Service-Oriented Matchmaking and Brokerage

Tom Goodale[1], Simone A. Ludwig[1], William Naylor[2], Julian Padget[2] and Omer F. Rana[1]
[1]School of Computer Science/Welsh eScience Centre, Cardiff University
[2]Department of Computer Science, University of Bath

Abstract

The GENSS project has developed a flexible generic brokerage framework based on the use of plug-in components that are themselves web services. The focus in GENSS has been on mathematical web services, but the broker itself is domain independent and it is the plug-ins that act as sources of domain-specific knowledge. A range of plug-ins has been developed that offer a variety of matching technologies including ontological reasoning, mathematical reasoning, reputation modelling, and textual analysis. The ranking mechanism too is a plug-in, thus we have a completely re-targettable matchmaking and brokerage shell plus a selection of useful packaged behaviours.

1 Introduction

How does the e-scientist or the e-scientist's software agent find the web service that does what they want? In practice, the reality for the e-scientist may be more the result of social interaction than scientific evaluation. While collegial recommendation, as an approach, has a number of positive attributes, it also underlines the weaknesses of current service description languages and service discovery mechanisms if a user prefers to use other means to find the "right" web service. To facilitate the re-use of generic components and their combination with domain specific components, a new low-overhead approach to building matchmakers and brokers is required—and one that gains leverage from the currency of the grid: web services while also bringing the process and the control closer to the user. Significant work has already been undertaken to support information services, such as the Globus MDS, LDAP and recently registry services such as UDDI. Most of these systems however are based on an "asymmetric" relationship between a client and a provider – generally requiring the client to make query to a service provider, and the provider enforcing some policy after a suitable service has been discovered. Each of these systems are also restricted by the types of queries that they can support.

In this paper we describe an architecture that simplifies the deployment of bespoke matchmakers and brokers. The matchmaker is comprised of re-usable components, the use of which is demonstrate through a set of examples. Whether a user wants the function of a matchmaker—to find suitable candidate services—or of a broker—to select from the candidate services and invoke or even construct a workflow—depends on just how much the user wishes to trust in the intelligence of matching and ranking mechanisms. This is somewhat similar to hitting the "I'm feeling lucky" button in Google, except here the user is committing to the use of a grid resource and that may have cost implications.

The flexibility of the brokerage framework stems from the fact that its architecture involves a component based approach, which allows the integration of capabilities through the use of web services. Thus, constructing a new broker becomes a matter of composing a workflow involving: (i) a range of sources of service descriptions; (ii) a range of matching services that will accept a service request and a service description and output some information about the relationship between the two; (iii) a ranking service that can order the service matching results information to determine the best fit; (iv) a service to invoke the selected service and deliver the results. For the user that would prefer greater control, instead of a ranking service, there could be a presentation service that contacts the user to display a list of options from which they may select. Whichever option is taken, the broker components employed, and other decision-making data may be recorded as provenance data in the answer document. It is also crucial to be able to provide feedback about why a particular service did or did not match—a form of explanation—since it is not just a matter of the presence or absence of a keyword as it is for Google finding and ranking a page. In the same way that humans choose carefully and subsequently refine their inputs to Google, we may expect users to want to do the same in identifying web services.

But how can a user express what they want of a web service? Keywords might help narrow down the search but they do not offer a language for describing the compatibility requirements, such as what inputs and what outputs are wanted. Furthermore a statement of the signature of a service says next to nothing about the actual function; for that we require the statement of relationships between the inputs and outputs, or more generally, statements of pre- and post-conditions. The reasonable conclusion is that we need a combination of syntactic, semantic and even social mechanisms to help identify and choose the right services.

We can therefore observe that each service will have a functional interface (describing the input/outputs needed to interact with it and their types) and a non-functional interface (which identifies annotations related to the service made by other users and performance data associated with the service). Being able to support selection on both of these two interfaces provides a useful basis to distinguish between services.

Even when a service (or a composition of a set of services) has been selected, it is quite likely that their interfaces are not entirely compatible. Hence, one service may have more parameters than another, making it difficult to undertake an exact comparison based just on their interfaces. Similarly, data types used within the interface of one service may not fully match those of another. In such instances, it would be necessary to identify mapping between data types to determine a "degree" of match between the services. Although the selection or even the on-the-fly construction of shim services is something that could be addressed from the matchmaking perspective [6, 10], we do not discuss this issue further in this paper.

