Distributed Policy Management and Comprehension with Classified Advertisements

Nicholas Coleman, Rajesh Raman, Miron Livny and Marvin Solomon
University of Wisconsin, 1210 West Dayton Street, Madison, WI 53703
{ncoleman,miron,solomon}@cs.wisc.edu, dr_rajesh_raman@yahoo.com

February 11, 2003

Abstract

Distributed systems present new challenges to resource management, which cannot be met by conventional systems that employ relatively static resource models and centralized allocators. Matchmaking paradigms based on identifying compatible classified advertisements (ClassAds) placed by providers and requesters of services work well in such environments. Condor is a production-quality distributed system with a distributed policy model that uses ClassAds for matchmaking. However, due to the distributed polices and dynamics of such environments, understanding why some ClassAds are not matched while others are can be a very complex task. We therefore present algorithms which not only identify problematic aspects of a policy, but also suggest modifications.

1 Introduction

A fundamental challenge of distributed systems is collaboration in highly dynamic, heterogeneous and untrusted environments. In such environments the service providers and requesters which comprise the federation must be able to specify policies which define the conditions under which they will collaborate. The difficulties involved in accommodating such a framework include a notation for *specifying* such usage policies, a method for *discovering* entities compatible with the stated policy, a simple yet flexible and scalable architecture to *implement* policies and analysis technologies to *comprehend* the implications of policies.

In this paper, we describe distributed policy management and comprehension in the context of Condor, a production-quality distributed high throughput computing system architected on a federated model, developed at the University of Wisconsin-Madison. The emphasis on policy management in Condor is on the specification and implementation of *resource allocation policy*. Condor operates in highly dynamic environments characterized by distributed management and distributed ownership. Distributed management introduces *resource heterogeneity*: Not only the set of available resources, but even the set of resource types is constantly changing. Distributed ownership introduces *policy heterogeneity*: Each requester has a unique idiosyncratic notion of what it requires of a provider, and *vice versa*.

Condor solves these problems by adopting a matchmaking paradigm. Users who request machines to run jobs and administrators who provide these machines to users are the primary principals. Agents for these principals send the Matchmaker structures called *classified advertisements* (ClassAds), which are declarative descriptions of the principal's characteristics, constraints and preferences. The Matchmaker uses a generic policy-neutral algorithm to discover and notify compatible agents, who activate a claiming protocol to establish a collaboration. Thus, in contrast to many conventional resource management systems, Condor does not impose a monolithic allocation and scheduling model on the resources in its purview. Instead, users and administrators independently and dynamically define allocation policies, and their jobs and machines respectively form dynamic collaborations when matched, realizing an *opportunistic computing* paradigm.

The simplicity and flexibility of this distributed policy approach has been validated in practice—Condor has been successfully deployed in both academic and industrial environments as a production quality system. Experience has shown that the ClassAd-based Matchmaking framework enables the description of sophisticated policies to accurately represent the expectations of the system's users. However, we have also discovered that understanding why certain ClassAds are not matched (while others are) can be a very complex task in the presence of complex policies and environment dynamics. In order to help users understand why ClassAds do not match, and therefore their jobs do not run, we have developed algorithms which not only identify problems, but also suggest modifications to the ClassAds.

2 Matchmaking

The underlying ideas of the matchmaking paradigm are intuitive and very simple. In this section, we briefly describe the fundamental processes and components of our matchmaking framework. Interested readers are referred to [12] for further details.

In our framework, human users who participate directly or indirectly in the system are called *principals*, and the software programs that represent principals in the system are called *agents*. Server and customer agents requiring matchmaking services express characteristics, constraints and preferences to a Matchmaker (illustrated as Step 1 in Figure 1). We call these agent descriptions *classified advertisements* in analogy to their newspaper counterparts. The task of the Matchmaker is to detect compatible advertisements in a generic manner (Step 2), which is performed by checking the constraints specified in the respective advertisements. When compatibilities are discovered, the Matchmaker notifies the respective advertising agents, discards the matched ClassAds and relinquishes any further responsibility for the match (Step 3). Matched agents then establish an allocation through a claiming process that does not involve the Matchmaker (Step 4).

