MATCHMAKING FRAMEWORKS FOR DISTRIBUTED RESOURCE MANAGEMENT

By

Rajesh Raman

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

2001

Abstract

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. Distributed environments, particularly those built to support *high-throughput computing* (HTC), are often 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 resource may have its own idiosyncratic allocation policy.

We propose a resource management framework based on a matchmaking paradigm to address these shortcomings. Matchmaking services enable discovery and exchange of goods and services in marketplaces. Agents that provide or require services advertise their presence by publishing constraints and preferences on the entities they would like to be matched with, as well as their own characteristics. A matchmaker uses a matching operation to discover pairings between compatible agents. Since the notion of "compatible" is completely determined by the content of agent *classified advertisements* (classads), a matchmaker can match classads from different kinds of entities in a general manner. Matched agents activate a separate claiming protocol to confirm the match and establish an allocation. The resulting framework is robust, scalable, flexible and evolvable, and has been demonstrated in Condor, a production-quality distributed high throughput computing system developed at the University of Wisconsin-Madison.

The goal of this dissertation is to show the power, flexibility, desirability and feasibility of resource management through matchmaking. We detail the architecture and operation of matchmaking frameworks, and describe mechanisms to implement the components and interactions in such systems.

We describe the architecture of a matchmaking framework that distinguishes itself by providing both bilateral and multilateral matchmaking (i.e., gangmatching) services. The classad language, a semi-structured agent specification language, is presented, and an indexing model for the classad data model is defined. The indexing solution tolerates the lax semantics of semi-structured data models, and indexes both classad attributes and constraints to efficiently identify compatible advertisements. Finally, algorithms that implement the proposed gangmatching model are described, and their performance characteristics analyzed.

Acknowledgements

No sufficiently complex enterprise can be accomplished alone, and graduate research is no exception. Miron Livny provided me with the opportunity of working with a top-class research group and helped me through my graduate years. This research could not have been accomplished if it were not for his insightful guidance as my advisor, and intrepid leadership of the Condor project. Marvin Solomon has been an invaluable source of ideas and technical wisdom, ranging from ASN.1 and "bottom-up semi-naive techniques" to the semantics of comments in Perl and make. Marv also greatly improved the quality of this manuscript by pointing out errors, inconsistencies and typos.

There are many people who have influenced the course of my life. My interest in computer science is almost entirely due to Akshay Kadam and Kiran Pamnany, who have been life-long cohorts and the best of friends. (We'll get down to writing that operating system some day.) Alan Zaring, Kathy Radloff and Jeff Nunemacher at Ohio Wesleyan motivated me to obtain a graduate degree and provided me with an absolutely fantastic undergraduate education. Dad and Mom have been a constant source of support and encouragement. If it were not for the sacrifices they made in getting me through college, none of this would have been possible. Amma and Appa (my other parents) provided an immense amount of support and enabled me to finish my graduate education.

I owe the greatest thanks to the many people who have developed for and been part of Condor Team. Mike Litzkow and Jim Pruyne patiently explained the intricacies of the system when I was still a newbie. Todd Tannenbaum, Derek Wright and Jim Basney are part of the core Condor Team and are largely responsible for Condor's success. I will never forget the heady days of hacking Version 6 with these extremely talented individuals. I also owe a note of sincere thanks to all my colleagues at Condor who have made the project what it is — there are too many of you to list by name, but you know who you are.

Finally, and most specially, I must thank my wife Vidya. Vidya's companionship and love has been instrumental in leading me out of some pretty rough times and bringing me to where I am. My graduate career would not have been possible if were not for Vidya's unconditional TLC, and most excellent back-rubs.

Contents

Chapter 1

Introduction

Conventional resource management systems assume a *system model*, which is an abstraction of the underlying resources, to describe the availability, performance characteristics and allocation policies of the resources being managed. A centralized allocator consults the system model to establish and maintain current and future allocation information in schedules. Indeed, one of the primary activities of conventional resource managers is the creation of efficient allocation schedules, which are usually constructed to optimize a given performance metric, such as response time, utilization, or system throughput. Such resource management strategies work well in *high performance scheduling* regimes, where resources are relatively static and dedicated, time constraints on schedules are tight, and resource usage patterns are firmly legislated and policed.

However, resources in many distributed environments *cannot* be described by system models, and therefore cannot be managed by conventional resource management systems. Issues that obstruct formulation of monolithic system models include:

1. *Distributed Ownership.* In many environments, resources such as powerful commodity workstations are assigned to (and therefore "owned" by) single users and small groups. Each resource in such *distributively owned* environments can exhibit a unique usage pattern and allocation policy, which can vary from indifferent to idiosyncratic. For example, an owner may institute a policy on a workstation that states that a foreign job can be run on the machine only if it was submitted by a member of a particular group, or if the job is run between 6 p.m. and 6 a.m., or if the keyboard has been idle for more than fifteen minutes and the load average is less than 0.3. The behavior of such resources, which transit between available and unavailable states nondeterministically, cannot be captured in predictive system models.

2. *Resource Failure.* Even the most carefully constructed allocation schedules can be frustrated by resource failure. System crashes due to hardware faults, software bugs or power outages are inherently unpredictable, and not uncommon. Furthermore, the probability of system failure rises as the number of resources in the environment increases. When resources that are relied on fail, guarantees made by the allocator to customers in terms of response time or reserved time slots must be retracted, greatly diminishing the benefits of creating allocation schedules.

3. *Heterogeneity.* The semantics of "allocating" a resource to a customer

depends significantly on the type of resource being allocated. For example, the specific semantics of allocating a compute node to a customer is very different from allocating network bandwidth or storage space. It is extremely difficult to capture the varied allocation semantics of both single resources and groups of resources (as might be required when co-scheduling) in a unified system model. The problem of heterogeneity is exacerbated by pool evolution (discussed shortly): even if a unified system model could be constructed, the model would have to be modified frequently.

4. *Evolution.* Most resource pools exhibit almost continuous evolution. Given the rapid development pace and cost depreciation of technology, it is common for resources to be modified, upgraded or even completely replaced relatively often. Pool evolution affects the construction of system models in two ways. First, the resource must be temporarily or permanently retracted from the pool, affecting allocation schedules. Second, the replaced or upgraded resource must be re-introduced into the environment, requiring a change to the system model.

In light of these difficulties, the need for a resource management paradigm that does not require a monolithic system model is clear. We propose a matchmaking resource management solution to address the above problems.

1.1 Matchmaking

The underlying ideas of matchmaking are simple. Servers and customers advertise their presence to a common advertising service by describing their characteristics in advertisements. These advertisements also contain qualitative descriptions of the entities the agents would like to be matched with. A matchmaker discovers compatible providers and customers with a generic matching operation and notifies the matched agents, which then employ a protocol to connect to each other and enable exchange of service.

The matchmaking-based resource management paradigm avoids the problems associated with defining system models and allocation schedules by using an *opportunistic scheduling* paradigm: Resources are used as soon as they become available, and applications are migrated from resources that become unavailable. While this paradigm is not optimal for high-performance computing, the paradigm works well for *high-throughput computing*, where robustness and availability of the environment are very significant factors [37]. The matchmaking scheme has been validated in the Condor [35, 36] high throughput computing system developed at University of Wisconsin-Madison. Condor is heavily used by scientists at numerous sites around the world, and derives much of its robustness and efficiency from the matchmaking architecture.

1.1.1 Philosophy and Architecture

A fundamental observation that drives most of our work is that the dynamics of large heterogeneous and distributed environments require resource management paradigms that approximate the discovery and exchange of goods and services in marketplaces. With this observation in mind, we identify three essential features that guide the design of our framework.

1. We require that the framework be decentralized not only at a physical level, but at a policy level as well. Thus, we require that there be no central authority that dictates how resources or customers may represent themselves, what their local policies may be, or how they may claim each other when matched.

2. Mechanisms and facilities provided to customers should also be made available to servers. Mechanisms available to one set of parties but not another reflect (usually undesirable) assumptions about the behavior of participating principals.

3. The environment must present the image of a single "clearing house" of providers and requesters. This requirement greatly simplifies the specification of policies for both servers and customers, since there is an easily definable global notion of the "best match candidate."

A detailed discussion of these requirements is presented in Chapter 3.

1.1.2 Components

The main actions involved in the matchmaking process are advertising, matching, notification and claiming. The above interactions between the matchmaker and other principals participating in the matchmaking environment motivate the definition of the following components of a matchmaking service.

1. A language for specifying the characteristics, constraints and preferences of principals. Our framework uses the *classified advertisement* (classad) language for this purpose. Figure 1 shows a classad representing a workstation. The `Constraint` attribute indicates that the workstation only runs jobs when it is idle (as determined by load average and keyboard idle time) and if the job's image size is 15 megabytes less than the physical memory size of the machine, with a preference for jobs submitted by user "raman." The classad language is a symmetric description language; both servers and customers use the same language to describe their respective characteristics, constraints and preferences. Among other constructs that allow entities to be easily represented, the language supports semi-structured "records" that are used as the descriptions themselves, and dynamic typing with special **undefined** and **error** values that enable robust evaluation semantics in heterogeneous environments.

2. The *Matchmaker Protocol* is composed of the *publishing protocol* and *notification protocol* that respectively describe how agents communicate with

```
[
  Type        = "Machine";
  Activity    = "Idle";
  KeybrdIdle  = '00:23:12'; // h:m:s
  Disk        = 323.4M;     // mbytes
  Memory      = 64M;        // mbytes
  State       = "Unclaimed";
  LoadAvg     = 0.042969;
  Mips        = 104;
  Arch        = "INTEL";
  OpSys       = "SOLARIS251";
  KFlops      = 21893;
  Name        = "foo.cs.wisc.edu";
  Rank        = other.Owner=="raman" ? 1 : 0;
  Constraint  = other.Type == "Job"
                   && LoadAvg < 0.3 && KeybrdIdle>'00:15'
                   && Memory - other.ImageSize>=15M
]
```

Figure 1: A classad describing a workstation

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 relates the contents of submitted classads and the state of the system to the matches that will be created. As part of this process, the algorithm defines a set of conventions (an *advertising protocol*) which binds meanings to certain classad attributes that will be used for special purposes. For example, a matchmaker may define that in any classad, the attributes named `Constraint` and `Rank` will be respectively treated as the constraints and preferences defined by the advertising entity.

4. The *Claiming Protocol* is activated between the matched parties to confirm the match and establish a working relationship. In our resource management framework, we require that the locus of control for claiming reside in the matched agents themselves. An important motivation for this requirement is that, in our framework, a match between A and B is not the same as allocating A to B. Instead, the match is *permission* for A and B to cooperate — it is the agents' responsibility to verify the match and decide if cooperation is still desirable. Either entity may choose to not go further and reject the match altogether.

1.1.3 Advantages

Realizing opportunistic scheduling through matchmaking addresses several of the problems associated with creating system models and allocation schedules.

1. The expressive nature of the classad language addresses the problems of heterogeneity and distributed ownership associated with conventional resource management systems. Sophisticated policies placed on resources by resource owners can be expressed and enforced. The language allows various kinds of entities to enter the environment and represent themselves effectively without requiring any change to the matchmaker in terms of schema evolution or resource-specific modifications. Thus a classad-based matchmaking framework facilitates management of distributively owned, heterogeneous resource environments.

2. By placing time-outs on advertisements, common problems associated with resource failure and evolution can be avoided: when a resource fails or is temporarily removed from the environment for upgrade, it "ceases to exist" after its advertisement expires. (A modified resource re-enters the pool in the same manner as a completely new resource.) Heart-beat protocols can be piggy-backed on advertisement updates for dynamic information (such as load average), with which the state of the environment can be deduced fairly accurately. Matches made with stale advertisements can be efficiently detected and handled by the claiming protocol, as will be discussed shortly.

A key idea of matchmaking is that "resource allocation" is a two step process: matching and claiming. Matching is performed in a centralized manner to enforce fairness with pool-wide priority mechanisms. Centralized matchmaking also provides all servers and customers with wide access to the classads of all potential agents they would like to be matched with and facilitates easier administration and control. In contrast, the claiming phase of the operation is performed in a distributed manner.

The separation of allocation into matching and claiming has several benefits.

Weak consistency requirements. Since the state of service providers and requesters may be continuously changing, there is a possibility that the matchmaker made a match with a stale advertisement. Claiming allows the provider and customer to verify their constraints with respect to their current state. This toleration of weak consistency makes the remainder of the system significantly simpler, more robust, and more efficient.

Authentication. The claiming protocol may use cryptographic techniques for the provider and customer to convince each other of their identities. A challenge-response handshake can be added to the claiming protocol at very little cost.

Bilateral specialization. In dynamic heterogeneous environments, it is very difficult to write a matchmaker that is aware of the specifics of allocating all the different kinds of resources that may be added to the environment. Indeed, the myriad kinds of resources already present in the environment may itself present the problem of packing all the resource specific allocation code in the matchmaker.

By pushing the establishment of allocation to the claiming stage, the details of allocation are contained in the entities that really need to interact with specific kinds of providers and customers. The matchmaker may be written as a general service that does not depend on the kinds of services and resources that are being matched.

Bilateral specification implies that since the system does not assume a single monolithic or static allocation model, the allocation models are supplied by the entities involved in providing and using services. The matchmaking framework thus allows several dissimilar "allocation models" to coexist in the same resource management environment.

End-to-end verification [45]. The principals involved in a match are themselves responsible for establishing, maintaining and servicing a match. The matchmaker only holds a "soft state" representation of active claims for accounting purposes, a fact that simplifies recovery in case of failure and makes the system more scalable.

1.2 Overview of the Dissertation

The goal of this body of work is to demonstrate the power, flexibility, feasibility and desirability of matchmaking as a resource management paradigm. As such, we identify all the primary components and interactions that must exist in a matchmaking environment, and describe mechanisms to address the difficulties that arise when deploying such a framework in a resource management context.

A fundamental notion in any matchmaking environment is entity description, which is accomplished with the use of a description language. In Chapter 2, we describe in detail the design goals, structure and semantics of the description language used in our matchmaking framework, the classified advertisement (or classad) language.

The architecture of our matchmaking model is detailed in Chapter 3. While utilization of the matchmaking paradigm in resource management is in itself a unique proposition, the proposed model also distinguishes itself in providing both bilateral and multilateral matchmaking services. The issues of advertisement and advertisement mechanisms are addressed, the semantics of matches are described, and the functionality implications of notification and claiming protocols are discussed.

Although the classad language affords a relatively light-weight mechanism to test for advertisement compatibility, the presence of a large number of servers and customers in a resource management environment can still place a significant load on the matchmaking process. However, due to the database representation of classads, indexing technologies may be utilized to significantly decrease the cost of identifying compatible classads. In Chapter 4, we present a complete indexing solution developed for the semi-structured classad data model which, when presented with a classad, efficiently identifies all possible candidate matches by indexing both attributes and constraints. The indexing scheme has been designed to work as a component of both bilateral and multilateral matchmaking algorithms, and makes very significant improvements to overall matchmaking performance.

Algorithms for multilateral matchmaking are significantly more complex than their bilateral counterparts. Multilateral matchmaking is fundamentally a combinatorial algorithm, which results in extremely large execution times even for problems of moderate size if solved with naive algorithms. In Chapter 5, we describe several gangmatching algorithms and present the results of a performance study.

Matchmaking for resource management is a new area, and although this body of work identifies and addresses many fundamental issues, the problem has many aspects that are worth further investigation. In Chapter 6, we summarize our work, present our contributions and identify directions for future research and study.

Chapter 2

The Classified Advertisements Language

A fundamental component of any matchmaking environment is the language used to describe entities in advertisements. The advertisement language serves as a common communication substrate to enable the various principals to interact with each other and communicate their notions of "compatibility" concisely and unambiguously. As such, the flexibility, robustness and inherent complexity of an advertisement language has profound implications on the functionality and efficiency of a matchmaking environment.

In this chapter, we describe the *classified advertisement* (classad) language, which is a simple, expressive and flexible language employed by our framework. We begin with a discussion of the desired properties of an advertisement language to motivate and justify our design. We then present a brief overview of the main characteristics of the language in the context of matchmaking, followed by a more detailed description of the language's structure and semantics. Concrete examples of non-trivial and practical policies are then encapsulated in classads and presented to exemplify the salient features of the language. Certain classad processing algorithms are extremely useful as fundamental operations in defining more complex algorithms that streamline the matchmaking process. We describe these algorithms and their usefulness in the context of the matchmaking scheme, and conclude with a brief survey of related work.

2.1 Design Goals

The matchmaking framework design principles of symmetry and decentralization provide very specific language design directions: clearly, both requests and offers must be similar in structure, and no centralized schema may be used. These points and other key design issues are enumerated below.

1. **Symmetric.** As discussed earlier, a key requirement of our matchmaking framework is that the matchmaking model and mechanisms be symmetric to both providers and requesters. This requirement includes the advertisement language too. The implication of this requirement is that the advertisement language must be powerful and flexible enough to subsume the functionality of traditional resource description and resource selection languages commonly found in conventional resource management systems. We also require the language to provide the dual properties of customer description and customer selection.

2. **Semi-structured.** The proscription of centralized control (and hence

centralized schema management) naturally suggests the use of a semi-structured model as the basis of the description language. Semi-structured data models (such as XML [9]) are finding widespread acceptance due to their flexibility in managing heterogeneous and distributed information. In the context of matchmaking, the use of a semi-structured data model would facilitate the representation of heterogeneous entities, but also introduce the problem of determining compatibility with partial or missing information. Our solutions to these (and other) problems associated with the adoption of a semi-structured model are discussed shortly.

3. **Declarative.** We require that the advertisement language be declarative rather than procedural. By this we mean that advertisements should describe notions of compatibility "qualitatively," rather than specifying a procedure for determining compatibility. The significant advantages that declarative specifications provide over their procedural counterparts, especially with regard to alternative implementation strategies and optimization, are well known. In addition, declarative specifications simplify the process of specifying, understanding and maintaining policy specifications.

4. **Simple.** It is extremely important for an advertising language to be simple both syntactically and semantically. A complex specification language

is less amenable to efficient and correct implementation. Complex languages also compound the process of specifying and understanding policies, making both manual and automatic policy management difficult. Experience with resource owners in practice have shown that although policies may be complex with respect to the number of variables involved, policies are fundamentally simple predicates which do not require a powerful Turing-complete language to be specified. The use of a non-Turing-complete advertisement language also avoids "Halting Problem" issues, so compatibility between classads may be tested at low, known costs.

5. **Portable.** A key advantage of matchmaking systems is their ability to manage heterogeneity naturally and efficiently. However, this property is predicated on the portability of the advertisement language itself. Specifically, the language must be amenable to efficient implementation on various hardware and software platforms. Thus, it is not reasonable to introduce language features that require specific features of the host architecture that may not be widespread.

2.2 Overview

The classad language is a simple expression-based language that has been designed to meet the above design goals. The central construct of the language is the *classad*, which is a record-like structure composed of a finite number of distinctly named expressions, as illustrated in Figure 2. Classads are used as attribute lists by entities to describe their characteristics, constraints and preferences. Since whole expressions (and not just scalar values) are bound to attribute names, classads can naturally accommodate the predicate-like constraints used by principals to define their policy requirements. Similarly, preferences are specified as expressions that are evaluated to numeric values denoting the “goodness” of candidate matches.

```
[
  Type         = "Job";
  QDate        = 'Mon Jan 11 10:53:31 1999 (CST) -06:00';
  CompletionDate = undefined;
  Owner        = "raman";
  Cmd          = "run_sim";
  WantRemoteSyscalls = true;
  WantCheckpoint = true;
  Iwd          = "/usr/raman/sim2";
  Args         = "-Q 17 3200 10";
  ImageSize    = 31M;
  Rank         = other.KFlops/1E3 + other.Memory/32;
  Constraint   = other.Type == "Machine" &&
       other.Arch=="INTEL" && other.OpSys=="SOLARIS251" &&
   other.VirtualMemory > self.ImageSize
]
```

Figure 2: A classad describing a submitted job

The classad language differentiates between *expressions* and *values*: Expressions are evaluable language constructs obtained by parsing valid expression syntax, whereas values are the results of evaluating expressions. The classad language employs *dynamic typing* (or *latent typing*), so only values (and not expressions) have types. The language has a rich set of types and values which includes many traditional values (numeric, string, boolean), non-traditional values (timestamps, time intervals) and some esoteric values, such as **undefined** and **error**. **Undefined** is generated when an attribute reference cannot be resolved, and **error** is generated when there are type errors. In a sense, all classad operators are *total functions*, since they have a defined semantics for every possible operand value, facilitating robust evaluation semantics in the uncertain semi-structured environment.

Classads may be nested to yield a hierarchical name-space, in which case *lexical scoping* is used to resolve attribute references. The scoping features of the language in context of the "match evaluation environment" established by the matchmaker to test matches result in the semantics that an attribute reference made from either customer or resource classad of the form "`other.`*attribute-name*" refers to an attribute named *attribute-name* of the other ad. In addition, every classad has a builtin attribute `self` which evaluates to the innermost classad containing the reference, so the reference "`self.`*attribute-name*" refers to an attribute of the ad containing the reference. If neither `self` nor `other` is mentioned explicitly, the evaluation mechanism assumes the `self` prefix. For example, in the `Constraint` of the job ad in Figure 2, the sub-expression `other.VirtualMemory > self.ImageSize` expresses the requirement that the target machine have sufficient virtual memory to accommodate the requirements of the job. The expression could also have been written `other.VirtualMemory > ImageSize`.

A reference to a non-existent attribute evaluates to the constant **undefined**. Most operators are “strict” with respect to this value—if either operand is **undefined**, the result is **undefined**. In particular, comparison operators are strict, so that

```
 other.Memory > 32,
 other.Memory == 32,
 other.Memory != 32,
```

and

```
!(other.Memory == 32)
```

all evaluate to **undefined** if the target classad has no `Memory` attribute. The Boolean operators `||` and `&&` are non-strict on both arguments, so that

```
other.Mips >= 10 || other.Kflops >= 1000
```

evaluates to **true** whenever either of the attributes `Mips` or `Kflops` exists and satisfies the indicated bound. There are also non-strict operators `is` and `isnt`, which always return Boolean results (not **undefined**), allowing explicit comparisons to the constant **undefined** as in

```
other.Memory is undefined || other.Memory < 32.
```

2.3 Types and Values

We view types as a partitioning of the universe of values in the language, where every partition is non-empty. To aid in the unambiguous definition of language semantics, we define fixed internal implementation representations for certain values (such as numbers), while leaving representations of other values unspecified.

Values in the classad language may be one of the following types.

Undefined. The undefined type has exactly one value: the **undefined** value. As its name suggests, the **undefined** value represents incomplete or unknown evaluation results due to absent information. The adoption of a semi-structured data model *requires* the inclusion of an **undefined** (or similar) value for robust evaluation semantics.

Error. The error type has exactly one value: the **error** value. Similar to the **undefined** value, the **error** value plays an important part in securing robust evaluation semantics in semi-structured environments. While the **undefined** value represents missing information, the **error** value represents incorrect or incompatible information, and is usually generated when operators are supplied with values that are outside the domains of their operands. For example, the quotient of a number and a string is **error**.

Boolean. There are exactly two distinct boolean values: **false** and **true**. Unlike their C and C++ counterparts, boolean values are not considered numeric values, and therefore cannot be directly used in numeric expressions.

String. String values are finite sequences of non-zero 8-bit ASCII characters. There is no *a priori* limit of the length of string values.

Integer. Integer values are signed 32-bit two's complement numbers.

Real. Real values are IEEE-754 double precision numbers.

Absolute Time. Absolute time values are non-negative discrete integral values recording the number of seconds elapsed between the UNIX epoch (i.e., 1 January 1970) and the timestamp represented by the value. Absolute time values must be able to represent the largest integer value as a valid timestamp.

Relative Time. Relative time values are discrete integral values that represent time intervals in seconds. Relative time values may be negative or zero. The cardinality of the relative time value set must be at least as large as the set of integer values.

Classad. Classad values are finite sets of (*identifier*, *expression*) pairs, where each identifier is distinct (ignoring case). Identifiers are strings of alphanumeric characters and underscores, which begin with non-numeric characters. Classad values additionally indicate (directly or indirectly) the presence of a *parent classad* (or parent scope), which is the closest enclosing classad. If a classad is not lexically nested, it is called a *toplevel* (or root) classad, and its corresponding value does not have a parent scope component.

List. List values are finite sequences of expressions.

Pedantically, classad and list values do not have external representations — only classad and list *expressions* do.[1] This situation is similar to lambda expressions and closures in LISP/Scheme.

2.4 Expressions and Evaluation Semantics

The majority of the classad language is straightforward and familiar, with some modest extensions. Most of the subtlety of the classad language lies in the treatment of attribute references, which operate in a lexical scoping formalism, but may also explicitly traverse the hierarchical classad namespace during an evaluation to access an attribute.

All expression evaluations occur in the context of a given classad, which may be nested arbitrarily deep inside other classads. However, for any given expression evaluation, there is a single unique outermost classad that is not nested. We designate this classad the *root* (or *toplevel*) classad.

[1]Pragmatically, however, the scope information associated with these values may be ignored to provide a usable external representation if interpreted in context.

2.4.1 ClassAd Expressions

A classad is constructed with the mixfix classad construction operator, as shown in the syntax schema below.

$$\texttt{[}name_0 \texttt{ = } expr_0 \texttt{ ; } name_1 \texttt{ = } expr_1 \texttt{ ; } \ldots \texttt{ ; } name_n \texttt{ = } expr_n \texttt{]}$$

Each $name_i$ is a unique identifier, and each $expr_i$ is an expression. A classad expression evaluates to a classad value. Every classad value has three implicit attributes: `self`, `parent` and `root`. These attributes are reserved in the concrete syntax and therefore may not be used as any of the $name_i$.

2.4.2 List Expressions

A list is constructed with the list construction operator as illustrated below.

$$\texttt{\{}expr_0 \texttt{ , } expr_1 \texttt{ , } \ldots \texttt{ , } expr_n \texttt{ \}}$$

A list expression evaluates to a list value, which can later be used as an array in subscript expressions.

2.4.3 Literals

Literals are atomic expressions that directly evaluate to scalar values (i.e., non-classad and non-list values). In this sense, literals directly represent the values that they evaluate to. Examples of literal expressions for values of the various types are provided below. (With the exception of string literals, all literals are case-insensitive.)

- Undefined: `undefined`
- Error: `error`
- Boolean: `false`, `true`
- String: `"foo"`, `"bar\n\t"` (C-style escapes are supported.)
- Integer: `10`, `0xff` (Hex), `0600` (Octal)
- Real: `3.141`, `6.023e23`, `2K` (i.e., 2048.0) The suffixes `B`, `M`, `G` and `T` representing scale factors of 2^0, 2^{10}, 2^{20} and 2^{30} are also supported.
- Absolute Time: `'Thu Aug 17 18:21:07 2000 (CDT) -06:00'`
- Relative Time: `'18:21:32'`, `'3d19:49:15'`

2.4.4 Operations

Operations are expressions that combine other expressions by means of unary, binary and ternary operators. The operators are essentially those of the C language, with certain operators excluded (e.g., pointer and dereference operators) and others added (e.g., non-strict comparison). Thus, a rich set of arithmetic, logic, bitwise and comparison operators are defined. The set of supported operators and their relative precedences are summarized in Figure 3.

In the following specification of operator semantics, it is to be assumed that unless otherwise specified, operators are strict with respect to the **undefined** and **error** values in all places, with **error** taking precedence over **undefined**.

Operator class	Operators	Associativity
Primary	`[]`	Left to right
Unary	`- + ! ~`	Right to left
Multiplicative	`* / %`	Left to right
Additive	`+ -`	Left to right
Shift	`<< >> >>>`	Left to right
Relational	`< <= > >=`	Left to right
Equality	`== != is isnt`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`?:`	Right to left

Figure 3: Classad language operators in decreasing order of precedence

Additionally, since most operators are meaningfully defined only over certain values, we define operations to evaluate to **error** when values outside the domain of an operator are supplied as operands. In other words, unless otherwise specified, the following implicit rules must be applied (in order) to all following specifications:

A. (Strictness Rule.) If any operand to an operator is **undefined** (**error**), the resulting value of the operation is also **undefined** (**error**). If both **undefined** and **error** are simultaneously supplied to an operator, the result is **error**.

B. (Domain Rule.) If the operands to the operator are outside the operator's domain, the resulting value of the operation is **error**.

We now informally describe the behaviors of operators in the classad language.

Arithmetic Operators

All arithmetic operators are binary, and follow both Strictness and Domain Rules. The domain for arithmetic operators is numeric values, i.e., the integer and real values. With the inclusion of the following rules, arithmetic in the classad language occurs in "the natural way."

1. If the divisor is zero in the case of the division (`/`) and remainder (`%`) operators, the evaluation result is **error**.
2. If one operand is integer and the other is real, the integer operand is promoted to a real, and the evaluation proceeds as a computation of real numbers. Unless the expression violates any of the previous rules, the type of the evaluation result is real.

Comparison Operators

All comparison operators are binary and, with the exception of the `is` and `isnt` operators, follow both Strictness and Domain Rules. The following rules define the behavior of strict comparison.

1. Only values of the same type may be compared. The only exception to this rule is that integers and reals may be compared — the integer is promoted to a real, and comparison proceeds as with real values.

2. Only scalar values may be compared. Comparison of aggregate values (i.e., classads and lists) results in **error**.

3. (Boolean specialization.) The **false** value is defined to be less than the **true** value.

