Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching

Rajesh Raman, Miron Livny and Marvin Solomon
{raman, miron, solomon}@cs.wisc.edu
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706

Abstract

Dynamic, heterogenous and distributively owned resource environments present unique challenges to the problems of resource representation, allocation and management. Conventional resource management methods that rely on static models of resource allocation policy and behavior fail to address these challenges. We previously argued that Matchmaking provides an elegant and robust solution to resource management in such dynamic and federated environments. However, Matchmaking is limited by its purely bilateral formalism of matching a single customer with a single resource, precluding more advanced resource management services such as co-allocation. In this paper, we present Gangmatching, a multilateral extension to the Matchmaking model, and discuss the Gangmatching model and its associated implementation and performance issues in context of a real-world license management co-allocation problem.

Keywords: *distributed resource management, matchmaking, gangmatching, heterogenous computing, Condor*

1. Introduction

Federated distributed systems present new challenges to resource management. Conventional resource managers are based on a relatively static resource model and a centralized allocator that assigns resources to customers. This model does not adapt well to 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 [3]. Distributed ownership introduces *policy heterogeneity*: Each resource may have its own idiosyncratic allocation policy. We previously argued that Matchmaking provides an elegant and robust solution to the problem of heterogeneous resource management in dynamic, distributed environments [13]. Matchmaking provides a powerful language for a consumer to express constraints and preferences on a resource *and* for the resource to express constraints and preferences on a consumer.

But Matchmaking has an important limitation: It matches each customer with a single resource. In many important application domains, a collection of resources may be required to perform an action. Often, complex consistency requirements hold between the consumer and the resources, and amongst the resources. This paper presents Gangmatching, a formalism that extends Matchmaking from a bilateral to a multilateral model, and discusses the implementation and performance issues associated with Gangmatching.

We begin with a brief introduction to Matchmaking in Section 2, and continue in Section 3 with a discussion of the necessity of a multilateral matchmaking model by presenting a real-world multi-resource problem that has no practical bilateral matchmaking solution. We then present the Gangmatching model in Section 4 and discuss implementation and performance of the Gangmatching model in Sections 5 and 6 respectively. Related work is presented in Section 7 and future directions are identified in Section 8.

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 [13] for further details.

Agents describe their capabilities and requirements by sending messages to a Matchmaker. These messages, which we call *classified advertisements* (classads) in analogy to their newspaper counterparts, contain both descriptive information about entities and policy constraints on compatible matches. The Matchmaker finds compatible pairs of classads and informs agents of the results. The agents may then use bilateral protocols to establish bindings based on these results. For example, Submission agents may inform the Matchmaker about Jobs waiting to be run, while Execution agents send classads describing Machines and their capabilities. Each Job classad describes the characteristics of the Job, constraints on Machines suitable for running it, and preferences to choose among compatible Machines. Similarly, each Machine classad may impose impose constraints and indicate preferences among Jobs it is willing to run. When the Matchmaker finds a compatible (Job, Machine) pair, it informs the corresponding Submission and Executions agents, which then engage in a *claiming* protocol to cement the relationship and start running the Job.

Our matchmaking framework is composed of 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 [10] records of $(name, expression)$ pairs which may be thought of as "attribute lists" that describe agents. The language has special **undefined** and **error** values, as well as special operator semantics to operate robustly in heterogeneous and semi-structured environments.

2. The *Matchmaker Protocol* describes how entities 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, establish the allocation and utilize the advertised services. Either party may choose to withdraw from a match by rejecting a claim, which may happen if the state of the agent has changed since the last advertisement was posted.

The flexibility and expressiveness of the classad language greatly contributes to the effectiveness of our Matchmaking framework. Figure 1 shows a classad describing a workstation in the University of Wisconsin–Madison Condor [7] pool.[1] While most attributes in the classad describe the machine's characteristics, the `Constraint` and `Rank` identify the advertising entity's constraints and preferences—i.e., the entity's *policy*. 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*. If A.`Constraint` and B.`Constraint` both evaluate to **true**, the two advertisements are deemed compatible and the `Rank` expressions of A and B may be evaluated to determine their respective preferences.