The remainder of the paper is laid as follows: (i) in the next-but-one section (3) we describe the architecture in detail and the design decisions that lead to it (ii) this is followed by a description of the range of plug-ins that have been developed during the MONET and GENSS projects and that are now being integrated through the KNOOGLE project (iii) the paper concludes with a survey of related work and a brief outline of developments foreseen over the next year.

2 eScience Relevance

The GENSS project has developed a flexible generic brokerage framework based on the use of plug-in components that are themselves web services. The focus in GENSS has been on mathematical web services, but the broker itself is domain independent and it is the plug-ins that act as sources of domain-specific knowledge. Thus, bespoke matchers/brokers can be created at will by clients, as we demonstrate later in conjunction with the Triana workflow system. Each broker takes into consideration the particular data model available within a domain, and undertakes matching between service requests and advertisements in relation to the data model. The complexity of such a data model can also vary, thereby constraining the types of matches that can be supported within a particular application. Within GENSS and its predecessor project MONET, a range of plug-ins were developed that offer a variety of matching technologies:

1. **Ontological reasoning:** input/output constraints are translated to description logic and a DL reasoner is used to identify query to service matches,
2. **Mathematical reasoning:** analyses the functional relationship expressed in the pre-conditions and effects of the service (can also be applied to service composition),
3. **Reputation modelling:** recommendations from the user's social network are aggregated to rank service recommendations,
4. **Textual analysis:** natural language descriptions of requests and services are analysed for related semantic information.

The purpose of the framework is to make brokerage technology readily deployable by the non-specialist, and more importantly, an effective but unobtrusive component for e-Science users. Specifically this indicates the following modes of use:

1. As a pre-deployed broker in a work-flow demonstrating pre-packaged functionality and utility
2. As a bespoke broker using pre-defined plug-ins demonstrating the construction of a new broker from existing services
3. The authoring and/or packaging of plug-in services other than those described above, demonstrating the flexibility and interoperability of the architecture
4. The exploration of meta-brokerage, where the web service description of broker components are published and selected by a meta-broker to instantiate new special-purpose brokers.

3 Service-oriented Matchmaking

3.1 Matchmaking Requirements

We begin by reiterating the basic requirements for matchmaking:

1. Sufficient input information about the task is needed to satisfy the capability, while the outputs of the matched service should contain at least as much information as the task is seeking, and
2. The task pre-conditions should at least satisfy the capability pre-conditions, while the post-conditions of the capability should at least satisfy the post-conditions of the task.

These constraints reflect work in component-based software engineering and are, in fact, derived from [23]. They are also more restrictive than is necessary for our setting, by which we mean that some inputs required by a capability can readily be inferred from the task, such as the lower limit on a numerical integration where by convention this is zero, or the dependent variable in a symbolic integration of a uni-variate function. Conversely, a numerical integration routine might only work from 0 to the upper limit, while the lower limit of the problem is non-zero. A capability that matches the task can be synthesised from the composition of two invocations of the capability with the fixed lower limit of 0. Clearly the nature of the second solution is quite different from the