4. (String specialization.) All string comparisons are case-insensitive, so `"FOO"`, `"fOO"` and `"fOo"` are all equivalent. Strings are ordered lexicographically, ignoring case.

5. (Absolute time specialization.) An absolute time value is defined to be less than another if the timestamp it represents temporally precedes the timestamp represented by the other comparand.

6. (Relative time specialization.) Shorter intervals are less than longer intervals.

The non-strict comparison operators `is` and `isnt` implement the "is identical to" and "is not identical to" predicates, and can therefore be used to test if given values are **undefined** or **error**. By definition, these operators follow neither Strictness nor Domain Rules — these operators *always* evaluate to **true** or **false**. The following rules, when applied in order, summarize the behavior of the `is` operator — the `isnt` operator is simply the boolean negation of the `is` operator.

1. If the types of the two comparands differ, the result of the comparison is **false**.

2. If the type of one comparand is **undefined** (**error**), the result of the operation is true if the other comparand is also **undefined** (**error**), and false otherwise.

3. Comparison of aggregate values is not allowed, so the result of the `is` operator is **false** if either operand is an aggregate value.

4. Comparison of string values is case sensitive. This behavior is different than that of the strict comparison operators.

5. Otherwise, the `is` operator behaves exactly like the equals comparison operator (`==`).

Bitwise Operators

The bitwise operators follow both Strictness and Domain rules, and are applicable only to integer values. The operators behave identically to their counterparts in the Java programming language.

Logic Operators

The logic operators OR (`||`) and AND (`&&`) are non-strict operators, and therefore do not follow the implicit Strictness Rule. Instead the operators follow the truth tables supplied below, in which T, F, U and E stand for **true**, **false**, **undefined** and **error** respectively. If any operand does not evaluate to a boolean, undefined or error value, the result of the operation is `error`.

AND	F	T	U	E
F	F	F	F	E
T	F	T	U	E
U	F	U	U	E
E	E	E	E	E

OR	F	T	U	E
F	F	T	U	E
T	T	T	T	E
U	U	T	U	E
E	E	E	E	E

NOT	
F	T
T	F
U	U
E	E

Miscellaneous Operators

The Subscript Operator The subscript operator is a binary operator that follows both Strictness and Domain Rules. It requires one list type operand (i.e., an array), and one integer type operand (i.e., an index). If the supplied index is not a non-negative integer less than the length of the array, the operation evaluates to **undefined**. Otherwise, the result of the operator is the value of the index'th expression in the array (with zero based indexing).

The Conditional Operator The conditional operator is the only ternary operator in the classad language. It follows the Strictness and Domain rules only with respect to its first operand (the condition), which is required to be boolean. The result of the evaluation is the value of the second operand (the true consequent) if the condition evaluates to **true**, and the value of the third operand (the false consequent) if the condition evaluates to **false**.

2.4.5 Attribute References

Attribute references in the classad language are similar to both variable references in programming languages like C and C++, and filenames in the UNIX filesystem. In the following description of the three variants of attribute reference expressions, *attr* denotes a case-insensitive identifier and *expr* denotes an arbitrary expression.

attr This attribute reference variant has two possible behaviors. If *attr* is one of the following special built-ins, the reference evaluates to certain predefined values.

1. The *self* attribute reference evaluates to the classad that serves as the current scope of evaluation.
2. The *root* attribute reference evaluates to the classad that serves as the root of the evaluation.
3. The *parent* attribute reference evaluates to the classad that is the lexical parent of the current evaluation scope. If the current evaluation scope is the root scope, the *parent* attribute reference evaluates to **undefined**.

If the reference is not one of the above three special built-ins, the reference evaluates to the value of the expression bound to the attribute named *attr* in the closest enclosing scope. (The obtained expression must be evaluated in the same scope that it was found.) If no such attribute is found, the reference evaluates to the **undefined** value.

.attr This attribute reference variant evaluates to the value of the expression bound to the name *attr* in the root scope, when evaluated in the root scope. If the root scope does not contain an attribute named *attr*, the value of the reference is **undefined**.

expr.attr This variant first evaluates the expression *expr*, which must evaluate to a classad. (If this expression evaluates to **undefined**, the value of the entire reference is **undefined**. Otherwise, if the value is not a classad, the value of the reference is **error**.) The value of the reference is the value of the expression bound to the attribute named *attr* in the closest enclosing scope *beginning with the classad scope identified by* expr. As with previous variants the identified expression must be evaluated in the scope it was obtained from, and if no such expression exists, the value of the reference is **undefined**.

2.4.6 Function Calls

The classad language provides a number of built-in utility functions to perform tasks such as string pattern matching, obtaining the current time of day, converting values from type to another and testing value types. A comprehensive list of functions and their behaviors is provided in Appendix A.

2.4.7 Circular Expression Evaluation

It is trivially possible for expressions in the classad language to refer to each other in a manner that would lead to an infinite loop during expression evaluation. For example, in the classad `[a=b; b=a]`, it is not possible to determine the value of either attribute. The classad language defines that circular expression evaluation result in the **undefined** value.

2.5 Example ClassAd Policies

2.5.1 Workstation Access Control

Figure 4 shows a classad that describes a workstation and demonstrates the flexibility of the mechanism in expressing fairly sophisticated policies. The `Constraint` attribute indicates that the workstation is never willing to run applications submitted by users “rival” and “riffraff,” it is always willing to run the jobs of members of the research group, friends may use the resource only if the workstation is idle (as determined by keyboard activity and load average), and others may only use the workstation at night. The `Rank` expression states that research jobs have higher priority than friends’ jobs, which in turn have higher priority than other jobs.

```
[
  Type       = "Machine";
  Activity   = "Idle";
  KeybrdIdle = '00:23:12'; // h:m:s
  Disk       = 323.4m;     // mbytes
  Memory     = 64m;        // mbytes
  State      = "Unclaimed";
  LoadAvg    = 0.042969;
  Mips       = 104;
  Arch       = "INTEL";
  OpSys      = "SOLARIS251";
  KFlops     = 21893;
  Name       = "foo.cs.wisc.edu";
  ResearchGp = { "raman", "miron", "solomon" };
  Friends    = { "calvin", "hobbes" };
  Untrusted  = { "rival", "riffraff" };
  Rank       = member(other.Owner, ResearchGp) ? 10 :
                member(other.Owner, Friends) ? 1 : 0;
  Constraint = !member(other.Owner, Untrusted) && Rank>=10 ? true :
                Rank>0 ?  LoadAvg < 0.3 && KeybrdIdle>'00:15' :
                DayTime()<'8:00' || DayTime()>'18:00'
]
```

Figure 4: Workstation Access Control

2.5.2 Time-Dependent Resource Preference

Customers may incorporate environment specific information to improve the quality of service delivered to their applications. For example, many of the university's workstations that are used for instructional purposes exist in computer laboratories that are locked during the night. Thus, it is beneficial for applications to run on these machines after hours, as they will not be preempted by machine owners during this time.

Figure 5 describes a job that has the policy of running only on INTEL machines with sufficient memory and disk space, running the Solaris 2.5.1 operating system. In addition, the `Rank` expression in the job classad expresses a preference for running on instructional machines during the night over running on a machine that has been idle for a long time (and is therefore likely to remain unused), which is in turn preferred over running on any other machine.

```
[
  Type           = "Job";
  CompletionDate = undefined;
  RemoteSyscalls = true;
  Checkpoint     = true;
  QDate          = 'Mon Jan 11 10:53:31 1999 (CST) -06:00';
  Owner          = "raman";
  Cmd            = "run_sim";
  Iwd            = "/usr/raman/sim2";
  Args           = "-Q 17 3200 10";
  ImageSize      = 31M;
  Rank           = DayTime()>'20:00' && DayTime<'8:00' &&
                     other.IsInstructional ? 10 :
                     other.KeybrdIdle>'3:00' ? 5 : 0;
  Constraint     = other.Type=="Machine" && Arch=="INTEL" &&
                     OpSys=="SOLARIS251" && Disk >= 10M &&
                     other.Memory >= self.ImageSize
]
```

Figure 5: Time Dependent Resource Preference

2.5.3 Time-Dependent Resource Constraints

We now present an example in which a customer varies the resource constraint over time. In the example illustrated in Figure 6, the customer waits for up to two hours for a resource with at least one gigabyte of memory. If a match hasn't been found with two hours, the customer downgrades the resource requirement to a resource that has at least half a gigabyte of memory. The `Rank` expression states that machines with larger memories are preferred. In this example, the

```
[
  Type           = "Job";
  CompletionDate = undefined;
  RemoteSyscalls = true;
  Checkpoint     = true;
  QDate          = 'Mon Jan 11 10:53:31 1999 (CST) -06:00';
  Owner          = "raman";
  Cmd            = "run_sim";
  Iwd            = "/usr/raman/sim2";
  Args           = "-Q 17 3200 10";
  ImageSize      = 31M;
  ElapsedTime    = CurrentTime( ) - QDate;
  Rank           = other.Memory;
  Constraint     = other.Type=="Machine" && Arch=="INTEL" &&
                    OpSys=="SOLARIS251" &&
                    other.Memory >= (ElapsedTime>'2:00'?0.5G:1.0G)
]
```

Figure 6: Time Dependent Resource Constraints

customer will reject machines with less than one gigabyte of memory for the first two hours, hoping for a better match. This policy is therefore fundamentally different from one that merely prefers machines with larger physical memories.

2.6 Useful ClassAd Processing Algorithms

The generality of classad expressions is the source of the classad language's flexibility and expressive power. However, this generality also incurs costs that affects the performance of the matchmaking scheme. In this chapter, we describe classad processing algorithms that assist in the efficient management of classads by discovering and simplifying the structure of classad expressions. These algorithms are extremely useful as fundamental operations in defining a more streamlined and efficient matchmaking process. We describe these algorithms and their usefulness in the context of the matchmaking scheme.

2.6.1 Specialization

Classad constraints are formulated as a combination of predicates that define conditions that must be met for a successful match. Common patterns of such conditions include time-specific predicates and comparisons of the attributes of candidate match classads with local attributes. For example, a machine may declare that it is unavailable between 9:00am and 5:00pm, and only applications with an image size less than the available virtual memory are admitted. Many variables involved in specifying such policies may be resolved before the classad is actually tested against candidate matches. For example, the advertising principal of the machine in the above scenario may decide to not advertise if it determines that it is unavailable given the current time of day. Since the current time of day is a constant at the time of advertisement, the specialization mechanism may be invoked to substitute this value into, and therefore simplify, the advertisement's constraint. In similar spirit, the known available virtual memory size may be substituted for the virtual memory variable in the example, to yield a simpler constraint.

We call the above process *specialization*, since a more general and complex expression is specialized to a simpler expression given known values. The process of specialization may be viewed either as *constant folding*, similar to the process commonly used in optimizing compilers, or as *partial evaluation*, since a constraint is simplified when faced with known static inputs and unknown dynamic inputs. Algorithmically, these view points are similar due to the absence of iterative structures and user-defined procedures in the classad language.

Specialization is intuitively similar to ordinary evaluation, except that the result of specializing an expression whose value is unknown is not the **undefined** value, but the expression itself. When adequate information is available, these "expression results" are squashed or propagated by operators analogous to the **undefined** value. For example, `false && other.x > 10` would specialize to **false**, and `true && other.x > 10` would specialize to `other.x > 10`. Furthermore, constants are aggregated appropriately to yield simpler expressions, so `3 + a + 7` would be specialized to `10 + a`.

The utility of specialization is threefold:

1. Specialization provides advertising principals with a mechanism to determine their current availability accurately and efficiently. In the context of the policy specified above, if a classad was advertised for the machine between 9:00am and 5:00pm, the absence of a match from the matchmaker could signify one of several possible situations.

    - There are no customers in the system.
    - The machine does not match the constraints of any customer.

- The machine is unavailable to all current customers, but is available to some (possibly hypothetical) customer.
- The machine is unavailable to all customers.

While identifying the first two situations is not very important to the machine's access control mechanism, differentiating between the last two situations is often very important. For example, the resource agent used in Condor uses a specialization mechanism to determine if it is in "owner state" (i.e., unavailable to all) or not. The agent's behavior differs considerably between owner and non-owner states.

2. Specialization reduces the number of classads that must be considered when matchmaking. If an agent can determine that it is unavailable to everybody given its current state and constraints, it need not publish an advertisement. The matchmaking algorithm may therefore typically be run on problem instances of reduced size compared to a framework without specialization.

3. Specialization makes the matchmaking algorithm more efficient, since specialized constraints consist only of the minimal expressions that *must* be verified *vis a vis* candidate matches.

2.6.2 External Reference Determination

Since principals in the system do not know which of their attributes are of interest to candidate matches *a priori*, there is an incentive for principals to describe themselves comprehensively to attract as many candidate matches as possible. Thus classads may include a very large number of attributes. At any given time, however, most of these attributes may not be accessed by any principal. It is therefore useful to identify exactly which attributes of principals are of interest at any point in time, following which processing techniques that focus on the efficient management of these attributes may be applied, leading to a more efficient matchmaking scheme.

The algorithm to determine the set of "interesting attributes" is a modification of a bound/free variable analysis algorithm that returns the set of free variable references given a classad and its constraints, i.e., the external references of a classad. Given a set of request classads, the set of offer attributes that are accessed by the requests is simply the union of the external reference sets of each request classad. Similarly, the set of request attributes accessed is the union of the external references of the offer classads.

The external reference algorithm is similar in some ways to the specialization algorithm discussed previously. However, the algorithm is simpler in many ways, since it is only an identification algorithm and does not include any notion of optimization. Indeed, external references may be determined by the specialization algorithm since the specialization algorithm must be able to identify external references for correct operation. Nevertheless, it is instructive to decouple these algorithms conceptually since their respective outputs are used in fundamentally different ways: while specialization facilitates intra-classad optimization and efficient verification of individual constraints, external reference determination facilitates inter-classad optimizations and efficient bulk matching, as detailed below.

1. While it is possible for every classad in the system to be unique, the external reference algorithm provides a basis for defining a notion of classad similarity. For example, the combination of various architectures, operating systems, virtual memory sizes and load averages may result in completely distinct machine classads. However, if all jobs only access the architecture and operating system attributes, all machines with the same architecture and operating system attributes are essentially identical (with respect to jobs). A summarization algorithm based on this observation is described in Section 5.8.

2. Identifying the external reference sets of resources and offers provides a concrete basis for formulating efficient matchmaking algorithms. For example, our framework employs an indexing scheme to efficiently identify compatible classads (see Chapter 4. Since the cost of indexing every classad attribute could be prohibitive, the algorithm only indexes attributes that are in the external reference sets of candidate matches.

2.7 Related Work

Semi-structured data models such as XML [9] and OEM [4] are finding widespread use due to their ability to represent and manage distributed and heterogeneous information. As a result, languages such as Lorel [4], XML-Query [12] and UnQL [7] are being developed to query these semi-structured databases. The classad data model distinguishes itself from these data models by including not only schema and data, but also a query in a single specification.

Features of the classad language distinguish our matchmaking framework from similar systems. Some multi-agent environments distinguish between the *messaging* language, which is used to post and retract advertisements, and the *content* language, which is used to describe services and requests that require matching. Since messaging interactions are abstracted by the matchmaker protocol in our framework, we neither require nor preclude the use of any specific messaging language. KQML [15] (Knowledge Query and Manipulation Language) is a common and popular messaging language, used in many systems including ACL and RETSINA, discussed below.

In general, knowledge-base description of agents is common in systems that facilitate inter-operation between general purpose autonomous agents. ACL [22] (Agent Communication Language) combines the KQML messaging language and the KIF [44] content language to enable inter-operation of programs. As in matchmaking, agents register their interests and capabilities to a facilitator (which behaves like a matchmaker). Unlike many other frameworks, the facilitator in ACL is capable of sophisticated processing. For example, facilitators are capable of forwarding requests to other agents that can handle them, decomposing requests to be handled by multiple agents and then combining results to form the answer, forwarding advertisement information to monitoring agents, and translating information to match agents' vocabularies. Facilitators use automated inference to reason about agent specifications and application-specific facts.

The RETSINA [48] multi-agent infrastructure uses the LARKS [50] language to represent services and requests. Advertisements in the language have input and output variables (whose types must be declared), on which "input" and "output" constraints may be specified. Constraints are expressed as Horn clauses (plans are underway to upgrade constraints to full Prolog programs), and support is provided for inferencing to enable automated reasoning. An ontological description of words used in the advertisement may be defined using the ITL concept definition language [51] and included in the advertisement.

ACL and RETSINA employ powerful languages so that general behavioral specifications of agents may be expressed in advertisements. Since we are not interested in a generalized matchmaking meta-architecture, but rather a specialized matchmaking framework for resource management, we employ the simpler classad language, which appears to be sufficiently powerful. In contrast to the knowledge-base representations used in KIF and LARKS, the classad language uses a database representation. Expression evaluation semantics are simple and lightweight, facilitating efficient and robust implementation.

Globus [17, 11] 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. The task of mapping specifications to actual resources is performed by a resource co-allocator, which is responsible for coordinating the allocation and management of resources at multiple sites. Using RSL, customers may provide very sophisticated resource requirements, but servers have no analogous mechanism.

Chapter 3

The Gangmatching Model

3.1 Goals

3.1.1 The Benchmark Problem: License Management

Our multilateral matchmaking research is primarily motivated by the following real-world license management problem that exposed the inadequacy of a purely bilateral matchmaking framework. The scenario consists of a number of jobs, each of which requires a machine and a license to run the application. Licenses are limited in quantity, and each license is only valid on some subset of machines. Thus the workstation and license resources required by each job are inter-dependent.

This license management scenario cannot be accommodated by a bilateral matchmaking framework. Due to the interplay between limited quantities of licenses and validity of individual licenses on several (but not all) machines, statically partitioning machines or jobs into "license categories" is not feasible. Licenses must therefore be treated as first-class resources with advertisement, matching and claiming phases. Matches in this scenario however now consist of *three* advertisements: job, machine and license.

A primary goal of our research is the formulation of a general multilateral matchmaking framework that can accommodate both the above license management problem and the original bilateral formalism as special cases. The license management problem serves not only to motivate our research efforts, but also as a as a "benchmark problem" that we use to measure the efficacy and efficiency of our proposals. We justify the choice of the benchmark problem as follows.

1. The license management problem has been encountered in practice, and is therefore real.

2. The problem is simple enough to only require a few modeling parameters. Thus a comprehensive study of the problem's parameter space may be conducted.

3. The problem is complex enough to shed light on the basic issues of multilateral matchmaking models and algorithms.

3.1.2 Decentralized Management

A key goal in our framework is facilitating the specification and implementation of policy at the granularity of single principals. By this we mean that we want to provide both servers and customers with a framework that enables representation of idiosyncratic policies effectively, as judged by the expectations of the principals themselves. This goal precludes the imposition of any kind of "global schema" to describe resources and customers, as the legislation of such a schema would require *a priori* knowledge of all resources that may ever participate in the environment. Such knowledge is impossible to obtain and fix in dynamic, heterogeneous and continually evolving environments. Such an approach also divests principals of the means to represent themselves individualistically. We therefore employ an expressive and flexible semi-structured data model to describe principals.

The use of a semi-structured data model with no centralized control, however, exposes two difficulties. First, there is a possibility of information missing from certain descriptions. The solution to this problem is provided by the **undefined** value of the classad language (see Chapter 2). The second difficulty with this stance is the possibility of semantic incompatibility, which can manifest itself in two ways. First, attributes with the same name across different descriptions may represent incompatible concepts. For example, it is possible for a customer to ask for a green fruit and be presented with a curvaceous translucent computer instead. The dual problem occurs when the same concept is mapped to different attributes across different descriptions, such as "Cost" and "Price." In the absence of a pre-defined schema, a common solution to the above problem is to employ a concept definition meta-language, which can then be used to map abstract properties to concrete attributes.

However, we believe that resource management environments evolve much like communities, and as with such communities, informal agreements and conventions are quickly established as the system is used because it is in the interest of both providers and requesters to reach such agreements.

3.1.3 Provider/Requester Symmetry

While all resource management systems allow resource customers (usually jobs) to qualitatively and/or quantitatively describe required resources, none of the systems provide resources with the same degree of control and expressiveness *vis a vis* customers, reflecting implicit assumptions about resource behavior and preferences. We argue from experience that such assumptions cannot be made when resources are distributively owned — we have observed the necessity of expressive distributed policy definition on (possibly) a per-resource basis.

We therefore believe that it is imperative that the resource management model invest customers and resources with the same facilities and mechanisms. It may be necessary for certain *implementations* of the matchmaking model to introduce asymmetries in certain environments, such as introducing priorities that affect customers but not resources. Nevertheless, these asymmetries should be encapsulated in replaceable modules, and should not pervade the overall model.

3.1.4 Single Clearing-House Abstraction

We believe that the system must provide both servers and customers with a single "clearing-house" abstraction. The absence of such an abstraction greatly complicates the specification and implementation of policies, as it would not be possible to easily determine if a principal's preferences have been honored correctly (i.e., if the global best match has truly been identified, or the match is only locally best). A single image abstraction provides an intuitive basis for defining the guarantees made by the matchmaker with respect to a principal's policies. From the resource management perspective, this abstraction also allows the flexibility of defining priorities over the entire system, to enforce pool-wide fairness.

The matchmaking architecture is inherently hybrid, with centralized matchmaking and distributed claiming. This architecture incorporates many of the advantages of both centralized and distributed implementations: centralized matchmaking simplifies administration and control, and defines a clear and intuitive basis for defining matchmaking algorithm semantics, while distributed claiming disperses the responsibility of establishment and management of allocations, resulting in a more scalable solution.

It is important to note that we are not necessarily advocating a centralized *implementation* of the matchmaker, although this approach is the simplest method of achieving a single system image. It is certainly conceivable for a distributed matchmaking implementation to provide a single system image abstraction, along with the implied benefits of parallelization: increased scalability, availability and reliability. Our implementation, however, employs a centralized matchmaker and therefore inherits the advantages of a centralized implementation: simplicity and ease of administration. The performance difficulties associated with this approach are addressed in this body of work.

3.1.5 Support for User and Administrative Policy

Matchmaking systems are distributed policy environments. Whereas conventional resource managers are driven by centralized mechanisms that emphasize system-wide administrative policy, the matchmaking model is motivated by distributed, fine grained per-principal user policy. However, user policy cannot completely supplant administrative policy in a resource management system. Administrative concerns about controlling preemption, pool usage and fairness cannot be expressed as user policy issues. We therefore state the necessity of administrative policy mechanisms as a goal of the matchmaking framework.

Due to the opportunistic, idiosyncratic and dynamic nature of matchmaking environments, administrative policies must also be expressed with similar mechanisms as the advertisements themselves. Administrative policy mechanisms therefore complement user policy mechanisms to define additional constraints and preferences with which customers and servers may be matched.

3.2 Language Representations

3.2.1 Attribute Interpretation and Meaning

The classad language is a partial solution to the substantial language subproblem of matchmaking. However, the classad language by itself is insufficient as a complete matchmaking solution because it does not include any mechanisms that define semantics to attribute names, and therefore does not provide a basis for "attribute interpretation." It is important to note that the problem of attribute interpretation occurs at three distinct levels.

1. Some attributes of the advertisement must be interpreted by the matchmaker so that the advertisement may be meaningfully included in the matchmaking process.

2. Other attributes are interpreted by the administrative policy enforced in a particular resource management environment.

3. Meanings of most other attributes are of interest to match candidates, who express user policies via constraints and preferences under basic assumptions and expectations of what the referred attributes mean.

We are careful to distinguish these three cases because we believe that the solutions to the above problems require different approaches. Since the matchmaker assumes the responsibility of ensuring that the requirements of entities are satisfied, the matchmaker is only interested in identifying the quantitative and operational aspects of the advertisement. This level of interpretation is different from that performed by the administrative policies specified in matchmaking environments, since the latter require certain attributes to be present in advertisements so that the defined policies may be enforced. While the interpretation performed by the matchmaker is a basic infrastructural necessity of all matchmaking environments, the interpretation performed by administrative policy only occurs in particular instances of matchmaking environments. In contrast to both of the above, motivated by their imperative to find an entity that satisfies their constraints and preferences, candidate partners are interested in the descriptive attributes of principals.

To enable unambiguous representation of principals to matchmakers, we define simple representation conventions as part of the *advertising protocol*, which is part of the matchmaking algorithm component of a matchmaking environment. These conventions merely enable the matchmaker to determine basic operational aspects of the advertising entity, such as how many candidates are required by the advertising entity to be satisfied, and where the constraints and preferences of the entity are expressed. Since these conventions only change when the entire matchmaking model is revised, this “fixing” of the semantics of some attributes does not hamper the agility of the framework in dealing with dynamic and heterogeneous principals — all principals that participate in the matchmaking environment must adhere to these simple conventions.

In contrast, we do not define explicit mechanisms to define the semantics of arbitrary attributes of principals, whether they are required by the administrative policy or by per-advertisement user policies. To accommodate the flexibility and expressivity requirements of our framework, any such semantic definition mechanism must be able to accommodate principals that use arbitrary attributes to represent themselves individualistically. In addition, semantic definition mechanisms must be able to adapt to the heterogeneity and dynamism evident in distributively owned matchmaking environments. We believe that rather than following a centrally legislated attribute semantics framework, communities of service providers and customers will instead naturally converge to common conventions regarding attribute use and semantics since it is in their best interest to do so. Furthermore, common guidelines regarding administrative policies of matchmaking environments will be made available to users who may configure their agents to include the required attributes in advertisements when participating in that environment.

It is nevertheless important to note that our framework does not preclude the use of formal semantic definition mechanisms. Concept definition and management languages usually involve the definition of abstract concepts that are then translated into a set of ground terms whose meanings are fixed and known. Thus, the attributes included in advertisements may be thought to be ground terms whose semantics are fixed by some mechanism external to the matchmaking framework.

3.2.2 Ports and Docking

Multilateral matchmaking is an important goal of our model. Single advertisements must therefore be able to represent the requirement of multiple inter-dependent matches. There are therefore two fundamental concepts that must be represented in the advertisement.

- Advertisements must incorporate the notion of multiple "interfaces" to enable simultaneous matching with several heterogeneous entities.
- Advertisements must also be able to represent dynamic dependencies between interfaces. By this we mean that it must be possible to qualitatively alter some properties of interfaces based on match results of other interfaces.

In order to realize the above goals, we have adopted a docking paradigm to matchmaking. Each advertisement defines an ordered list of labeled ports, each of which denotes a request for a "submatch." A multilateral match occurs by docking the individual ports of distinct advertisements, thus forming tree-shaped "gangs" of linked classads, as illustrated in Figure 7. Ports may be thought of as docking interfaces defined by the hosting advertisement. As such, in addition to the label associated with the port, each port provides a namespace that may be used to define attributes to express characteristics of the advertising entity to candidates expected to dock at that port. Each port also includes constraint and preference expressions (named `Constraint` and `Rank`

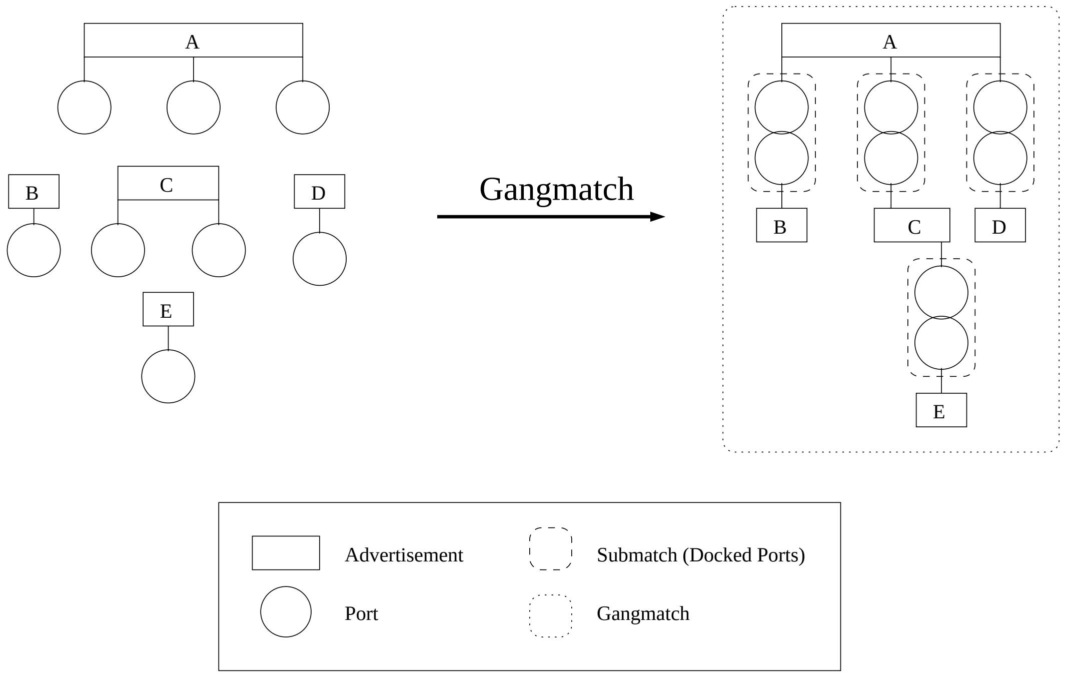

Figure 7: The Gangmatching Operation

respectively) which are used to select the candidate to dock at that port. Since gangmatching cannot proceed without these essential items of information, the port list, port label, port constraints and preferences are part of the advertising protocol of the gangmatching model. The detailed semantics of port constraints and preferences will be shortly discussed.