The classad language specifies the syntax and semantics of the expressions in Figure 1, while the matchmaking protocol and algorithm give special significance to the keywords `Constraint`, `Rank`, and `other`. All policy information is expressed in the `Rank` and `Constraint` expressions; the classad language and matchmaking provide the mechanism for enforcing it. For example, the workstation in Figure 1 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 low image sizes are preferred between 9am and 5pm.

```
[
  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;
  Constraint  = other.Type=="Job" &&
                  other.Owner!="riffraff" &&
                  LoadAvg < 0.3 && KeybrdIdle>'00:15'
]
```

Figure 1. Classad describing a Machine

Many interesting and useful policies may be easily defined within this framework; interested readers are referred to reference [14] for more sophisticated examples derived from the policies of real-world users of the Condor system.

3. Motivation for Gangmatching

One of the first indications of practical limitations to bilateral matchmaking arose in the context of Condor. A Condor user had purchased licenses for various software packages. Jobs that use such packages need to allocate both a machine and a license before they can run. Licensing terms impose a variety of constraints on running instances of an application. For example, in addition to issuing a limited number of licenses, some licenses may be valid only on some workstations, while others may be valid on certain

[1] The Wisconsin Condor pool is currently composed of over 900 nodes, running seven different architecture/operating system combinations. The pool is used continuously as a production system to provide computation services for several research projects.

subnets. Other licenses may "float" throughout the site, but once claimed on a particular machine, may be valid for several instances of the application on that machine. In the context of such policies, it becomes necessary to treat software licenses as resources that must be managed with the same degree of flexibility and robustness as other resources in a resource management system.

Due to the dependencies between job, workstation and license, conventional bilateral matchmaking is inadequate for solving this problem. An attempt to work around the problem might use two interactions with the Matchmaker. On the first round, the submission agent would submit an ad describing a Job and seeking a matching Machine. On receiving a response, it could then send an ad describing the (Job, Machine) pair and looking for a suitable license. However, if licenses are in short supply, the first match might tie up a machine for lengthy periods while waiting for a license. In the worst case, deadlock is possible. A similar problem arises if the Submission agent requests the license first and then the machine. Strategies that allocate one resource and subsequently free it if the other other is not available are highly inefficient and can lead to livelock and starvation. The need for a matchmaking scheme that can marshal a consistent aggregation of dependent classads in an atomic operation is therefore clear.

It is important to note that we are not merely proposing a mechanism to solve specific license management scenarios, which may be individually solved by *ad hoc* mechanisms that are simpler than our Gangmatching solution. Our goal is to develop a single method of multilateral matchmaking that is agnostic to the kinds of resources being matched and thus capable of marshaling consistent resource aggregates whose composition and inter-dependencies are not known to the Matchmaker *a priori*.

4. The Gangmatching Model

The challenge of developing a multilateral matchmaking model is in defining a solution that inherits and extends the full generality of the bilateral matchmaking scheme. The power of the Matchmaking model is in managing resources whose properties and dependencies are not known *a priori*. In direct analogy, we required our multilateral model to be able to marshal candidate groups in which the specific kinds of candidates and their inter-dependencies are defined only by the candidates themselves. There are many consequences of this requirement.

The most important requirement of a multilateral matchmaking model is a scheme to express the need to marshal an arbitrary number of candidates. Since no central schema is legislated, it is important that the "interfaces" of these different candidate resource types be separated to prevent namespace collisions and ambiguity. Next, the solution must provide the ability to relate the properties of multiple candidates through arbitrary constraints defined on candidate individuals and groups. Since the constraints may themselves be defined in different advertisements, a mechanism must be provided for properties about some candidates to be conveyed to other candidates who might not know the full composition of the group. For example, a compute server may need information about a data file without knowing if the file data is expressed as part of the user's job attributes, or if the user is marshaling in a replica from a storage server as part of the multilateral match. Finally, the solution must be amenable to efficient implementation.

Gangmatching is our solution to the multilateral matchmaking problem. The Gangmatching model follows a docking paradigm, where aggregate "gangs of classads" are created by binding together (i.e., "docking") individual classads with a matching operation. Intuitively, Gangmatching extends regular matchmaking by replacing a classad's single implicit bilateral match imperative with an explicit *list* of required bilateral matches, with the additional ability of allowing classads to access information from other bilateral match localities.

Each basic docking operation occurs between *ports* of classads. The port abstraction serves multiple purposes. First, ports serve as matchmaking interfaces, allowing information to be provided to candidates independent of how the information is generated (i.e., some constant value, or from properties of other candidates in the gang). Second, the port abstraction separates the namespaces of candidates, and provides a naming scheme that allows the properties of candidates to be accessed from other localities. Finally, the abstraction imposes a structure on the aggregate gang that simplifies the definition and implementation of algorithms that operate on the entire group.