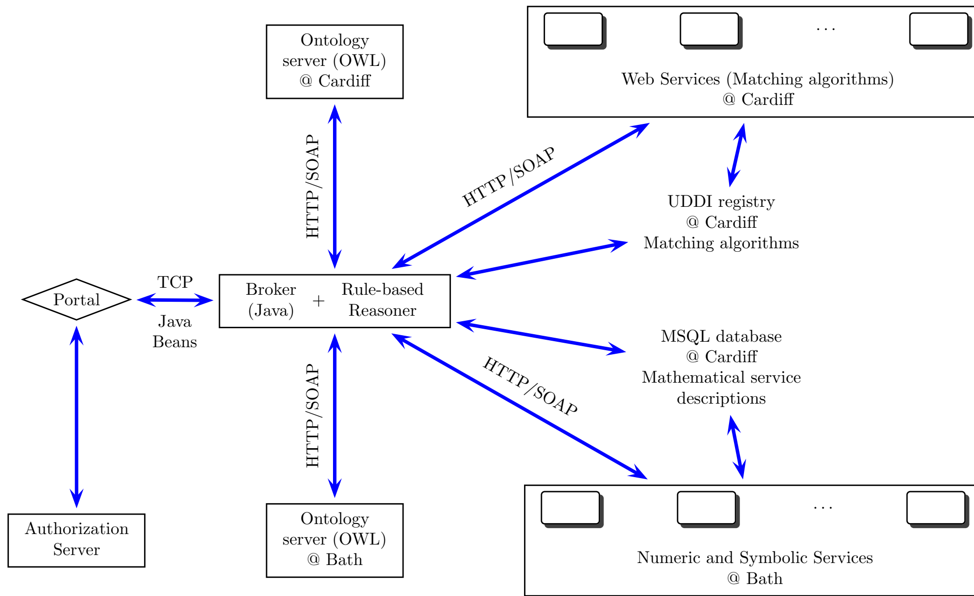

Figure 1: Architecture

first, but both serve to illustrate the complexity of this domain. It is precisely this richness too that dictates the nature of the matchmaking architecture, because as these two simple examples show, very different reasoning capabilities are required to resolve the first and the second. Furthermore, we believe that given the nature of the problem, it is only very rarely that a task description will match exactly a capability description and so a range of reasoning mechanisms must be applied to identify candidate matches. This results in:

> **Requirement 1:** A plug-in architecture supporting the incorporation of an arbitrary number of matchers.

The second problem is a consequence of the above: there will potentially be several candidate matches and some means of indicating their suitability is desirable, rather than picking the first or choosing randomly. Thus:

> **Requirement 2:** A ranking mechanism is required that takes into account pure technical (as discussed above in terms of signatures and pre- and post-condition) and quantitative and qualitative aspects—and even user preferences.

3.2 Matchmaking Architecture

Our matchmaking architecture is shown in Figure 1 and comprises the following:

1. The Authorization service, Portal: these constitute the client interface and are employed by users to specify their service request.
2. The Broker and Reasoner: these constitute the core of the architecture, communicating with the client *via* TCP and with the other components *via* SOAP.
3. The matchmaker: this is in part made up of a reasoning engine and in part by the matching algorithms, which define the logic of the matching process.
4. Mathematical ontologies: databases of OWL based ontologies, derived from OpenMath Content Dictionaries (CDs), GAMS (Guide to Available Mathematical Software) *etc.* developed during the MONET [15] project.
5. A Registry Service: which enables the storage of mathematical service descriptions, together with the corresponding endpoint for the service invocation.
6. Mathematical Web Services: available on third party sites, accessible over the Web.

There are essentially two use cases:

Use Case 1: *Matchmaking with client selection:* which proceeds as follows:

1. The user contacts the matchmaker.
2. The matchmaker loads the matching algorithms specified by the user via a look-up in the UDDI registry. In the case of an ontological match a further step is necessary. This is, the matchmaker contacts the reasoner which in turn loads the corresponding ontology.
3. Having additional match values results in the registry being queried, to see whether it contains services which match the request.
4. Service details are returned to the user via the matchmaker.

The parameters stored in the registry (a database) are service name, URL, taxonomy, input and output signatures,

pre- and post-conditions. Using contact details of the service from the registry, the user can then call the Web Service and interact with it.

Use Case 2: *Brokerage:* where the client delegates service selection via a policy statement. This proceeds essentially as above except that the candidate set of services is then analysed according to the client-specified policy and one service is selected and invoked.

Details of the various components of the architecture are discussed in [13].

4 Workflow integration

Workflow-based tools are being actively used in the e-Science community, generally as a means to combine components that are co-located with the workflow tool. Recently, extensions to these tools which provide the ability to combine services which are geographically distributed have also been provided. To demonstrate the use of the matchmaker as a service, we have integrated our Broker with the Triana workflow engine.