One may note that the classads shown in previous figures represent policies in a bilateral matchmaking framework. Bilateral matchmaking is a special case of gangmatching, where every advertisement possesses exactly one port. To simplify the notation of this special case, advertisements without an explicit port list are assumed to have a single implicit port labeled “other.”

The rules and semantics of port labels are the key to providing much of the advanced functionality of the gangmatching model. Port labels are required to be *locally unique*; i.e., the port labels of any given advertisement must be distinct, so that in the context of any advertisement, a label is unambiguously associated with at most one port. The port label serves as the name that identifies the corresponding port of the candidate docked at that port. Thus port labels are the fundamental mechanism by which constraints and preferences are expressed on candidates.

An important aspect of port labels is that their scope of validity is not confined to the port of declaration. Instead, the scope of a port label extends from the port of declaration to the end of the port list, so expressions in later ports may refer to labels of ports earlier in the list, but not *vice versa.* This definition of label scopes allows information to be conveyed from one match locality to another. Specifically, constraints and preferences on candidates docked at ports later in the port list may be defined using information from candidates docked to ports earlier in the port list. These ideas are discussed further below.

3.2.3 User Policy Specification

Constraints and preferences are the primary mechanism via which agents specify their user policies. The port constraint defines properties that must be satisfied by candidates that dock at that port, thus allowing principals to define idiosyncratic notions of compatibility on a per-port and per-advertisement basis. The extended scopes of labels allow inter-port dependencies to be expressed since constraints of some ports may refer to attributes of entities docked in previous ports.

The ports of two advertisements may successfully dock if the constraint expressions defined in the ports both evaluate to **true**. If either constraint evaluates to **false** or non-boolean values such as **undefined** or **error**, the ports are deemed to be incompatible and may not be docked.

Each port may also include a preference expression which is used to select among several compatible advertisements. The rank expression is regarded as a user defined "goodness metric," where greater values denote more desirable candidates. The "greater than" comparison operator of the classad language is used to compare rank values, so rank expressions may evaluate to any value types that are comparable in the language.

Care must be taken in defining the semantics of port rank expressions. Although gangmatching is naturally combinatoric in nature, we must not define model semantics that would force gangmatching algorithms to consider all possible combinations. The latter situation would be necessary if the semantics of rank expressions were defined to produce the best *combination* of submatches by maximizing a function that incorporates the rank expressions of all ports. Since functions are not necessarily maximized by maximizing their components independently, it would be necessary to examine a large number of advertisement combinations to identify the most preferred bundle. We therefore define preference semantics so that the rank expressions of ports are considered independently in a left to right order, with the best candidate being selected on a per-port basis.

These ideas are illustrated in Figure 8, which shows a single advertisement that requires two machines in a single match operation, which are both required to be on the same subnet. The port labels “Host1” and “Host2” are used to define the constraints on the two ports. Note that the constraint expression in the latter port refers to both port labels in defining the subnet co-location constraint. The `Ports` attribute (a list of two classads) and the `Label`, `Constraint` and `Rank` expressions in each port are required by the advertising protocol of the matchmaker.

3.2.4 Administrative Policy Specification

User policies expressed as constraint and rank expressions in advertisements define local criteria to determine match validity and desirability. Administrative policies complement user policies by reconciling the competing forces of customer priorities, request preferences and offer preferences. Administrative policies also control the state of the system by disallowing matches that may, for example, overload the system.

Administrative policy is specified through three mechanisms.

1. **Root identification.** There is no intrinsic information in a classad to differentiate a request from an offer. In the abstract, it is possible to

```
[
   Owner    = "raman";
   Address  = "<froth.cs.wisc.edu:2020>";
   Priority = 23.7;
   Ports    =
     {
        [
           Label      = Host1;
           Executable = "a.out";
           ImageSize  = 25.7M;
           MemoryReqs = 17.2M;
           Constraint = Host1.Arch == "INTEL" && Host1.OpSys == "LINUX" &&
                            Host1.VirtualMemory > ImageSize;
           Rank       = Host1.MIPS
        ],
        [
           Label      = Host2;
           Executable = "b.out";
           ImageSize  = 22.7M;
           MemoryReqs = 11.2M;
           Constraint = Host2.Arch == "INTEL" && Host2.OpSys == "LINUX" &&
                            Host2.VirtualMemory > ImageSize &&
                            Host1.Subnet == Host2.Subnet
           Rank       = Host2.KFlops
        ]
     }
]
```

Figure 8: Using port labels to define dependent matches

perform the matchmaking process either by choosing offers and finding compatible requests, or by choosing requests and finding compatible offers. Choosing one set of advertisements to "seed" the matchmaking process emphasizes the satisfaction of these advertisements in comparison to the remaining set. We call this process *root identification*, since it identifies the advertisements that will serve as the roots of classad gangs.

In Condor, root advertisements are usually resource requests issued by submitted jobs, reflecting the resource management system's task of completing submitted jobs. We henceforth assume that our roots are exactly the requests submitted to the system.

2. **Root Ordering.** While the root identification process identifies a set of advertisements to seed the matchmaking process, the root ordering process sequences the roots so that they are matched in decreasing order of preference. Root ordering may be used to implement common resource management formalisms like priority. The root ordering scheme is specified by an expression that is evaluated in the context of every root to yield a corresponding value. The roots are then ordered by the values obtained from evaluating the sort expression.

3. **Docking Vector.** The final administrative policy mechanism is the docking vector, which provides a basis for selecting a single submatch for a port when the port is compatible with several candidates. The docking vector is a vector of expressions that are evaluated in the context of the port being matched to yield a vector of values. Value vectors are obtained for all candidates compatible with the port in question, and vectors with **undefined** or **error** components are discarded. The candidate chosen for the port in question is the one that has the "largest" value vector when the vectors are ordered lexicographically.

We now provide an example of how these administrative policy mechanisms may be used to implement the matchmaking policy of the Condor system. Job advertisements requesting workstation resources serve as the root advertisements in Condor. The root ordering criterion is the priority of the job's submitter. Thus, the root identification and root ordering mechanisms essentially produce a list of job advertisements sorted by priority, which are then considered in order by the matchmaking algorithm.

Given a list of mutually compatible candidates for a given root, Condor's candidate selection policy is as follows. First, if the job's preferences indicates a uniquely preferred candidate, then that candidate is chosen as the match. If there are several equally ranked candidates, Condor attempts to match the job with a candidate that is currently unused by any other customer. If no such resources are available, the system must preempt some workstation.

An attempt is first made to identify a machine that strictly prefers the new job to the customer currently being served; i.e., preemption for machine rank. If rank preemption is not possible, preemption for priority is considered. Priority preemption is possible if the new customer's priority is better than the customer currently being served *and* the machine prefers or is at least indifferent between the two customers. In addition to these criteria, rank and priority preemption can only occur if an administrator specified preemption constraint holds. If the preemption constraint does not hold an alternate resource must be considered. By default, this constraint is satisfied only if the customer being served has been serviced for at least three hours, thus preventing "preemption thrashing."

```
{
  self.Rank                               // job's rank
,
  label.RemoteUser is undefined  ? 2 :    // no preemption
  label.CurrentRank < label.Rank ? 1 :    // rank preemption
  label.RemoteUserPrio > UserPrio*1.1 &&  // priority preemption
    label.CurrentRank <= label.Rank &&    //   (machine must like job)
    label.ServiceTime > '3:00'   ? 0 :    //   (preemption constraint)
      error                               // otherwise fail
,
  label.RemoteUser is undefined ? 0 :     // preemption?
  label.RemoteUserPrio*10000-label.ImageSize //   (preemption rank)
}
```

Figure 9: Condor's policy expressed as a docking vector

If, at this point, there are several equally feasible workstations, a candidate is chosen on the basis of a rank expression specified by the administrator. The current default is only defined during preemption, when the machine serving the customer with the worst priority and smallest image size is chosen.

Condor's administrative policy is expressed as a docking vector in Figure 9. Note that the attribute `label` is used as a generic label to access the namespace of the docked candidate. This technique works because the expression `label` itself evaluates to the label specified for the port.

The docking vector formulation of Condor's policy is far easier to understand. Furthermore the mechanism allows the administrative policy to be modified or extended. For example, one may append the expression `-label.Memory` to the docking vector, which would have the effect of giving out the machine with the least amount of memory to the job, thereby conserving machines with larger memories to jobs that actively require or prefer them. This functionality is a significant extension to Condor's current capabilities.

3.2.5 Summary of Representation and Discussion

We now summarize the main aspects of the proposed gangmatching advertisement representation and proceed to discuss the representation's implications and functionality.

Summary of Representation

Advertisements in the gangmatching model are composed of distinctly labeled ports, each of which includes a constraint and a rank expression. Ports may be regarded as interfaces through which advertisements are matched by means of a docking operation. Port labels are used to refer to the attributes of candidates that dock at the labelled port, and since the scope of port labels extends from the port of declaration to the end of the port list, expressions in later ports may refer to attributes of candidates docked at previous ports. The list of ports, each of which contains a label, constraint and preference, is required by the matchmaker for the advertisement to be meaningfully included in the matchmaking process. Thus, these structures of the advertisement are fixed by the matchmaker's advertisement protocol.

The root identification and root ordering administrative policy components are invoked to identify and sequence a subset of advertisements that will serve as the seeds of classad gangs. Root ordering enforces a priority scheme to ensure that gangs for some roots are marshaled before others. When marshaling a gang, the matchmaker ensures that the constraints specified in the respective ports of candidates are satisfied. The docking vector is then used to produce value vectors for all compatible candidates. The best candidate is identified by lexicographically ordering value vectors that do not have **undefined** or **error** components, and choosing the candidate with the "largest" vector.

Functionality Implications of Representation

The proposed representation of advertisements has several interesting functionality implications. The use of the classad language to formulate constraints and preferences provides an extremely flexible and general basis for matchmaking. However, due to the fixed number of ports in advertisements, the proposed gangmatching model can only marshal gangs that have fixed "branch out" factors at each gang node. While it is certainly possible to envision scenarios in which this feature is a limitation, we have not encountered real-world situations that require this functionality. Another implication of the model is the implicit conjunction of port requests — the candidate is satisfied if and only if *all* the ports of the advertisement are successfully docked, resulting in an implicit AND model of resource allocation. Thus, there is no mechanism to specify OR models or arbitrary AND-OR resource allocation models.

The "declare before use" semantics of scope labels introduces a left-to-right bias to the model, and results in many interesting consequences. First, although the model does not explicitly require a gangmatching algorithm that fills ports in left to right order, the use of such an algorithm is naturally suggested. Second, the label scope semantics ensures that there are no circular dependencies between ports. A consequence of this feature is that inconsistent gangs may be detected before the entire gang of classads is marshaled, greatly increasing the efficiency and feasibility of gangmatching algorithms.

The symmetry of the gangmatching model implies that both requests and offers can marshal gangs of advertisements. Given a designated root advertisement, the topology of the gang is a rooted tree. Thus the model not only supports aggregation for roots but for non-root advertisements as well, providing a hierarchical aggregation functionality.

The multi-ported advertisements in the gangmatching model are in some ways similar to Horn clauses in Datalog-like languages. However, while Horn clauses have a uniquely designated term that serves as the head of the clause, advertisements in the gangmatching model do not have a specific port that is designated to serve as a "parent link" in a gang. Since root advertisements do not have a parent in the gang tree, they do not have parent links. Thus, the omission of a designated parent link preserves the symmetry between root and non-root advertisements.

Since the parent link of a non-root advertisement may exist anywhere in its port list, the gangmatching model facilitates the representation of interesting abstract services. Figure 10 shows an advertisement for a graphics rendering service. The advertisement consists of three ports, of which the first behaves as the parent link. The next two ports request a workstation and a license for the rendering application respectively. The advertisement therefore behaves as an intermediary that converts requests for an abstract rendering service to concrete requests for resources that are required to perform the render.

```
[
  Ports =
    {
       [
          Label      = request;
          Type       = "render_server";
          Constraint = request.Type=="render_client" &&
                          request.Owner!="rival";
          Rank       = 0
       ],
       [
          Label      = cpu;
          ImageSize  = 27.2M;
          MemoryReqs = 15M;
          Executable = "do_render";
          StdIn      = request.sceneFile;
          StdOut     = request.outputFile;
          Constraint = cpu.Arch=="INTEL" && cpu.OpSys=="LINUX" &&
                cpu.VirtualMemory>ImageSize;
          Rank       = cpu.Memory
       ],
       [
          Label      = license;
          HostName   = cpu.Name;
          Constraint = license.App=="do_render"
          Rank       = 0
       ]
   }
]
```

Figure 10: Advertisement for a rendering service

Figure 11 illustrates an intriguing functionality of the gangmatching model. The advertisement marshals two co-located compute nodes and then re-advertises a multi-processor computing service that is the aggregate of the two marshaled nodes.

3.3 The Matchmaking Framework Architecture

The matchmaking framework is composed of three kinds of principals: agents that provide services, agents that request services and entities that constitute the matchmaking service itself.

3.3.1 Provider and Requester Architecture

The most numerous kind of principals in matchmaking environments are service providers and requesters. In discussing these agents, it is important to note that the provider and requester characterizations are made only with respect to particular interactions — it is possible for a single agent to be a provider with respect to one service and a requester with respect to another. However, for clarity of discussion we assume agents to be dedicated providers or requesters.

In addition to the actual actions involved in producing or consuming a service, the matchmaking related activities of these agents are advertising, receiving notifications and claiming. The architecture of providers and requesters is simple: in addition to the modules required to provide, request and activate

```
[
  Ports =
    {
       [
          Label         = cpu1;
          Type          = "cpu_request";
          Constraint    = cpu1.Type == "ComputeNode" &&
                            cpu1.IsDedicated;
          Rank          = 0
       ],
       [
          Label         = cpu2;
          Type          = "cpu_request";
          Constraint    = cpu2.Type == "ComputeNode" &&
                            cpu2.IsDedicated         &&
                            cpu1.Host == cpu2.Host
          Rank          = 0
       ],
       [
          Label         = request;
          Type          = "multi_processor";
          HostName      = cpu1.Name;
          Arch          = cpu1.Arch;
          OpSys         = cpu1.OpSys;
          NumCpus       = cpu1.NumCpus + cpu2.NumCpus;
          Memory        = cpu1.Memory  + cpu2.Memory;
          Disk          = cpu1.Disk    + cpu2.Disk;
          VirtualMemory = cpu1.VirtualMemory + cpu2.VirtualMemory;
          Constraint    = request.MemoryReqs < Memory - 15M;
          Rank          = 0
       ]
   }
]
```

Figure 11: Advertising two co-located compute nodes as an multi-processor

the specific services of concern, each agent includes an *advertisement creation module*, a *notification reception module* and a *claim maintenance module.* The functionalities of these modules will be discussed in detail shortly.

3.3.2 Matchmaker Architecture

Although we have previously referred to the matchmaker as a single entity, the matchmaker may be conceptually decomposed into a number of components These components are fairly independent, and may therefore be implemented as separate threads or processes.

Offer Collector The Offer Collector component is responsible for collecting advertisements issued by agents that provide services. The Offer Collector implements the advertising protocol of the matchmaker.

Request Collector The dual component of the Offer Collector, the Request Collector collects advertisements issued by agents that request services. The Request Collector is also cognizant of the matchmaker's advertising protocol, and ensures that the advertisements sent to it are valid.

Matchmaking Engine The Matchmaking Engine encapsulates the matchmaking algorithm and is responsible for creating matches. The Matchmaking Engine is periodically activated to perform a "matchmaking cycle," which consists of obtaining the offer and request advertisements from the respective collectors and then proceeding to create gangs in accordance to the user and administrative policies specified. As matches are created, the Matchmaking Engine sends the matched gangs to the Notification Engine for further processing.

Notification Engine The Notification Engine is responsible for posting match notifications to all entities involved in the match. The responsibility for notifying matched entities is off-loaded from the main Matchmaking Engine because posting notifications can be a relatively slow process due to the required number of network connections and associated round-trips. In addition to notifying all the gang members in a match, the Notification Engine also posts a notification to a Match Registrar containing the contents of the entire gang.

Match Registrar The Match Registrar maintains a record of all the matches in a matchmaking environment. The information from the registrar may be used maintain accounts about current and historical resource usage for users. The registrar may also be consulted to quickly and accurately gauge the state of the system.

The above components and their interactions are illustrated in Figure 12.

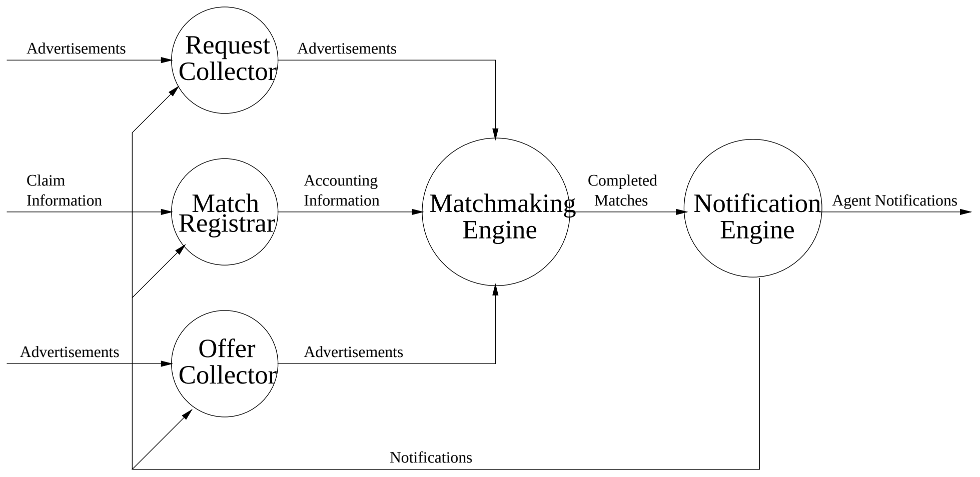

Figure 12: Architecture of the Matchmaker

3.4 Advertising

3.4.1 Advertisement Creation Mechanism

The information required to create complete advertisements is distributed in various locations. Advertisement creation is therefore a process of collating information with which advertisements are augmented or rewritten. We envision advertisement creation to be performed by a "pipeline" of *advertisement writer* objects, each of which each obtains information from some source and augments or rewrites advertisements passing through it. It is important to note that not all stages of the pipeline exist within the provider or requester agent. Some attributes of the agent, such as priority information, are inserted by accounting processes that reside within the Match Registrar.

For example, consider the workstation advertisement example in Figure 13

```
[
  Name                  = "cobra.cs.wisc.edu";
  Machine               = "cobra.cs.wisc.edu";
  IsInstructional       = FALSE;
  RebootedDaily         = FALSE;
  OffHoursOnly          = FALSE;
  CkptServer            = "condor-ckpt.cs.wisc.edu";
  IsDedicated           = FALSE;
  IsComputeCluster      = FALSE;
  VirtualMachineID      = 1;
  VirtualMemory         = 746304;
  Disk                  = 384802;
  CondorLoadAvg         = 0.997985;
  LoadAvg               = 1.000000;
  KeyboardIdle          = 70526;
  ConsoleIdle           = 70530;
  Memory                = 251;
  Cpus                  = 1;
  Arch                  = "INTEL";
  OpSys                 = "LINUX";
  UidDomain             = "cs.wisc.edu";
  FileSystemDomain      = "cs.wisc.edu";
  Subnet                = "128.105.166";
  TotalVirtualMemory    = 746304;
  TotalDisk             = 384802;
  KFlops                = 63685;
  Mips                  = 550;
  LastBenchmark         = 971987587;
  TotalLoadAvg          = 1.000000;
  TotalCondorLoadAvg    = 0.997985;
  TotalVirtualMachines  = 1;
  State                 = "Unclaimed";
  EnteredCurrentState   = 972068166;
  Activity              = "Idle";
  EnteredCurrentActivity = 972206722;
  Rank                  = 0.000000;
  CurrentRank           = 0.000000;
  Constraint            = (((LoadAvg - CondorLoadAvg) <= 0.300000) &&
                              KeyboardIdle > 15 * 60) &&
                              (other.ImageSize <= ((Memory - 15) * 1024));
  LastHeardFrom         = 972223116;
]
```

Figure 13: Condor Workstation advertisement

which describes a real advertisement of a workstation in the University of Wisconsin-Madison pool. The advertisement consists of several descriptive attributes such as load average, keyboard idle time and available virtual memory which are obtained by probing the physical state of the workstation. Most of the attributes used to describe the machine including the main constraint and rank expressions which express the owner's policy, are obtained from configuration files. Other aspects of the advertisement, such as the `IsInstructional` attribute which describes whether the machine is an instructional machine that is physically located in a common laboratory, are similarly obtained from static and dynamic configuration mechanisms. Finally, the advertisement is augmented with accounting information from Condor's equivalent of the Match Registrar, including information such as the priority of the customer currently using the machine (if any). The corresponding pipeline of advertisement writer objects is illustrated in Figure 14.

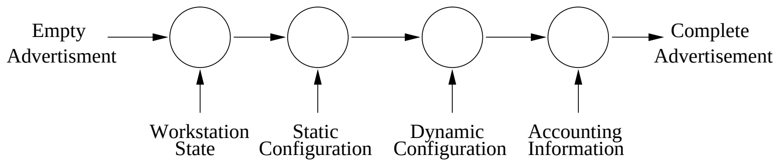

Figure 14: Machine advertisement creation pipeline

The process of request creation in Condor is equally involved. Much of the information required to create a request advertisement, such as executable name, arguments, environment variables, constraints and preferences, may be specified in description files. When information is not provided in the description file, reasonable defaults are created from information obtained directly from the executable itself, or the submission environment. For example, if the image size of the executable is not specified, the size of the executable is used to estimate an image size. Similarly, if constraints and preferences are not specified, defaults are created to obtain a workstation that approximates the submission environment. The corresponding pipeline of advertisement writer objects for Condor requests is illustrated in Figure 15.

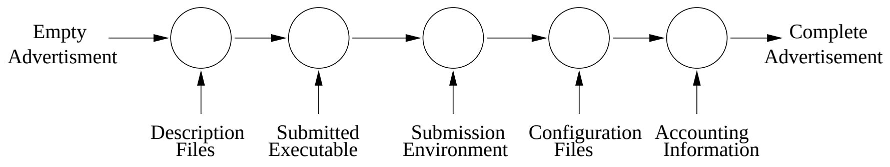

Figure 15: Job advertisement creation pipeline

3.4.2 Advertisement Identity and Lifetime

Advertising, like many other activities in matchmaking environments, is periodic in nature. Thus entities repeatedly send advertisements with updated attributes to the matchmaker so that matches may be created with current information. Periodic advertising provides many of the important functionality advantages of matchmaking environments: fault tolerance and natural support for both pool evolution and dynamic agent attributes. These functionalities are facilitated by "timing out" stale advertisements, introducing new advertisements and updating old advertisements respectively, all of which involve advertisement identity and lifetime issues.

Every advertisement posted by an agent is created with a globally unique identifier, which may be created by, for example, concatenating the network address of the agent with the timestamp of the advertisement's creation. The advertisement identifier is used as a handle when attributes in the advertisement need to be updated, thus preserving the identity of the updated advertisement. Every advertisement has a specific lifetime associated with it. These lifetimes may only be extended by updating the advertisement. If no updates are received from an agent for any interval beyond the advertisement's lifetime, the agent is assumed to have crashed or withdrawn from the resource management environment, resulting in the destruction of the agent's advertisement.

Advertisements are also consumed by the matchmaking process. Agents that have been matched cannot continue to update their already matched advertisements — these updates will be discarded by the appropriate collector. Instead, matched agents must create advertisements with fresh identities whenever they receive match notifications from the matchmaker.

3.5 Match Creation

The main complication of match creation is the efficient identification of matches in the gangmatching model. The inherently combinatoric aspects of the model require interesting indexing techniques and search strategies to enable a feasibly efficient implementation. A detailed study of this topic is provided in Chapter 5.

After a gang has been identified by the matchmaking algorithm, the advertisements that comprise the gang are removed from the respective advertisement pools, thus ensuring that matched advertisements are not reused in subsequent gangs. The remaining steps in match creation are trivial given that a mutually compatible gang has been marshaled. Unique identifiers are created and issued to all the various submatches which comprise the gang, and to the gang as a whole. These identifiers may be subsequently used as handles to obtain information about the match. The gang is then passed on to the Notification Engine.

3.6 Notification

Notification is conceptually the simplest of the matchmaking activities since it only involves sending copies of the matched gang to every agent whose advertisement is involved in the gang. Notification of the gangmatch is also sent to the Match Registrar (to track the status of the pool) and to the appropriate Collectors (to discard the matched advertisements).

Gang notification is by nature a latency-intensive operation because it requires several socket connection operations which are each followed by the transmission of relatively small messages. Fortunately, gang notification is a "pleasantly parallel" operation and can therefore be decomposed into independent tasks that may be performed concurrently by several worker tasks. By creating a suitably large number of worker tasks, the throughput of gang notification may be balanced with the match creation throughput of the matchmaking algorithm.

3.7 Claiming

The presence of claiming as an additional step in resource allocation distinguishes matchmaking from conventional resource management. The semantics of a match is defined to provide matched entities with an opportunity window to complete the claiming process. When notified of a match by the matchmaker, notified agents enter into a "matched state" for a fixed duration during which time the claiming protocol is activated. If the claiming protocol is not completed within the opportunity window, the agents abort the process and re-advertise their availability.

An important feature of the claiming interaction is the possibility of veto. Due to the autonomy invested in providers and customers, agents may choose to not proceed with the claim and reject the match completely. Thus the locus of control resides with the agents participating in the matchmaking framework and not the matchmaker. Since any one of the several agents involved in a gangmatch may reject the match, the claiming protocol for an entire gang of agents must be sufficiently robust to ensure that all agents in the gang are left unclaimed if the gangmatch is rejected.

To avoid the duplication of functionality and complexity across different claiming protocols, we define a composite claiming interaction composed of a *generic claiming protocol* and a *specialized claiming protocol*. The generic claiming protocol encapsulates the complexity of both marshaling and maintaining a gang of distributed agents, and is common to all agents. The specialized claiming protocol performs the specific actions required to claim the service of interest, and therefore differs across agents representing different services.

Given the implicit AND request model of advertisements, one might be tempted to assume that the entire gangmatch must always be rejected if any agent in the gang vetoes the match or leaves the match after the gang is claimed. However, we believe that this logic should not be legislated by the gangmatching model. Advertisements, such as those illustrated in Figure 10 and 11, may employ the parent link and hierarchical marshaling features of the gangmatching model to provide services by aggregating other services. In this paradigm it may be possible for an entity to provide the service offered even if one of the aggregated services is not available. For example, an advertisement may have marshaled two workstations to provide a particular service, with one workstation being unessential after an initialization phase. The agent can therefore continue to provide the promised service with a reduced subgang. Thus, while most agents may not exercise the option to continue service with a reduced subgang, agents must be allowed to make this decision rather than legislating one via the model.

Due to the presence of a gang root, the generic claiming protocol may be formulated as a recursive two-phase protocol consisting of VERIFY and COMMIT phases, much like the two-phase transaction commit protocol [24]. After being notified, the root of the gang begins the VERIFY phase by asking each of its subordinates to verify if they want to join the gang. If these subordinates do not have subordinates of their own, they immediately reply with OK or ABORT messages indicating their position on joining the gang. If the root receives OK messages from all subordinates, the COMMIT phase of the protocol is activated, when the root agent sends COMMIT messages to all subordinates, completing the claiming process successfully. If any of the subordinates sends an ABORT message in the VERIFY phase and the root consequently decides to abort the claim, ABORT messages are sent to all subordinates, thus terminating the claiming protocol. Of course, if the root's subordinates have subordinates of their own, the VERIFY and COMMIT phases of the protocol are performed recursively, with the verdict of each phase being returned to the root as the agent's decision. The specialized claiming protocol may be activated after the gang has been claimed by the generic protocol. By definition, other details of the specialized protocol are not specified by the model.

Although the two-phase claiming protocol is similar to its database counterpart, the match claiming protocol is far simpler due to the relatively lax semantics of matches and claims. Since matches are only valid for a fixed time interval prior to claiming and since gang members may legitimately withdraw from the gang at any time, the complex failure modes associated with the two-phased transaction commit protocol are absent.

After the gang has been claimed, the generic claiming protocol also maintains the match by periodically sending "heart beat" messages to all immediate neighbors in the gang tree. If expected heart beat messages are not received by an agent for a suitable time interval, the agent assumes that the corresponding gang member has crashed or retired from the gang. Depending on the semantics of the situation, the agent then itself retires from the gang, or continues to provide the advertised service as necessary. Heart beat messages are also sent by every gang member to the Match Registrar, so that the registrar's view of active matches is maintained accurately.

3.8 The Condor Matchmaking Scheme

Condor [35, 36] is an high throughput computing (HTC) environment that can manage very large heterogeneous collections of distributively owned resources. The architecture of the system is structured to provide sophisticated resource management services at the resource, customer and application levels to both sequential and parallel applications [42]. This section briefly describes aspects of the Condor system that are relevant to the problem of matchmaking.

Resources in the Condor system (computers capable of running Condor jobs) are represented by *Worker Agents* (WA),[1] which are responsible for enforcing

[1]The current Condor system uses slightly different terminology, for historical reasons.

the policies stipulated by resource owners. An WA periodically probes the resource to determine its current state, and encapsulates this information in a classad along with the owner's usage policy. Customers of Condor are represented by *Customer Agents* (CAs), which maintain per-customer queues of submitted jobs (represented as classads). Matching is coordinated by a *central manager*, which consists of three entities, a *collector*, an *accountant*, and a *negotiator*. In relation to the described architecture, the collector process is equivalent to the Offer Collector, the accountant is equivalent to the Match Registrar, and the negotiator acts as the Request Collector, Matchmaking Engine and Notification Engine.