We introduce the Gangmatching model by discussing the license management problem in context of the components of a matchmaking framework: classad representation, matchmaking algorithm, matchmaking protocols and claiming protocols. More complex examples of the capabilities of the Gangmatching model are provided in reference [12].

4.1. ClassAd Representation

The classad representing the job in the license management problem is illustrated in Figure 2. The most notable feature of the example is the `Ports` attribute. A classad's ports define the number and characteristics of matching ads required for that classad to be satisfied. In the Gangmatching model, bilateral matchmaking occurs between the ports of classads instead of entire classads.

Each port defines a `Label` that names the candidate bound to that port, replacing the fixed `other` attribute of bilateral matching. The scope of a label extends from the

```
[ Type  = "Job";
  // some common attributes
  Owner = "raman";
  QDate = 'Mon Feb 28 14:22:22 2000 (CST) -06:00';
  Cmd   = "run_sim";
  Ports = {
     [ // request a workstation
       Label      = cpu;
       ImageSize  = 28M;
       Rank       = cpu.KFlops/1E3 + cpu.Memory/32;
       Constraint = cpu.Type=="Machine" &&
                      cpu.Arch == "INTEL" &&
                      cpu.OpSys == "LINUX" &&
                      cpu.Memory >= Imagesize;
     ],
     [ // request a license
       Label      = license;
       Host       = cpu.Name; // cpu name
       Rank       = 0;
       Constraint = license.Type=="License" &&
                      license.App == Cmd;
     ]
   }
]
```

Figure 2. A Gangmatch request

port of declaration to the end of the port list. Thus, expressions in the second port with declared label "license" can refer to the "cpu" label declared in the first port, but not *vice versa*. Furthermore, port labels are private and local to the hosting classad, preventing namespace pollution and collisions.

In this example, as in many cases of Gangmatching, constraints on some matches are influenced by the attributes of other classads participating in the match. Specifically, the validity of a license depends on the particular machine that has been chosen to host the application. By allowing the scopes of labels to extend beyond the port of declaration, the Gangmatching mechanism allows the ability to convey information from one match locality to another. Thus, the license request port can convey the location of the chosen workstation to the license offer via the `Host` attribute. Note that labels of succeeding ports may not be referred to by preceding ports, which limits the dependency relations between ports and makes the model more amenable to efficient implementation. Specifically, the scoping rules guarantee that the first port of an advertisement is not dependent on any other port.

Example advertisements of workstations and licenses (as would be advertised by workstation and license agents) are illustrated in Figures 3 and 4 respectively. The workstation classad is very similar to its bilateral matchmaking counterpart, except for the presence of a port to explicitly indicate its imperative to match one entity.

The most noteworthy aspect of the license classad is the presence of `requester.Host` in the `Constraint` expression which, refers to the `Host` attribute of the matching port. Since this attribute was defined as `cpu.Name` in Figure 2, the referenced value is the name of the workstation

```
[ 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";
  Ports        = {
    [ Label      = requester;
      Rank       = 1/requester.ImageSize;
      Constraint = requester.Type=="Job" &&
                     requester.Owner!="riffraff" &&
                     LoadAvg < 0.3 && KeybrdIdle>'00:15'
    ]
  }
]
```

Figure 3. Workstation Advertisement

```
[ Type         = "License";
  App          = "sim_app";
  ValidHost    = "foo.cs.wisc.edu";
  Ports        = {
    [ Label      = requester;
      Rank       = 0;
      Constraint = requester.Type=="Job" &&
                     requester.Host==ValidHost
    ]
  }
]
```

Figure 4. License Advertisement

chosen to run the job. Thus, the license's constraint is satisfied only if the chosen workstation is the single valid host "foo.cs.wisc.edu." Clearly, more complex constraints may be expressed in the license constraint to implement sophisticated license management policies.

Note that the Gangmatching model requires the job classad to convey host information to the license classad via the appropriate port. The license ad cannot directly access the `Name` attribute of the machine ad. This scoping restriction was deliberate. The `license` port declared in the Job ad acts as an abstract "interface" to potential license ads. The job classad in turn "implements" that interface by exporting a matching workstation's name through its attribute `Ports[1].Host`.

4.2. Matchmaking Algorithm

The role of the Matchamking algorithm in the Gangmatching scheme is to marshal a consistent "gang" of classads for the job classads in the system. Conceptually, a gang is constructed by starting with a degenerate gang composed of a single *root* classad, and then binding each unbound port of the gang to a compatible port of a new classad (one not already in the gang) until all ports in the gang are bound and the gang is consistent (i.e., all constraints are satisfied).