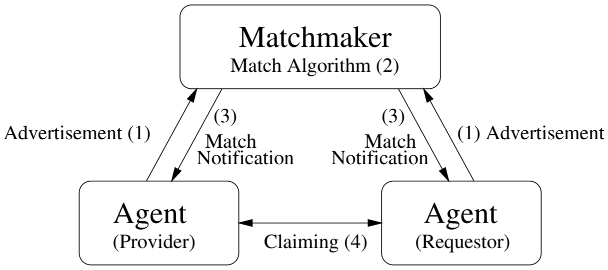

Figure 1: Actions involved in the Matchmaking process

Our matchmaking framework can be naturally decomposed into the following components:

1. A language for specifying the characteristics, constraints and preferences of agents. Our framework uses the *classified advertisement* (ClassAd) language for this purpose. ClassAds are semi-structured [9] sets of $(name, expression)$ pairs which may be thought of as “attribute lists” that describe agents.

2. The *Matchmaker Protocol* describes how agents communicate with the Matchmaker to post advertisements and receive notifications.

3. The *Matchmaking Algorithm* is used by the Matchmaker to create matches. In the abstract, the matchmaking algorithm transforms the contents of submitted advertisements and the state of the system to the set of matches created.

4. *Claiming Protocols* are activated between matched parties to confirm the match and establish a working relationship.

The simplicity, flexibility and expressiveness of the ClassAd language greatly contributes to the effectiveness of our Matchmaking framework. Figures 2 shows a ClassAd describing a workstation in the University of Wisconsin-Madison Condor [5] pool.[1]

Most attributes of the workstation (e.g, `Name`, `Memory`, `OpSys`) describe the machine's characteristics. The `Requirements` and `Rank` attributes are of special interest to the Matchmaker since these attributes identify the advertising agent's constraints and preferences. When testing the compatibility and preferences of two advertisements A and B, the Matchmaker places the two advertisements in an evaluation environment such that in ClassAd A, the reference `other` evaluates to B, and *vice versa*. Thus, the workstation in Figure 2 has the following policy: Jobs belonging to user “riffraff” are never accepted, and jobs are only serviced when the machine has a low load average and its console has been idle for at least fifteen minutes. Furthermore, jobs with small image sizes are preferred between 9 a.m. and 5 p.m. Similarly, the job shown in Figure 3 requires an INTEL workstation running the LINUX operating system with at least 128 MB of memory. Among all such workstations, the job prefers machines with better KFLOPS ratings and more memory.

Many interesting and useful policies may be easily defined within this framework; interested readers are referred to [13] for more sophisticated examples derived from the policies of real-world users of the Condor system.

3 ClassAd Analysis

Occasionally in Condor a submitted job's ClassAd does not match with any machine ClassAds. This situation occurs when none of the machines meet the submitted job's requirements, when the job does not meet the requirements of the machine candidates, or a combination of these two circumstances. In this paper we will treat the first two problems seperately and assume that they are not related.[2]

Requirements expressions are most commonly in *disjunctive normal form* (DNF) with atoms in the form of

[1] The Wisconsin Condor pool is currently composed of over 800 nodes, running nine different architecture/operating system combinations. The pool is used continuously as a production system to provide computation services for several research projects.

[2] An example where this is not the case: the job requirements expression contains the predicate `other.Memory >= ImageSize` and the requirements expressions of the machine ClassAds reference the job attribute `ImageSize`.

```
[
  Type        = "Machine";
  Activity    = "Idle";
  KeybrdIdle  = '00:23:12'; // h:m:s
  Disk        = 323.4M;     // mbytes
  Memory      = 256M;        // mbytes
  State       = "Unclaimed";
  LoadAvg     = 0.042969;
  Mips        = 104;
  Arch        = "INTEL";
  OpSys       = "LINUX";
  KFlops      = 21893;
  Name        = "foo.cs.wisc.edu";
  Subnet      = "128.105.175";
  Rank        = DayTime() >= '9:00' &&
            DayTime() <= '17:00' ?
            1/other.ImageSize : 0;
  Requirements= other.Type == "Job"
            && other.Owner != "riffraff"
            && LoadAvg < 0.3
            && KeybrdIdle > '00:15'
]
```

Figure 2: ClassAd describing a Machine

```
[
  Type          = "Job";
  QDate         = 'Mon Feb 10 10:53:31
                    2003 (CST) -06:00';
  Owner         = "raman";
  Cmd           = "run_sim";
  WantRemoteSyscalls = true;
  WantCheckpoint = true;
  Iwd           = "/usr/raman/sim2";
  Args          = "-Q 17 3200 10";
  Memory        = 31M;
  Rank          = KFlops/1E3 +
                   other.Memory/32;
  Requirements = other.Type == "Machine"
                   && other.Arch == "INTEL"
                   && other.OpSys == "LINUX"
                   && other.Memory >= 128M
]
```