Each CA periodically sends the collector *submitter* classads describing users who have submitted jobs. The WAs also send the collector *worker* ads describing their state. The collector only stores the most recent ad for each worker and each submitter. These ads conform to an advertising protocol that states that every classad should include expressions named `Constraint`[2] and `Rank`, which denote the requirements and preferences of the advertising entity. The protocol also requires the advertising parties to include "contact addresses" with their ads, and allows an WA to include an "authorization ticket" in each worker ad.

Periodically, the negotiator enters a *negotiation cycle*. It retrieves from the collector the current ad for each worker and submitter. It asks the accountant to

What we call a worker agent here is called a start daemon (startd) in Condor, and what we call a customer agent is called a scheduler daemon (schedd).

[2]Constraint expressions are currently named `Requirements` expressions by the Condor system

prioritize submitters based on their past usage. In then cycles through the submitters in priority order and contacts CAs, requesting from them *job* classads. The negotiator matches each job ad with a compatible worker ad. Since the notion of "compatible" is completely determined by `Constraint` expressions, classads may be matched in a general manner. In addition, `Rank` expressions are used as goodness metrics to identify the more desirable among the compatible matches. The algorithm used by Condor's for preemption and for resolving job and machine preferences was described in Section 3.2.4.

When the negotiator determines that two classads match, it removes the worker ad from its set of available workers and invokes the matchmaking protocol to contact the matched principals at the contact addresses specified in their classads and send them each other's classads. The manager also gives the CA the authorization ticket supplied by the WA.

The CA then follows the claiming protocol by contacting the WA and sending the authorization ticket. The WA accepts the resource request only if the ticket matches the one that it gave the collector, and the request matches the WA's constraints with respect to the updated state of the request and resource, which may have changed since the last advertisement. If the request is accepted, the workstation runs the customer's job and informs the accountant of the resources used. When the CA finishes using the resource, it relinquishes the claim, and the WA advertises itself as unclaimed by sending a new ad to the collector. The WA may also send an ad when it starts running the job, indicating that although the workstation is currently busy, it is still interested in hearing from higher priority customers. The specification of what constitutes “higher priority” is completely under the control of the WA.

The central manager is highly fault tolerant. A very simple *Condor master* process ensures that a collector process and a negotiator process are always running. If the collector should die, a new one started by the master will quickly learn of the states of all the CAs and WAs from their periodic update messages. The negotiator regenerates its lists of ads from the collector at the start of each negotiation cycle. Only the accountant maintains persistent state.[3] Established associations between CAs and WAs are not affected by a malfunction of the central manager.

3.9 Related Work

Although details of current distributed resource management systems vary dramatically, there are aspects that they share. Instead of providing a survey of a large number of systems, we briefly discuss the basic matching mechanisms of some resource management environments to highlight the differences between conventional resource allocation and matchmaking.

Systems such as NQE [43], PBS [27], LSF [54] and Load-Leveler [10] process jobs by finding resources that have been identified either explicitly through a job

[3]The accountant is currently a library of functions invoked by the negotiator and collector, which store usage information in a file on disk.

control language, or implicitly by submitting the job to a particular queue that is associated with a set of resources. Customers of the system have to identify a specific queue to submit to *a priori*, which then fixes the set of resources that may be used, and hinders dynamic qualitative resource discovery. Furthermore, system administrators have to anticipate the services that will be requested by customers and set up queues to provide these services. Over time, the system may accumulate a large number of queues whose service semantics differ to various extents, complicating the process of finding the appropriate queue for a job.

Legion [25] takes an object-oriented approach to resource management, formulating the matching problem as an *object placement problem* [31]. The identification of a candidate resource is performed by an object mapper, whose recommendation is then implemented by a different object. The Legion system defines a notation [31] that is similar to classads, although it uses an object-oriented type system with inheritance to define resources [34], as contrasted with the simple attribute-oriented Boolean logic of classads. Legion supports autonomy with a jurisdiction magistrate (JM), which may reject requests if the offered requests do not match the policy of the site being managed by the JM. While the JM gives a resource veto power, there is no way for a resource to describe those requests that it would rather serve.

Distributed computing environments such as Seti@Home [2] and Distributed.net [1] exemplify the power of federated computing. However, these systems do not provide general and flexible mechanisms to specify resource usage and access policy, running tasks in "screen saver" priority instead. However, this policy may neither be necessary nor sufficient to many resource owners. Furthermore, the infrastructure to match customers to resources is also rudimentary when compared to the matchmaking system.

The JINI system [53] being developed by Sun Microsystems has similar notions as the classad based Condor system: resources advertise their presence, customers discover their presence through a lookup service and claim them for computation. The JINI architecture is closely coupled to the Java platform, and the lookup service used by customers essentially locates object instances that implement the interface specified by the customer. Constraint based queries may also be specified by the customer, but the query language is significantly less rich than the classad language. Furthermore, JINI does not provide a symmetric interface to providers and customers.

UDDI [52] and eSpeak [28] are two specifications being defined to enable automation of business-to-business interactions. Both systems use XML as a specification language to describe services, and define a rich framework for service discovery. Like most other systems, neither UDDI nor eSpeak exports a symmetric interface to servers and customers. Furthermore, since the emphasis in these frameworks is on service discovery and not resource allocation, the matchmaker provides a list of candidate servers to the customer, who then chooses one or more servers based on subjective criteria.

Chapter 4

ClassAd Indexing

4.1 Intuition

Consider a simple scenario in which all classads have a single implicit port named "other," and that offer classads have a single numeric attribute named y, and all request classads have a single numeric attribute named x. Furthermore, assume each classad constrains the numeric attribute of its candidate match with a single inequality. Examples of classads that follow the above assumptions are illustrated in Figure 16. The fundamental intuition we wish to communicate

Offers	Requests
`[ x=10; Constraint=other.y>=7 ]`	`[ y=8; Constraint=other.x>=8 ]`
`[ x=7; Constraint=other.y<=5 ]`	`[ y=3; Constraint=other.x<=15 ]`

Figure 16: Simple classads for indexing

is that these classads may be represented as (degenerate) rectangles in the x-y plane, and that intersecting offer and request classad rectangles signify candidate matches. For example, the first offer may be represented by the point set $\{(x, y)|x = 10, y >= 7\}$, which is a ray in the x-y plane extending from the point $(10, 7)$ vertically, a very thin long rectangle. Similarly, the first request is

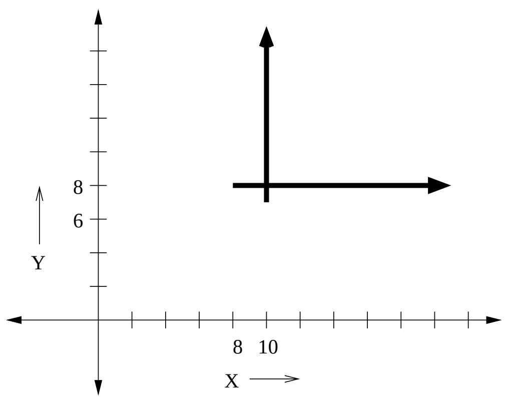

Figure 17: Classads as rectangles in the x-y plane

a horizontal ray. Since the two sets intersect, the classads match (Figure 17). The availability of sophisticated data structures for rectangle management and querying enables us to efficiently identify compatible classads using this spatial approach to matchmaking. The spatial data structures may be thought of as indexes on both the descriptive attributes *and* constraints of classads.

Despite the obvious intuition obtained from the above example, the actual indexing scheme is considerably more complicated due to the semi-structured nature of the classad data model. Difficulties include the presence of common attribute names in both offers and requests, the possibly large number of attributes in classads resulting in a large number of indexes, similarly named attributes with different types, and attributes completely absent from classads and constraints of classads. We first describe a restricted data model on which we base our indexing infrastructure, and discuss the basic indexing algorithms ignoring the above concerns. We then sequentially augment our indexing model to address each of the above difficulties.

Finally, it is important to note that due to the embedding of the matchmaking framework in a shared resource environment, there are externally defined factors (such as priority schemes) that define the order in which requests must be satisfied. Since the priority scheme may not, in general, have any correspondence with the rectangle geometry of the requests themselves, bulk processing techniques based on rectangle geometry (such as the plane-sweeping algorithms commonly used to detect rectangle intersections) are not appropriate. Instead, only offer classad rectangles are stored in a spatial index, and each request is used in turn as a *window query* to probe the index and identify compatible offers. After compatible candidates are identified, algorithms that reconcile candidate preferences, system priorities and other policies are invoked to select a single candidate, after which the corresponding rectangles of the candidate are removed from the index, and the process is repeated for the next request.

4.2 The Indexing Data Model

The classad data model is extremely general: each named expression may be arbitrarily complex. In practice, however, most attributes found in classads are equivalent to single values either because they are already literal expressions, or because they are *constant expressions* which can be completely evaluated locally since they do not involve attributes from candidate match classads. In other words, most attributes resolve to single values after applying the specialization algorithm discussed in Section 2.6.1. Similarly, we note from practice that most constraints are boolean combinations of comparisons of foreign attributes with constant expressions. We use these observations to define an indexing scheme that is optimized to operate with classads in this observed form.

We first define our *idealized indexing data model* (IIDM) and then describe the equivalence of classads in IIDM form to (hyper) rectangles. We then identify forms of classads that deviate from the IIDM, but which can still be accommodated in the indexing scheme, albeit at reduced efficiency.

4.2.1 IIDM Structure

The IIDM is a restriction of the classad data model, where attributes are constant expressions, and constraint expressions are in *disjunctive profile form* (DPF).[1] An expression is in DPF if it is a disjunct of one or more *profiles*, where each profile is a conjunct of one or more *properties*, and each property is a comparison between an attribute and a constant expression.

These definitions are captured in BNF notation in Figure 18. It is important to bear in mind that the BNF grammar only describes the *logical* structure of indexable classads, and not their physical syntactic structure. Thus, although the grammar indicates that attributes must exist within the port of a classad, the attributes may actually exist in a lexical super scope of the port in question. Furthermore, the non-terminal *ConstantExpression* describes a evaluation-time

[1]The structure of rank expressions does not concern us, since we are only interested in identifying compatible classads.

$$\begin{array}{rcl}
\mathit{IIDMClassAd} & \Rightarrow & \mathit{IIDMPort}^{*} \\
\mathit{IIDMPort} & \Rightarrow & \mathit{Attribute}^{*}\ \mathit{Constraint} \\
\mathit{Attribute} & \Rightarrow & \mathit{ConstantExpression} \\
\mathit{Constraint} & \Rightarrow & \mathit{Profile}\ (\texttt{||}\ \mathit{Profile})^{*} \\
\mathit{Profile} & \Rightarrow & \mathit{Property}\ (\texttt{\&\&}\ \mathit{Property})^{*} \\
\mathit{Property} & \Rightarrow & \mathit{AttributeName}\ \mathit{Relation}\ \mathit{ConstantExpression} \\
\mathit{Relation} & \Rightarrow & \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>}
\end{array}$$

Figure 18: Informal IIDM grammar

characteristic rather than the syntactic structure of an expression. The set of values in the IIDM is also restricted from the general classad data model to string, numeric, boolean, absolute time, and relative time values. Integer and real values are identified by the IIDM as “numbers,” and undefined, error, classad and list value types are not considered for indexing.

4.2.2 Rectangle Equivalence

Each indexable classad (i.e., each classad in IIDM form) can be converted into a set of rectangles, which are in turn cross-products of one or more independent intervals. In general, each port yields one or more rectangles that are obtained from the attributes and constraints expressed in that port. Each disjunct in a constraint (i.e., each profile) corresponds to a rectangle, and each conjunct in the profile (i.e., property) corresponds to an end-point of an interval in that rectangle.[2] The rectangle conversion algorithm therefore is mostly a recursive traversal of a classad's constraint expression that creates new rectangles for each profile, and adds interval bounds for each property encountered within a profile.

It is interesting to note that certain kinds of consistency errors in constraints that cannot be easily detected through other means (like specialization) can be detected as part of the rectangle conversion process. Consider the constraint `other.x < 10 && other.x > 12`, which obviously cannot be satisfied. We can detect this error easily because the lower bound of the corresponding interval must exceed the upper bound of the interval — a contradiction. We call this error a *value consistency error* because it violates the consistency requirements of an interval in the context of the interval's domain, which, in this case, is the set of real values. Note that although the expression is contradictory, it does not evaluate to **false** under specialization (i.e., partial evaluation) because the attribute x is foreign and absent.

A second consistency error arises when the same attribute is compared to values of different types in the same profile, which we call a *type consistency error*. We postpone the discussion of type consistency errors until we consider the generalization of the indexing infrastructure to multiple types.

[2]Due to two minor complications, properties and profiles do not correspond bijectively to the set of resulting intervals and rectangles. First, some constraints result in two-sided intervals (e.g., `other.x > 10 && other.x < 50`), so several properties may collectively define a single interval. Second, unlike all other comparison operators, the *not equal* operator (`!=`) renders into two disjoint intervals in the same dimension, so a single property may yield more than a single rectangle.

4.2.3 Deviation from the IIDM

Although the IIDM describes the vast majority of classads observed in practice, we also cover some of the exceptions. Since many different attributes, properties and profiles constitute the classad, minor deviations from the IIDM may still be overcome by using remaining information from the classad to include or exclude it as a candidate match.

We consider two relaxations: attributes that are arbitrary expressions, and properties that are arbitrary expressions. It is easily seen that these relaxations are very substantial, as the relaxed IIDM describes arbitrary [3] classads! Thus, in a certain perverse sense, the indexing model can actually index arbitrary classads. We nevertheless do not make this claim, as it does not accurately capture the motivations and assumptions underlying our indexing model.

We postpone further discussion of these relaxations until Section 4.4.5, where many details of indexing the semi-structured data model are disclosed.

4.3 Indexed Matchmaking with Rectangles

Matchmaking with rectangles is simple. Conceptually, the change is analogous to using an index nested loops join algorithm instead of a tuple nested loops algorithm, where both indexed data and index query keys are rectangles (i.e., a spatial join). See Algorithm 1.

[3] If the constraint deviates from DPF, the "outermost non-conformant" expression, which in the worst case may be the entire constraint, may be treated as the arbitrary property.

Algorithm 1 Brief Indexed Matchmaking Algorithm Outline

```
1: $O \leftarrow$ offer classads
2: $R \leftarrow$ request classads
3: $O_{rect} \leftarrow$ classads in $O$ converted to rectangles
4: $O_{indexed} \leftarrow$ indexed intervals of rectangles in $O_{rect}$
5: $R_{sorted} \leftarrow R$ sorted by priority
6: for each $C$ in $R_{sorted}$ do
7:     $R \leftarrow$ convert $C$ into a rectangle
8:     $M \leftarrow$ all rectangles in $O_{indexed}$ intersecting $R$
9:     $B \leftarrow$ pick best match from $M$
10:    output($B$,$R$)
11:    remove $B$ from $O_{indexed}$
12: end for
```

Assume that all requests and offers follow respective schemas and place constraints on all attributes of the target match, so that all rectangles involved in the match are of the same dimensionality. (In effect, the above assumption imposes a structured data model, as opposed to the semi-structured model in actual use, to simplify the exposition of the indexing and querying process.) Under this assumption, the rectangles may be stored in any of the many excellent multidimensional indexing structures like grid files, R-trees and hB-trees (see [20] for an excellent survey of multidimensional data access methods). The index interrogation in line 8 is a window query that returns all possible matches for the request R, from which the best match B is obtained through a process that is outside the scope of the discussion. Finally, the matched offer is removed from the index, thus "consuming" the offer, after which the next request is matched.

We now describe the indexing scheme for semi-structured data by iteratively relaxing aspects of the above assumption of uniform structure, and discuss the required modifications.

4.4 Indexing Semi-Structured Data

4.4.1 High and Variable Dimensionality

The first level of complexity we consider is the high and variable dimensionality of classads. In practice, it is common for request and offer classads to access approximately six to eight attributes from each other, requiring eight dimensional rectangles. Spatial data structures that require strictly interleaving levels for various dimensions, or naive nesting of data structures to represent higher dimensional rectangles scale poorly and impose high overhead to represent the relatively moderate numbers of high dimensional rectangles prevalent in matchmaking scenarios.

Another problem of extreme importance is that not all rectangles are of the same dimensionality. The very distributed nature of our framework implies that different offers and requests both represent themselves and constrain each other in possibly distinct and idiosyncratic ways. The problem is exacerbated by the presence of heterogenous resources, such as workstations and software licenses, which may have completely different namespaces and therefore, in a certain sense, do not meaningfully belong to the same "rectangle space."

Our solution is to independently index each dimension with a one-dimensional data structure, which makes it trivial to store rectangles of varying dimensions. Rather than a single multi-dimensional index, we therefore have a collection of one-dimensional indexes, each identified by the name of the attribute being indexed (i.e., the dimension name).

We use the interval tree data structure [13] as our index mechanism. The interval tree is a time and space optimal in-memory data structure that can efficiently store a large number of intervals and, when presented with a “window” interval, retrieve the stored intervals that intersect the window.

The advantage of storing rectangles intervals in independent structures is space efficiency. The only intervals stored in an index are those that belong to rectangles which have an interval in the dimension of concern. Thus, small dimensional rectangles do not impact the space requirements of indexes on dimensions they do not have.

However, there are two disadvantages to this scheme. First, the spatial pruning effects that are obtained when querying a single spatial data structure are lost. By this we mean that during the query process, if it is determined that certain rectangles do not intersect the query window along some dimension, then the other intervals of those rectangles do not have to be considered further since those rectangles cannot possibly be in the answer set. By decoupling the dimensions into separate indexes, we cannot benefit from the above pruning effects. The second disadvantage of our scheme is the necessary complication in both specification and implementation of multiple index management.

While the first disadvantage is inherent in our approach, the second disadvantage is marginal since the presence of multiple indexes (of some sort) is required to handle the case of similarly named attributes with different types, as discussed in Section 4.4.4.

4.4.2 Similarly Named Attributes

Consider the following two completely valid compatible classads.

```
[ x = 7 ; Constraint = other.x > 1 ]
[ x = 3 ; Constraint = other.x < 9 ]
```

It is clear that these classads cannot be correctly represented as conventional rectangles. While this example appears contrived, it illustrates the difficulty imposed by the unreasonable assumption of disjoint namespaces between requests and offers, especially in a decentralized data model — commonly named attributes such as *Name* and *Type* may exist in many classads.

Our solution is to partition the intervals of every rectangle into *exported reference* and *imported reference* sections, which are generated from the constraints and attributes of classads respectively. Separate indexes are maintained for exported and imported intervals. Furthermore, the exported dimensions of the query rectangle only query the indexes of imported attributes and *vice versa*.

In the above example, the two classads would be rendered into the following rectangles.

	ClassAd 1	ClassAd 2
Imported	$x : [7, 7]$	$x : [3, 3]$
Exported	$x : (1, \infty)$	$x : (-\infty, 9)$

As can be seen, the imported interval of classad 1 intersects the exported interval of classad 2 and *vice versa*, correctly identifying the two matching classads.

4.4.3 Identifying Attributes to Index

A recent snapshot of Condor revealed almost one hundred and fifty distinct attributes in request and offer classads. Clearly, the overhead of indexing all these attributes is prohibitive especially since only a very small number of attributes are actually accessed in any matchmaking cycle.

Rather than indexing all classad attributes, we employ an adaptive approach in which only attributes that are accessed are indexed. The set of attributes that are accessed by offers (the *offer external reference* set) is identified by taking the union of the external references of all offer constraints. Similarly, the attributes accessed by requests (the *request external reference* set) is determined by taking the union of all external references of all request constraints. The external reference set of a single classad corresponds exactly to the dimension names of its exported intervals. Furthermore, since the external references of offers map into the attributes of requests, the exported reference set of offers is also the imported reference set of requests (and *vice versa*). Thus, detecting the external reference sets of offers and requests completely determines the sets of indexes to be constructed.

External reference set determination is one of the first steps performed when matchmaking with indexes. The subsequent rectangle conversion algorithm is performed in the context of given exported and imported reference sets, so that missing or deviant attributes and constraints in each rectangle may be identified and accounted for. Further details of this process are presented in Section 4.4.5.

4.4.4 Type Heterogeneity

An important consequence of a semi-structured data model is the possibility of similarly named attributes with different types, as with x illustrated below.

```
[ x = 73282; Constraint = other.y == "foo" ]
[ x = "asd"; Constraint = other.y == "bar" ]
```

To handle this situation, we create separate indexes for each distinct type (in the classad language sense) of attribute encountered. Since indexes are keyed by attribute name, we augment index names with type information by means of a simple name mangling scheme.

Partitioning indexes on the basis of type as above is a valid approach for the classad data model since it is not possible for a single expression in property form to be satisfied by attributes of multiples types. (This trick would not work if the classad language automatically converted strings to numbers as required by context.) With respect to the above two classads, the query `other.x > 10` will only be satisfied by the first classad, while the query `other.x > "a"` will only be satisfied by the second. Thus, we note that all information required to select the index to query is available in the query itself: the attribute name is obtained from the left hand side of the comparison, and the type information required to mangle the name is obtained from the value on the right hand side of the comparison.

The observation on partitioned indexes also motivates the definition (and detection) of *type consistency errors* as discussed in Section 4.2.2. Since a single classad can only answer at most one of several "type inconsistent" property queries on the same attribute, a profile consisting of a conjunct of such inconsistent properties cannot be satisfied. Such type consistency errors are detected during the rectangle conversion process to identify defective classad constraints prior to the matchmaking process.

4.4.5 Absent Attributes and Constraints

The most involved issue in the indexing model is accounting for absent attributes in classads, unspecified properties in constraints, and other deviations from the IIDM. Our solution is to use a more detailed description of rectangles to provide sufficient information to account for differences. A rectangle's intervals are still partitioned into imported and exported intervals, which respectively correspond to the attributes and constraints of the relevant classad. However, additional distinctions are introduced to identify dimensions that are either absent from the idealized fully-structured model or otherwise deviant from the IIDM due to the presence of arbitrary expressions that cannot be rendered into sets of named intervals.

The matchmaking algorithm then performs the rectangle intersection algorithm with respect to the available specified intervals, and "patches" the results appropriately to account for missing or deviant intervals.

4.5 The Complete Indexing Solution

We now integrate solutions to all the above issues and present the complete indexed matchmaking solution. We first present an overview of the algorithm, and then discuss details of the algorithm and specific data structures.

4.5.1 Overview

The indexed matchmaking algorithm proceeds in two phases. In the Setup phase, the required classads are obtained, preprocessed and stored in efficient data structures. The Setup phase is then followed by the Match phase, when request classads are used in turn to probe the indexed structures and matches are identified.

Offer and request classads are obtained in the first step of the Setup phase, after which offer and request external reference sets are determined. External references are identified by applying the algorithm detailed in Section 2.6.2; essentially a bound/free variable identification algorithm that only reports the free variables reachable from a classad's constraints. The offer and request external reference sets are then used as exported and imported reference sets respectively to convert the offer classads to a set of rectangles. This conversion process and the details of representing single rectangles are discussed in Section 4.5.2. During the conversion process each rectangle is assigned a unique rectangle identity number (RID), which serves as a convenient handle to name rectangles. Offer rectangles are then aggregated, indexed and stored in efficient structures as discussed in Section 4.5.3. Finally, the request classads are sorted by an externally defined priority order, ending the Setup phase.

In the Match phase, request classads are considered one at a time to identify matches. Each request classad is first converted into rectangle form, and the resulting structure is used as a window query to the index structures. Section 4.5.4 discusses the query algorithm that is applied to yield the set of all candidate matches. The best match from the candidate set is then selected by criteria outside the scope of this chapter. The match is recorded and the matched offer is removed from the index structures.

4.5.2 Rectangle Conversion

Rectangle conversion occurs in the context of given sets of imported and exported references, which in essence, define a "super-schema" encompassing *all* the "schemas" required by individual classads. Considering the possibility of missing and unstructured information in individual classads, each rectangle R

obtained from a classad C may be characterized exactly (for the purposes of indexing) by the following information.

$\mathit{Imp}(R)$ A set of intervals named by attributes in the the imported reference set that are constant expressions in the classad in question. Since constant expressions evaluate to single values, the intervals in this set are degenerate (i.e., single points).

$\mathit{Imp}_{\mathit{absent}}(R)$ A subset of the imported reference set corresponding to attributes that are absent from the classad in question.

$\mathit{Imp}_{\mathit{deviant}}(R)$ A subset of the imported reference set corresponding to attributes that are present in the classad, but are not constant expressions.

$\mathit{Exp}(R)$ A set of named intervals corresponding to the constraints established on attributes named in the exported reference set, which are present as properties in the DPF constraint.

$\mathit{Exp}_{\mathit{absent}}(R)$ A subset of the exported reference set corresponding to attributes that are not present in $\mathit{Exp}(R)$.

$\mathit{Exp}_{\mathit{deviant}}(R)$ A single boolean flag that is true if the profile in the constraint corresponding to the rectangle R has a sub-expression that is not in property form.

The following points about the above characterization are worth noting.

1. Every reference from the imported reference set applied to the classad will appear in exactly one of $\mathit{Imp}(R)$, $\mathit{Imp}_{\mathit{absent}}(R)$ and $\mathit{Imp}_{\mathit{deviant}}(R)$.

2. A reference from the exported reference set applied to the classad will appear in exactly one of $\mathit{Exp}(R)$ and $\mathit{Exp}_{\mathit{absent}}(R)$.

3. The names of intervals in $\mathit{Imp}(R)$ and $\mathit{Exp}(R)$ are name mangled, but the attribute names in $\mathit{Imp}_{\mathit{absent}}(R)$, $\mathit{Imp}_{\mathit{deviant}}(R)$ and $\mathit{Exp}_{\mathit{absent}}(R)$ are not name mangled.

4.5.3 Index Structures

The rectangle characterizations described above are obtained for each offer rectangle, collated and stored in the following data structures.

Imported Interval Index Map A map of mangled names to indexes. For each mangled name M, the corresponding index indexes intervals in the M dimension from $\mathit{Imp}(R)$ for each offer rectangle R. Given a query interval (or window) W and mangled name M, the operation $\mathit{ProbeImported}(M, W)$ probes the Imported Interval Index Map and returns the set of RIDs of rectangles that have an interval that intersects W in the M dimension.

Unimported Dimensions Map A map of names to sets of RIDs, this structure records the rectangles that have absent imported attributes for any given named attribute. Given an unmangled name N, the operation

ProbeUnimported(N) probes the Unimported Dimensions Map and returns the set of RIDs of rectangles that do not have an imported interval for attribute N.

Deviant Imported Intervals Map Also a map of names to sets of RIDs, the Deviant Imported Intervals Map records the rectangles that have deviant imported attributes for each named attribute. For an unmangled name N, the operation *ProbeDeviantImported*(N) probes the Deviant Imported Intervals Map and returns the set of RIDs of rectangles that have N as a deviant imported attribute.

Exported Interval Index Map Similar to the Imported Interval Index Map, the Exported Interval Index Map is a map of mangled names to indexes that index intervals from *Exp*(R) along the named dimension for each offer rectangle R. Given query interval W and mangled name M, the operation *ProbeExported*(M, W) returns the set of RIDs of rectangles that have an interval that intersects W in the M dimension.

Unexported Dimensions Map A map of names to sets of RIDs, this structure identifies rectangles that do not have exported intervals for the specified dimensions. Thus, given an unmangled name N, this data structure enables the operation *ProbeUnexported*(N), which returns the set of RIDs of rectangles that do not have any exported interval for attribute N.

Deviant Exported Rectangle Set A set of rectangle IDs that identifies rectangles that have a deviant expression in their constraints. Since this structure is itself a set of RIDs, no higher-level operations are required to be defined for its use.

In addition to these primary index structures, maps of RIDs to classads and ports are also maintained, so that given an RID, the port containing the named rectangle, and the classad containing the port in question can be efficiently identified.

4.5.4 Querying and Match Identification

The query window used to probe the indexed offers is obtained by converting the relevant request classad into rectangle form. Thus, the query window too has imported and exported sections, each of which is composed of normal, absent and deviant components, requiring a detailed case analysis of the query components *vis a vis* the index structures.

It is important to note that due to the presence of deviant components in both offers and requests, the validity of a match cannot always be completely determined from indexed information alone. Our indexing scheme returns a superset of compatible classads, and actual compatibility of individual candidates from the result set must be verified via constraint evaluation prior to match creation.

The overall process of querying given query rectangle Q is as follows.

1. We begin by inserting the RID of every rectangle in the (provisional) result set.

2. Given a (mangled) name-interval pair (M, I) from $Imp(Q)$, the set of rectangles satisfied by the interval must exist in one of the following three cases.

   (a) All rectangles that export intervals that intersect I in dimension M.

   (b) All rectangles that do not export an interval in dimension N, where N is the demangled name of M. These are the classads that do not place a constraint on the attribute N.

   (c) All deviant exported rectangles. Since nothing is known about the structure of constraints in deviant exported rectangles, they must be considered as candidates.

   Since any rectangle that exists in the eventual result set *must* exist in one of the above three cases, we intersect the provisional result set with the union of the RID sets obtained from these three cases. This process is repeated for each name-interval pair in $Imp(Q)$.