4.1 Triana

Triana was initially developed by scientists in GEO 600 [7] to help in the flexible analysis of data sets, and therefore contains many of the core data analysis tools needed for one-dimensional data analysis, along with many other toolboxes that contain units for areas such as image processing and text processing. All in all, there are around 500 units within Triana covering a broad range of applications. Further, Triana is able to choreograph distributed resources, such as web services, to extend its range of functionality. Additional web service-based algorithms have also been added recently to Triana to support data mining [17]. Triana may be used by applications and end-users alike in a number of different ways [21]. For example, it can be used as a: graphical workflow composition system for Grid applications; a data analysis environment for image, signal or text processing; as an application designer tool, creating stand-alone applications from a composition of components/units; and through the use of its pluggable workflow representation architecture, allowing third party tool and workflow representation such as WSDL and BPEL4WS.

The Triana user interface consists of a collection of toolboxes containing the current set of Triana components and a work surface where users graphically choreograph the required behaviour. The modules are late bound to the services that they represent to create a highly dynamic programming environment. Triana has many of the key programming constructs such as looping (do, while, repeat until etc.) and logic (if, then etc.) units that can be used to graphically control the dataflow, just as a programmer would control the flow within a conventional program using specific instructions. Programming units (i.e. tools) include information about which data-type objects they can receive and which ones they output, and Triana uses this information to perform design-time type checking on requested connections to ensure data compatibility between components; this serves the same purpose as the compilation of a program for compatibility of function calls.

Triana has a modularized architecture that consists of a cooperating collection of interacting components. Briefly, the thin-client Triana GUI connects to a Triana engine (Triana Controlling Service, TCS) either locally or via the network. Under a typical usage scenario, clients may log into a TCS, remotely compose and run a Triana application and then visualize the result locally – even though the visualization unit itself is run remotely. Alternatively, clients may log off during an application run and periodically log back on to check the status of the application. In this mode, the Triana TCS and GUI act as a portal for running an application, whether distributed or in single execution mode.

4.2 Triana Brokering

To support matchmaking for numerical services in Triana, the broker has been included within Triana in two ways:

1. A service in the toolbox: as illustrated in figure 2, the broker is included as a “search” service within the Triana toolbox. In order to make use of the service, it is necessary for a user to instantiate the service within a workflow, including a “WSTypeGen” component before the service, and a “WSTypeViewer” after the service. These WSType components allow conversion of data types into a form that can be used by a web service. Double clicking on the WSTypeGen component generates the user interface also shown in figure 2, requiring a user to chose a match mode (a “structural” mode is selected in the figure), specify pre- and post-conditions, and the query using OpenMath syntax. Once this form has been completed, hitting the “OK” button causes the search service to be invoked, returning a URL to the location of a service that implements the match. If multiple matches are found, the user receives a list of services and must manually select between them (as shown in figure 3). If only a single service is found, the location of a WSDL file may be returned, which can then be used by a subsequent service in the workflow (if the matchmaker is being used as a broker).
2. A menu item: in this case, the search service is invoked from a menu item, requiring the user to complete a form, as shown in figure 3. The user specifies the query using the form also shown in the figure, and selects a matching mode (a “Structural match” is selected in the figure). The result is displayed in a separate window for the user to evaluate. In this instance, it is not necessary to match any data types before and after a match service, and some additional components will need to

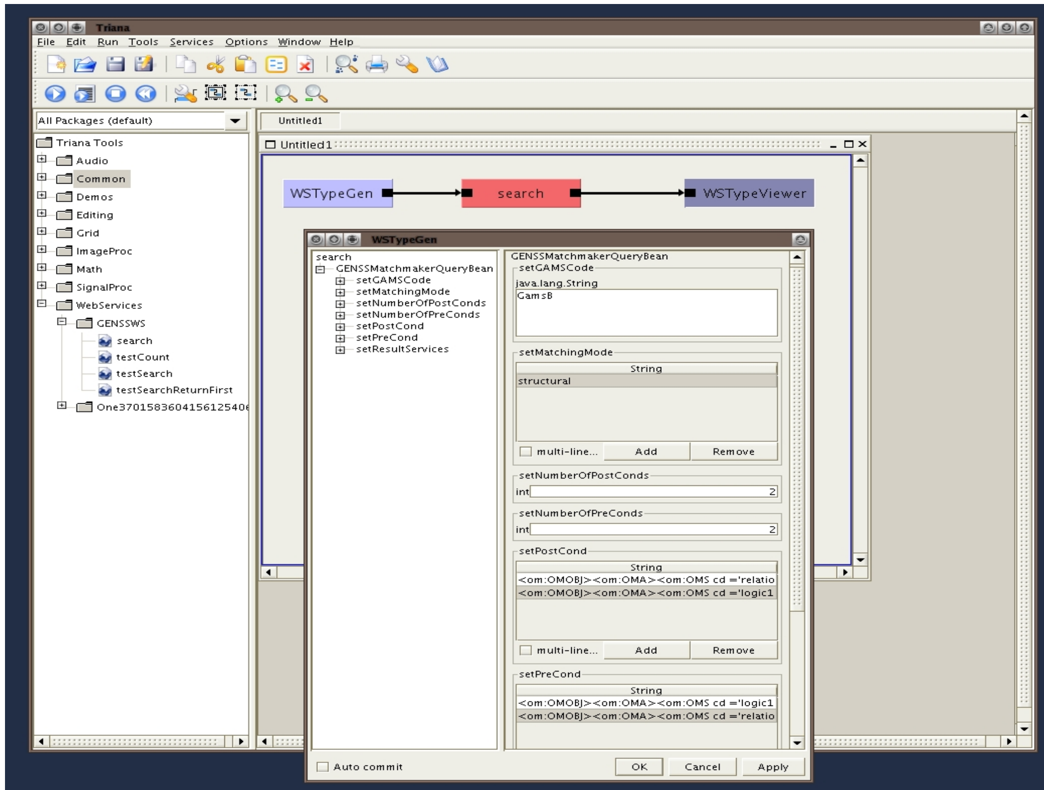

Figure 2: A Triana workflow with matchmaker

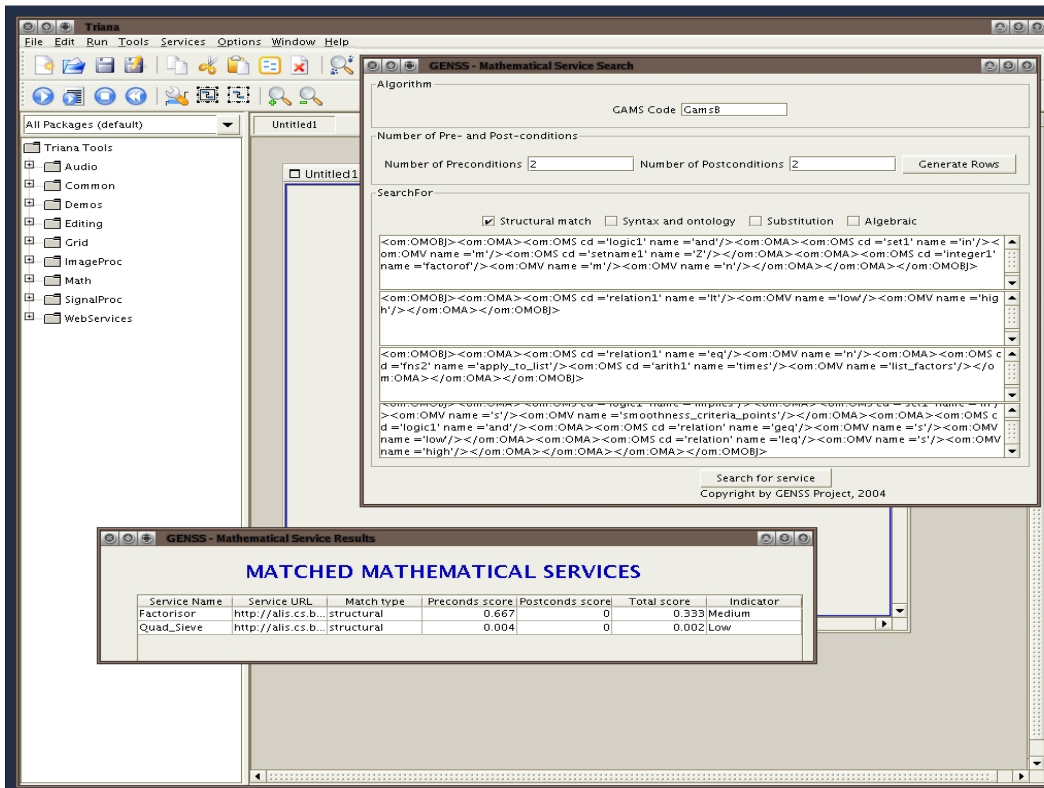

Figure 3: Results from the matchmaker in Triana

be implemented by the user to achieve this. This mode of use is particularly useful if a user does not intend to use the matchmaker within a workflow, but prefers to investigate services of interest, before a workflow is constructed graphically.

Using the matchmaker as a search service allows it to be used as a standard component in Triana. In this case, a user constructs a workflow with a search service being a component in this workflow. During enactment, the search service invokes the matchmaker and returns one or more results. Where a single result is returned (i.e. where the matchmaker is being used as a broker), the workflow continues to the next service in graph without any requirement for user input. As Triana already allows for dynamic binding of components to service instances, the search service essentially provides an alternative form of dynamic binding.

5 Complexity issues

An important issue for brokerage systems is whether the system scales well with respect to the number of services registered with the system. That is, the number of services should only have a small factor in any expression calculating the complexity of the system. For the brokerage system described in this paper, the complexity costs will be dependant on the complexity costs of the matchmaker plugins to the service. Generally the more powerful the plugin, the higher the complexity cost. One class of matchmaker plugins, that we utilise is ontology based plugins, these generally involve some traversal of an ontology and as such, the costs will be dependant more on the height than the absolute size of the ontology. Some figures indicating the actual performance of our system appear in Appendix A

6 Planned Future Work

A particular system which manages allocation of jobs to a pool of processors, is the GridSAM system [8]. The GridSAM system takes job descriptions given in JSDL [2] and the executables from Clients. It then distributes the jobs over the processors which perform the processing, GridSAM then returns the results to the Clients. It also provides certain monitoring services. Grimoires [9] is a registry service for services, which allows attachments of semantic descriptions to the services. Grimoires is UDDIv2 [1] compliant and stores the semantic attachments as RDF triples, this gives scope for attachments with arbitrary schema including those for describing mathematical services. A restriction that has been identified with the current GridSAM architecture, is that it doesn't incorporate any brokerage capabilities. Furthermore it appears that the Grimoires registry does not provide the resource allocation provided by GridSAM. A future project will look at integration of these two approaches in such a manner that it can be used in coordination with the architecture described in this paper.