Different concrete algorithms may be employed to implement this conceptualization. We describe a few specific algorithms developed to exploit the known structure of the Gangmatching problem in Section 5.

In the license example, a gang consists of three ads: a job, a machine and a license. If one of the ads, for example the license, had another port, the match would not be complete until another ad was found to match against the "unbound" port. In general, a match consists of a *tree* of ads. Each pair of adjacent ads is bound by choosing a port ad from each and checked by evaluating the `Constraint` attributes of the two port ads. Each `Constraint` expression can refer to attributes of its own ad as well as attributes of the `Port` ads bound to its port and earlier ports in its containing ad.

Although this conceptual model suggests a "left-to-right" evaluation of `Constraint` expressions, it does not require it. In fact, we shall see in Section 5 that other orderings can be substantially more efficient.

4.3. Matchmaking Protocols

The Gangmatching model does not require any modification to the the advertisement protocol that is used by agents to send their classads to the matchmaker. However, the notification protocol (used by the matchmaker to notify matched agents) requires a modest and straightforward extension: The matchmaker must now notify every agent whose classad was matched in the gang, and it must include in the notification all relevant information about the other ads it matched. A simple way to provide this information is simply to send the entire gang to each agent, letting the agents extract whatever parts of it they find useful.

4.4. Claiming Protocols

The claiming protocols for Gangmatching must proceed in a two-phase manner to accommodate the possibility of any agent in the gang rejecting the match. Claiming is initiated by the agent whose advertisement serves as the root of the gang (the *root agent*), and proceeds recursively in a top-down manner. The root agent begins by checking whether each of its gang neighbors is willing to participate in the transaction. During the delay between the initial message from the agent to the matchmaker and the time when it is contacted during the claiming protocol, its local conditions may have changed so that it no longer considers the match valid. If so, it responds in the negative and the root agent tells all members of the gang to discard the match and start over. If the match is still valid and it is not a leaf in the gang tree, it contacts its children to validate the match. Affirmative responses pass up the tree to the root. If the root agent get affirmative responses from all its children, it enters the second phase, in which claims and allocations are established.

5. Gangmatching Implementation

Given a set of classads, Gangmatching proceeds as a periodic activity by identifying a subset of classads that will serve as gang roots, and sorting these roots using some criterion. For example, job classads may be identified as roots and sorted by priority order. Thereafter, the Gangmatching algorithm attempts to marshal a consistent gang for each root (in order), eliminating matched ads from further consideration. Note that our implementations handle the full generality of the Gangmatching scheme; however, we restrict our descriptions to the license management problem for purposes of clarity and simplicity.

We begin by describing a simple *naive* algorithm that serves to illustrate the obvious solution, and provide context to the algorithms that follow. The naive algorithm is a simple recursive backtracking search. Given a root classad, the matchmaker proceeds sequentially through its ports from first to last, attempting to find a candidate compatible with that port. If a candidate is found to be incompatible, another candidate is tried, until the a successful candidate is discovered, or all candidates are exhausted. If no candidate could be found for a particular port, the algorithm backtracks to the previous port and attempts to replace the incumbent at that port with another compatible candidate, proceeding forward if the attempt was successful. If the algorithm reaches the end of the root's port list, a gang has been successfully marshaled, and if the algorithm attempts to backtrack from the first port, no consistent gang exists. If a matching ad has more than one port (so that the resulting gang is a multi-level tree), the algorithm is called recursively to iterate through potential sub-gangs using it as a root.

Although the naive algorithm will give the correct result, it may be very inefficient, depending on the nature and distribution of the available ads. In relational database terminology, the Gangmatching problem is similar to a multi-way join[2]. Relational database systems use two techniques for optimizing such queries: reordering the pairwise joins, and choosing the right algorithm for each join.

The naive algorithm performs each pairwise join using simple nested loops. We use a classad indexing scheme that indexes both attributes and constraints to efficiently exclude large numbers of incompatible candidates. The basic idea is

[2]The difference between a database join and matchmaking is that when multiple matches are found, a join returns them all, while matchmaking chooses one. Also, a match "consumes" the matched ads.

to use conventional search trees for attribute indexes and interval trees to index the constituent comparison expressions of constraints, which may be thought of as intervals over some domain. Due to the efficiency of the indexing scheme and its ability to provide tight supersets of candidates, it is also used in heuristic functions to guide the strategy of the Gangmatching algorithm by estimating candidate set sizes for ports.