3. At this stage in the process, every rectangle whose constraints are satisfied by the defined attributes of the query are included. We must, however, now *exclude* rectangles that have placed constraints on attributes that are absent in the query. Thus, for every name N in $Imp_{absent}(R)$, we remove all rectangles that place a constraint on N from the provisional result.

4. We now consider the processing of exported components in query window. Given a (mangled) name-interval pair (M, I) from *Exp*(Q), the rectangles that satisfy this interval must exist in one of the following cases.

   (a) All rectangles that import intervals that intersect I in dimension M.

   (b) All rectangles that have deviant imported attributes named N, the demangled version of M.

   As in the imported intervals case, since any rectangle that exists in the eventual result *must* exist in one of the above two cases, we intersect the provisional result set with the union of the RID sets obtained from the above two cases. However, we must now exclude rectangles that do not import the attribute that is being constrained, so we subtract the set of RIDs of rectangles that do not import N, the demangled version of M. This procedure is then repeated for every name-interval pair in *Exp*(Q).

5. The provisional result remaining at this time is the final result.

The correctness of this procedure is not clear at first. Intuitively, we expect a more complicated algorithm to account for all the cases that exist: since each kind of imported component (imported, unimported and deviant imported) must be considered in turn with each kind of exported component (exported, unexported, deviant exported) it is natural to expect nine cases for the imported section of the query window, and nine more for the exported section of the query window.

The following observations are the key to understanding the absence of these cases.

1. Our querying procedure is an *exclusionary* algorithm that begins by assuming that all rectangles exist in the result set and then rejects rectangles known to not intersect the query rectangle. Thus, the absence of additional cases in the algorithm only signifies that the interaction between the exported and imported components in those cases do not provide any information that would allow the rejection of additional rectangles.

2. Rectangles rejected by one component of the imported or exported section cannot be re-admitted by another component of either section. This property holds because of an extension of the principle that two iso-dimensional rectangles intersect if and only if their projected intervals intersect in *every* dimension. In direct analogy, if the interval of some rectangle does not intersect the query window's interval along that dimension, that rectangle can be safely excluded from the result set irrespective of the presence, absence or complexity of other attributes and constraints of the rectangle.

For example, from the informal query procedure description we see that the imported intervals are indeed considered against all three variants of exported components. However, unimported attributes in the query window are only considered against exported intervals, and not unexported or deviant exported components. Upon reflection, it is clear that these cases do not allow the rejection of any rectangles: the interaction between absent attributes *vis a vis* absent constraints and arbitrarily complex constraints does not let us identify any rectangles that may be excluded.

The algorithm for determining query results is illustrated in Algorithm 2. Except for a few code motion optimizations, the algorithm is a straightforward transcription of the overview presented previously to psuedo-code. The algorithm assumes that the query is in DPF, and is therefore composed of several rectangles. The query procedure described previously is applied to each rectangle, and the query results are aggregated through a set union operation.

4.6 Performance Study

The goal of the following performance study is not to merely establish that the indexing scheme outperforms the naive record-at-a-time expression evaluation mechanism — this result is expected. We instead wish to compare the relative performance of the two schemes to determine how much more efficient the indexing scheme is in workloads that vary parameters such as number of attributes, domain size of attribute values, constraint complexity and number of advertisements. We begin with a discussion of the workload used to perform these experiments, and then present the experiment results.

Algorithm 2 Query Algorithm

```
1: **Input:** Query window $Q$, index structures
2: **Output:** $T$, set of rectangle IDs
3:
4: $T \leftarrow \emptyset$
5: for each query rectangle $R$ in $Q$ do
6:     $o \leftarrow \mathcal{U}$ {Universal Set of all RIDs}
7:
8:     {Process imported intervals}
9:     $o \leftarrow$ all RIDs in Deviant Exported Rectangle Set
10:    for each $(M, I)$ in $Imp(R)$ do
11:        $t \leftarrow ProbeUnexported(Demangle(M))$
12:        $t \leftarrow t \cup ProbeExported(M, I)$
13:        $o \leftarrow o \cap t$
14:    end for
15:
16:    {Remove candidates that constrain absent attributes}
17:    $u \leftarrow \emptyset$
18:    for each $N$ in $Imp_{absent}(R)$ do
19:        $u \leftarrow u \cup ProbeUnexported(N)'$ {Set complement}
20:    end for
21:    $o \leftarrow o \setminus u$
22:
23:    {Now process exported dimensions}
24:    for each $(M, I)$ in $Exp(R)$ do
25:        $t \leftarrow ProbeImported(M, I)$
26:        $t \leftarrow t \cup ProbeDeviantImported(Demangle(M)$
27:        $t \leftarrow t \setminus ProbeUnimported(Demangle(M))$
28:        $o \leftarrow o \cap t$
29:    end for
30:
31:    $T \leftarrow T \cup o$
32: end for
```

4.6.1 Workload

Each experiment in the workload consists of a number of *object advertisements* and an equal number of *query advertisements*. Experiments identify the number of symmetric matches for each query using both mechanisms, and measure the elapsed time, memory usage and overheads of these two mechanisms. In spirit, these experiments measure the efficiency of performing joins with a tuple nested loops algorithm and an indexed nested loops algorithm. As explained earlier, although we expect the indexed algorithm to outperform its counterpart, we wish to know how the algorithms behave as parameters of the workload are varied.

Each experiment in the workload is characterized by three parameters: number of advertisements, number of attributes in each advertisement and domain size of each attribute. The constraint complexity of each advertisement is dictated by the number of advertisement attributes of the workload since every attribute of the candidate match set is constrained by either an upper or lower bound. Attribute domain size is varied on a per attribute basis, with attributes having a domain size of 10, or the the number of advertisements in the experiment itself (denoted N). All advertisements in each experiment belong to one of the following three domain size mixes: all attributes have a domain size of 10, half the attributes have a domain size of 10 and the rest have domain size N, all attributes have domain size N. These mixes are annotated as T, M and D in the ensuing performance graphs. Attribute values are constructed

```
[
   A          = 3;
   B          = 6;
   C          = 7;
   D          = 5;
   Constraint =
       other.Z<=5 &&
       other.Y<=6 &&
       other.X>=1 &&
       other.W<=7
]
```

Figure 19: Object advertisement from the index performance workload

by picking a uniformly random number between 0 and the domain size of that attribute. Constraints are created by similarly picking a random value, and choosing (with equal probability) a greater than or less than comparison with the chosen value. The final constraint for each advertisement is the conjunction of these one-sided intervallic constraints on each attribute. Finally, to create a semi-structured workload, the attributes or intervallic constraints of an advertisement may be omitted with a 5% probability. Figures 19 and 20 show sample object and query advertisements from the workload.

4.6.2 Results

Figure 21 compares the elapsed time performance of the expression based algorithm with the index algorithm for workloads of various size. The legend in the graph employs the notation $A/n/D$, where A is the algorithm (E for expression-based algorithm, I for indexed), n is the number of attributes in

```
[
   Z          = 0;
   Y          = 9;
   W          = 6;
   Constraint =
       other.A>=0 &&
       other.B<=6 &&
       other.C>=1 &&
       other.D>=7
]
```

Figure 20: Query advertisement from the index performance workload

each advertisement and D is the domain size mix (T, M or D).

There are several interesting features in this graph. First, we note that many of the curves for the expression based mechanism are co-incident, reflecting the fact that attribute domain size does not affect expression evaluation. For clearer exposition, only one representative curve from each of these bands is illustrated in Figure 22.

The graph in Figure 22 also shows that the indexing algorithm outperforms the expression-based algorithm by an enormous margin — the performance curves of the indexed algorithm are almost co-incident with the x-axis when compared to the performance of the expression-based algorithms. Finally, we note that constraint complexity markedly affects the performance of the expression-based algorithm.

Figure 23 shows the performance of the indexed algorithm in isolation. This graph shows the elapsed time performance of the algorithm on workloads containing up to 16,000 advertisements — eight times the workload size of the

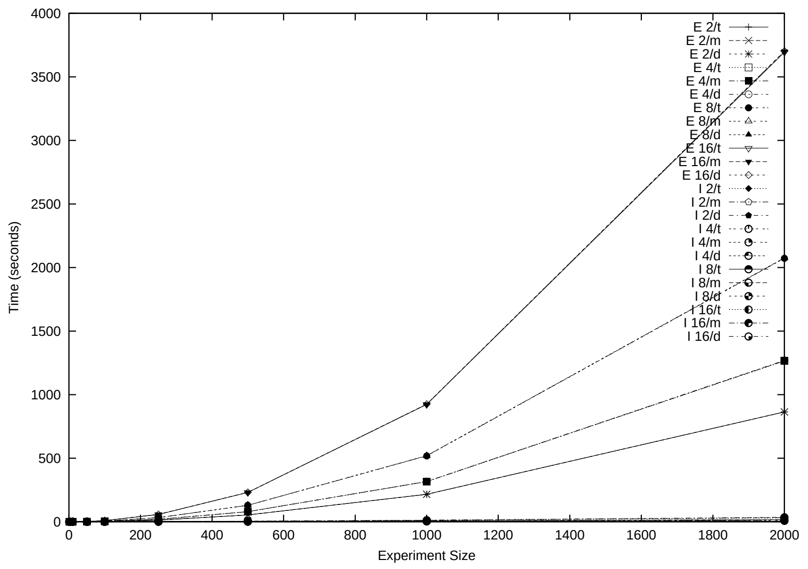

Figure 21: Elapsed time performance of the expression based and indexed query algorithms.

expression based algorithm. From this graph we observe that both constraint complexity and attribute domain size alters the performance of the algorithm.

Figure 24 illustrates the amount of memory used for creating and maintaining the data structures required by the indexed algorithm. We observe that the memory requirements of the algorithm scale linearly with the number of the advertisements, with the slope being proportional to the constraint complexity and attribute domain size parameters.

We finally consider the elapsed time overheads induced by the initialization phase of the indexed algorithm. These overheads are incurred by the external reference determination and rectangle conversion phases of the algorithm. Since

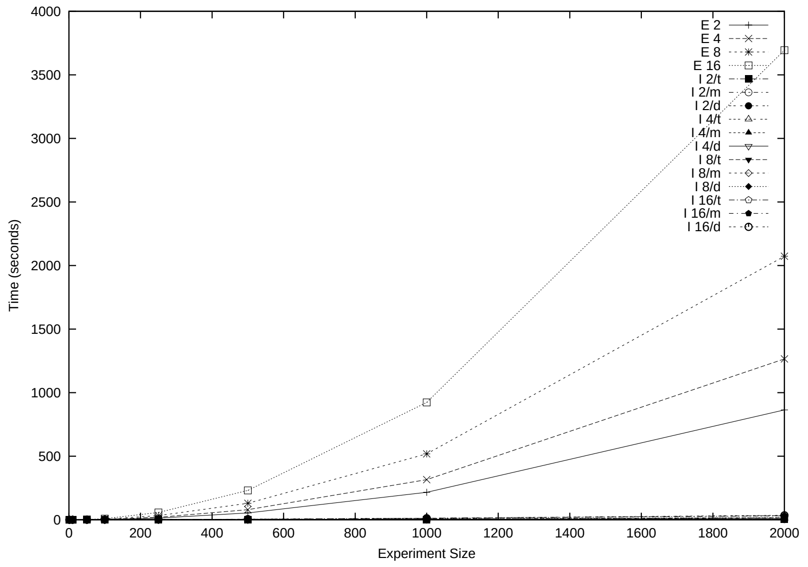

Figure 22: Representative performance curves of the expression-based and indexed query algorithms.

these processes are essentially traversals of the syntax tree of constraint expressions, both external reference determination and rectangle conversion algorithms are independent of attribute domain size Furthermore, the per advertisement costs of these processes are not dependent on total input size. We therefore present only the amortized per advertisement elapsed time overheads of these algorithms plotted against constraint complexity in Figure 25.

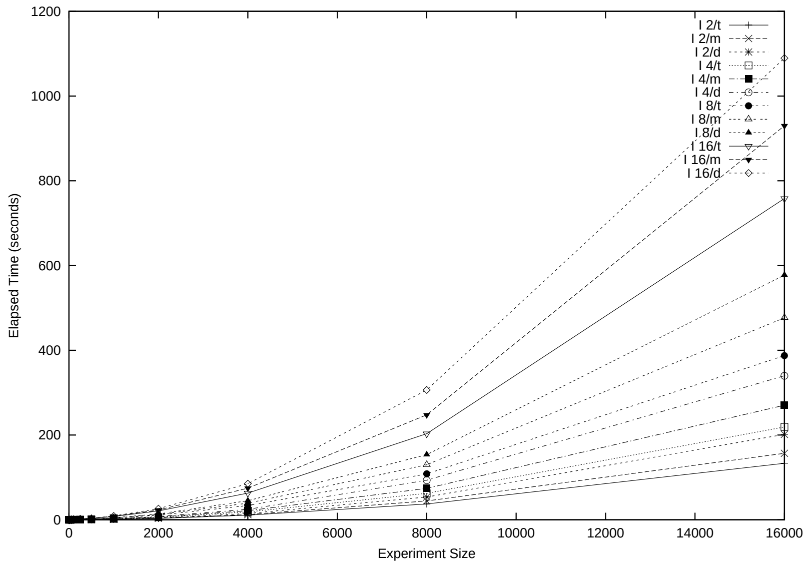

Figure 23: Elapsed time performance of the indexed query algorithm.

4.7 Related Work

Lorel [4] is a query framework developed for semi-structured environments that includes a sophisticated indexing model [38]. The Lorel indexing model emphasizes the querying of hierarchical information via generalized path expressions and pattern matching, and has many sophisticated mechanisms to tolerate weakly typed data. The classad data model is relatively flat and has a stronger typing system, but requires constraint indexing mechanism.

Kanellakis et. al. [30] describe a constraint indexing scheme that uses interval trees [13]. However their work is not formulated for a semi-structured data environment.

The classad notation is very similar to that of generalized tuples found in

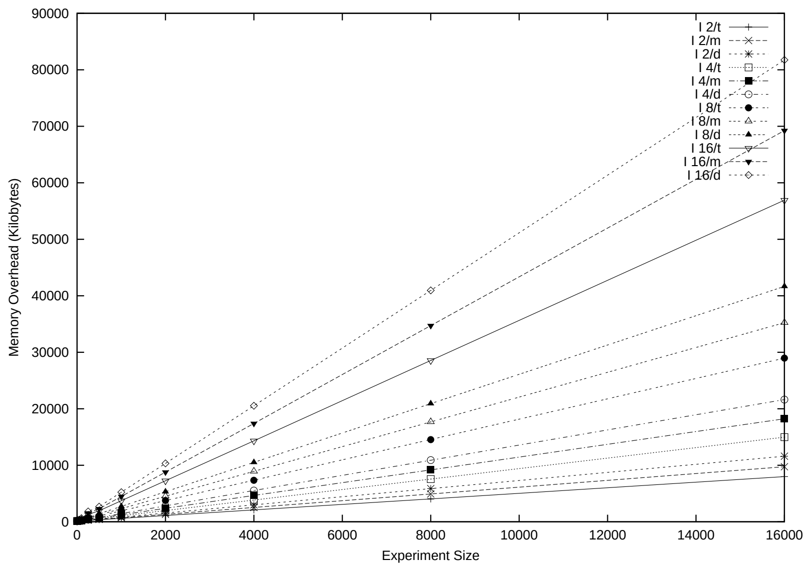

Figure 24: Memory overhead of the indexed query algorithm.

constraint databases [21]: the classad mechanism also employs a system of equations to specify regions of the parameter space that are of interest. The matchmaking operation then intuitively reduces to a spatial join [40] between server and customer classads. Indeed, these similarities have motivated the design of the proposed indexing model. However, there are several differences between classads in matchmaking and generalized tuples in constraint databases.

First, matchmaking differs from a spatial join in that matchmaking "consumes" classads during the matching process — a matched classad is removed from further consideration in the matchmaking process. Thus, while a given classad may never occur in more than one match in matchmaking, a spatial join over the classad domain will result in the generation of all valid matches. Disambiguation and match selection must then be performed by a second "filtering"

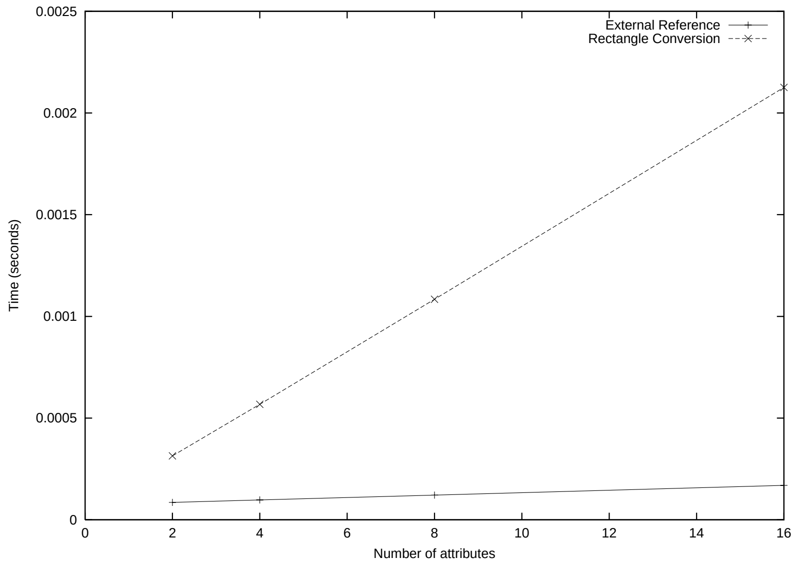

Figure 25: External reference and rectangle conversion overheads.

process that is independent of the spatial join.

A second difference between constraint databases and classads is that classads employ a semi-structured data model — each classad in a collection may potentially carry a distinct schema. In contrast, constraint databases require fixed schemas over which generalized tuples are defined.

Chapter 5

Gangmatching Algorithms

The gangmatching model described in Chapter 3 provides an expressive declarative basis for a matchmaking framework. We now consider the problem of efficiently identifying gangs; i.e., implementing gangmatching algorithms.

We begin with a discussion of some basic correctness and performance issues in gangmatching algorithm implementation. The methods used to measure algorithm efficiency are then presented, including a description of the base workloads used in these measurements. We then present a number of gangmatching algorithms and discuss the main characteristics of the algorithms as observed under the base workloads.

5.1 Gangmatching Algorithm Issues

5.1.1 Alternative Algorithms, Combinatorics and Efficiency

Gangmatching is fundamentally a combinatoric algorithm, since it involves marshaling consistent aggregates of advertisements. In the abstract, the gangmatching problem may be solved by enumerating all possible combinations of advertisements (i.e., all possible gangs), and only selecting those combinations (if any) in which all defined constraints are satisfied. Needless to say, such brute-force techniques are untenable: Even when only a few thousand classads are involved in gangs of size three, the total number of possible combinations is on the order of hundreds of billions.

Fortunately, the gangmatching model is declarative — rather than defining and legislating a procedure for for finding gangs, it only defines what valid gangs are. Thus we may employ alternative strategies that are far more efficient at identifying valid gangs. For example, the "declare before use" nature of advertisement ports structures the gangmatching problem so that inconsistent gangs may be detected before entire gangs are marshaled. Thus, exhaustive gang enumeration is not required for even the simplest gangmatching algorithms. The performance of the various gangmatching algorithms presented here vary primarily due to the efficacy of their mechanisms at further reducing the number of combinations that need to be considered.

5.1.2 Preferences

The gangmatching model provides a sophisticated set of mechanisms for defining and resolving preferences. The preference expression and docking vector mechanisms described in Chapter 3 collectively define a complete model of gangmatching preferences that may be used as the behavioral specification of a preference-aware gangmatching algorithm. Furthermore, implementation of the defined preference model is straightforward in the restricted bilateral matching case. This is however not the case for general gangmatching — the problem of efficient preference-free gangmatching is considerable in itself, and must be understood before general preference-aware algorithms are addressed.

In this dissertation we therefore only investigate the problem of preference-free gangmatching. Since gangmatching algorithms that consider preferences are extensions of preference-free algorithms, we wish to first establish fundamental insights and techniques that will enable the formulation of more advanced and capable algorithms. In Chapter 6 we show how the mechanisms developed here may be used to formulate preference-aware algorithms.

5.1.3 Absence of Deadlock

Gangmatching provides the key functionality of "resource co-allocation" in a matchmaking resource management framework. As such, the gangmatching formalism must guarantee a principal correctness criterion of multi-resource allocation schemes: absence of deadlock. In the context of gangmatching, the direct analogue of deadlock occurs when two (or more) partially marshaled gangs are circularly dependent in the sense that each partial gang requires to be matched with an advertisement that is currently a member of some other partial gang.

Deadlock elimination is straightforward in gangmatching due to the formalism of root ordering, which dictates that higher priority roots must be completely satisfied before lower priority roots are considered. Deadlock may therefore be eliminated by only matching high-priority roots advertisements completely, or not at all. When the unsatisfiability of a gang is detected, all advertisements currently marshaled in the gang are returned to the advertisement pool. Thus, in a sense, gangmatch operations are atomic since they either occur completely or not at all. The above property is sufficient to guarantee absence of deadlock [8].

5.2 Performance Evaluation Methods

Before presenting the various gangmatching algorithms, we first describe the methods used to measure algorithm performance. Every algorithm is first evaluated under a common *base workload*, which consists of various instances of the job-machine-license gangmatching problem. This workload has been engineered to be relatively simple, so that the tendencies and characteristics of algorithms may be easily isolated and identified. However, the workload is also fairly realistic, and includes many parameters that would be encountered in solving the job-machine-license problem in practice. Algorithms are also selectively examined under *specialized workloads* highlight their strengths and weaknesses.

5.2.1 The Base Workload

The base workload consists of several experiments, each of which tests the efficacy of the algorithm on inputs with various numbers of machines, licenses and jobs, where the number of advertisements range from a few hundred to several thousand. Furthermore, the relative numbers of licenses and jobs, and the constraints associated with licenses are manipulated to exercise algorithms under different conditions.

In any given experiment, the number of jobs is always the same as the number of machines. However, the experiment may belong to one of two regimes: one in which there are as many licenses as jobs (the 100% regime), or one in which there are half as many licenses as jobs (the 50% regime). Furthermore, the experiment may belong to one of four selectivity indexes, namely 1, 2, 4 and 8. For each selectivity index n, the machine and license sets associated with the experiment are each divided into n disjoint partitions, and licenses from any given partition p are only valid on machines in the corresponding machine partition p. For example, workloads with a selectivity index of 2 partition machines and licenses into two partitions. Licenses from the first license partition would only be valid on machines in the first partition and *vice versa*. For any given experiment size, the full parameter space of license percentage regimes and selectivity indexes is explored.

In graph legends, the performance curves of algorithms are annotated as $A/L/n$, where A is the algorithm being considered, L is the license percentage (i.e., 50 or 100), and n is the license selectivity index. The algorithm component of the annotation is occasionally omitted if the algorithm of interest is known from context.

Machine Model

The characteristics of machines have been roughly modeled on the composition of the University of Wisconsin-Madison Condor pool as of March 2000. Each machine is assigned an architecture and operating system combination with probabilities as shown in Figure 26. The physical memory of each machine

Architecture/Operating System	Probability
INTEL/LINUX	0.3191
INTEL/SOLARIS26	0.2127
INTEL/WINNT40	0.2411
SGI/IRIX6	0.0212
SUN4u/SOLARIS26	0.0390
SUN4u/SOLARIS27	0.0390
SUN4x/SOLARIS26	0.0766
SUN4x/SOLARIS27	0.0524

Figure 26: Base workload machine architecture/operating system distribution

is assumed to be independent of its architecture and operating system, and is assigned with probabilities as shown in Figure 27. Total virtual memory of the machine is always double the modeled physical memory, and available virtual

Memory (Megabytes)	Probability
32	0.070
64	0.400
128	0.200
256	0.200
800	0.130

Figure 27: Base workload memory distribution

memory is uniformly random between 20% and 90% of total virtual memory. Finally, each machine is assigned a unique integer key in $[0, n)$, where n is the number of machines being modeled. An example machine advertisement from the base workload is shown in Figure 28.

```
[
   Key    = 143;
   Type   = "Machine";
   Arch   = "INTEL";
   OpSys  = "WINNT40";
   Memory = 64M;
   VirtualMemory = 83.235176M;
   Ports =
     {
       [
         Label = Job;
         Constraint = Job.MemoryReqs<Memory-15M
       ]
     }
]
```

Figure 28: Example machine advertisement from the base workload.

License Model

Licenses do not have any attributes that need to be modeled stochastically. However, the constraints generated for individual licenses are varied to implement license selectivity indexes. Specifically, given selectivity index k, the set of generated licenses are virtually partitioned into k disjoint partitions that are numbered from 0 through $k-1$. The constraint generated for licenses in partition i is that the `HostID` provided by the job must be in the interval $[\lfloor \frac{n}{i} \rfloor i, (\lfloor \frac{n}{i} \rfloor + 1)i)$, where n is the number of machines generated for the workload. In other words, licenses from partition i are only valid on machines in the corresponding machine partition i. Finally, the number of licenses created (relative to the number machines) is also varied to implement the 50% and 100% license regimes. An example license advertisement from the base workload is shown in Figure 29.

```
[
   Type  = "License";
   App   = "sim_app";
   Ports =
     {
       [
         Label = Site;
         Constraint = Site.HostID >= 0 && Site.HostId < 1000
       ]
     }
]
```

Figure 29: Example license advertisement from the base workload.

Job Model

In contrast to machines and licenses which only have one port each, all jobs have two ports: one to dock with machines, and one to dock with licenses. The contents of the machine port are modeled as follows. The image size of a job is modeled as a random variable whose value is uniformly between 20% and 80% of twice the memory size, where memory size is first generated using the probability distribution of Figure 27. The memory requirements of a job is a uniformly random number between 20% and 50% of the job's image size. Finally, the architecture and operating system constraints of the job are generated using the probability distribution shown in Figure 26, and a final constraint requiring that the candidate machine's available virtual memory be greater than the job's image size is appended.

The main attribute of interest in the license port of the job is the `HostID` attribute, which evaluates to the *Key* attribute of the machine docked at the `Cpu` port.

An example job advertisement from the base workload is shown in Figure 30.

5.2.2 Performance Evaluation Method

Each experiment uses the gangmatching mechanism under consideration to create gangs for each of the request advertisements (i.e., jobs). The requests are

```
[
  Ports =
    {
      [
        Label      = Cpu;
        ImageSize  = 13461450.030324;
        MemoryReqs = 2860414.100368;
        Constraint = Cpu.Arch=="INTEL" &&
              Cpu.OpSys=="LINUX" &&
              Cpu.VirtualMemory > ImageSize
      ],
      [
        Label      = License;
        HostId     = Cpu.Key;
        Constraint = License.App=="sim_app"
      ]
    }
]
```

Figure 30: Example job advertisement from the base workload.

considered in the same order as presented to simulate the presence of an externally defined request prioritization. Thus, each experiment simulates a "matchmaking cycle," an abstraction that encompasses the periodic matchmaking activity in the Condor system. Of course, aspects such as match notification (which is common to all algorithms) are not considered in these experiments.

While elapsed time is arguably the most meaningful metric of algorithm efficiency, we also record "secondary statistics" that count and measure the activities and costs of gangmatching algorithms. The two statistics of import are number of expression evaluations, and number of index probes. The roles that these operations play in the ensuing algorithms, and the corresponding implications on performance are noted in the discussion section for each algorithm. In most cases, the insight that secondary statistics provide allows comparisons between algorithms purely on the basis of these statistics. As will be seen, such comparisons provide better intuitions of algorithm behavior.

Finally, it is important to note the number of successfully matched jobs is dependent on the search strategy used. (The reasons for this effect are discussed below in Section 5.5.2.) Thus, it is expected that the number of jobs successfully matched given the same workload changes with the algorithm used. The deviations are minor, however, and are usually within 5% of the size of the workload. More serious deviations (if they occur) are noted in the performance study of the algorithm in question.

5.3 Naive Gangmatching

The first algorithm that we consider is the simplest and most "natural" solution — a top-down gangmatching algorithm that uses the classad expression evaluation mechanism to determine compatibility between advertisement ports.