Figure 3: ClassAd describing a Job

predicates in which an attribute is related to a value or constant by means of a comparison relation. An example of such a predicate is

```
other.OpSys == "LINUX".
```

We shall often refer to *clauses* that are conjunctions of predicates in this form. An example of such a clause is the following:

```
(other.OpSys == "LINUX") &&
(other.Arch == "INTEL") &&
(other.Memory >= 512M)
```

Thus, requirements expressions are generally disjunctions of such clauses. We shall assume all requirements expressions are in DNF, as any atom that is not in the form of a predicate as we have described may be treated as an atom that can not be modified. Additionally any logical expression that is not in DNF can be converted to DNF, though there may be an exponential blowup in the size of the expression.

3.1 I Don't Like Anyone

First, we shall examine the case in which no available machines match a submitted job's requirements expression. Using a dating service analogy this situation may be described as *I Don't Like Anyone*. Depending on one's point of view, the problem is either with the requirements expression of the job, or with the attributes of the various machine ClassAds which are referenced in the job's requirements. As ClassAd analysis is primarily concerned with aiding the user who has submitted the job we shall focus on the job's requirements expression. First we must indicate which predicate or combination of predicates in a given clause is causing the problem. Once the offenders have been identified we may use the machine ClassAds to suggest possible modifications to the expression. In addition we can detect *conflicts* within the requirements expression, that is two or more predicates which are incompatible. It may well be that the job requirements are non-negotiable, and may not be relaxed or modified. In this case the analysis is still pertinent as it provides useful information about the current pool of available machines.

3.1.1 Suggesting Modifications

On the assumption that the job requirements expression may be modified, we shall examine how to form useful suggestions to the user in this regard. Our goal in this end is to find the least drastic modification to the expression that results in a successful match. In order to achieve this algorithmicly we must define a precise metric for the degree to which an expression is modified. One such metric (M1) is simply the number of predicates in a clause that are modified or removed. The algorithm employing this metric turns out to be quite straight forward with a little bookkeeping; however, the metric does not take into account how drastically each individual predicate must be changed. We can define a more robust metric (M2) by taking into account the difference between the old value and the new value in the predicate as well as the range of values for the machine attribute referenced by the predicate.

In either case, we need to define exactly what constitutes a modification to a predicate. For our purposes we will allow either a modification to the value part of a predicate, or the complete removal of the predicate. If the predicate has an equality operator the value may be changed to anything as long as it has the same type as the original value. In the case of an inequality the value should only be modified so as to relax the predicate, as a stricter predicate will get us nowhere. If the operator in question is a not-equals operator, the only sensible modification is to remove the predicate all together. Removal is also the best strategy if the attribute is not defined in any machine ClassAd.

We begin the algorithm for M1 by generating a table of boolean values from each clause in the job requirements expression. For example, given the following clause:

```
(other.Arch == "ALPHA") &&
(other.OpSys == "SOLARIS") &&
(other.Memory >= 512M)
```

Our table might look like this:

Machine ClassAd	other.Arch == “ALPHA”	other.OpSys == “SOLARIS”	other.Memory >= 512M	Total True
1	T	F	F	1
2	F	F	F	0
3	**F**	**T**	**T**	**2**
4	F	F	T	1
5	**T**	**F**	**T**	**2**
6	**F**	**T**	**T**	**2**
7	F	F	F	0
8	F	T	F	1

The columns of this table correspond to the predicates in the given clause and the rows correspond to machine ClassAds which serve as contexts for the job requirements expression. In addition we keep a tally of the number of **true** values in each row, so we can determine the fewest number of predicates that must be changed in order for the entire clause to evaluate to **true**. To generate our suggestions we find the fewest number of predicates that need to be modified, and we pick the combination of predicates to modify that will yield the most machines.

In the table above we see that there are three machine ClassAds (3, 4, and 5) in which two of the three predicates in the clause evaluate to true. In these three rows there are two different configurations: **F T T** and **T F T**. There are two rows with the first configuration, but there is only one row with the second configuration. In order to match with the most machines we choose the first configuration, and thus suggest the removal (or modification) of the first predicate, `(Arch=="ALPHA")`.

If we are simply suggesting the removal of predicates from the clause, we need only pick the combination of predicates with the highest number of corresponding rows. Removing the offending predicates from the clause will yield exactly that number of machines. However, if we are suggesting modifications we must consider the actual values of each relevant attribute as defined in a given machine ClassAd. We construct a second table, whose columns correspond to attributes referenced in the job requirements expression and whose rows correspond to the machine ClassAds. Continuing with our example, the second table is as follows:

Machine Classad	Arch	OpSys	Memory
1	“ALPHA”	“LINUX”	256
2	“INTEL”	“LINUX”	256
3	**“INTEL”**	**“SOLARIS”**	**1024**
4	“SPARC”	“LINUX”	512
5	**“ALPHA”**	**“LINUX”**	**512**
6	**“SPARC”**	**“SOLARIS”**	**1024**
7	“SPARC”	“LINUX”	256
8	“INTEL”	“SOLARIS”	256

Notice that our assumption about selecting the first predicate is no longer true, We can gain only one machine if we change the value from “ALPHA” to either “INTEL” or “SPARC.” Therefore we must take into account the actual values in a given row before tallying how many machines we will match.

The algorithm using M2 throws out the notion of finding the least number of predicates to be modified, and replaces it with a composite distance function. The value part of each predicate represents a point in a set of literal values and we can calculate the distance from this point to another point representing a new value. In the case of numerical values we simply take the absolute value of the difference between the two points and normalize over the size of the range of values defined in our machine ClassAds. In the case of string or boolean values we assume a distance of zero between two equal values and a distance of one between two non-equal values. Clearly some attributes may be harder to change than others, but absent further information from the user to this effect we can not take this into account.

For each row (machine ClassAd) in our value table we sum up the normalized distances. This gives us a measure of distance between the query represented by the requirements expression and any given point corresponding to a machine ClassAd. A machine ClassAd with the smallest distance is chosen (perhaps as before by determining the changes that will yield the most machines), and the values of the attributes in this ClassAd are used to suggest modifications to the job requirements expression.

3.1.2 Detecting Conflicts

Another way of looking at the I Don’t Like Anyone situation is to find predicates which conflict with one another, that is, predicates that may be satisfied by machines on their own, but are not satisfied in conjunction. For example, a Condor pool may have many machines running Solaris and several machines with Alpha processors, but no Alpha machines running Solaris. In this case the expression

```
(other.OpSys == "SOLARIS") &&
(other.Arch == "ALPHA")
```

represents two conflicting predicates, each of which evaluate to **true** in the context of some machine ClassAds, but in conjunction will always evaluate to **false**. Alternately, an expression may contain a conflict which will evaluate to **false** regardless of the context it is evaluated in. An example of such a conflict is the expression

```
(other.Arch == "ALPHA") &&
(other.Arch == "INTEL")
```

In this case we have two predicates that may be satisfied on their own, but together they will never be satisfies as the `Arch` attribute can only have one value.

Detecting the former kind of conflict requires the evaluation of the individual expressions in the context of machine ClassAds, whereas the latter kind may be identified in isolation. To dectect the latter we must separate the predicates in a clause by attribute reference. For each attribute referenced we convert the predicates to points or intervals depending on the type of the values. If the intersection of these intervals is empty we have identified a conflict. The notion of predicates as intervals is discussed further in the next section. The remainder of this section is devoted to conflicts which are dependent on the values of the machine attributes.

To better understand the problem of conflict detection it is helpful to think of a clause as set of predicates and to construct a subset lattice, with the full clause on the top and an empty clause (semantically equivalent to **true**) on the bottom. In Figure 4 we see a lattice representation of the subexpressions of the clause

p_1 && p_2 && p_3 && p_4

where p_1 to p_4 are the predicates. Each subset corresponds to subexpression of the clause generated by removing certain predicates.

A given subset *succeeds* (marked with a **T**) if the corresponding expression evaluates to **true** in the context of some machine ClassAd and *fails* (marked with an **F**) otherwise. Any set in the lattice that fails and has no failing subsets is a Minimal Failing Subexpression (MFS), enclosed by a dashed oval. Any set that succeeds and has no succeeding superset is a Maximal Succeeding Subexpression (MSS), marked by a solid oval. This terminology comes from work in database query analysis [3]. We have substituted the term subexpression for subquery. The conflicts we are looking for are those that correspond to MFSs, that is expressions that always evaluate to **false**, but whose subexpressions evaluate to true in some context.

Godfrey [3] shows that in the general case finding all MFSs is NP-Hard, but proposes a linear time algorithm for finding one MFS and a polynomial time algorithm for finding a fixed number k of MFSs. However, these algorithms are measured in terms of the number of database

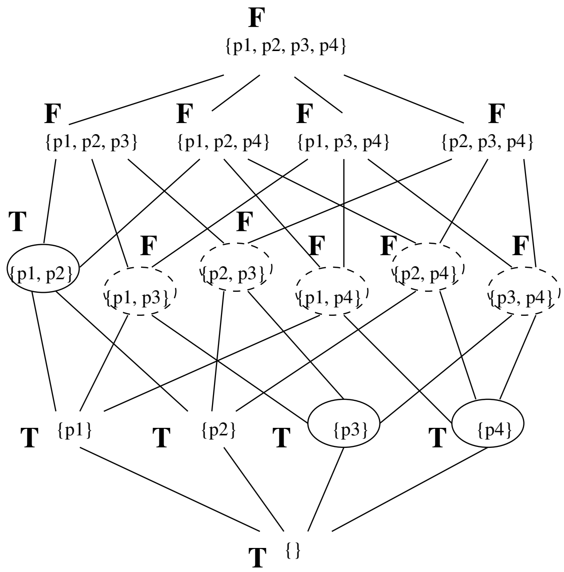

Figure 4: A subset lattice representing subexpressions of a clause. The solid ovals are MSSs and the dashed ovals are MFSs.

queries needed to produce the desired information. If we have a table of boolean values (which can be generated in $m \times n$ time where m is the length of a clause and n is the number of contexts) we do not need to make a series of queries as we have all of the information we need.

It can be shown that to generate the set of all MFSs we may use the set of all MSSs to express the set of all succeeding subexpressions as a formula in DNF, negate this formula, convert the result to DNF, and prune out any logically redundant subformulas. For example in Figure 4 the set of succeeding subexpressions may be represented by the formula:

$$(\neg p_3 \wedge \neg p_4) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_4) \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3)$$

Its negation is:

$$(p_3 \vee p_4) \wedge (p_1 \vee p_2 \vee p_4) \wedge (p_1 \vee p_2 \vee p_3)$$

Converting this to DNF, and pruning out redundancies we get:

$$(p_1 \wedge p_3) \vee (p_1 \wedge p_4) \vee (p_2 \wedge p_4) \vee (p_2 \wedge p_3) \vee (p_3 \wedge p_4)$$

which represents exactly the set of MFSs. To get the set of all MSSs we simply collect all of the unique rows of the boolean table and prune out any rows that do not correspond to MSSs. The main drawback to this process is that converting the negated formula (which is in CNF) to DNF may result in an exponential blow up in the size of the formula. This is not a grave concern, as in practice these requirements expressions are not very long, at least with respect to the number of machine ClassAds.

3.2 Nobody Likes Me

The converse of the I Don't Like Anyone situation is *Nobody Likes Me*. Instead of the job ClassAd requirements expression rejecting all machines, all of the machine ClassAd's requirements expressions reject the job. Therefore we must examine multiple expressions in DNF in the context of a single job ClassAd. Our focus is providing information for the user who submits the job, so we must look at this in terms of the attributes of the job ClassAd. Just as we sought to suggest modifications to the job requirements expression in the previous section, we shall endeavor to find potential modifications to the job ClassAd's attributes. It is even possible that crucial attributes may be missing from the job ClassAd entirely.

We may look at this situation geometrically, where the collection of attributes with literal values in a ClassAd is represented by a point in n-dimensional space where each dimension corresponds to a single attribute. Clauses are represened in this space by n-dimentional hyper-rectangles. The situation described in the previous section was that of a single hyper-rectangle (the job requirements expression) and many points (the machine ClassAds) that did not lie within the hyper-rectangle. The algorithm using the M2 metric in effect finds the closest point to the hyper-rectangle and generates suggestions to expand the hyper-rectangle to include it. Now, given a single point, we wish to find the closest of several hyper-rectangles. We shall use this to suggest changes to the job attributes so that the point may be relocated within the closest hyper-rectangle, and thus the job ClassAd will be accepted by some machine ClassAd's requirements expression.

Figure 5 shows the geometric equivalent of two clauses where clause 1 is

```
(ImageSize >= 128M) &&
(MemoryRequirements >= 512M)
```

and clause 2 is

```
(ImageSize >= 64M) &&
(MemoryRequirements >= 1024M)
```

One way to accomplish this would be to apply the algorithm using the M2 metric discussed in the previous section. This method is sufficent for finding the nearest point, or smallest overall change to the attributes in the job ClassAd. However, one might wish for more detailed information, such as how many machines would match with the job if the changes described above are made. In order to be concise we should partition the space covered by the hyper-rectangles representing machine requirements expressions into equivalence classes. Each partition corresponds to a range of job attribute values which satisfy the requirements of a unique set of machines.

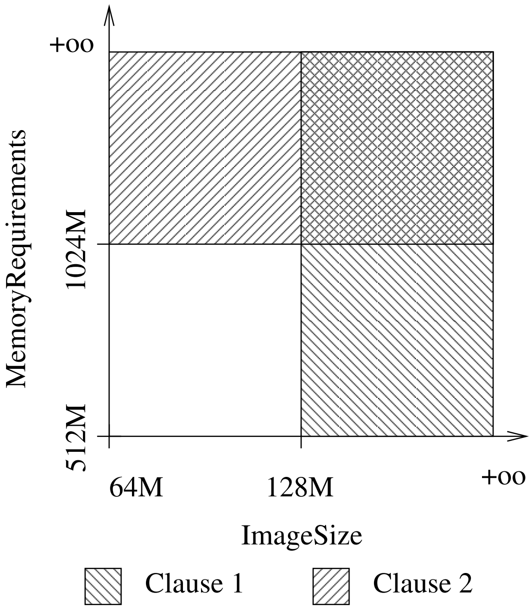

Figure 5: A geometric representation of two clauses

The first step in this process is generating the hyper-rectangles. If a machine requirements expression contains more than one clause, we create a seperate hyper-rectangle for each clause, as the union of all such hyper-rectangles represents the space covered by the entire expression. Given a clause, we may treat each predicate as an interval (or set of intervals) in the dimension corresponding to its attribute.

An equals operator in the predicate defines a point, a not-equals operator defines the union of two open intervals comprising all values except for the value in the predicate, and any other inequality operator defines an open or closed interval from the value to positive or negative infinity. If there are multiple predicates with the same attribute, we find the intersection of all of the intervals they represent. If this intersection is empty, we have found a conflict as described in the previous section. Since we are dealing with machine ClassAds, we simply throw out this clause as it will never be satisfied.

The second step is partitioning along each dimension seperately, taking care to keep track of which partition any given machine belongs to. For example, clause 1 (c_1) in Figure 5 contains the predicate `other.Memory >= 512M` and clause 2 (c_2) contains the predicate `other.Memory >= 1024M`. This creates two intervals: $[512, 1024)$ and $[1024, +\infty)$ coresponding to the sets $\{c_1\}$ and $\{c_1, c_2\}$. We continue this process until all clauses in each machine requirements expression are processed, then repeat the process for all dimensions. If there is an attribute that is referenced in some clauses but not others we simply represent that clause as $(-\infty, +\infty)$, the set of all strings, or the set of all boolean values depending on the inferred type of the attribute (determined by the first value associated with that attribute).

One tricky aspect is dealing with a predicate like `other.Owner != "ncoleman"`. If the type of the value were boolean the solution would be trivial, and we have already defined how a not-equals operator is to behave among numerical values. In this case we invent a special string value called *AnyOtherString*. Thus if we have several string values plotted as points in the dimension corresponding to the `Owner` attribute, we add the new clause to the sets associated with any string that is not `"ncoleman"` and also add the clause to the set associated with AnyOtherString. We shall see that it is important to keep track of which strings are not represented by AnyOtherString in a given dimension.

The third step is constructing the n-dimensional partitions by taking the cross product of the vectors of intervals in each dimension. Given interval $[64, 128)$ in the `ImageSize` dimension with clause set $S1 = \{c_2\}$ and interval $[1024, +\infty)$ in the `MemoryRequirements` dimension with clause set $S2 = \{c_1, c_2\}$ we create a rectangle defined by $[64, 128) \times [1024, +\infty)$, and associate it with the intersection of $S1$ and $S2$, namely $\{c_2\}$. What this means is that job ClassAds with attribute values in the range defined by $[64, 128) \times [1024, +\infty)$ will match the machine corresponding to c_2. We continue the process with all intervals, and with all dimensions.

If we run into the AnyOtherString placeholder in a dimension with string values, we make note of all of the other string values we have encountered in that dimension and annotate any hyper-rectangle created using this instance of AnyOtherString with these values. For example, if the dimension corresponds to the `Owner` attribute and the other string values are `"ncoleman"` and `"raman"`, then AnyOtherString means any value for `Owner` except `"ncoleman"` and `"raman"`.

Finally we have a set of hyperrectangles each associated with a subset of clauses (and therefore a subset of machines) which partitions the space. We may need to clean up this set by adjoining hyperrectangles corresponding to identical sets of machines.

We can now not only find the closest range of values to those in our job ClassAd, we can specify how many and which machines will match with a job ClassAd with attribute values in that range. In addition we can present several alternative value ranges, each with a distance defined by our M2 metric and a set of matching machines. This extra information opens the door for more complex policies for determining suggestions based on distance as well as user preferences for certain machines.

3.3 ClassAd Analysis in Condor

One ideal application of ClassAd analysis (indeed the one that originally motivated this research) is the Condor user tool `condor_q`, a command line interface to local or remote Condor job queues. Users can view information about the jobs in a given queue, including the owner, how long the job has been in the queue, and the status of the job. A job may have a status of IDLE for various reasons, including our two scenarios: *I Don't Like Anyone* and *Nobody Likes Me*. For a given job `condor_q -analyze` will report how many machines the job rejected as well, how many machines rejected the job, as well as information about priorities and preferences which we will not go into here.

Using the ClassAd analysis library, we can provide detailed information to supplement the numbers. For example, a job with the following requirements expression is submitted:

```
(other.Foo == "bar") &&
(other.Memory > 2096) &&
(other.Arch == "INTEL") &&
(other.OpSys == "LINUX") &&
(other.Disk >= 14)
```

The results of `condor_q -analyze` are:

```
Run analysis summary.  Of 839 machines,
  839 were rejected by the job's
  requirements
  No successful match recorded.

  Predicate          Matches Suggestion
  ---------          ------- ----------
1 Foo == "bar"       0       REMOVE
2 Memory > 2096      0       MODIFY TO 1911
3 OpSys == "LINUX"   616
4 Arch == "INTEL"    740
5 Disk >= 14         835
```

In this case there is no machine ClassAd with the `Foo` attribute, hence the suggestion to remove predicate 1. On the other hand, there are machine ClassAds where the `Memory` attribute is defined, so a modification suggestion is made for predicate 2.

The above example contained no conflicting predicates. We now examine the following requirements expression:

```
(other.OpSys == "SOLARIS") &&
(other.Arch == "ALPHA") &&
(other.Disk >= DiskUsage)
```

The results of `condor_q -analyze` are:

```
Run analysis summary.  Of 839 machines,
  839 were rejected by the job's
  requirements
  No successful match recorded.
```

```
  Predicate          Matches Suggestion
  ---------          ------- ----------
1 Arch == "ALPHA"    4       MODIFY TO
                             "INTEL"
2 OpSys == "SOLARIS" 120
3 Disk >= 14         838

Conflicts:
  predicates: 1, 2
```

This time we are given the additional information that the first two predicates conflict with one another.

Finally we examine a case where the job requirements expression is satisfied by some machine ClassAds, but one of the job attributes, `ImageSize`, is too large:

```
Run analysis summary.  Of 837 machines,
  807 were rejected by the job's
     requirements
  30 rejected the job

  Predicate          Matches Suggestion
  ---------          ------- ----------
1 Memory > 1024      30
2 OpSys == "LINUX"   618
3 Arch == "INTEL"    741
4 Disk >= 14         836

The following attributes should be added
or modified:

Attribute          Suggestion
---------          ----------
ImageSize          use a value <= 1220608
```

4 Related Work

The ClassAd-based matchmaking framework for resource management has parallels with several fields: generic matchmaking systems, resource management systems and constraint database systems.

Matchmaking is widely studied in generic agent systems. The advertising languages of ACL [2] and RETSINA [14] support reasoning so that very general behaviors may be described and inferred. In contrast to these knowledge-base representations, ClassAds employ a database representation.

Many resource management systems [15, 10, 4] process jobs by using resources that are identified explicitly through a job control language, or implicitly, by submitting the job to a queue associated with a resource set. Jobs requiring multiple resources must be submitted to special queues — there is no general mechanism to marshal a unique mix of resources. In Globus [1], customers describe required resources in a resource specification language (RSL) based on a pre-defined schema of the resources database. However, resources cannot place constraints on requests.

The matching operation is similar to a spatial join between generalized tuples of a constraint database. However, matchmaking is different in that it employs a semi-structured data model, and ClassAds are consumed during the matching process.

Most of the work relevant to ClassAd analysis is in literature on databases, particularly on cooperative query answering. In [8] a mechanism called SEAVE is presented for extracting and verifying presuppositions from queries. This mechanism identifies queries which result in null answers, then finds more general queries by weakening or deleting query subexpressions. The result is a set of maximally general erroneous presuppositions, which may be of more value to the user than a simple null answer.