7 Related Work

Matchmaking has quite a significant body of associated literature, so we do not attempt to be exhaustive, but survey just those systems that have been influential or we believe are especially relevant to the issues raised here, namely architecture, flexibility and matching technologies.

Although generalizations are risky, broad categorizations of matchmaking and brokerage research seem possible using criteria such as domain, reasoning mechanisms and adaptability.

Much of the published literature has described generic brokerage mechanisms using syntactic or semantic, or a combination of both, techniques. Some of the earliest systems, enabled by the development of KIF (Knowledge Interchange Format) [5] and KQML (Knowledge Query and Manipulation Language) [20], are SHADE [11] operating over logic-based and structured text languages and the complementary COINS [11] that operates over free text using well-known term-first index-first information retrieval techniques. Subsequent developments such as InfoSleuth [14] applied reasoning technology to the advertised syntax and semantics of a service description, while the RETSINA system [19] had its own specialized language [18] influenced by DAML-S (the pre-cursor to OWL-S) and used a belief-weighted associative network representation of the relationships between ontological concepts as a central element of the matching process. While technically sophisticated, a particular problem with the latter was how to make the initial assignment of weights without biasing the system inappropriately. A distinguishing feature of all these systems is their monolithic architecture, in sharp contrast to GRAPPA [22] (Generic Request Architecture for Passive Provider Agents) which allows for the use of multiple matchmaking mechanisms. Otherwise GRAPPA essentially employs fairly conventional multi-attribute clustering technology to reduce attribute vectors to a single value. Finally, a notable contribution is the MathBroker architecture, that like the domain-specific plug-ins of our brokerage scheme, works with semantic descriptions of mathematical services using the same MSDL language. However, current publications [3] seem to indicate that matching is limited to processing taxonomies and the functional issues raised by pre- and post-conditions are not considered. The MONET broker [4], in conjunction with the RACER reasoner and the Instance Store demonstrated one of the earliest uses of a description logic reasoner to identify services based on taxonomic descriptions coming closest to the objective of the plug-ins developed for GENSS in attempting to provide functional matching of task and capability.