As in the case of relational queries, reordering joins can yield orders of magnitude speedup. For example, consider our license example when there are many compatible machines but a small number of highly selective licenses. The naive algorithm would iterate through all matches of jobs to machines, trying to find an appropriate license for each such match. Most of the effort spent matching jobs to machines would be wasted. A much more efficient approach would be to first match a job to a license, and then look for a compatible machine. The license's `Constraint` expression makes indirect reference to the `Name` attribute of a machine ad, so it cannot be fully evaluated until a machine ad is chosen. However, it can be partially evaluated. Perhaps a license candidate can be rejected out of hand. For example, if the clause `&& Owner == "solomon"` were added to the license ad of Figure 4, a match with the ad of Figure 2 could be rejected regardless of machine. If the partial gang cannot be rejected at this stage, it can still be represented as a classad with a `Constraint` expressions to be matched against potential machine ads. The indexing techniques mentioned above can be use in this step as well.

Since it is always possible (and often likely) for a fixed strategy to work extremely poorly in realistic workloads which constantly change, it is important to develop an adaptive algorithm that chooses the right strategy dynamically. We augmented the naive algorithm with heuristics to fill ports in ascending order of candidate set sizes. Thus, for example, if there are a large number of compatible machines but a small set of licenses, the algorithm first chooses a license and then attempts to find a machine that is compatible with the license, dynamically shunting constraints from the license classad to the cpu port of the job classad to accomplish the task. Furthermore, if there are no licenses, the algorithm can immediately terminate without having to check for machine compatibility at all.

6. Gangmatching Performance

In this section, we present the performance of the fixed-order, fixed-order with indexes, and dynamic algorithms. Classads used for these experiments are similar in structure to the job, workstation and license classads presented in Figures 2, 3 and 4, except that only architecture, operating system, physical memory and virtual memory attributes were modeled for workstations, with corresponding constraints imposed on them by jobs. In addition we modeled a memory size attribute for jobs, which is constrained by the workstation classads.

Workloads are characterized by three parameters: N, the number of job and workstation classads (always equal), L, the license density (the relative number of license classads, either 50% or 100% of N), and S, the selectivity index of license classads (1, 2, 4 or 8). For selectivity index S, the sets of license and workstation classads are each divided into S disjoint equally sized partitions; licenses in each partition are only valid on workstations from the corresponding partition.

Figure 5 illustrates the elapsed time performance of the naive fixed-order algorithm on this benchmark (Intel Pentium Pro Linux workstation with 256MB RAM). Curves fall into two bands, those with license density L of 50% (above) and those with license density 100% (below). Two points are worth noting: First, the naive algorithm is especially inefficient when matches cannot be made due to scarcity of licenses. When licenses are scarce, the algorithm must frequently backtrack, leading to a running time that is high and grows quadratically with the number of ads. Second, the costs of backtracking and evaluation dominate the elapsed time so that the selectivity S has nearly no additional effect on performance.

Figure 6 shows what happens when indexes are used to speed up the search. As in Figure 5, the upper curve corresponds to the 50% license density workloads. Although this graph looks similar to Figure 5, the reader should note the scale; performance is improved by an order of magnitude overall. The indexed algorithm handles upto 4000 job classads in at most 580 seconds, compared to only 500 classads in 800 seconds, in the worst case. Again, selectivity is not seen to be a significant factor.

The elapsed time of the indexed algorithm is closely aligned with the number of index probes, as shown in Figure 7. Therefore, we present the rest of our results in probe counts rather than seconds of elapsed time, as probe counts are less sensitive to implementation details such as processor and memory speeds.

Figure 8 exhibits the performance of the dynamic algorithm on the same workload, as measured by the number of index probes. In contrast to the previous algorithms, the performance of the dynamic algorithm does not differ significantly between the 50% and 100% license density regimes, demonstrating the efficacy of the dynamic scheme.