Similar techniques are discussed more formally in [3]. Godfrey discusses identification of *minimal failing subqueries* (MFSs) and *maximal succeeding subqueries* (MSSs). Godfrey's MFSs are analogous to the erroneous presuppositions generated by Motro's SEAVE mechanism. The MSSs are the least general generalizations of the initial query which succeed. An algorithm called ISHMAEL is presented which enumerates MFSs and MSSs. This algorithm is NP-hard for queries of arbitrary length, but remains polynomial for fixed length queries.

Finally, in [7] the notion of a *query difference operator* is introduced to indicate missing information in query results. The authors discuss a system of resource agents, brokers, and user agents which resembles the distributed framework used by condor. The primary focus of this work is to indicate the incompleteness of query answers. The query difference operator is used to generate the description of the set of results covered by the query, but not covered by the query answer. This set is expressed in relational algebra and can presumably be converted into a pseudo-English response for the user.

Our work with ClassAd analysis uses similar techniques and notions to provide useful information regarding matchmaking failure. As discussed previously, our conflict detection algorithm covers similar territory as [3]. One key difference in our research is the semi-structured data model which unlike the relational model discussed in the cited papers does not require a fixed schema. Another important difference is the reflexive nature of ClassAds. In database terms a ClassAd contains both a query (the requirements expression) and a record (the set of attributes with literal values). Nevertheless, many of the issues encountered in ClassAd analysis are applicable in database query analysis, web search, or any other field in which boolean expressions are used as constraints.

5 Conclusions and Future Work

As the matchmaking process is used to deal with larger groups of resources, and the ClassAd language is used to represent more complex policies, the need for ClassAd analysis will only increase. A framework is needed that will be optimized for the common case (disjunctive normal form) but be powerful enough to analyze expressions in any form. It follows that a ClassAd analysis tool must be very robust, but it must also be usable. If the presentation of the results of the analysis is not clear and concise the tool will be of no help to the user. The adjustments to the `condor_q` tool are a preliminary effort to this effect, but a more comprehensive and interactive ClassAd analysis interface is needed. For example, we would like to incorporate input from the user as to which predicates in the job requirements expression are harder to change than others.

Not only will the number of available resources increase, but new matchmaking paradigms may be needed to deal with more complicated resource allocation. Say, for example, that a job must match with both a machine and a license for the software used to run the job. Any of the three principals may have requirements on any of the others, and thus a more complex matchmaking model is needed. The Gang Matching model deals with just this situation and is described in [11].

Even Gang Matching is insufficient to deal with a principal requesting an unspecified and possible large number of resources. Constraints on these resources may be aggregate, such as total memory across a number of machines. A Set Matching model is presented in [6] to address exactly this situation in the context of Grid resource selection. Several new set operations have recently been added to the ClassAd language to facilitate Set Matching.

With new matchmaking models comes a need for more complex ClassAd analysis. The algorithms described here are not sufficient for these new paradigms, but they will provide a firm foundation for future work in this area.

References

[1] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems.

[2] M. Genesereth, , N. Singh, and M. Syed. A distributed anonymous knowledge sharing approach to software interoperation. In *Proc. of the Int'l Symposium on Fifth Generation Computing Systems*, pages 125–139, 1994.

[3] P. Godfrey. Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems (IJCIS)*, 6(2):95–149, June 1997.

[4] R. Henderson and D. Tweten. Portable Batch System: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[5] M. J. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. *IEEE Workshop on Experimental Distributed Systems*, 1990.

[6] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11)*, July 2002.

[7] M. Minock, M. Rusinkiewicz, and B. Perry. The identification of missing information resources by using the query difference operator. Technical report, MCC, April 1999.

[8] A. Motro. SEAVE: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4):312–330, October 1986.

[9] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring Structure in Semistructured Data. In *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[10] B. Clifford Neumann and S. Rao. The prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, June 1994.

[11] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, Madison, 2000.

[12] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high-throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, July 1998.

[13] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distibuted resource management. *Cluster: Journal of Software, Networks and Applications. (Special Issue on High Performance Distributed Computing)*, 2(2), 1999.

[14] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, pages 36–46, dec 1996.

[15] S. Zhou. LSF: Load sharing in large-scale heterogenous distributed systems. In *Proc. Workshop on Cluster Computing*, 1992.