In contrast, matching and brokerage in the grid computing

domain has been relatively unsophisticated, primarily using syntactic techniques, such as in the ClassAds system [16] used in the Condor system and RedLine [12] which extends ClassAds, where match criteria may be expressed as ranges and hence are a simple constraint language. In the Condor system, the use of ClassAds is to enable computational jobs find suitable resources, generally using dynamic attributes such as available physical and virtual memory, CPU type and speed, current load average, and other static attributes such as operating system type, job manager etc. A resource also has a simple policy associated with it, which identifies when it is willing to accept new job requests. The approach is therefore particular focused to work for managing job execution on a Condor pool, and configured for such a system only. It would be difficult to deploy this approach (without significant changes) within another job execution system, or one that makes use of a different resource model. The RedLine system allows matching of job requests with resource capabilities based on constraints – in particular the ability to also search based on resource policy (i.e. when a resource is able to accept new jobs, in addition to job and resource attributes). The RedLine description language provides functions such as *Forany* and *Forall* to be able to find multiple items that match. The RedLine system is however still constrained by the type of match mechanisms that it supports—provided through its description language. Similar to Condor, it is also very difficult to modify it for a different resource model. Our approach is more general, and can allow plug-ins to be provided for both RedLine and Condor as part of the matchmaker configuration. In our model, therefore, as the resource model is late-bound, we can specify a specialist resource model and allow multiple such models to co-exist, each implemented in a different configuration of the matchmaker.

8 Conclusion

We have outlined the development of a brokerage architecture whose initial requirements were derived from the MONET broker, namely the discovery of mathematical services, but with the addition of the need to establish a functional relationship between the pre- and post-conditions of the task and the capability. As a consequence of the engineering approach taken in building the GENSS matchmaker/broker, the outcome has been a generic matchmaking shell, that may be populated by a mixture of generic and domain-specific plug-ins. These plug-ins may also be composed and deployed with low overhead, especially with the help of workflow tools, to create bespoke matchmakers/brokers. The plug-ins may be implemented as web services and a mechanism has been provided to integrate them into the architecture. Likewise, the use of web services for the plug-ins imposes a low overhead on the production of new components which may thus encourage wider authorship of new, shareable, generic and specific matching elements (as reported in the Appendix). This would also provide the basis for defining a policy about how results from multiple matching techniques may be combined.

9 Acknowledgements

The work reported here is partially supported by the Engineering and Physical Sciences Research Council under the Semantic Grids call of the e-Science program (GENSS grant reference GR/S44723/01) and partially supported through the Open Middleware Infrastructure Institute managed program (project KNOOGLE).

References

[1] A.E. Walsh. UDDI, SOAP, and WSDL: The Web Services Specification Reference Book, 2002. UDDIORG.

[2] Stephen McGough Ali Anjomshoaa, Darren Pulsipher. Job Submission Description Language WG (JSDL-WG), 2003. Available from `https://forge.gridforum.org/projects/jsdl-wg/`.

[3] Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. A Web Registry for Publishing and Discovering Mathematical Services. In *EEE*, pages 190–193. IEEE Computer Society, 2005.

[4] Olga Caprotti, Mike Dewar, and Daniele Turi. Mathematical Service Matching Using Description Logic and OWL. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2004.

[5] M. Genesereth and R. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical report, Computer Science Department, Stanford University, 1992. Available from `http://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps`.

[6] C. Goble. Putting Semantics into e-Science and Grids. *Proc E-Science 2005, 1st IEEE Intl Conf on e-Science and Grid Technologies, Melbourne, Australia, 5-8 December*, 2005.

[7] EO 600 Gravitational Wave Project. `http://www.geo600.uni-hannover.de/`.

[8] GridSAM - Grid Job Submission and Monitoring Web Service, 2006. Available from `http://gridsam.sourceforge.net/2.0.0-SNAPSHOT/index.html`.

[9] (Grimoires: Grid RegIstry with Metadata Oriented Interface: Robustness, Efficiency, Security , 2004. Available from `http://twiki.grimoires.org/bin/view/Grimoires/`.

[10] Duncan Hull, Robert Stevens, Phillip Lord, Chris Wroe, and Carole Goble. Treating "shimantic web" syndrome with ontologies. In John Domingue, editor, *First Advanced Knowledge Technologies workshop on Semantic Web Services (AKT-SWS04)*, volume 122. KMi, The Open University, Milton Keynes, UK, 2004. Workshop proceedings available from CEUR-WS.org. ISSN:1613-0073. `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-122/`.

[11] D. Kuokka and L. Harada. Integrating information *via* matchmaking. *Intelligent Information Systems 6(2-3), pp. 261-279*, 1996.

[12] Chuang Liu and Ian T. Foster. A Constraint Language Approach to Matchmaking. In *RIDE*, pages 7–14. IEEE Computer Society, 2004.

[13] Simone Ludwig, Omer Rana, William Naylor, and Julian Padget. Matchmaking Framework for Mathematical Web Services. *Journal of Grid Computing*, 4(1):33–48, March 2006. Available via `http://dx.doi.org/10.1007/s10723-005-9019-z`. ISSN: 1570-7873 (Paper) 1572-9814 (Online).

[14] W. Bohrer M. Nodine and A.H. Ngu. Semantic brokering over dynamic heterogenous data sources in InfoSleuth. In *Proceedings of the 15th International Conference on Data Engineering, pp. 358-365*, 1999.

[15] Mathematics on the Net - MONET. `http://monet.nag.co.uk`.

[16] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *HPDC*, pages 140–, 1998.

[17] O. F. Rana, Ali Shaikh Ali, and Ian J. Taylor. Web Services Composition for Distributed Data Mining. In D. Katz, editor, *Proc. of ICPP, Workshop on Web and Grid Services for Scientific Data Analysis, Oslo, Norway June 14*, 2005.

[18] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Journal of Autonomous Agents and Multi Agent Systems*, 5(2):173–203, June 2002.

[19] Katia P. Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph A. Giampapa. The RETSINA MAS Infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.

[20] D. McKay T. Finin, R. Fritzson and R. McEntire. KQML as an agent communication language. In *Proceedings of 3rd International Conference on Information and Knowledge Management, pp. 456-463*, 1994.

[21] Ian Taylor, Matthew Shields, Ian Wang, and Roger Philp. Grid Enabling Applications using Triana. *Workshop on Grid Applications and Programming Tools. In conjunction with GGF8. Organized by: GGF Applications and Testbeds Research Group (APPS-RG) and GGF User Program Development Tools Research Group (UPDT-RG)*, 2003.

[22] D. Veit. *Matchmaking in Electronic Markets*, volume 2882 of *LNCS*. Springer, 2003. Hot Topics.

[23] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

A GENSS performance indicators

Performance indicators for the GENSS matchmaker are reported here. An "end-to-end" test, with a user in Bath and a registry in Cardiff, produced a wall-clock time of 8 seconds. A "soak" test to repeatedly run the same search (client/matching process – as a Web Service – on a machine in Bath with registry in Cardiff) produced the following results:

Structural matcher

Iterations	real	user	sys
1	0m14.495s	0m4.362s	0m0.172s
10	1m2.898s	0m5.766s	0m0.318s
100	9m59.015s	0m12.173s	0m1.566s

Indicating that amortized elapsed time/query is around 6 seconds using the structural matcher.

Ontological matcher

Iterations	real	user	sys
1	0m22.089s	0m4.786s	0m0.173s
10	1m49.031s	0m5.882s	0m0.299s
100	17m13.894s	0m12.682s	0m1.203s

Indicating that amortized elapsed time/query is just over 10 seconds using the ontological matcher. Performing the same test as above, but where a command line Java application was used to search the service registry, linux system monitoring tools report that on the machine carrying out the search process that program used approximately 25.4MB and 36.06% CPU.