To contrast the performance of the dynamic and the indexed fixed order (i.e., "left-to-right") algorithms, we compare the number of probes issued by each algorithm in the two license density regimes. Figure 9 illustrates representative curves of the number of probes issues by the two algorithms in the 100% license density case with selectivity $S = 1$. We note that the dynamic algorithm actually

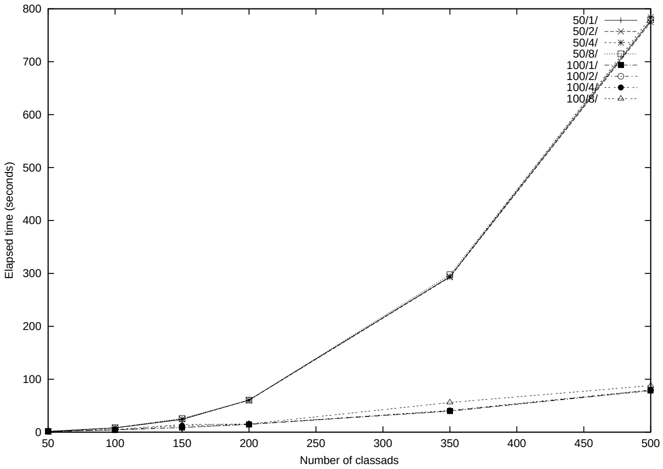

Figure 5. Naive Algorithm Performance

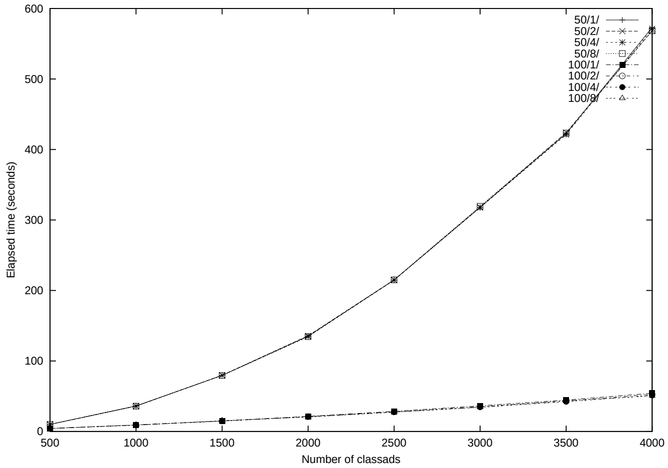

Figure 6. Indexed Left-to-Right Algorithm: Performance

issues *more* probes than the indexed fixed-order algorithm due to the additional probes required by the heuristic function, which then chooses the same "left-to-right" strategy as the fixed-order algorithm. However, the situation is drastically reversed in Figure 10, which illustrates the number of probes issued when the license density is only 50% — the fixed-order algorithm issues upto 425,000 probes, while the dynamic algorithm issues less than 11,000 probes.

To highlight the performance gained by the dynamic algorithm due its agility, we present the performance of a fixed-order indexed algorithm that runs *right-to-left*. The performance of the algorithm is dual to the left-to-right algorithm in that it performs well in the 50% license density case. However, in the 100% workload, the algorithm performs worse than the left-to-right algorithm in the 50% workload. This is due to the fact that unlike workstation that have attributes such as operating system and architecture, there are no attributes to differentiate licenses from one another. Thus, the right-to-left algorithm is severely handicapped by having to consider every single license as a possible candidate. Furthermore, selectivity plays a significant role in this experiment. For higher selectivity values, the reduced machine to license ratio of the 100% workload makes it less likely to obtain a workstation matching a job's constraints once a license has already been picked from a particular partition. Thus, higher selectivity indexes provoke correspondingly worse performance from the right-to-left algorithm.

Thus, we see that the dynamic algorithm is distinctly better than any fixed-order algorithm. The heuristic allows the dynamic algorithm to avoid the pathological cases of the fixed-order algorithms. The stability of the algorithm under workload variations, as characterized by the proximity of the 50% and 100% regime curves in Figure 8 indicate that it is a far better choice than any fixed-order algorithm.