5.3.1 Algorithm Description

The input to the algorithm is an advertisement, which has been designated as the *root advertisement* (or root) of the required gang. In keeping with the inherent left-to-right bias of the gangmatching model, the algorithm attempts to fill each port in the order they are listed in the `Ports` attribute (see Figure 31). Compatibility between ports of advertisements is determined by evaluating the

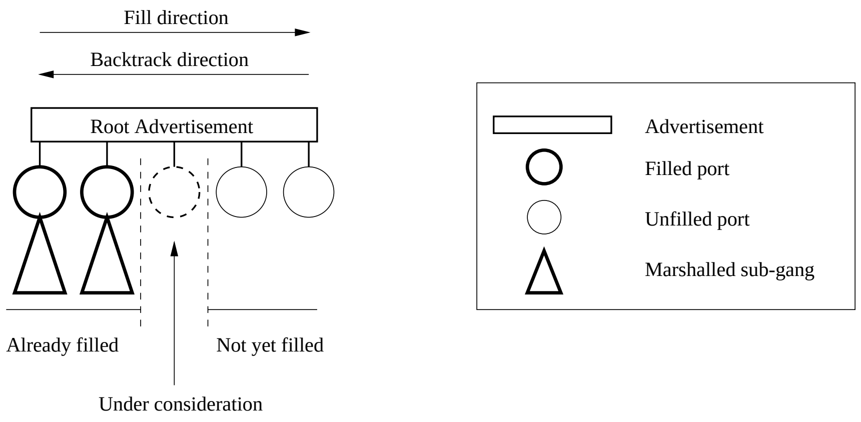

Figure 31: Operation of the Naive Gangmatching Algorithm

constraints defined in those ports. Constraint evaluation is performed by inserting the respective advertisements in an evaluation environment specially constructed by the matchmaker in which port labels evaluate to docked candidate ports. The flexibility of the classad language and the semantics of attribute references allows this evaluation environment to itself be represented by a classad. Figure 32 shows the evaluation environment after the complete gang has been marshaled. Each advertisement is encapsulated in a “context,” which serves as a namespace in which the desired label expressions are defined.

```
[
_ctx_0 = [ Cpu = _ctx_1._ad.ports[0];
           License = _ctx_2._ad.ports[0];
           _ad = [ Ports =
                     { [ Label      = Cpu;
                         MemoryReqs = 2.86041e+06
                         ImageSize  = 1.34615e+07;
                         Constraint = cpu.Arch == "INTEL" &&
                             cpu.OpSys == "LINUX" &&
                             cpu.VirtualMemory > ImageSize;
                       ],
                       [ Label      = License;
                         HostId     = cpu.Key;
                         Constraint = License.App == "app1";
                       ]
                     }
                 ]
         ];
_ctx_1 = [ Job = _ctx_0._ad.ports[0];
           _ad = [ Key              = 90
                   OpSys            = "LINUX";
                   VirtualMemory    = 427.836M;
                   Memory           = 800M;
                   Arch             = "INTEL";
                   Type             = "Machine";
                   Ports =
                     { [ Label      = Job;
                         Constraint = Job.MemoryReqs < Memory - 15M;
                       ]
                     };
                  ]
         ]
_ctx_2 = [ Job = _ctx_0._ad.ports[1];
          _ad = [ Type              = "License";
                  App               = "app1"
                  Ports =
                    { [ Label      = Job;
                        Constraint = Job.HostID>=50 && Job.HostId<100;
                      ]
                    };
                 ]
         ];
]
```

Figure 32: Expression evaluation environment for gangmatching

If, at any time, a particular port of the root advertisement cannot be filled (because no compatible candidates are found), the algorithm *backtracks* to the previously filled port (which is always the immediately preceding port) and attempts to refill that port with another candidate. Depending on the success of the refill operation, the algorithm then either continues to the succeeding port, or backtracks yet again to the preceding port. The gangmatching algorithm fails for the designated root if a backtrack is attempted from the left-most port of the root.

As with all backtracking algorithms, care must be taken to ensure that the algorithm doesn't visit an already visited configuration as a result of the backtrack operation, as this would lead to a non-terminating algorithm. To prevent this situation, each port maintains a *history set* of candidate ports with which docking attempts have already been made. During the refill operation, only candidate ports that do not exist in the history set are considered.

The history set of a port is cleared when backtracking from that port. The reason for doing so is as follows. If the algorithm backtracks from a port and then revisits it again, at least one (and possibly more) of the port's predecessors have new candidates docked to them. Since the information from these new candidates may admit matches that were previously inadmissible, the history of the port is not valid after the backtrack.

It is easy to see that the algorithm maintains the invariant that all ports that precede the one under consideration are docked and all ports that succeed the port are undocked. This invariant greatly simplifies the process of history and backtrack management, and also ensures that all inter-port references allowed by the gangmatching model are naturally resolved during the progression of the algorithm.

The final detail of this algorithm that must be described is the process of filling ports. Since the non-root advertisements of a gang may themselves have multiple ports, the above gangmatching algorithm must be applied recursively to each (non-root) advertisement included in the gang. A fundamental difference between non-root advertisements and the root advertisement in that one of the ports of a non-root advertisement must serve as a "parent link" in the gang tree. However, due to the generality of the gangmatching model, the port that will serve this role is not known *a priori* (unless, of course, the non-root advertisement has only one port). Since the parent link identification process must be performed before the gangmatching algorithm is recursively applied, the ports that precede the identified parent link will not have been docked yet (see Figure 33). Thus, the specialization algorithm described in Section 2.6.1 must be employed to identify parent link ports, If the constraints between the parent link port and its counterpart in the parent advertisement specialize to **true** or **false**, the compatibility of the parent link port is known. Otherwise, the port is marked as a "tentative yes," and used as the parent link.

Tentative parent link ports are rigorously checked for compatibility during the recursive marshaling phase when all predecessor ports of the parent link have

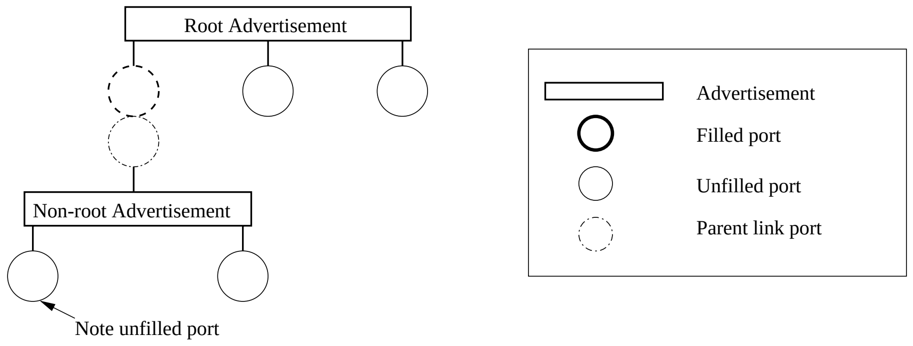

Figure 33: Identifying parent links requires specialization due to possible presence of undocked ports preceding the parent link.

been filled. If the parent link is indeed compatible, the port parent link status is changed from "tentative yes" to a "positive yes," after which the algorithm proceeds as usual. However, if the parent link port is not in fact compatible, the algorithm attempts to backtrack, incrementally refilling the ports preceding the parent link port, and re-testing the parent link port for compatibility. If the algorithm then attempts to backtrack from the left-most port (i.e., run out of all options), the tentative parent link is re-established at the next port that is possibly compatible with the parent advertisement's port, after which the algorithm proceeds as if a new candidate was being recursively matched. Of course, if a parent link (tentative or otherwise) could not be established at any of the candidate advertisement's ports, the entire candidate is rejected.

5.3.2 Performance and Observations

The performance of the naive algorithm on the base workload is illustrated in Figure 34, which plots the elapsed time of the experiment against the number of classads in the workload. The graph consists of two “bands” of curves corresponding to the 50% and 100% workloads. The bands are composed of the curves corresponding to the various license selectivity indexes, showing that the algorithm is not sensitive to this parameter in this workload. Comparison of representative curves from these bands (Figure 35) with Figure 36 shows that, as expected, expression evaluation dominates the cost of the algorithm. Although expression evaluation is a relatively light-weight operation, we see that the number of evaluations required to marshal a consistent gang of size three is quite high even for workloads with relatively small numbers of advertisements.

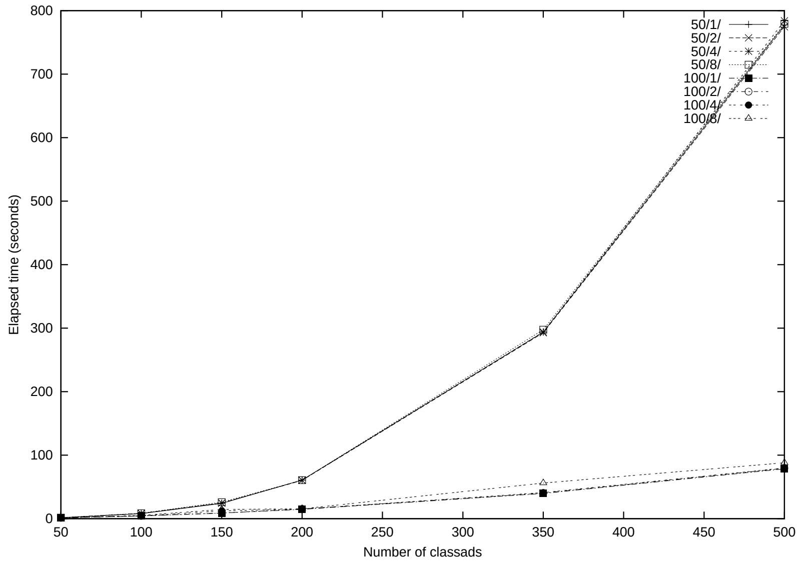

Figure 34: Elapsed time performance of the naive algorithm.

The algorithm’s basic method of testing ports of advertisements one at a

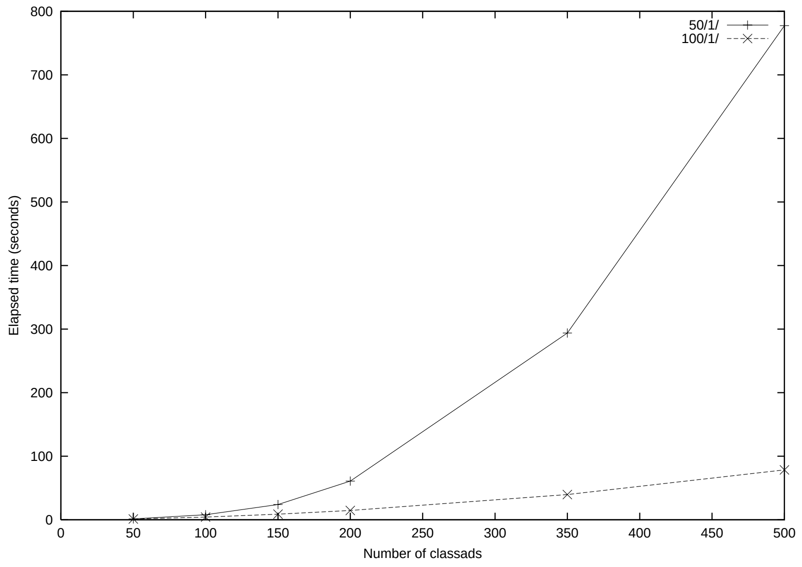

Figure 35: Representative elapsed time performance curves for the naive algorithm.

time also makes performance heavily dependent on the order in which candidates are tested. The early presence of compatible machines and licenses greatly reduces both the number of expression evaluations required to identify a compatible port, and the possibility of backtrack. To illustrate the consequences of a favorable ordering, consider the best case workload which consists of a given number of jobs, each of which is compatible with every machine and every license, and machines and licenses are perfectly interleaved. In this case, the first resource offer tested at any point is always the correct choice, minimizing the number of expression evaluations, and consequently, elapsed time.

Needless to say, such favorable orderings can only be observed in completely artificial and contrived workloads. In practice (as in realistic workloads), the advertisements of machines and licenses may exist in arbitrary permutations.

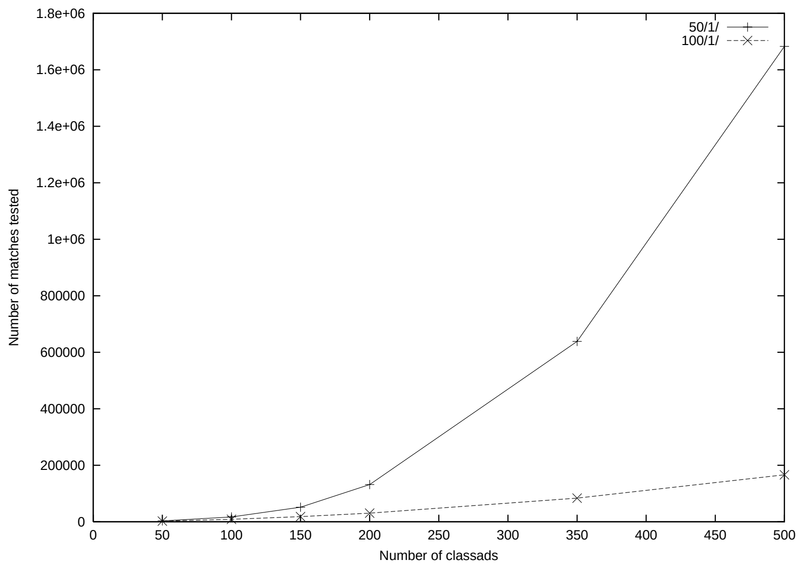

Figure 36: Representative number of matches tested by the naive algorithm.

Furthermore, machine, license and job advertisements would be far more selective in defining mutual compatibility. Finally, the natural dynamics and heterogeneity of pools would result in varying numbers of available resources at various times. Unfortunately, the above very likely scenario also exposes the worst case behavior of the naive algorithm.

As with many algorithms that "search for solutions," detecting the absence of a solution (i.e., a consistent gang) is far more expensive than constructing a correct solution (given that one exists) in the naive algorithm. This phenomenon is clearly exemplified in the upper curves in Figures 35 and 36, which illustrate the performance of the naive algorithm on workloads that have half as many licenses as machines and jobs. In these workloads, at least half of the jobs will not be able to find successful matches due to the absence of licenses. The naive algorithm does not have any mechanism to detect this situation. Instead, the vacant license port of the root is tested with the port of every remaining machine advertisement. When no candidate is found, the algorithm backtracks and replaces the already docked machine advertisement with another compatible candidate, and repeats the whole process yet again. When all compatible machine candidates have been considered and exhausted in the root's first port, the algorithm finally fails.

The worst case performance of the naive algorithm may be easily derived and expressed in terms of the number of expression evaluations performed. Given r requests (i.e., roots), n machine advertisements, no licenses, and the assumption that the machine port of each root is compatible (on average) with k of the n machines, the number of expression evaluations performed by the naive algorithm is $r(n + kn)$. Thus, the algorithm is $O(n^3)$ in the worst case, when both r and k are equal to n. Of course, the algorithm's worst case behavior is dependent on the workload and the gang tree topology imposed by the workload — the cubic polynomial is merely an artifact of our workload, which only marshals gangs of size three.

5.4 LR: Indexed In-Order Gangmatching

5.4.1 Motivation

The naive algorithm's exclusive reliance on the expression evaluation mechanism is its fundamental weakness. Since there is no bulk processing facility that can efficiently partition advertisements into "potential candidates" and "definite non-candidates," the naive algorithm is forced to detect these sets using the "record-at-a-time" expression evaluation mechanism. However, the classad indexing scheme presented in Chapter 4 is exactly such a bulk processing mechanism. It is therefore natural to augment the naive algorithm with the indexing scheme to obtain a far more efficient algorithm.

5.4.2 Algorithm Description

The LR algorithm is a straightforward extension of the naive algorithm. The basic processes of the algorithm, such as left-right progression, right-left backtracking and history set management, are left unchanged. However, the process of filling docks with candidate advertisements is preceded with an index probe, which results in considerably decreasing the number of candidates that must be considered in order to fill the port. The details of the algorithm are presented below.

As discussed in Chapter 4, the classad indexing scheme requires several preprocessing steps. The LR gangmatching algorithm commences by finding the external reference sets of all the offer and request advertisements, converting all offers to rectangles in context of the obtained external references, and finally indexing the offer rectangles. In addition, mappings between the offer rectangles and the advertisements and ports from which they were derived are constructed.

When presented with a root advertisement, the algorithm commences from the left-most port, like the naive algorithm, and proceeds towards the right-most port. However, when attempting to fill a port for the first time, the indexed gangmatching algorithm first converts the attributes and constraints of the port under consideration to a "window," which is used to probe the constructed indexes and efficiently identify all candidate match rectangles. The expressions used to construct the window query are first specialized to incorporate all present information, and therefore derive a specific query customized to current contents of the gang. For example, after the machine port has been filled, the subsequent index probe issued on the license port is specialized to find only those licenses that are compatible with the machine chosen and (of course) the job itself.

The result obtained from the query is associated with and stored in the advertisement's port, much like the history set. The mappings constructed during index creation are used to map matched rectangles to the candidate advertisements and ports. Thus, in addition to excluding candidates that are in the history set during fill and refill operations, only candidates that exist in the query result are considered. The query result associated with a port is cleared when backtracking from that port for the same reasons that the history set is — since the query result is derived from a window that is specialized to the predecessor candidates of the port, changing any of the predecessors requires invalidation of the query results.

5.4.3 Performance and Observations

The elapsed time performance of the LR gangmatching algorithm on the base workload is presented in Figure 37. As with the naive algorithm, the graph consists of two bands of curves corresponding to the 50% and 100% workloads, from which representative curves are presented in Figure 38. The first observation to be made is that although the shape of the performance curves of the LR algorithm is similar to the naive algorithm, LR is *far* more efficient. Whereas the naive algorithm marshals only 500 gangs in 800 seconds, LR matches 4000 gangs in less time. A comparison of the number of expression evaluations corresponding to the above elapsed time measurements as illustrated in Figure 39 shows that the accuracy of the indexing scheme relegates the use of expression evaluation to only confirm candidates, rather than identify them.

We also note that the number of index probes performed by the algorithm (Figure 40) is equivalent to the number of expression evaluations, showing that the indexing mechanism is extremely accurate in this workload — every expression evaluation merely confirmed the results of the query, requiring no additional matches to be tested. It is more convenient to analyze algorithm performance

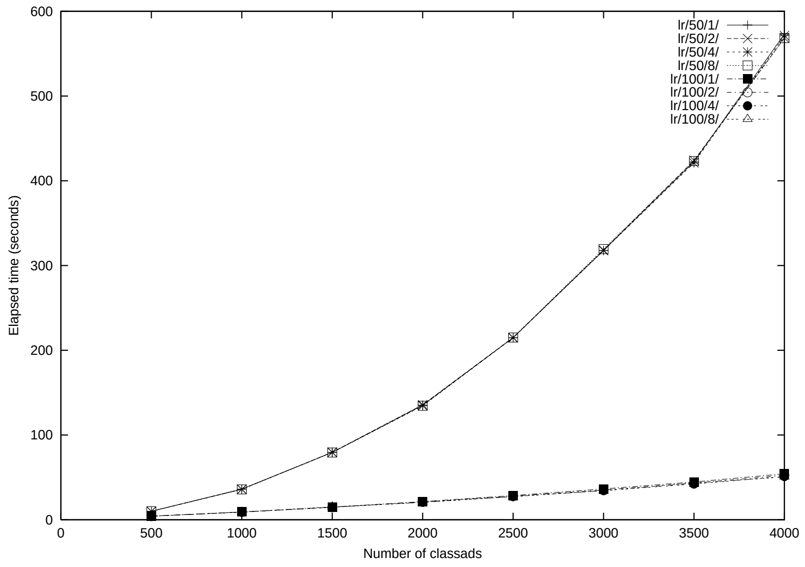

Figure 37: Elapsed time performance of the LR algorithm.

in terms of number of index probes, as this operation will continue to play a key role in subsequent algorithms. Further analysis of the algorithm's behavior is therefore performed with respect to the number of index probes.

While the indexed algorithm is clearly superior to the naive algorithm in many respects, is also inherits some of the naive algorithm's weaknesses. Specifically, the algorithm performs poorly when some resources are scarce or absent. The upper curve in Figure 40 illustrates the number of index probes performed for various experiment instances, on workloads that have half as many licenses as machines or jobs. When compared to to the lower curve in the same graph one notes that the inherited weakness of the algorithm in this scenario remains. However, in order to address the weaknesses of the algorithm (and preserve its strengths), it is first necessary to understand the algorithm's behavior in various

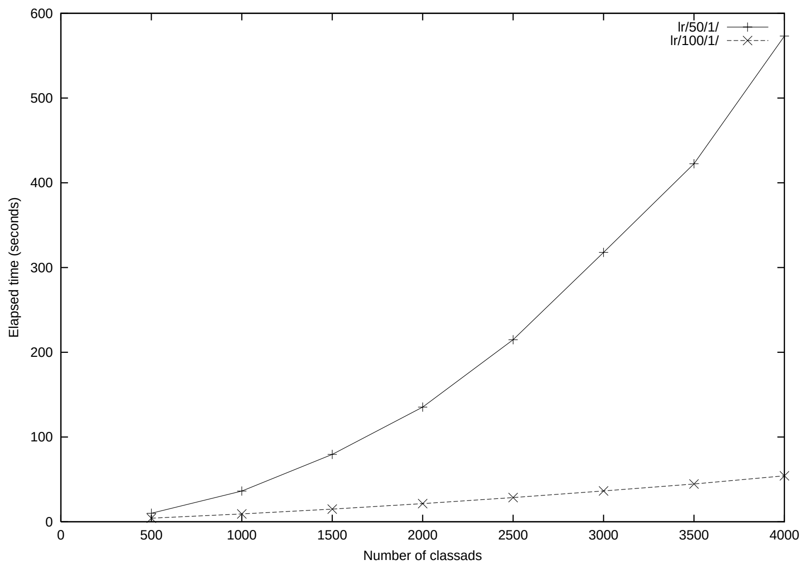

Figure 38: Representative curves of LR's elapsed time performance.

scenarios.

When both machine and license advertisements are abundant and widely compatible, the index probe on the machine port of the root will identify several candidate machines. Due to the abundance of licenses, choosing any one machine will result in the identification of a valid license when a probe is attempted on the vacant license port, resulting in a complete gang in just two index probes. Thus, the algorithm performs well in an environment with abundant resources, and this suspicion is borne out by the lower curves in Figures 38 and 40. The algorithm does not suffer when compatible machines are not present either — the initial index probe on the root's machine port will result in an empty query result, immediately resulting in match failure in just one probe.

The weakness of the algorithm is exposed when there are a large number

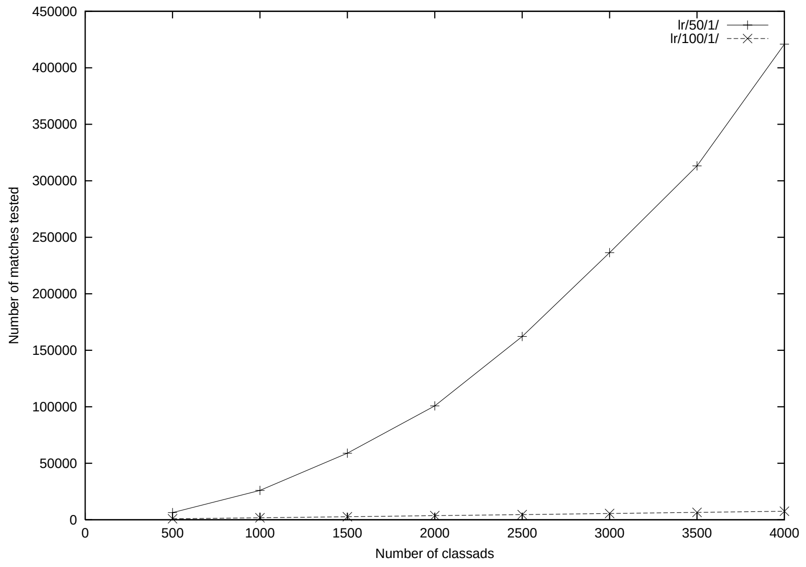

Figure 39: Representative number of matches tested by LR.

of compatible machines, but few (or worse, no) licenses. In this case, the first probe made on the root's machine port results in a large set of compatible candidates. An index probe is then issued from the root's license port for each candidate machine, but most of these probes are ineffectual because the few available licenses are only compatible with a small set of machines, which may not be included in the candidate machine set. Even worse, there may be no licenses available, in which case *all* probes issued from the root's license port are ineffectual.

Using the notation introduced in the previous section, assuming r roots, n machines, no licenses and an average machine compatibility of k, the worst case number of index probes issued is $n(1 + k)$, which in the worst case is $O(n^2)$. While this algorithm is clearly a substantial improvement over the naive

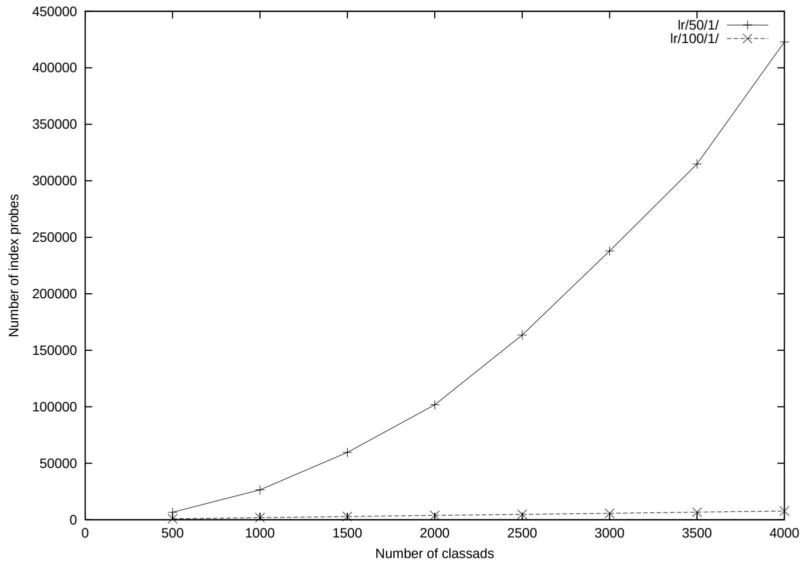

Figure 40: Representative number of index probes performed by LR.

algorithm, we can in fact do better.

5.5 DYN: Dynamic-Order Gangmatching

5.5.1 Motivation

In the particular license management example, the fixed left-to-right operation of the indexed gangmatching algorithm defines that the algorithm always pick a machine first and then find a compatible license for it. If no compatible licenses are found, another compatible machine is chosen, and the process repeated until either a compatible machine-license pair is found, or all compatible machines are exhausted without finding a suitable license. We have seen that this algorithm performs well if machines and licenses are abundant, or if compatible machines are scarce (irrespective of whether licenses are scarce or not). The algorithm performs poorly when machines are abundant, but licenses are scarce. The solution to the weakness of the algorithm, of course, is that when faced with abundant compatible machines and scarce licenses, the algorithm must begin by picking a license, and then proceed to find a machine compatible with the license.

While the insight is easily stated, it is not obvious if this strategy is practicable. First, the inherent left-right bias of the gangmatching model must be overcome. Second, a mechanism that directs the algorithm to either proceed left-right or right-left must be developed. The details of these problems and solutions to them are discussed below.

Overcoming Left-Right Bias

The"declare-before-use" semantics of port labels introduces a natural left-right bias to the gangmatching model, which, as we have identified, must be overcome. However, we must first identify exactly what it means to "overcome the left-right bias."

Consider the normal left-right matching situation with a docked machine candidate and vacant license port (Figure 41). The query window that needs to be created for the license port can be created by the straightforward process of specializing the port's attributes and constraints. Specifically, the information required to find a compatible license is available because the value of the `HostID`

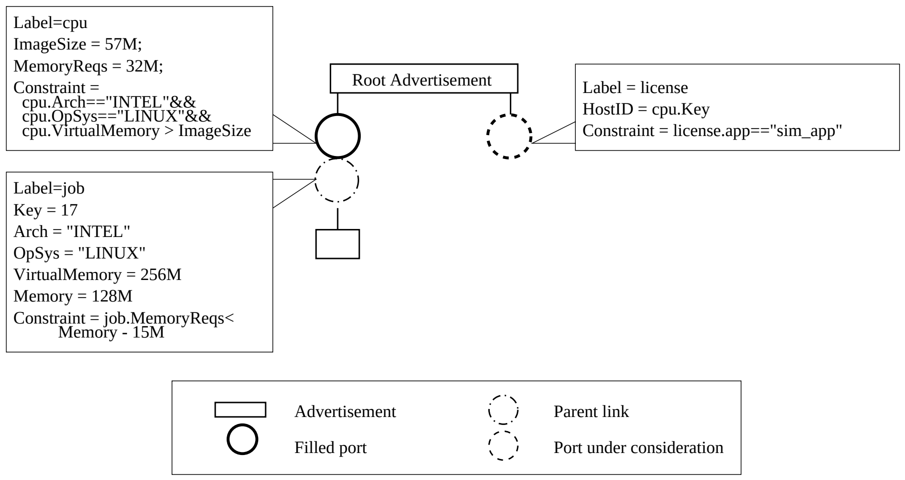

Figure 41: Left-right gangmatching

attribute in the license port is known.

In contrast, consider Figure 42 which illustrates the dual scenario of a docked license candidate and a vacant machine port. The query we wish to generate on the machine port is `cpu.Arch=="INTEL" && cpu.OpSys=="LINUX" && cpu.VirtualMemory>ImageSize && cpu.Key >= 10 && cpu.Key < 20`. However, the elements required to generate this query are distributed at several locations: the *machine port* in the root contains the attributes and constraints that will determine a large part of the query, the docked *license candidate* places the constraint on the `HostID` attribute which needs to be translated to a constraint on the `Key` attribute of the machine, and the *license port* of the root contains the information necessary to perform this translation in the `HostID` attribute expression.

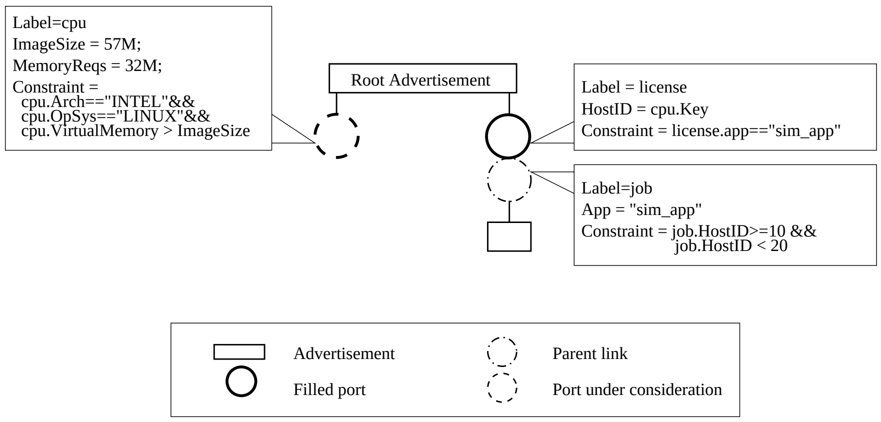

Figure 42: Right-left gangmatching

The basic challenge of performing right-left gangmatching is *constraint shunting*, which is comprised of the following three sub-problems: *identifying* necessary information, *translating* information from source locations to be meaningful in target locations, and *collating* information into a single window query.

1. **Identification.** The identification step ensures that all relevant information sources are used. Importantly, identification also ensures that irrelevant information sources are ignored. For example, if the license port of a root does not access any information from the machine port, the two ports are independent, and the complicated process of constraint shunting would not be necessary. The first step of identification is therefore discovering the dependency relations between the ports of an advertisement. Inter-port dependencies may be detected by a free variable analysis of all the expressions present in a port: free variables corresponding to port names determine inter-port dependencies. For example, the license ports

of request advertisements are dependent on the machine port due to the presence of the `HostID` attribute, which has a free reference to the port label `cpu`. Note that these dependence relations are acyclic, since only "backward" dependencies are possible given declare-before-use semantics. Thus, when filling the machine port, it is necessary to consider the contents of the license port, which is identified by inverting the dependence relation.

When attempting to fill a port, it is in general necessary to consider information from all ports that are transitively dependent on it. For example, if port C is dependent on port B, which is in turn dependent on A, it is possible for constraints defined by the entity docked at C to translate to a constraint on A. We must consider the contents and docked candidates (if any) at both B and C when filling A; i.e., all ports in the transitive closure of the inverse dependent relation.

To summarize the identification step, dependencies between an advertisement's ports are determined through a free variable analysis. The transitive closure of the reverse dependency relation is constructed and consulted when filling ports out of order. The information of interest from a reverse dependent port exists both in the reverse dependent port itself, and the candidate docked at that port (if present).

2. **Translation.** Translation is the process of transcribing information from

one port so that it is usable in another. It is both interesting and informative to view translation as a function inversion process. The "translation attribute" `HostID` in the license port may be viewed as a function of one argument, `cpu.Key`. In this case, the function is simply the identity function. Constraints issued by the license advertisement, such as `job.HostID >= 10 && job.HostID < 20`, may be viewed as applications of the function `HostID` which need to be inverted so that they may be applied on the machine port. Since the function is only the identity function, the inversion results in `cpu.Key >= 10 && cpu.Key < 20`.

However, if the attribute instead was `HostID = cpu.Key + 10`, a more able translation mechanism can still proceed. In this case the license constraint `job.HostID >= 20 && job.HostID < 30` would be translated to `cpu.Key >= 10 && cpu.Key < 20`. In general, the translation step may be carried out as long as the "translation attribute," when viewed as a function, is invertible. (If the translation attribute is not invertible, the constraint shunting process cannot be performed.) Of course, if the translation is being performed in a port that is several hops away in the inverse dependence relation, the inversion functions for each hop must be composed to determine the final translation.

3. **Collation.** Information obtained from the various dependent ports must finally be collated to form a single query. Essentially, each item of information being collated is a query fragment, which, as we have noted in

Section 4.2.2, may be a set of rectangles. Collation of query fragments is therefore slightly involved: since the constraints of all already filled ports must be simultaneously satisfied, the final query is a conjunction of these query fragments, which must itself be expressed as a set of rectangles.[1] Alternatively, the index mechanism can be trivially extended to accept and process logical combinations of independent query windows.

In summary, the collation step involves the conversion of several query fragments into a single query window (i.e., set of rectangles).

Determining Match Strategy

The motivation for performing right-left gangmatching is based on the intuition that scarce resources should be incorporated into the gang first. Therefore, the goal of the strategy mechanism is to detect which of the ports require scarcer resources. Specifically, the mechanism must detect the number of resource candidates for each port, and direct the gangmatching algorithm to fill ports in increasing order of candidate availability.

This problem is solved by the indexing mechanism. Before attempting to fill ports of an advertisement, the algorithm first performs a preliminary pass over the ports, issuing an *initial index probe* for each port. Initial probes are generated using the identical mechanisms used to issue regular probes during

[1] The collation step is analogous to converting the conjunction of several predicates in DNF to DNF, and therefore (in general) results in a large number of conjuncts in the resulting DNF expression. Analogously, collation may result in a query window with a large number of rectangles.

the fill and refill operations, but some of the information necessary to completely define the probe may be absent. For example, when issuing the initial probe on the license port, the `HostID` attribute would be missing from the query window. Tolerance of "incomplete" queries is however one of the basic functionalities of our indexing model, which has been designed to operate in a semi-structured environment. Since the result of the index probe is a guaranteed superset of the final answer, the cardinality of the probe result may be used as an upper-bound estimate of the number of resources compatible with the port. Thus, the port fill order is determined by sorting ports in ascending order of initial query result cardinalities.

5.5.2 Algorithm Description

The operation of the indexed dynamic order algorithm is similar to, but more involved than that of the indexed in-order algorithm. The pre-processing phase of the algorithm includes all the steps of the in-order algorithm: external reference determination, rectangle conversion, index creation, and rectangle-to-port and rectangle-to-classad map construction. However, the pre-processing phase of the dynamic algorithm also includes inter-port dependence analysis (performed through a free variable identification algorithm) and transitive closure of the reverse dependence relation. These relations are maintained on per advertisement.

The main operation of the algorithm is also very similar to, but more involved than, the in-order algorithm. Before filling the ports of root (or non-root) advertisements, but after identifying the parent link of a non-root advertisement, the dynamic algorithm first performs an initial query (or index probe) for each non-parent link port of the advertisement in question. The ports are then sorted in increasing order of the query result cardinalities. The algorithm then performs fill and backtrack ports in the sorted port order. In a sense, the algorithm still operates in a left-right formalism, except that the orientation of the algorithm is abstracted to a logical rather than a physical ordering.

When a fill or refill operation is attempted on a port, the translation and collation mechanisms discussed previously are activated to generate the necessary query window. The history set and query result management strategies are unchanged from the in-order algorithm.

Finally, the expression evaluation mechanism is invoked when candidates are incorporated into the gang to ensure that the candidates identified by the indexing mechanism do in fact satisfy the constraints of the advertisement. The constraints of all ports (and possible candidates docked at those port) in the transitively closed reverse dependent relation must be evaluated.

It is important to note that applying different algorithms to the same workload can result not only on different gangs being marshaled for the same root, but also different numbers of successfully marshaled gangs. The first effect is easily understood given the situation when machines near the beginning in the machine port query result are compatible with licenses near the end of the license port query result. The second effect is a consequence of the first because matching resource advertisements to gangs has associated opportunity costs. If request A is compatible with two offers X and Y, whereas request B is only compatible with X, the advertisement matched to A clearly alters the number of matches possible. Consequently, the number of matches created by the dynamic and in-order algorithms may vary. Nevertheless, the number of matches never differs by more than 5% of the workload size in the tests presented here.

5.5.3 Performance and Observations

The performance of the DYN algorithm on the base workload is contrasted with the LR algorithm in Figures 43 and 44 which illustrate the 100% and 50% scenarios respectively. Representative curves from these graphs are illustrated in Figure 45 and 46. We note that on the 100% license workloads, the dynamic algorithm issues more queries than the in-order algorithm. Furthermore, the number of additional queries issues is within a constant factor of the in-order algorithm. Since licenses are abundant in this workload, the algorithm almost invariably performs left-right gangmatching after performing the initial queries. The difference between the performance of the two algorithms is therefore due to initial query overhead.

The performance of the algorithm under the 50% license workloads (Figure 46) shows a dramatic improvement over the indexed in-order algorithm,

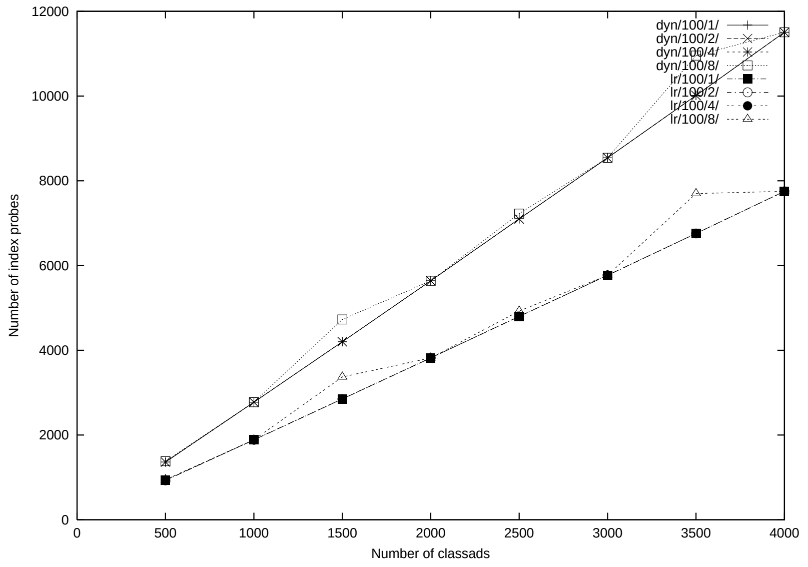

Figure 43: Number of probes issued by DYN and LR on the 100% workload.

confirming our intuitions. The improvement is due to the fact that the algorithm switches between left-right and right-left strategies as appropriate. Thus, the large number of ineffectual index probes issued by the in-order algorithm are avoided.

Thus, we see that the dynamic order algorithm performs similarly (within a small factor), or dramatically outperforms all the other algorithms considered till now by using a simple heuristic to determine port fill order. As with our previous experiences, we wish to understand the strengths and weaknesses of the dynamic order algorithm, so that the reasons for the performance advantage may be understood.

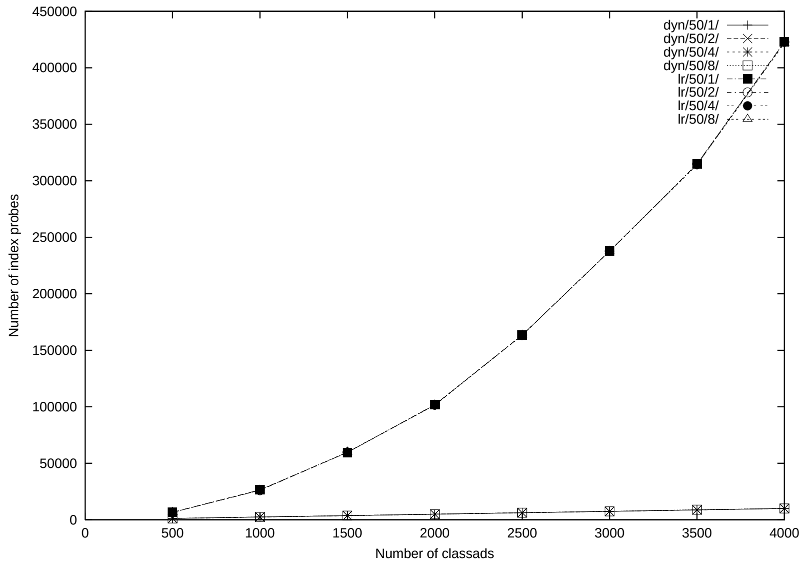

Figure 44: Number of probes issued by DYN and LR on the 50% workload.

5.6 The Dynamic Algorithm's Advantage

The dynamic order algorithm is an adaptive algorithm that modifies its behavior depending on the nature of the workload. The intuition that drives dynamic order matchmaking is similar to that which motivates the use of the smaller relation as the outer relation in a nested-loops join. While in databases the benefit is realized as fewer I/O operations, the benefit in gangmatching is fewer index probes.

While it is natural (and essentially correct) to attribute all the performance gains to the agility of the algorithm, the process of dynamically determining fill order actually has *two* beneficial consequences. First, as discussed previously, scarcer resources are incorporated first, reducing the number of required index probes. Second, the algorithm immediately aborts if desired resources are

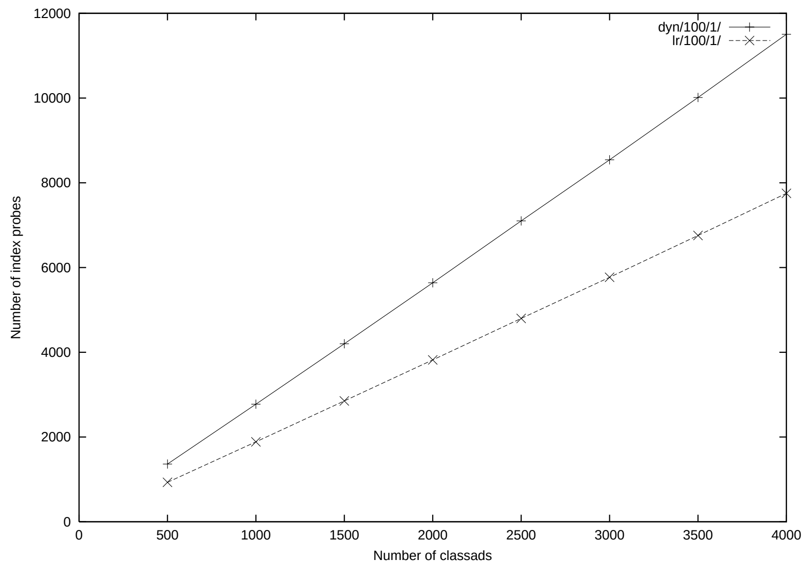

Figure 45: Representative number of probes issued by DYN and LR on the 100% workload.

unavailable.

The latter property is a consequence of the port order determination step. Specifically, sorting the port list on initial query result size naturally places ports with cardinality zero at the head of the list. However, the presence of ports with zero result cardinality implies the impossibility of a successful match. Futile match attempts are therefore immediately detected and aborted.

It is instructive to differentiate and isolate these effects to better understand the reasons for the indicated performance gains. While we certainly do expect performance contributions from both effects, our goal is to understand the potential magnitude of these contributions, and identify the kinds of workloads in which these separate effects succeed or fail in comparison to the full dynamic algorithm. To this end we introduce three variants of previous algorithms.

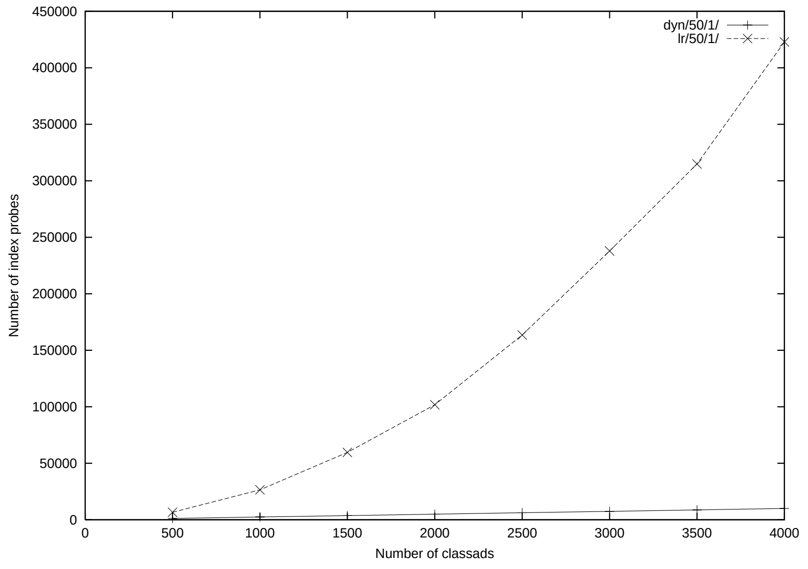

Figure 46: Representative number of probes issued by DYN and LR on the 50% workload.

Fixed right-left algorithm (RL) As the name implies, the algorithm always starts from the right-most port and proceeds to the left-most port.

Left-right algorithm with Checks (LRC) A modification of the indexed in-order algorithm (LR), LRC first performs initial queries on all ports to determine if any of them do not have candidates. If any such ports are detected, the algorithm immediately aborts. Otherwise, the algorithm proceeds with the conventional LR algorithm.

Right-left with Checks (RLC) An analogous modification of the RL algorithm, RLC performs initial queries to check candidate availability. Again, the algorithm proceeds with RL if candidates are available, and aborts immediately if not.

5.6.1 The RL Algorithm

The performance of the RL algorithm under the base workload is presented in Figure 47. As anticipated, the RL algorithm's performance is dual to the per-

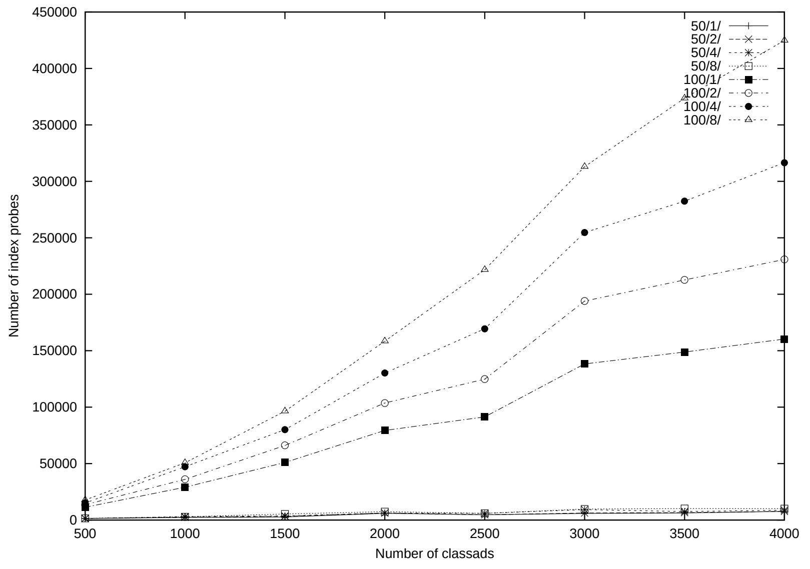

Figure 47: Number of probes issued by the RL algorithm.

formance of the LR algorithm, performing well when LR does poorly and *vice versa*. Specifically, the algorithm performs well in the 50% workloads, when licenses are scarce. However, the algorithm's performance in the 100% workloads is far worse than the performance of LR under the 50% workloads. This is primarily due to the fact that licenses do not have any attributes to differentiate them — the query result of the root's license port contains *every* license. Thus the cardinality of the first query is always substantially larger in the RL algorithm than the LR algorithm, amplifying the weakness of the algorithm.

Nevertheless, the performance of the algorithm under the 50% workloads confirms that some of the dynamic algorithm's advantage is due to the ability of the algorithm to match right-left when required.

Due to the complementary performance of the LR and RL algorithms, and the simultaneous and consistent success of the dynamic algorithm in these workloads, we can conclude that the agility of the dynamic algorithm is indeed an important ingredient for its success.

5.6.2 The LRC and RLC Algorithms

The performance of the LRC algorithm is presented in Figure 48, which seems to indicate that the performance of this algorithms under the base workload is essentially the same as the dynamic algorithm. In fact, one may be tempted to

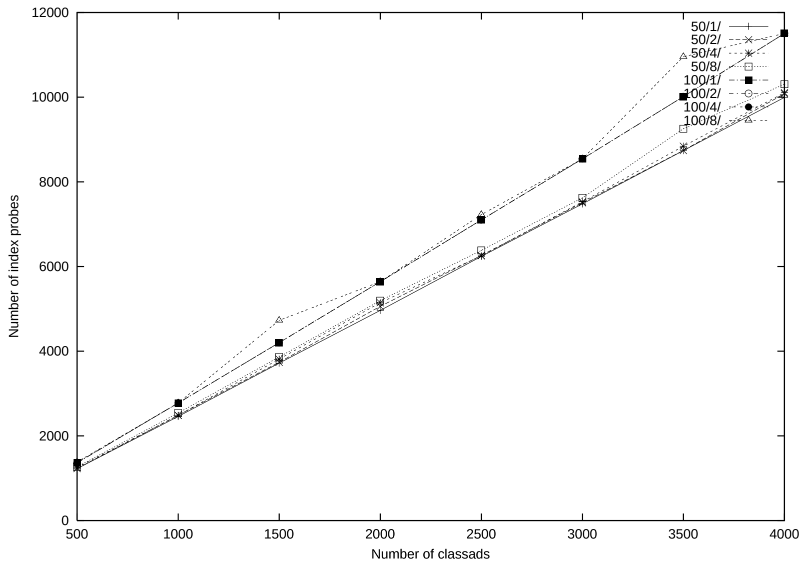

Figure 48: Number of probes issued by the LRC algorithm.

claim that the complexity of the dynamic algorithm is unwarranted given the performance competitiveness of the substantially simpler LRC algorithms. Intuitively, the LRC algorithm only provides a performance advantage if resources are completely exhausted. If resources are merely scarce (but not exhausted), the LRC and RLC algorithms should perform like the LR and RL algorithms respectively. Thus there must be a "performance cliff" between no resources and one resource for this algorithm that is not being provoked by the workload.

These intuitions are correct, as indicated by the performance of RLC on the same base workload (Figure 49). The two sharp discontinuities show the

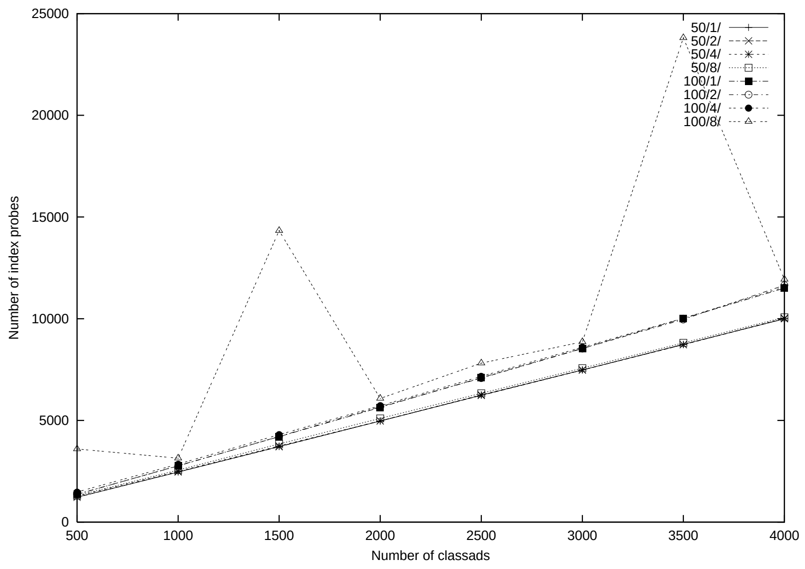

Figure 49: Number of probes issued by the RLC algorithm.

presence of the performance cliffs in these experiments.

To further demonstrate this effect, the resource advertisements of the base workload were augmented with eight additional advertisements: one advertisement representing a license whose constraints on `HostID` can never be satisfied by any machine, and seven advertisements representing machines of various architectures, which do not satisfy the constraints of any license. It is important to note that these "resource dud" advertisements have been engineered so that any query in the workload, be it on licenses or machines, always returns exactly one of these advertisements. Thus, the workload ensures that no queries are empty.

Figures 50 and 51 show that our intuitions are indeed correct — the behavior of LRC and RLC under the engineered workload are essentially identical to the LR and RL algorithms. The performance of the dynamic algorithm on the same workload is shown in Figure 52, showing that the dynamic algorithm's performance advantage is not solely dependent on the early detection of futile matches.

These results are hardly surprising. However, our goal was not to demonstrate that there exists workloads that can provoke pathological algorithm behavior. Instead, we want to emphasize the almost negligible change made to the base workload which results in this behavior. Thus, although LRC and RLC are occasionally efficient, their performance is *highly* sensitive to workload composition. In practice, the presence of Incompatible resources is almost guaranteed, rendering these algorithms infeasible.

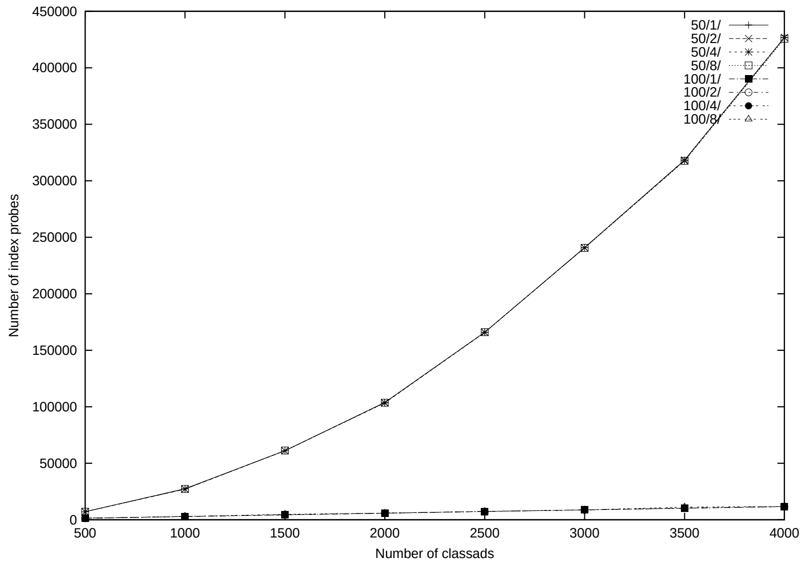

Figure 50: Number of probes issued by the LRC algorithm on the specialized workload.

5.7 Heuristic Failure

In light of the previous performance study one may conclude that the dynamic-order gangmatching algorithm is superior to the static in-order gangmatching algorithm. The heuristic of filling ports having fewer candidates first is intuitive and correct (under our current assumptions), and the algorithm performs well under various workloads.

Nevertheless, it is reasonable to ask: when is the dynamic algorithm's heuristic wrong? In other words, when is it incorrect to choose to fill the port with the smallest number of candidates first? If nothing more about the workload is known, the dynamic algorithm's heuristic is *always* better in the sense that it minimizes the worst case number of index probes. However, if the workload

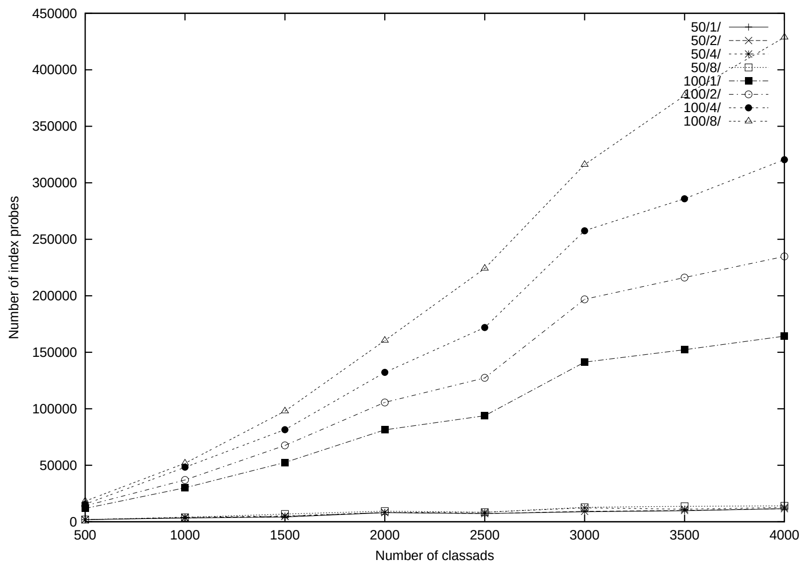

Figure 51: Number of probes issued by the RLC algorithm on the specialized workload.

has certain special properties, it may be more efficient to ignore the heuristic.

Consider a situation in which there are a 500 compatible machines and 1000 identical licenses, all of which are valid only on the last 100 machines. In other words, 80% of the machines are not compatible with any license. The dynamic algorithm's heuristic is defeated in this case: the heuristic decides to fill the machine port first, and like the LR algorithm, results in a large number of ineffectual index probes. In contrast, it is clear that filling the license port first would result in fewer index probes for the first 500 matches, after which, of course, performance would sharply degrade.

We can however define an algorithm that performs as well as RL for the first 500 matches without the subsequent performance degradation on the basis of the following insight: The identical nature of the 1000 licenses should drive

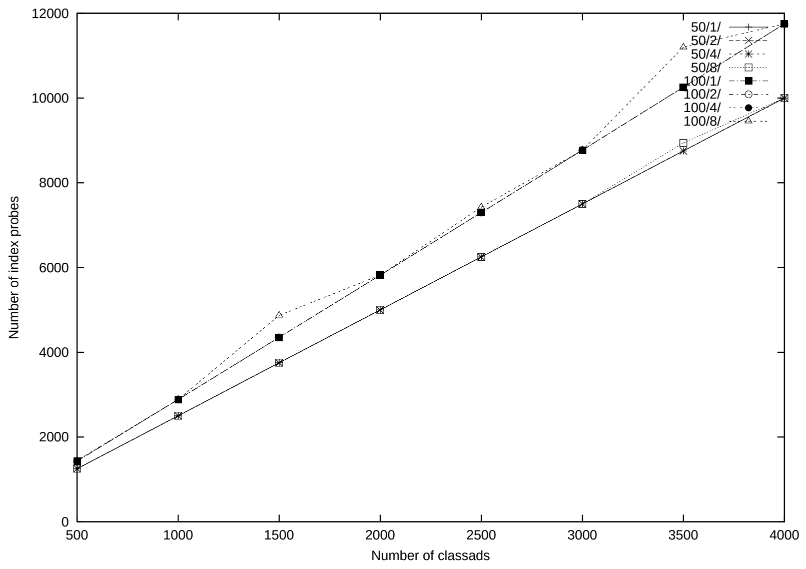

Figure 52: Number of probes issued by the dynamic algorithm on the specialized workload.

us to consider all the 1000 replicas as a single license for algorithmic purposes. In other words, the heuristic should not base its decision on the number of rectangles intersecting the query window, but the number of *unique* rectangles intersecting the query window. Thus, we propose a variant of the dynamic algorithm formulated on unique rectangles.

5.8 DYNS: DYN with Summarization

5.8.1 Algorithm Description

We now augment the dynamic algorithm with a simple summarization algorithm that places rectangles into "buckets" such that each bucket only contains identical rectangles. The summarization algorithm works by creating a "signature" for each rectangle by concatenating representations of the rectangle's imported and exported intervals. If a particular interval is absent, a special substring denoting the interval's absence is inserted instead of the interval. Representing deviant components is slightly more complex. We conservatively assume that no two deviant components are alike, and therefore include unique substrings in the rectangle's signature thereby ensuring that the signature is itself unique.

The string signatures of rectangles are used to hash into rectangle buckets in which the rectangles are stored, and therefore only identical rectangles are placed in the same bucket. Only single rectangle "representatives" from each bucket are indexed, thereby ensuring that the index only contains unique rectangles.

The dynamic algorithm then operates as before, with two minor additions. First, a multi-stage retrieval is performed after a successful index probe: the first stage translates the index result to a specific bucket, and the second stage retrieves a specific rectangle from the bucket. (The final mapping from the rectangle to the advertisement remains unchanged, and proceeds as before.)

The second change is that rectangles are purged from the index only when the rectangle's corresponding bucket is empty.

5.8.2 Performance and Observations

The performance of the dynamic algorithm with summarization (DYNS) on the base workload is compared with that of the regular dynamic algorithm (DYN) in Figures 53 and 54. While the performance of the two algorithms is essentially

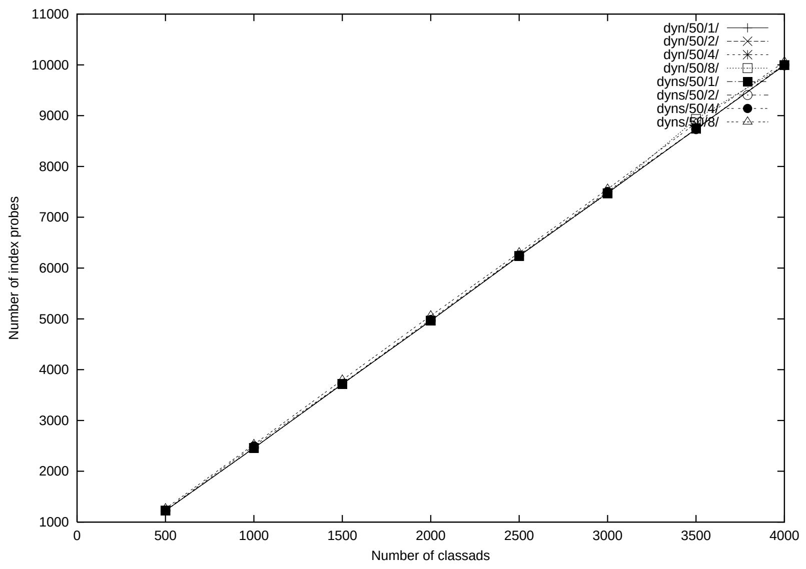

Figure 53: Number of probes issued by DYNS and DYN in the 50% workload

identical for the 50% workloads, we notice that DYNS issues more probes than the DYN algorithm in the 100% workloads. Furthermore, the number of probes issued increases with the license selectivity index of the experiment.

The reason for this unintuitive result is that the DYNS algorithm effectively groups identical licenses for matchmaking by placing them in buckets. The

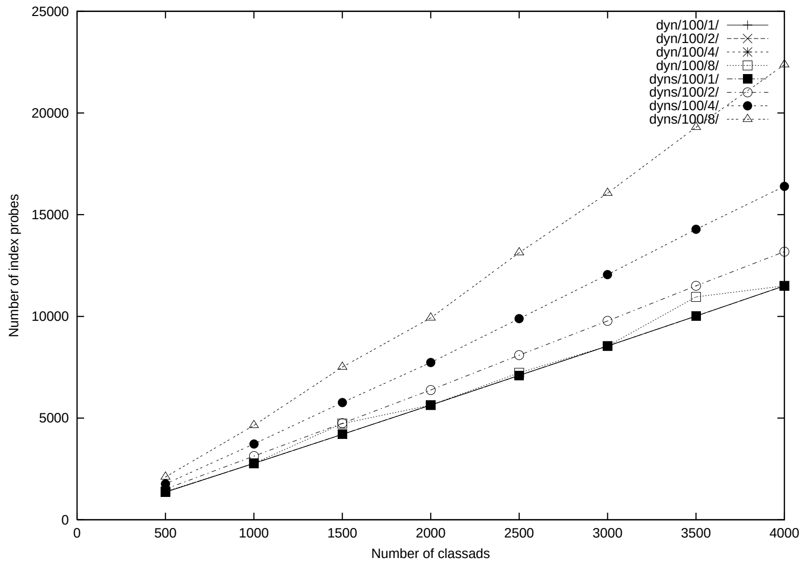

Figure 54: Number of probes issued by DYNS and DYN in the 100% workload

algorithm therefore sequentially uses and exhausts identical licenses from each bucket. However, not all licenses are consumed in the 100% workload, and license buckets therefore contain a small number of "license dregs" that are left unused. Since the algorithm proceeds sequentially through the result set for the license port, these dregs are continually chosen as the license candidate for each new root, resulting in a large number of ineffectual probes on the machine port. Furthermore, since these probes are issued for each license bucket, an increased number of distinct licenses (i.e., larger license selectivity indexes) results in a correspondingly large number of index probes.

5.8.3 DYNSR: DYNS with Random Start

The ineffectual probes performed by DYNS can be avoided if sequential consumption of license buckets is prevented. With this goal in mind, we add an additional modification to the DYNS algorithm — the "random start" — to yield the DYNSR algorithm. The DYNSR algorithm differs from the DYNS algorithm only in the manner in which it traverses an index probe's result set. While the DYNS algorithm always begins at the top of the set and proceeds sequentially to the end, the DYNSR algorithm begins at a random starting position that is chosen uniformly over the size of the result set, and then proceeds sequentially, "wrapping over" as necessary to traverse the entire set. Thus, DYNSR avoids the sequential consumption problem by being non-deterministic.

The comparative performance of DYNS and DYNSR on the 100% workload is presented in Figure 55. The upper four curves in the graph correspond to the performance of the DYNS algorithm as previously seen, and the lower band of curves illustrates the performance of the DYNSR algorithm. The graph shows that as expected, the DYNSR algorithm issues far fewer probes than the DYNS algorithm.

Unfortunately, the random-start strategy introduces a substantial deficiency — DYNSR makes 10%–15% fewer matches than either DYN or DYNS on identical workloads. Figure 56 graphs the number of matches made by the DYN, DYNS and DYNSR algorithms on a representative workload. While the curves for DYN, DYNS and DYNSR are co-incident for the 50% workload, DYNSR

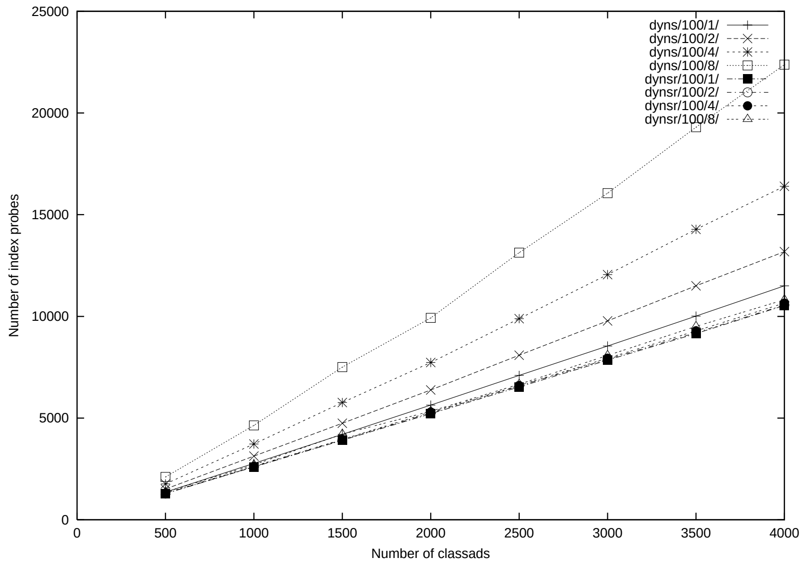

Figure 55: Number of probes issued by DYNSR and DYNS in the 100% workload

makes fewer matches in the 100% workload than the other algorithms. The reason for this behavior is subtle. Since the fixed probe result traversal performed by DYNS is equivalent to a "first fit" strategy, matches on the machine port of roots get successively "tighter" as the algorithm is run for several roots. By this we mean that successively matched machine advertisements include just enough resources to be compatible with the job. In contrast, the random-start strategy is an "any fit" algorithm that may match machines that have resources far in excess than those required to satisfy the job. Thus, the random-start strategy has a higher "opportunity cost" in the sense that the matches made with this strategy in general prevent more matches from occurring by depriving a larger number of jobs of their potential matches.

To verify this hypothesis, we constructed a purely bilateral workload (i.e., no

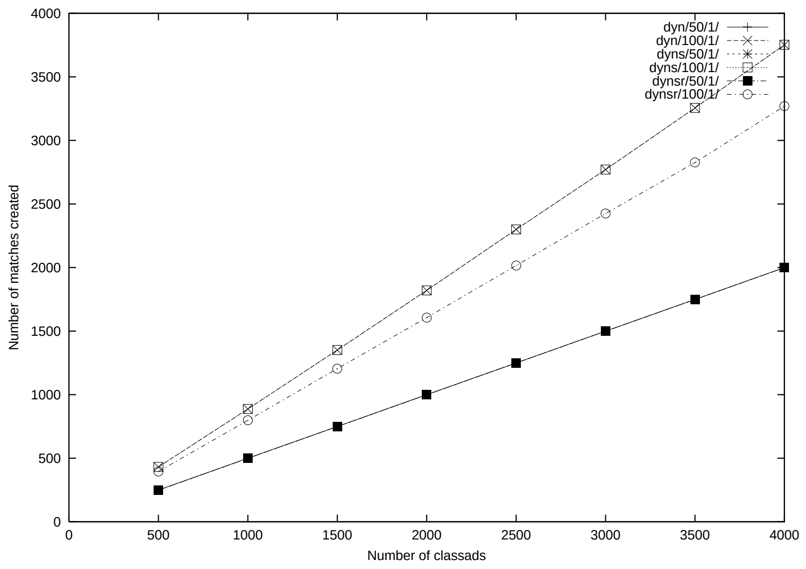

Figure 56: Matches created by DYN, DYNS and DYNSR in a 100% workload

licenses) in which we only modeled the architecture and operating system parameters of requests and offers. Each request was assigned a random uniformly distributed “tag value” between 0 and 10, and each offer included an additional constraint that required a tag that was above a certain threshold. The threshold for each offer was obtained by choosing a random value uniformly distributed between 0 and k, where k is a parameter of the workload that models machine discrimination. Thus, in general, machines modeled in workloads with higher discrimination values were compatible with fewer jobs. Workloads with 4000 requests and offers were generated for $k = 1, 4, 6, 8, 10$.

The experiment proceeded as follows. First, the compatibility set of each machine in the workload was determined by finding the set of jobs compatible with it. Algorithms with and without the random-start strategy were then run on the workloads, and the total opportunity costs of these algorithms were measured by adding the opportunity cost of each match as each algorithm progressed. The opportunity cost of each match was determined by subtracting the set of already matched jobs from the compatibility set of the matched machine, and taking the cardinality of the resulting set.

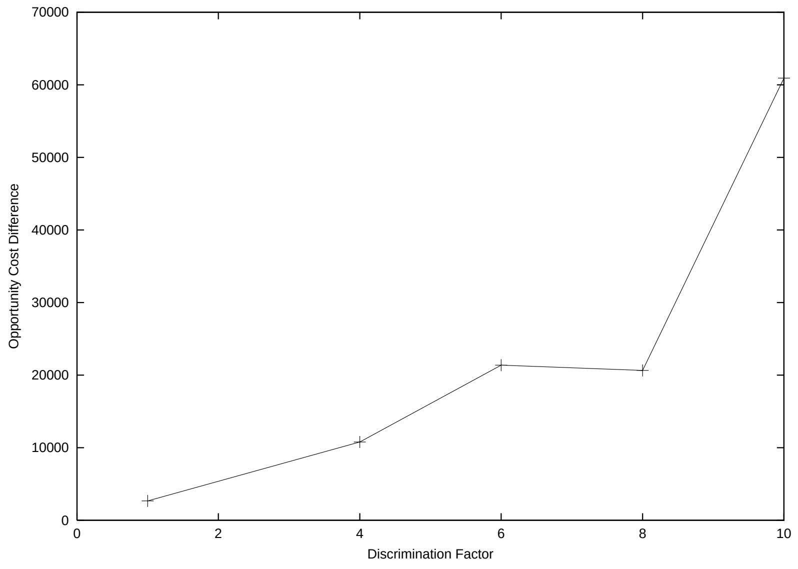

Figure 57: Opportunity cost difference imposed by random start

Figure 57 plots the *difference* between the opportunity costs of the algorithm that uses the random-start strategy against the algorithm that does not, for various discrimination values. When the discrimination factor of machines is low (i.e., for low values of k), the offers are essentially equivalent, so there are no large opportunity costs incurred when the random-start strategy is employed. However, as the discrimination factor increases, we see that the random-start strategy incurs far larger opportunity costs, explaining the behavior of the

DYNSR algorithm.

5.9 Summary of Gangmatching Algorithms

We have described several gangmatching algorithms in this chapter. The naive gangmatching algorithm uses the classad expression evaluation mechanism to identify and confirm matches. The algorithm introduces elements that are used in all subsequent algorithms such as backtracking, history set management, parent link identification and recursive marshaling. The naive algorithm performs poorly due to the relative inefficiency of the expression evaluation mechanism, and the fixed left-to-right strategy used to marshal gangs.

Based on these observations we defined LR, an algorithm that replaces the expression evaluation mechanism with an index probe to identify compatible candidates. LR is far more efficient than the naive algorithm, but inherits the naive algorithm's weakness when operating in workloads composed of scarce or absent resources. We therefore formulated a dynamic algorithm (DYN) that uses the index mechanism to not only identify compatible candidates, but also to dynamically determine the port fill order. DYN uses a constraint shunting mechanism to enable right-to-left port filling. The dynamic algorithm minimizes the worst-case number of index probes by first filling the port that has the fewest number of match candidates.

DYN's heuristic is optimal (in the worst-case) if nothing more about the workload is known. However, the heuristic fails when workloads are composed of large numbers of similar or identical advertisements. To rectify this problem, we formulated DYNS, a variant of DYN that summarizes advertisement rectangles by collecting identical rectangles in buckets. DYNS makes strategy decisions using the number of unique rectangles as the criterion, instead of the raw number of rectangles.

Unfortunately, DYNS issues more queries than DYN on the base workload due to the grouping of licenses in buckets, leading to sequential consumption effects. Some license dregs left in each bucket are not compatible with any machine in the workload, and thus result in a large number of ineffectual index probes. The DYNSR algorithm attempts to rectify this problem introducing an element of non-determinism when traversing index probe results. While DYNSR eliminates the sequential consumption problem of DYNS, it introduces a new one by its inability to create as many matches as either DYN or DYNS. This deficiency is due to the fact that DYNSR is essentially an "any-fit" algorithm while DYNS and DYN are "first-fit" algorithms. We have shown that the "random-start" strategy employed by DYNSR incurs larger match opportunity costs, and therefore results in fewer matches.

The dynamic algorithms are in general more efficient than their static counterparts, but each algorithm variant (DYN, DYNS, DYNSR) exhibits distinct strengths and weaknesses. The relative superiority of these algorithms is therefore highly sensitive to workload composition.

The trade-offs imposed by the algorithm variants largely exist due to the lax match semantics of our current preference-free gangmatching algorithms. The lack of preferences introduces a degree of freedom that will not exist in preference-aware schemes. Unconstrained by other external factors, these algorithms make different choices, leading to divergent results. If the choices available to these algorithms are narrowed by the semantics of the model, the effects of the above trade-offs will be greatly diminished. Nevertheless, the effects of these trade-offs cannot be completely eliminated. It would be interesting to see if algorithms without these shortcomings can be formulated.

Chapter 6

Conclusions and Future Directions

The distributed computing community has only recently realized the enormous capacity and potential of distributively owned resource environments. The increasing prevalence of computational grids and distributed resources in sophisticated federated environments will place high demands on the capabilities of distributed resource management systems. We believe that matchmaking frameworks have the potential to play a central role in these environments. In this chapter we summarize our contributions and identify topics for further research in the matchmaking area.

6.1 Conclusions and Contributions

Conventional resource management systems are very efficient at managing static and dedicated resources for high-performance computing, but cannot handle the complexity and dynamism of distributively owned high-throughput computing environments. In this dissertation we have discussed how matchmaking systems overcome these deficiencies by using an opportunistic scheduling model. We have identified the architecture, components and protocols that comprise a matchmaking system and have validated many key issues through practical design and implementation.

Our contributions to the field of resource management systems include the classified advertisements language, a powerful and flexible language that may be used as the language substrate of distributed frameworks. Classads enable the specification of many interesting and useful resource and customer policies facilitating the operation of market-like resource federations.

We have developed a complete indexing solution to the classad data model. Our indexing solution efficiently identifies compatible advertisements by indexing both the constraints and attributes of classads. The relative efficiency of the indexing mechanism and its toleration of semi-structured information enables it to be used to identify compatible advertisements *en masse*, and also as an exploratory aide in making strategic decisions.

Multilateral matchmaking is a substantial contribution of this research effort. We have developed a symmetric, flexible and expressive declarative model for aggregating an arbitrary number of advertisements in a single match operation. The resulting gangmatching mechanism is strictly more powerful than the previous bilateral matchmaking framework, and can solve many real-world problems such as license management. We have also defined formal mechanisms to express administrative policy through the root identification, root ordering and docking vector mechanisms, extending the functionality of previous matchmaking frameworks.

Finally, we have developed many techniques to solve the gangmatching problem. Several gangmatching algorithms have been developed, and their behaviors have been studied in various workloads. By successively addressing the weaknesses of these algorithms while maintaining their strengths, we have created a dynamic algorithm that adapts its match strategy to minimize the cost of marshaling a gang. The algorithm uses several interesting techniques such as exploratory index probes, expression specialization, constraint shunting and classad summarization to efficiently identify compatible gangs of advertisements. These techniques are useful for classad management both in and outside the matchmaking context, and therefore form the basis of a rich set of classad management tools.

6.2 Future Directions

Resource management through matchmaking is a relatively new area, and is therefore rich with many interesting subproblems that require further study. We identify topics for future matchmaking research in general, and gangmatching in particular.

6.2.1 General Matchmaking

Accounting

The emphasis of this body of work has been on matchmaking infrastructure. Thus, we have developed a framework of components, mechanisms and protocols to enable the construction of robust matchmaking systems. However, we have not addressed many of the policy issues raised by such frameworks. An important topic of investigation is accounting for matchmaking systems. Issues of interest include:

1. How should resource usage be measured in heterogenous environments?
2. How should past usage affect current usage when resources have constraints and preferences? What is a "fair" allocation in these environments?
3. Should resource owners be compensated for services rendered? If so, how?
4. How can different resource management environments create resource sharing agreements?

Advertisement Replacement

Matchmaking currently treats each advertisement as an atomic entity: the resource described in an ad is considered to be completely consumed when the ad is matched. In practice, this is often not the case. For example, a symmetric multi-processing computer can usually accommodate several customers simultaneously before the resource is deemed to have been consumed. We currently rely on the resource agent to identify the consumed portion of the resource when matched, and re-advertise the remaining portion. However, an automatic mechanism is required to replace a classad with its residual after matching.

Shared Resource Environments

Related to the above problem, the current matchmaking framework treats each match as an independent activity. However, shared resources such as network bandwidth may prevent subsequent matches from occurring due to over-subscription of the network caused by earlier matches. The problem, of course, is the implicit participation and yet lack of representation of network resources in the matchmaking process. The difficulty of representing network resources in the matchmaking process is due to the necessity of "side-affecting" network bandwidth classads appropriately as and when matches are made.

Scalability and Reliability

Our current matchmaking framework can handle resource management environments composed of thousands of principals. However, the advent of computational grids realizes the possibility of environments composed of tens of thousands of resources and hundreds of thousands of jobs. The scalability and reliability issues of the matchmaking infrastructure must be revisited to address the concerns of these large systems.

6.2.2 Gangmatching

The gangmatching model proposed in this dissertation is an extremely flexible and expressive formalism to represent multiple dependent requests. Although some aspects of the gangmatching problem have been studied in detail, there are many interesting and important topics that require further investigation.

Preferences

The gangmatching algorithms presented in this dissertation ignore the problem of preferences. A substantial problem remaining in gangmatching is the formulation of preference-aware gangmatching algorithms. Although the semantics of port preferences allows the port rank expressions to be "maximized" independently, preference-aware gangmatching algorithms cannot treat ports in the same independent manner as preference-free algorithms because different port fill orders will result in different gang compositions. Thus, one may face situations when an RL strategy would be better, but an LR strategy would be forced by preference semantics.

Fortunately, many of the mechanisms developed for preference-free algorithms may be adopted. First, a dependence analysis may be performed on the ports. If the ports are independent, the choice of algorithm will not forced. Second, if the ports are dependent and, for example, an LR strategy is forced even though the RL strategy is recommended and if the number of distinct candidate rectangles for the right port is relatively small, the entire set of candidate rectangles may be used as a query window to filter the candidates of the left port. Matchmaking may then proceed with the LR algorithm. This technique provides the advantages of the RL algorithm without actually performing an RL match, albeit with additional overhead. Of course, other techniques would have to be developed for situations in which the number of candidate rectangles is large.

Support for Wider and Deeper Gangs

The gangmatching algorithms have been created for and tested on problems requiring small gangs. However, there are many additional techniques and algorithms which may be developed to marshal larger gangs. For example, in the context of the license management problem, the gangmatching algorithm always ensures that at any instant, the current partial gang being processed is consistent in the sense that, to the best of the matchmaker's knowledge, all the current gang members are compatible with each other. Unfortunately, the dynamic algorithm does not have this property for larger gangs.

Consider an example in which a job requires a machine and *two* licenses, which indirectly constrain the machine in the usual way. Assume that the rightmost license port is filled first. and the license places the constraint `job.HostID > 10`. Now assume that the remaining license port is filled, and the candidate license places the constraint `job.HostID < 5`. It is obvious that the two licenses are incompatible. However, the dynamic algorithm cannot detect this condition because the dependency between the two ports is indirect — the license ports are both dependent on the cpu port. Thus, detecting this dependence takes two hops in the inter-port dependency graph.

In the terminology of constraint programming, the current dynamic algorithm implements *node consistency* and *arc consistency* [21]. What is required therefore is a mechanism to implement *path consistency*. The ingredients to implement this mechanism are all present: the dependency relation is already maintained for each advertisement and the indexing code may be used to determine consistency through its type and value consistency checks. The distributed information in the partial gang must be aggregated to form the appropriate window query to ensure that only completely consistent gangs are marshaled.

Bottom-Up Gangmatching Algorithms

The use of bottom-strategies is an intriguing possibility for gangmatching algorithms. Since all the proposed algorithms are top-down, they inherit the general weaknesses of top-down algorithms, including the possibility of exponential behavior on some workloads. For example, if a job requested six machines via six ports, but only five machines are available, all of the proposed algorithms would attempt the 5! machine permutations before failing to satisfy the match. The use of bottom-up algorithms and the implications of these strategies on the rest of the framework needs investigation.

Generalized AND/OR port relationships

The current gangmatching formalism requires *all* the ports of an advertisement to be satisfied for the advertisement to be considered matched. Thus, the model only supports the AND multi-resource allocation paradigm — there is no functionality to specify more complex relationships between the ports of the advertisement. The most flexible scheme would be a formalism that allows a hierarchical method of grouping ports, where each level of the hierarchy states the minimum number and maximum number of entries that are required to be satisfied. The generality of this scheme is very attractive, but it also introduces many difficulties. Issues such as label naming, label semantics, search strategies and preferences must all be reformulated to this more general formulation.

Gangmatching for other problems

We have approached the gangmatching problem with a strong goal-directed philosophy of solving the license management problem. However, as mentioned in Chapter 3, the gangmatching model is extremely flexible and capable of some very intriguing functionality. It would be interesting to apply the gangmatching formalism to other problems such as the management of abstract services and distributed access control.

Diagnostics

A significant problem currently faced in our framework is explaining the behavior of the matchmaking process to software agents, human users and system administrators. The matchmaking process currently behaves like a black-box, with little indication why certain classads are matched with other particular classads or why some classads never find matches. The reasons for such outcomes are complex and depend not only on the contents of the classad in question, but all candidate match classads which collectively determine the "state" of the system. Although work is currently underway to provide diagnostic services to human users for the bilateral case, it would be interesting to provide these services for gangmatching as well. In addition to assisting human users in comprehending policies, diagnostic functionality in gangmatching would enable dependency directed backtracking strategies instead of the naive temporal backtracking strategy currently used.

Bibliography

[1] Distributed.net Project Page. http://www.distributed.net.

[2] Seti@home Project Page. http://setiathome.ssl.berkeley.edu.

[3] S. Abiteboul. Querying Semi-structured Data. In *Proceedings of ICDT*, January 1997.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[5] J. Basney and M. Livny. Improving Goodput by Co-scheduling CPU and Network Capacity. *International Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[6] P. Buneman. Semistructured Data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, 1997.

[7] Pi. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1996.

[8] E. G. Coffman, M. J. Elphick, and A. Shoshani. System Deadlocks. *Computing Surveys*, 3(2):67–78, 1971.

[9] XML Core Working Group. Extensible Markup Language (XML) 1.0 second edition. http://www.w3.org/TR/2000/REC-xml-20001006.

[10] International Business Machines Corporation. *IBM Load Leveler: User's Guide*, September 1993.

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

[12] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW8)*, 1999.

[13] H. Edelsbrunner. A New Approach to Rectangle Intersections (Part 1). *International Journal of Computer Mathematics*, 13:209–219, 1983.

[14] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[15] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proc. of the Third Int'l Conf. on Information and Knowledge Management, CIKM-94*. ACM press, nov 1994.

[16] L. Foner. A Multi-Agent, Referral-Based Matchmaking System. In *Proceedings of Autonomous Agents*, 1997.

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

[18] M. Freeston, G. Kuper G, and M. Wallace. Constraint databases. In *Conference on Information Technology and its Use in Environmental Monitoring and Protection*, 1995.

[19] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint Logic Programming: An Informal Introduction. *Logic Programming in Action, Lecture Notes in Artificial Intelligence*, 1992.

[20] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computer Surveys*, 30(2):123–169, 1998.

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

[22] 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.

[23] J. Gosling, B. Joy, and G. Steele. The Java Language Specification, 1996. Available from http://www.java.sun.com/docs/books/jls/index.html.

[24] J. Gray. *Notes in Database Operating Systems*, pages 393–481. Springer Verlag, 1979.

[25] A. S. Grimsaw and W. A. Wulf. Legion—A View from 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[26] D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, 1989.

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

[28] Hewlett Packard Inc. espeak: The Universal Language of E-Services. http://www.e-speak.net/.

[29] W. Johnston, S. Mudumbai, and M. Thompson. Authorization and Attribute Certificates for Widely Distributed Access Control. In *Proc. of IEEE 7th Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises: WETICE'98*, 1998.

[30] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. In *Proc. of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 299–313, Apr 1990.

[31] J. F. Karpovich. Support for Object Placement in Hetergenous Distributed Systems. Technical Report CS-96-03, University of Virginia, January 1996.

[32] B. W. Kerninghan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[33] C. Lai, G Medvinsky, and C. Neuman. Endorsements, Licensing and Insurance for Distributed System Services. In *Proc. of the Second ACM Conf. on Computer and Communications Security*, 1994.

[34] M. J. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proc. of the Fifth IEEE Int'l Symposium on High Performance Distributed Computing*, August 1996.

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

[36] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A Hunter of Idle Workstations. In *Proc. of the 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[37] M. Livny and R. Raman. *The GRID: Blueprint for a New Computing Infrastructure*, chapter High Throughput Resource Management, pages 311–336. Morgan Kaufman, 1999.

[38] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical report, Stanford University, Department of Computer Science, January 1998.

[39] M. Muttka and M. Livny. The Available Capacity of a Privately Owned Workstation. *Performance Evaluation*, 1991.

[40] J. Patel and D. DeWitt. Partition Based Spatial Merge Join. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1996.

[41] F. P. Preperata and M. I. Shamos, editors. *Computational Geometry.* Springer, 1988.

[42] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

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

[44] M. R.Genesereth and R. E. Fikes. Knowledge Interchange Format: Version 3, Reference Manual. Technical report, Stanford University, 1992.

[45] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, August 1984.

[46] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[47] B. Stroustrup. *The C++ Programming Languaage.* Addison-Wesley, third edition, 1997.

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

[49] K. Sycara, K. Decker, and M. Williamson. Matchmaking and Brokering. In *Proc. of the Second Int'l Conf. on Multi-Agent Systems (ICMAS-96)*, Dec 1996.

[50] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic Service Matchmaking among Agents in Open Information Environments. *SIGMOD Record*, 1999.

[51] K. Sycara, J. Lu, and M. Klusch. Interoperability among Heterogeneous Software Agents on the Internet. