Surfer: An Extensible Pull-Based Framework for Resource Selection and Ranking*

Paul Z. Kolano
NASA Advanced Supercomputing Division, NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035 U.S.A.

E-mail: kolano@nas.nasa.gov

Abstract

Grid computing aims to connect large numbers of geographically and organizationally distributed resources to increase computational power, resource utilization, and resource accessibility. In order to effectively utilize grids, users need to be connected to the best available resources at any given time. As grids are in constant flux, users cannot be expected to keep up with the configuration and status of the grid, thus they must be provided with automatic resource brokering for selecting and ranking resources meeting constraints and preferences they specify. This paper presents a new OGSI-compliant resource selection and ranking framework called Surfer that has been implemented as part of NASA's Information Power Grid (IPG) project. Surfer is highly extensible and may be integrated into any grid environment by adding information providers knowledgeable about that environment. Surfer invisibly and seamlessly correlates results from different providers into a single unified view seen by the user.

1. Introduction

Grid computing [4] aims to connect large numbers of geographically and organizationally distributed resources to increase computational power, resource utilization, and resource accessibility. In order to effectively utilize grids, users need to be connected to the best available resources at any given time. In small, single organization grids, users may be able to adequately select resources for simple requests on their own. Even small grids, however, are in constant flux with resource availability changing minute by minute due to user activity and system failures. Large, multi-organization grids are much more chaotic where resource types, connectivity, support levels, software environments, availability, etc. may vary greatly with a greater probability of resource failures. In such an environment, users cannot possibly keep up with the configuration and status of the grid nor potentially even which resources they may have access to, thus they will tend to choose only those resources they are directly familiar with. This leads to imbalanced resource utilization, which results in longer wait times and a decrease in productivity. To maximize the benefits of grid computing, users must be provided with automatic resource brokering for selecting and ranking resources meeting constraints and preferences they specify.

Although resource brokering is a fundamental service that is vital to the usability of every grid environment, it is difficult to provide a general purpose solution for all such environments since each one has its own idiosyncrasies such as job models, resource types, and sources of information. Typical resource brokers are tightly entangled with these idiosyncrasies, eliminating the possibility of reuse across the grid community. By removing any dependence on such idiosyncrasies, and, specifically, decoupling the access to resource information from its utilization in the resource selection process, it becomes possible to provide a general framework upon which resource brokers for any environment can be built. The key features required in such a framework include an expressive and extensible constraint language, easy integration and correlation of new resource types and information sources, and accommodation of preexisting information retrieval optimizations.

This paper presents Surfer, the **S**election and **R**anking **F**ramework for **E**xtracting **R**esources. The job of Surfer is to surf the pool of potential grid resources and extract the highest ranked resources meeting user specified constraints and preferences. Surfer has no built-in bias towards any job model or selection policy, thus is suitable for inclusion in any grid environment by adding information providers knowledgeable about that environment. These providers determine the types of resources that are selectable and sup-

*This work is supported by the NASA Advanced Supercomputing Division under Task Order A61812D (ITOP Contract DTTS59-99-D-00437/TO #A61812D) with Advanced Management Technology Inc.

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

ply functions constrainable in resource requests. Provider functions are allowed to have arbitrary argument and return types or may be macros to be expanded before processing. Function definitions may hide arbitrarily complex back-end information retrieval that may be optimized as desired. Surfer has been implemented as an OGSI-compliant grid service that can also be embedded directly into Java applications through its APIs or into non-Java applications through its XML-based command-line interface.

Surfer is part of NASA's Information Power Grid (IPG) project [7]. The goal of the IPG is to develop new technologies to facilitate the use of the grid and enable scientific discovery. Several prototype services have been implemented including the Execution Service for submitting and managing jobs, the Prediction Service [18] for estimating execution, wait, and transfer times, the Cardea [10] service for dynamic resource access control, the Naturalization Service [8] for automatically establishing the execution environment for user applications, and the Surfer framework, which is the subject of this paper.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the selection and ranking framework. Section 4 describes a prototype resource broker implemented for the IPG using this framework. Finally, section 5 presents conclusions and future work.

2. Related Work

The most well known resource selection framework is the Condor matchmaker [15]. In this framework, providers and consumers describe their properties and requirements as classified advertisements (classads), which are pushed to a central matchmaker that does the matching. Each classad is a mapping between attributes and values that may additionally contain a constraint, which describes the requirements that must be met by any matching classads, and a ranking function, which describes the order of preference when several classads satisfy the constraint.