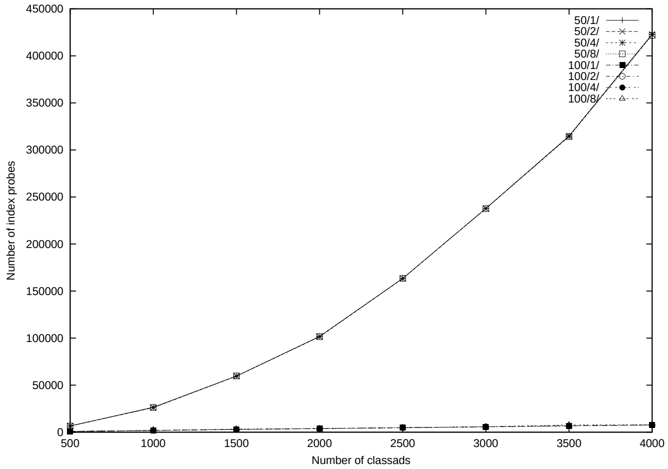

Figure 7. Indexed Left-to-Right Algorithm: Probes issued

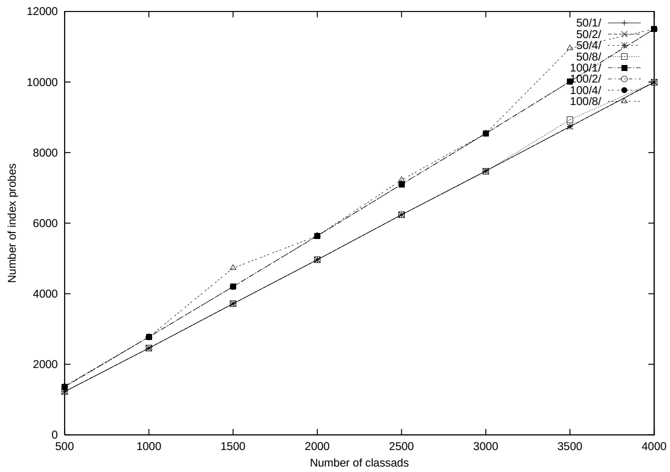

Figure 8. Dynamic Algorithm: Probes issued

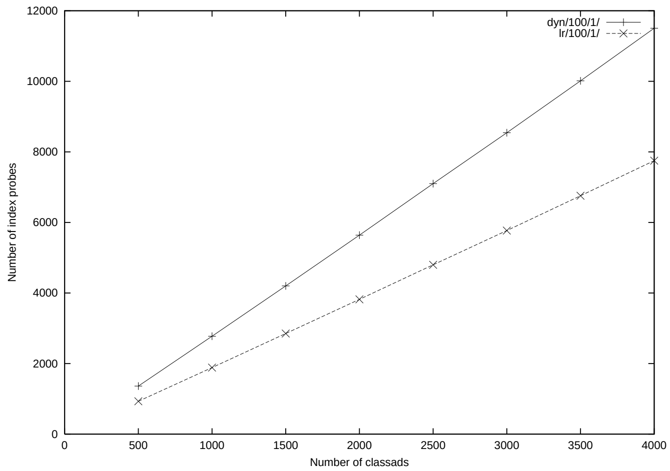

Figure 9. Left-to-Right vs. Dynamic: 100% license density

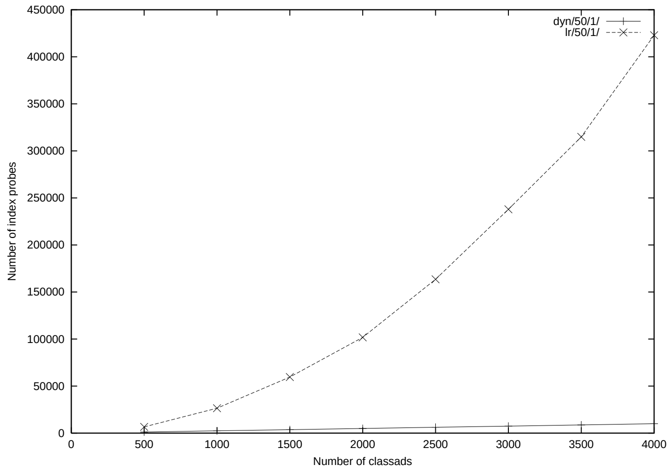

Figure 10. Left-to-Right vs. Dynamic: 50% license density

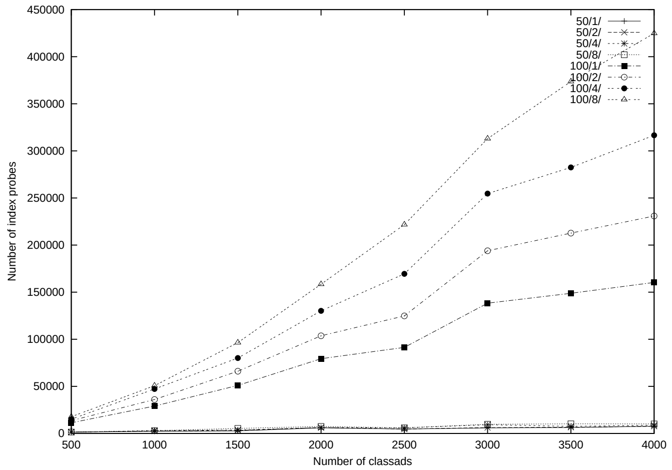

Figure 11. Indexed Right-to-Left Algorithm: Probes issued

7. Related Work

The concept of matchmaking is not new in itself since the topic is widely studied for agent systems. Agents systems such as ACL [5] and RETSINA [16, 17] employ powerful advertisement languages with inferencing capabilities so that general behavioral specifications of agents may be described and reasoned about. In contrast to the knowledge-base representations used in these systems, the classad language uses a database representation. Expression evaluation semantics are simple and lightweight, facilitating efficient and robust implementation.

