to technologies like Internet and the Web, a number of interesting approaches [3, 16]to the problem of searching for software components, as well as a number of interesting papers analyzing new and existing solutions [9–10], have been proposed.

In [3], *Odyssey Search Engine* (*OSE*), a search engine for Components is presented. OSE is an agent system responsible for domain information (i.e., domain items) search within the Odyssey infrastructure. It is composed of an interface agent (*IA*), filtering agents (*FAs*), and retrieval agents (*RAs*). The IA is the agent responsible for display the search results according to the users' profile. These profiles are modeled by identifying groups of users with similar preferences, stereotypes and so forth. The FAs match user keywords and the textual description of each component, returning those with the greatest number of occurrences of user keywords. Finally, the RAs are the agents responsible for searching for relevant components among different domain descriptions.

In [16], the *Agora* components search engine is described. Agora is a prototype developed by the Software Engineering Institute at the Carnegie Mellon University. The object of this work is to create an automatically generated, indexed, worldwide database of software products classified by component type (e.g., JavaBean, CORBA or ActiveX control). Agora combines introspection with Web search engines in order to reduce the cost of bringing software components to, and finding components in, the software marketplace. It supports two basic functions: the location and indexing of components, and the search and retrieval of a component. For the first task, Agora uses a number of agents which are responsible for retrieving information through introspection. At the time the paper was written, Agora supported only two kind of components: JavaBeans, and CORBA components. The task of searching and retrieving components is split into two distinct steps: in the first, a keyword-based search is performed and then, once the results have been presented to the user, s/he can refine or broaden the search criteria, based on the number and quality of matches. interesting features of Agora is the capability of discovering automatically the sites containing software components. The technique adopted to automatically find components is quite straight-forward but appears to be effective. Agora simply crawls the Web, as a typical Web crawler does, and whenever it encounters a pages containing an `<APPLET>` tag, it downloads and indexes the related component.

In [9], Frakes and Pole analyze the results of an empirical experiment with a real Component Search Application, called *Proteus*. The study compares four different methods to represent reusable software components: attribute-value, enumerated, faceted, and keyword. The authors tested both the effectiveness and the efficiency of the search process. The tests were conducted on a group of thirty-five subjects that rated the different used methods, in terms of preference and helpfulness in understanding components. Searching effectiveness was measured with recall, precision, and overlap values drawn from the Information Retrieval theory [20]. Among others, the most important conclusion cited in the paper is that no method did more than moderately well in terms of search effectiveness, as measured by recall and precision.

In [10], the authors cite an interesting technique for ranking components within a set of given programs. Ranking a collection of components simply consists of finding an absolute ordering according to the relative importance of components. The method followed by the authors is very similar to the method used by the Google search engine [7]to rank Web pages: PageRank [4]. In ComponentRank, in fact, the importance of a component[2] is measured on the basis of the number of references (imports, and method calls) other classes make to it within the given source code.

Another interesting project is Prospector[3]. It is a search engine able to seek out code examples that use any or all of J2SE 1.4, Eclipse 3.0, and Eclipse GEF (Graphical Editing Framework) code. IBM is working with the U.C. Berkeley Computer Science Department to fund the venture with a fraction of its $1 billion annual developer budget. Prospector searches the graph for paths from the "have" class to the "want" class and then converts the paths into legal Java source code. The approach proposed by Prospector can be very interesting in designing tools for component bridging.

The Knowledge Grid project [6]at University of Calabria also shows an interesting strategy. It offers a development environment, called VEGA, where the user can sketch a Data Mining application to be run on a set of resources known as Knowledge Grid. Using an ad-hoc ontology for data mining application (representing data sources, algorithms and so on), the system can show the user a set of data bases and tools that the user can connect to run their application.

Our approach differs in that we want to limit the introduction of standards (i.e. ontologies) from above, but rather we want to utilize naturally emerging social patterns and links among existing software. Nonetheless, the features and goals of their environment are of strong interest.

4. 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.

When this becomes available, building a Grid application will become a straight-forward process. A non-expert user, aided by a graphical environment,

[2]Only Java classes are supported in this version.
[3]http://snobol.cs.berkeley.edu/prospector-bin/search.py

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.

Acknowledgments

This work has been partially supported by the MIUR GRID.it project (RBNE-01KNFP) and the MIUR CNR Strategic Project L 499/97-2000.

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] R.M.M. Braga, C.M.L. Werner, and M. Mattoso. Odysseysearch: An agent system for component. In *The 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, Portland, Oregon - USA, May 2003.

[4] S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.

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

[6] Mario Cannataro, Antonio Congiusta, Andrea Pugliese, Domenico Talia, and Paolo Trunfio. Distributed data mining on grids: Services, tools, and applications. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS PART B: CYBERNETICS*, 34:2451–2465, December 2004.

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

[8] Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, and Andrea Pugliese. Fast Detection of XML Structural Similarity. In *SEBD 2002*, pages 193–207, 2002.

[9] William B. Frakes and Thomas P. Pole. An empirical study of representation methods for - reusable software components. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 20(8):617–630, August 1994.

[10] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th international conference on Software engineering*, pages 14–24, Portland, Oregon, May 2003. IEEE, IEEE Computer Society.

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

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

[13] Andrew Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, 2002.

[14] 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.

[15] Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services, held in conjunction with The 14th International Conference on Automated Planning and Scheduling , (ICAPS 2004), Whistler, British Columbia, Canada, June 3-7 2004*, 2004. Available at http://www.isi.edu/ikcap/icaps04-workshop/.

[16] Robert C. Seacord, Scott A. Hissam, and Kurt C. Wallnau. Agora: A search engine for software components. Technical Report ESC-TR-98-011, Carnegie Mellon - Software Engineering Institute, Pittsburgh, PA 15213-3890, 1998.

[17] Fabrizio Silvestri, Diego Puppin, Domenico Laforenza, and Salvatore Orlando. Toward a search engine for software components. In *Proceedings of IEEE Web Intelligence*, Beijing, China, September 20-24, 2004.

[18] 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.

[19] Giovanni Tummarello, Christian Morbidoni, Joakim Petersson, Francesco Piazza, Mauro Mazzieri, and Paolo Puliti. Toward widely deployable semantic web p2p: tools, definitions and the rdfgrowth algorithm. In *ISWC'04 workshop on Semantic Web Technology for Mobile and Ubiquitous Applications, 7th November 2004, Hiroshima, Japan*, 2004.

[20] C.J. Van Rijsbergen. *Information Retrieval*. Butterworths, 1979. Available at http://www.dcs.gla.ac.uk/Keith/Preface.html.

[21] K. Vanthournout, G. Deconinck, and R. Belmans. A small world overlay network for resource discovery. In *Euro-Par 2004, Pisa, Italy, Aug-Sep 2004*, 2004.

[22] 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.