The original matchmaking framework only allows a request to be matched with a single classad, but has been extended in several ways to overcome this limitation. In [12], a request may be matched to a set of classads of the same type, where requests may contain aggregate requirements that all classads of the set must satisfy. The aggregation functions include min, max, and sum. In [16], classads have been extended by allowing them to contain multiple *ports*, each of which may be matched with other classads of specific types and characteristics. Thus, a single request may be matched with multiple heterogeneous classads.

For matchmaking schemes to work, all parties must utilize the same vocabulary of attribute names and values. To facilitate interoperability between disparate vocabularies, [19] extends matchmaking with ontologies, which allows conditions to be defined under which differing attribute values may still be matched (e.g. "Linux" and "FreeBSD" match "Unix"). This approach, however, does not yet support multi-classad matching. RedLine [11] incorporates all three classad extensions by allowing heterogeneous classad sets, aggregate set requirements, and ontological vocabularies. Requests are specified as a set of constraints, which are solved as a constraint satisfaction problem.

All of the matchmaking models suffer from the same limitation. Namely, they rely on a push-based model of information, where every possible attribute of interest must be computed a priori and stored within a classad to be utilized during selection. Many attributes critical to resource selection, however, are too dynamic or complex to precompute and/or too large to store temporarily. Examples include access control, which may be completely dynamic based on a resource's current state and the user's grid identity as in [10] and network bandwidth, which may include measurements between a fully connected network of thousands of resources. In general, a more flexible pull-based model is needed to utilize such information, which can be computed on demand to control size and complexity.

A variety of resource brokers are available with their respective grid environments/scheduling systems. Examples include resource brokers for Legion [2], European Data Grid [9], ALICE Environment [17], UNICORE [13], Fraunhofer Resource Grid [6], Nimrod/G [1], and GridLab [14]. In general, these brokers were designed for their specific grid environment, thus are not easily extensible, nor are easily incorporated into other environments. In some cases, they are dependent on a specific job model or job submission functionality. In other cases, they are dependent on specific information sources that may not be available elsewhere. Finally, many of them have built-in selection policies such as a specific load-balancing scheme that may be unsuitable and/or undesirable in other environments.

3. Surfer Framework

Surfer is a framework for the selection and ranking of resources, where a resource is considered to be any entity that may require selecting such as computers, storage systems, software, data, etc. Figure 1 shows the architecture of Surfer, which is described in greater detail in section 3.5. Processing begins when a user or client application makes a resource selection request. The request is rewritten from a boolean expression to a set expression utilizing function calls and queries to providers that supply information. The rewritten request is then evaluated into an actual set of resource sets, which is finally ranked and returned to the user. The following sections describe this process in detail.

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

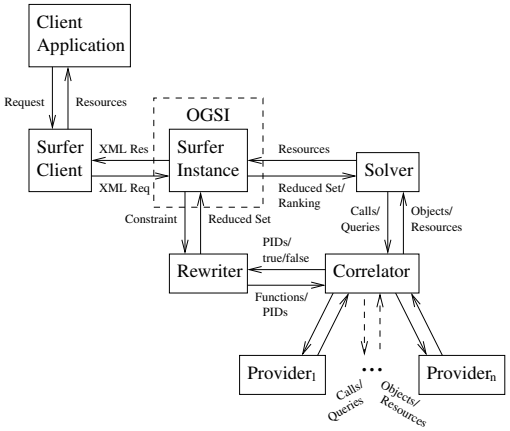

Figure 1. Surfer architecture

3.1. Requests

A Surfer request consists of a set of individual resource requests, a global constraint, a global ranking function, and a number of resource tuples to return. Each individual request consists of an identifier, a resource type, a local constraint, and a local ranking function. The global constraint describes the requirements that must be met by the complete set of resources while each local constraint describes the requirements that must be met by any resources selected for that particular resource request. Similarly, the global ranking function describes how complete resource sets should be ordered while each local ranking function describes how individual resources should be ordered. Note that the global/local distinction is purely for notational convenience and natural encapsulation as resources are evaluated against the *composite constraint* in which all global/local constraints are conjoined together and the *composite ranking function* in which all global/local ranking functions are multiplied together.

Figure 2 shows an example request for two resources: a storage resource with more than 20 GB of free disk space and a compute resource running IRIX64 with at least 128 free CPUs and more than 10 GB of free physical memory, where the host name of each resource must be the same. The final set of resource pairs must be ordered by the value of 100 times the compute resource's free CPUs plus the storage resource's free disk space. Note that this figure is only a text representation of a request object and does not depict a specific syntax for making requests with the exception of the constraints and the ranking functions.

Constraints and ranking functions are written in typed first-order logic without quantification, which consists of constants, variables, functions, and boolean, arithmetic, and relational operators. The constants include numbers,

```
Global:
  Constraint:
      $c1.host == $s1.host
  Ranking:
      100 * $c1.freeCpus +
       $s1.freeMb
Resource:
  Id: s1
  Type: StorageResource
  Constraint:
      freeMb > 20000

Resource:
  Id: c1
  Type: ComputeResource
  Constraint:
        freeCpus >= 128
   && freePhysicalMemoryMb > "10G"
   && operatingSystem == "IRIX64"
```

Figure 2. Resource Broker request

strings, and the boolean values true and false. The variables consist of the set of resource identifiers given in the individual requests and are used to reference resource attributes. In figure 2, the variables are "c1" and "s1", which are used in the global constraint to reference the host name of the compute and storage resource, respectively.

Supported operators include standard boolean, arithmetic, and relational operators as well as "contains" and "!contains" for strings and collection types such as sets and lists and the inline conditional operator "?:" a la C and Java. All of the arithmetic, relational, and set operators are extensible based on the Number, Comparable, and Collection classes of Java, respectively. That is, any operands conforming to these interfaces may be used. For the relational operators, if only one operand conforms to the Comparable interface, but its associated type has a constructor based on the other operand type, an appropriate instance will be created to make the two operands comparable to each other. This is especially useful for handling different unit types. For example, in figure 2, the "freePhysicalMemoryMb" function returns a Comparable-conforming type "MemorySize", which has both Number and String constructors allowing memory sizes such as "10G" to be easily normalized and compared to purely numeric values.

The key element of the specification language is the set of functions available, which varies depending on the *providers* that have been integrated into the system. Surfer has no built-in functions. Every function available is defined by a provider that has access to the information that function supplies. Figure 2 shows some of the functions available in the example resource broker discussed in section 4, which was developed using the Surfer framework. In the figure, "freeCpus", "freePhysicalMemoryMb", "operatingSystem", "freeMb", and "host" are all functions supplied by a specific provider. Users are not required to know which provider supplies which function. Surfer invisibly and seamlessly correlates results from different providers to give a single unified view of the functions available.

While Surfer requests are similar to the gangmatch classad requests of [16], its pull-based model of information

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

allows its language to be significantly more expressive. Information that is too large, complex, or dynamic to be precomputed and pushed to a central matchmaker, can be easily utilized by calling an appropriate provider function. Surfer also does not suffer from the vocabulary mismatch issues associated with classads as users can only utilize the specific strongly-typed functions that are supplied by providers hooked into the system.

The result of a Surfer request is a *tuple set*, which is a set of tuples such that each tuple is a map from variable names (i.e. resource ids) to resources that has been optimized for space and intersection performance. Figure 7 shows three example tuple sets. Each column shows a different variable value. For example, S2 has three tuples with variables v1 and v2 where (v1, v2) takes the values (a, a), (b, a), and (b, b). The tuple set resulting from a request contains the specified number of tuples and is ordered by value of the composite ranking function from highest to lowest. Each tuple contains a resource selection for each resource id requested and satisfies the conjunction of all specified constraints.

3.2. Providers

The core of the resource brokering framework are the providers that supply functions that may be constrained and queries that may be executed to produce resources with appropriate attribute values. Functions can be defined to take objects of any number and class type as arguments and to return objects of any class type. This allows significant flexibility and arbitrary extensions to the constraint and ranking language. Functions may also be *macro*s, which are processed during rewriting and transform strings to strings. Macros provide an easy abstraction mechanism for hiding complexity and grouping commonly used expressions.

Queries take a set of variable names and a boolean expression utilizing those variables and return a tuple set of resources satisfying the expression. The set of functions supplied by a provider defines the functions that are constrainable within the boolean expression of queries to that provider.