The classad notation is very similar to that of generalized tuples found in constraint databases [4]. The bilateral matchmaking operation is intuitively similar to a spatial join between server and customer classads. Matchmaking differs from a spatial join in that matchmaking "consumes" classads during the matching process — a classad may be matched at most once. Also, in contrast to constraint database systems, our framework employs a semi-structured data model.

Globus [2, 1] defines an architecture for resource management of autonomous distributed systems with provisions for policy extensibility and co-allocation. Customers describe required resources through a resource specification language (RSL) that is based on a pre-defined schema of the resources database. Although Globus provides flexible APIs to perform more sophisticated co-allocation, these requirements cannot be stated in RSL.

Most resource management systems such as LSF [18], Prospero [11], PBS [6] and NQE [15] process user submitted jobs by finding resources that have been identified either explicitly through a job control language, or implicitly, by submitting the job to a particular queue that is associated with a set of resources. Jobs that require multiple resources must be submitted to queues that can service the special requirements of these jobs. There is no mechanism for a job to marshal a unique mix of resources to service its particular needs.

Set-matching extends the ClassAds language to provide a multilateral matchmaking mechanism where the number of resources is not known *a priory*[9]. However, the set matching mechanism is not capable of marshaling a heterogeneous mix of resources.

The RedLine system casts multilateral matchmaking as a constraint satisfaction problem[8], and is thus capable of using many of the techniques developed for constraint programming. The scheme borrows and extends many of the principles and techniques developed in our matchmaking framework, but uses a substantially more complex advertising language. The recency of the system and lack of experience with it makes more detailed comparison difficult.

8. Conclusions and Future Work

Dynamic, heterogenous and distributively owned resource environments present unique challenges to the problems of resource representation, allocation and management. We have previously demonstrated that the matchmaking paradigm offers a natural solution to these problems, and has been demonstrated to work well in practice. In this paper, we have introduced a multilateral matchmaking extension to address the problem of heterogenous resource co-allocation, motivated by the intent of solving a real problem encountered by production users of the Condor system.

Our contribution is in defining the abstractions that comprise the Gangmatching scheme, and accompanying implementations of multiple algorithms that implement the model, demonstrating the feasibility of the Gangmatching solution. A semi-structured data indexing method and a heuristic-driven dynamic algorithm have been used to efficiently implement the Gangmatching model.

A significant goal of our future work is to incorporate preferences into the gangmatching algorithm, and develop more sophisticated algorithms to cope with the possibility of larger (i.e., both "wider" and "deeper") classad gangs.

There remain some useful extensions to be made to the gangmatching model. A shortcoming of our current formulation is that the number of resources required for a co-allocation must be known *a priori*. While this restriction is reasonable for heterogenous resource co-allocation, there are many situations, such as workstation allocation for parallel computations, when a dynamic number of relatively homogeneous resources are required. It would be interesting to generalize the gangmatching model to address this possibility.

References

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

[2] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. To appear in *International Journal of Supercomputer Applications*.

[3] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*, chapter High Throughput Resource Management, pages 311–336. Morgan Kaufman, 1999.

[4] V. Gaede and M. Wallace. An informal introduction to constraint database systems. *Lecture Notes in Computer Science*, 1191:7–52, 1996.

[5] 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.

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

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

[8] C. Liu and I. Foster. A constraint language approach to grid resource selection. In *Proceedings of the Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12).*, June 2003.

[9] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resourceselection framework for grid applications. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Distributed Computing, July 2002.*, 2002.

[10] 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.

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

[12] R. Raman. *Matchmaking Frameworks for Distributed Resource Man-agement. PhD thesis*. University of Wisconsin-Madison, 2000.

[13] 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.

[14] 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.

[15] C. Research. Document number in-2153 2/97. Technical report, Cray Research, 1997.

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

[17] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 1999.

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