Providers are not required to supply queries, but users can only request resource types for which there exists a query in some provider. This is by necessity as only queries supply raw sets of resources. Resources are considered to be objects with a set of identifying attributes that are unique to each resource (e.g. host and directory for a storage resource) and a set of dynamic attributes that may vary over time (e.g. free disk space). Figure 3 shows the Java interface that every resource must implement. The getAttributes method must return the map from attribute names to values. The isMergeable method takes a resource and must return whether that resource has the same identifying attributes as the current resource. Finally, the mergeAttributes method takes a resource and merges its attributes into the current resource's attributes. This method is used to merge the attributes of resources with the same identifying attributes that may have been produced by different providers during evaluation. A BaseResource class provides default implementations for getAttributes and mergeAttributes.

```
public interface Resource {
    public Map getAttributes();
    public boolean isMergeable(Resource res);
    public boolean mergeAttributes(Resource res);
}
```

Figure 3. Resource interface

Figure 4 shows the Java interface that every provider must implement, which consists of two methods for functions and two for queries. The getFunctions method must return the set of functions that the provider supplies. The hasQuery method takes a list of resource types and must return a boolean indicating whether that type of query is supported. The callFunction methods takes a function name and an array of arguments and must produce an actual value based on these arguments. The runQuery method takes a set of variable names and a boolean expression utilizing those variables and must return the resources satisfying that expression. Section 3.5 describes how new providers are integrated into the framework.

```
public interface Provider {
    public Set getFunctions();
    public boolean hasQuery(List types);
one of { public Object callFunction(String name, Object[] args);
         public Object[] callFunctions(String name, Object[] argArrays);
one of { public TupleSet runQuery(Set vars, AST ast);
         public List runQueries(List varSets, List asts);
}
```

Figure 4. Provider interface

Providers are responsible for optimizing access to their back-end information sources. In most cases, this involves caching results for future use. A simple built-in caching optimization supplied by the framework is to cache the dynamic attributes of any resources returned in queries. When this is done, functions based on these attributes can be called with minimal cost. The attributes are kept consistent across different providers using the isMergeable and mergeAttributes methods of the Resource interface during evaluation. For some providers, it may also be possible to increase overall throughput by processing function calls and queries in batches. This is especially likely in cases where a provider uses other services on different hosts to

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

implement its back-end functionality. To avoid limiting any potential provider optimizations, providers can implement alternative batch interfaces for callFunction and runQuery. A BaseProvider class provides implementations of each in terms of the other, thus whichever is not implemented will be based on the one that is.

3.3. Rewriting

In order to produce an appropriate tuple set from a boolean constraint, that constraint must first be rewritten into an expression using the functions and queries that the providers supply. Four set operations are utilized to describe these expressions: intersections, unions, *queries*, and *reductions*. Intersections and unions are defined as normal but with special handling for tuple sets. A query takes a provider id, a set of variables, and a boolean expression in those variables and returns a tuple set such that each tuple has a resource mapping for every variable that together satisfy the boolean expression. A reduction takes a tuple set and a boolean expression and returns the subset of the given set that satisfies the boolean expression. In general, a query is more efficient than a reduction as each element in a reduction set may necessitate a call to multiple providers.

Rewriting begins with a request's composite constraint, where conjunctions are changed to intersections, disjunctions to unions, and negations are distributed along the way. True constants are changed to universal sets (i.e. the set representing all possible resource combinations) and false constants to empty sets. Any macros are expanded by calling the appropriate provider functions. Relations are changed to queries if all functions within the relation belong to the same provider and that provider supports queries in the requisite number and types of variables. All other relations are changed to reductions on the universal set, with each function transformed into an explicit call to an explicit provider.

Although the expression generated from this initial transformation could be evaluated as is, several optimizations are possible. In particular, it is desirable to (1) eliminate universal sets as they are expensive to compute, (2) minimize the number of queries to run by combining queries to the same provider, and (3) minimize the number of function calls by minimizing the size of each reduction set.

To achieve these goals, the unification axioms shown in figure 5 are applied to the initial set expression. Lower numbered axioms are applied before higher numbered axioms. Axiom 1 actually consists of four separate axioms, the most important of which is the second, which will eliminate a universal set when it is intersected with anything else. Axioms 2 and 3 are used to combine queries when possible. Axioms 4 and 5 are used to minimize reduction set size. Although axiom 5 can always be applied in place of axiom 4, it is undesirable to choose the order in which the reductions occur at this point since the number of elements in each set is unknown until they are actually evaluated. Finally, axiom 6 is used both to eliminate universal sets and to minimize reduction set size by pushing intersections into unions to maximize the impact of the other axioms. The end result of rewriting will be an expression of the form $\bigcup_i reduce_i(\bigcap_j query_j, \bigwedge_k t_k)$.

1. $A \cap \emptyset = \emptyset \qquad A \cap 1 = A \qquad A \cup \emptyset = A \qquad A \cup 1 = 1$
2. $query(p, V_1, t_1) \cap query(p, V_2, t_2) = query(p, V_1 \cup V_2, t_1 \wedge t_2)$ when p has queries in $V_1 \cup V_2$
3. $query(p, V_1, t_1) \cup query(p, V_2, t_2) = query(p, V_1 \cup V_2, t_1 \vee t_2)$ when p has queries in $V_1 \cup V_2$
4. $reduce(A, t_1) \cap reduce(B, t_2) = reduce(A \cap B, t_1 \wedge t_2)$
5. $A \cap reduce(B, t) = reduce(A \cap B, t)$
6. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Figure 5. Unification axioms

Figure 6 shows the composite constraint of figure 2 after rewriting to reduced set form. All of the macros have been expanded from an easily readable form to the specific names used by one of the providers. Note that "freeCpus" has been expanded to the value representing 100 times the number of free CPUs in the last minute divided by 100. This is an example of how complex details can be hidden from the user by supplying appropriate macros. All universal sets have been eliminated, queries have been combined as much as possible, and the one reduction set has been minimized by bringing all other terms inside. Note that the two remaining queries cannot be combined even though their providers are the same because that particular provider does not support queries of multiple types at once.

```
reduce(query(pid2, {s1}, $s1.Mds_Fs_freeMB > 20000)
    ∩ query(pid2, {c1},
                $c1.Mds_Cpu_Total_Free_1minX100 / 100 >= 128
            && $c1.Mds_Memory_Ram_Total_freeMB > "10G"
            && $c1.Mds_Os_name == "IRIX64"),
      call(pid2, Mds_Host_hn, [$c1]) == call(pid2, Mds_Host_hn, [$s1]))
```

Figure 6. Rewritten constraint

3.4. Evaluation

After a request has been rewritten into reduced set form, it must be evaluated to produce an actual tuple set, which must then be ordered according to the composite ranking function. This involves utilizing the functions and queries supplied by the providers and computing the results of set, arithmetic, and relational operations. Although the rewriter

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

has performed expression-level optimizations, it is the responsibility of the solver to optimize the actual evaluation of rewritten form. The main goals of optimization are to (1) utilize available provider optimizations, (2) run identical queries only once, and (3) control set expansion.

To take advantage of any available provider optimizations, function calls and queries are always grouped into batches and sent to the batch interfaces of callFunction and runQuery in the providers. Identical queries may result from the application of axiom 6 in figure 5. To reduce the negative impact of this axiom, queries are gathered, run once, and the results duplicated where necessary.

Since Surfer allows an arbitrary number of resources to be selected at once, the number of tuples that can be generated during evaluation is exponential in the number of resources requested as each position within the tuple can potentially take any resource value of the appropriate type. Thus, the most critical optimization of the solver is to control this complexity to the greatest extent possible. This exponential expansion occurs during intersections between tuple sets with disjoint variable spaces as each tuple in one set may generate a new tuple for each tuple of the other set. Unions do not have this problem as the new set size is guaranteed to be at most the sum of the sizes of the two sets.

Although the final size of an intersection involving multiple sets is fixed, the sizes of the intermediate sets may vary with the order in which the individual intersections are evaluated. The intermediate sizes affect the total number of tuples that must be compared, thus directly affect execution time. Figure 7 shows three tuple sets and the number of tuple intersections required for each evaluation order. As can be seen, even for small sets, the number of tuple intersections is significantly impacted by the evaluation order.

S1	S2		S3			
v1	*v1*	*v2*	*v2*	*v3*	$\cap$ order	$\cap$'s
a	a	a	a	a	$(S1 \cap S2) \cap S3$	7
	b	a	a	b	$(S1 \cap S3) \cap S2$	16
	b	b	b	a	$(S2 \cap S3) \cap S1$	18
			b	b		

Figure 7. Intersection order

Surfer optimizes intersection evaluation order in two ways. The basic assumption of these optimizations is that sets will share variables that help reduce the resulting set size. When intersections occur outside of a reduction, the evaluation order is selected according to *estimated set size*. For two sets S1 and S2, let the set of variables shared between tuples of S1 and S2 be denoted by V and the maximum number of resources possible for each v_i in V be denoted by σ_i. The estimated set size of $S1 \cap S2$ is then defined as $|S1| \cdot |S2| / \prod_{i=1}^{|V|} \sigma_i$. That is, for each tuple of S1, the number of tuples of S2 that can be expected to match that tuple and be added to the resulting set decreases by a factor of σ_i for each variable shared. Thus, sets that are very sparse will be intersected before denser sets and sets with more shared variables will be intersected before those with less. For simplicity, the implementation uses the same σ_i for all resource types, which may be configured as described in section 3.5.

When intersections occur within reductions, a different strategy is used. After rewriting, all reductions will be in the form $reduce(\bigcap_{j=1}^{q} S_j, \bigwedge_{k=1}^{r} t_k)$. This form can be rewritten to $reduce(reduce(\bigcap_{i=1}^{s} S_i, t_1) \cap \bigcap_{j=s+1}^{q} S_j, \bigwedge_{k=2}^{r} t_k)$ when no variable of term t_1 is a variable of any set S_j. Thus, any intermediate result can be reduced by an individual term as long as that result contains all of the sets relevant to that term. In this case, the evaluation order is chosen based on terms. Terms are chosen in ascending number of variables and by involved estimated set size when the number of variables is the same. The idea is that the fewer variables a term involves, the fewer sets will have to be intersected before they can be reduced. After a term is chosen, intersections are performed as in the non-reduction case.

Figure 8 shows the time required to generate different numbers of resource tuples by intersection on a 750 MHz Pentium 3 system with 256 MB of memory running FreeBSD. The curves show the three different methods by which the given number of tuples were generated based on an initial request for four resources with 120 possible values for each. In the queries only case, each of the four resource dimensions was reduced by the same factor using a constraint on each resource variable (e.g. $s1.freeMb > 10000). In the reductions only case, relations between pairs of variables were added to reduce the possible set size (e.g. $s1.freeMb + $s2.freeMb > 10000). Finally, in the queries and reductions case, both techniques were used. When the number of resource tuples generated is small compared to the number of tuples possible, reductions have a higher overhead than queries. When the number of tuples generated becomes high enough, however, only the intersections dominate the computation.

Although the intersection times for even fairly large sets are reasonable, once the final tuple set is generated, each tuple must still be evaluated against the composite ranking function. If the resulting set is very large and the ranking function is complex, this evaluation may take significant time. To guard against this possibility, Surfer provides a configurable threshold that limits the maximum number of tuples that are evaluated when intersecting two sets. When more than the threshold tuple pairs are to be evaluated, a sampling function is generated that selects threshold pairs of tuples from the two sets. Those tuples are intersected

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

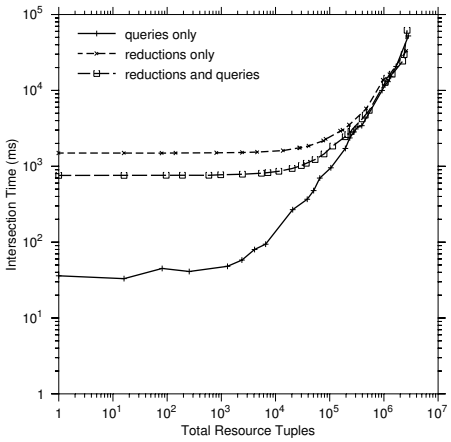

Figure 8. Intersection time vs. # tuples

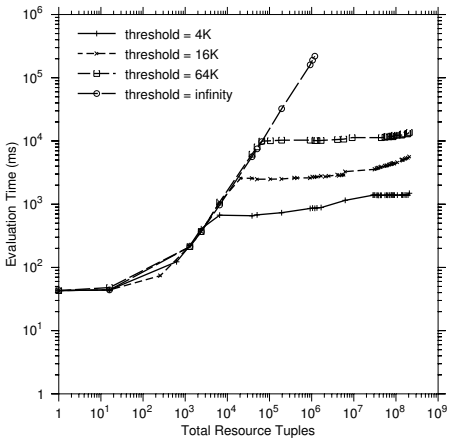

Figure 9. Evaluation time vs. # tuples

to produce the resulting set, while the remaining tuples are discarded.

With a threshold limitation comes a loss of precision as all possible resource combinations are no longer evaluated. There is also no guarantee how many tuples the resulting set will contain since providing such a guarantee could potentially require every tuple pair to be evaluated. In an attempt to keep the most desirable tuples, Surfer uses the composite ranking function as its sampling function. Each tuple set is ranked individually according to this function and then the square root of the threshold number of the highest ranked tuples of each set are intersected. Since the ranking function may reference resources that are not defined in intermediate tuple sets, the evaluation routines replace undefined values with constants.

Once the final tuple set has been produced, the last step of evaluation is to order this set by the composite ranking function. First, the solver's evaluation routines are used to compute a rank for each tuple. The tuple set is then sorted based on these values. Finally, the requested number of the highest ranked tuples are returned to the user. Figure 9 shows the total evaluation time of the queries only case of figure 8 for different threshold values. As can be seen, the evaluation time is relatively constant once the threshold is reached allowing a maximum acceptable response time to be set based on the threshold, if desired.

3.5. Implementation

Surfer is implemented in Java as an Open Grid Services Infrastructure (OGSI) compliant service within the Open Grid Services Architecture (OGSA) framework [5]. In the OGSA model, all grid functionality is provided by named *grid services* that are created dynamically upon request. The reference implementation of OGSI is the Globus Toolkit [3], which provides grid security through the Grid Security Infrastructure (GSI), low-level job management through the Globus Resource Allocation Manager (GRAM), data transfer through the Grid File Transfer Protocol (GridFTP), and resource/service information through the Monitoring and Discovery Service (MDS).

Figure 1 shows the architecture of the Surfer brokering framework. In this figure, a client application uses the Surfer client API to request a given number of resources of specific types and characteristics. This request is converted to XML and transmitted to an instance of the Surfer service running within an OGSI container. The rewriter component is then used to rewrite the request into a single constraint and from there to the reduced set form of that constraint. This expression is then given to the solver component, which evaluates the given set. Any calls and queries are evaluated using the correlator component, which provides a common interface for accessing all the provider functionality. The solver also ranks the resource tuples based on the ranking function given by the client application. Finally, the resulting set is returned back to the client and converted from XML form back to a tuple set of resources meeting the specified characteristics.

Extending Surfer from a framework into a usable resource broker for a specific grid environment involves two steps. First, a set of providers must be implemented that conform to the Provider interface of section 3.2. These providers should reflect the information available in the given grid environment. The higher the quality of information supplied within the functions of these providers, the higher the quality of the selection results. Once the provider implementations are ready, the second step is modifying Surfer's Web Service Definition Language (WSDL) parameters. These parameters include the sampling threshold, the expected size measure σ_i, and, most importantly,

In *Proc. of the 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, Chicago, IL, Apr. 19-22, 2004

the providers parameter, which supplies the fully qualified class names of the provider implementations. The providers parameter is used by the correlator component to load all the available providers into the framework.

After configuration, Surfer is available as a usable resource broker. Users or client applications can request the types of resources that are selectable, the functions that are constrainable, or can make a request to return resources of the appropriate types with specific characteristics.

4. IPG Resource Broker

An initial prototype of a resource broker for the NASA IPG has been developed using the Surfer framework. Eventually, this broker will have providers for most of the IPG services under development and will allow constraints on resource access based on Cardea, software locations, versions, and dependencies based on the Naturalization Service, wait, execution, and transfer predictions based on the Prediction Service, etc. For the initial prototype, two providers were implemented: an MDS provider and an MDS macro provider. The MDS provider supplies functions and queries based on the fields available in the MDS (version 2) servers of the IPG. The MDS macro provider supplies macros to abstract the sometimes cryptic names of MDS into more human readable forms.

The MDS provider supplies information about two types of resources: compute resources and storage resources. Compute resources consist of a host name, a queue name, and a queue type while storage resources consist of a host name and a directory name. Together, the two providers supply 80 functions with 65 of them coming from the MDS provider and 15 from the MDS macro provider.

Figure 2 shows a request to the IPG Resource Broker. This request uses only functions from the MDS macro provider, which are expanded to functions of the MDS provider as shown in the rewritten request of figure 6. This request examined 10,920 resource combinations and ran in 6.14 seconds, the bulk of which was associated with querying six separate MDS servers.

5. Conclusions and Future Work

This paper has described Surfer, the **S**election and **R**anking **F**ramework for **E**xtracting **R**esources. Surfer allows new information providers and resource types to be easily and seamlessly integrated into the system and allows arbitrary extensions to its constraint and ranking language through provider functions and flexible evaluation of built-in arithmetic, relational, and set operators. Its pull-based model allows information sources too large and/or too complex for push-based models to be efficiently incorporated into the resource selection process. By decomposing resource brokering into a framework independent of any specific grid environment, Surfer simplifies resource broker development and can reduce duplication of effort across the grid community.

There are a number of directions for future research. Some efficiency and accuracy improvements may be possible in the evaluation routines of the solver. One optimization is to allow providers to give the cost of calling their functions, which may allow term reductions to be evaluated in a more efficient order. Another possibility is to allow user-defined sampling functions for pruning intermediate resource sets since sampling based on the composite ranking function may not always yield optimal results. Finally, the accuracy of estimated set size computations may be improved by using different heuristics or by allowing the expected size measure σ_i to be configurable for each resource type or be automatically derived during evaluation.

Another area requiring more study is handling conflicts between providers such as different providers that supply the same information about the same resources or supply the same information, but about different subsets of resources. The former case could potentially be handled by allowing providers to supply a freshness measure to their information while an aggregate provider class may handle the latter.

Additional functionality may be added to the specification language as necessary such as a richer set of built-in operators. Finally, The IPG resource broker will be greatly enhanced in the near future. Namely, a variety of new information providers will be integrated into the system as they become available as part of the IPG project.

References

[1] Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. 4th Intl. Conf. on High Performance Computing, May 2000.

[2] Chapin, S.J., Katramatos, D., Karpovich, J., Grimshaw, A.S.: The Legion Resource Management System. 5th Wkshp. on Job Scheduling Strategies for Parallel Processing, Apr. 1999.

[3] Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Intl. J. Supercomputer Applications. 11(2) (1997) 115-128.

[4] Foster, I., Kesselman, C. (eds.): The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann, San Francisco, CA (1999).

[5] Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Ar-

chitecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, Jun. 2002.

[6] Hoheisel, A, Der, U.: An XML-based Framework for Loosely Coupled Applications on Grid Environments. 3rd Intl. Conf. on Computational Science, Jun. 2003.

[7] Johnston, W.E., Gannon, D., Nitzberg, B.: Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. 8th IEEE Intl. Symp. on High Performance Distributed Computing, Aug. 1999.

[8] Kolano, P.Z.: Facilitating the Portability of User Applications in Grid Environments. 4th IFIP Intl. Conf. on Distributed Applications and Interoperable Systems, Nov. 2003.

[9] Lee, W., McGough, S., Newhouse, S., Darlington, J.: Load-balancing EU-DataGrid Resource Brokers. UK e-Science All Hands Meeting, Sep. 2003.

[10] Lepro, R.: Cardea: Providing Support for Dynamic Resource Access in a Distributed Computing Environment. 19th Annual Computer Security Applications Conf., Dec. 2003.

[11] Liu, C., Foster, I.: A Constraint Language Approach to Grid Resource Selection. Technical Report TR-2003-07, Dept. of Computer Science, Univ. of Chicago, Mar. 2003.

[12] Liu, C., Yang, L., Foster, I., Angulo, D.: Design and Evaluation of a Resource Selection Framework for Grid Applications. 11th IEEE Intl. Symp. on High Performance Distributed Computing, Jul. 2002.

[13] MacLaren, J.: Resource Management and Resource Brokering Using UNICORE. Global Grid Forum 7 Wkshp. on Grid Scheduling Architecture, Mar. 2003. Available at `http://www.gridsched.org/ggf7/GGF7_Talk3.ppt`.

[14] Nabrzyski, J.: GridLab Resource Management System. Global Grid Forum 7 Wkshp. on Grid Scheduling Architecture, Mar. 2003. Available at `http://www.gridsched.org/ggf7/GGF7_Talk2.ppt`.

[15] Raman, R., Livny, M., Solomon, M.: Matchmaking: Distributed Resource Management for High Throughput Computing. 7th IEEE Intl. Symp. on High Performance Distributed Computing, Jul. 1998.

[16] Raman, R., Livny, M., Solomon, M.: Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. 12th IEEE Intl. Symp. on High Performance Distributed Computing, Jun. 2003.

[17] Saiz, P., Buncic, P., Peters, A.J.: AliEn Resource Brokers. Conf. for Computing in High Energy and Nuclear Physics, Mar. 2003.

[18] Smith, W., Foster, I., Taylor, V.: Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. 5th Wkshp. on Job Scheduling Strategies for Parallel Processing, Apr. 1999.

[19] Tangmunarunkit, H., Decker, S., Kesselman, C.: Ontology-based Resource Matching in the Grid - The Grid Meets the Semantic Web. 1st. Wkshp. on Semantics in Peer-to-Peer and Grid Computing, May 2003.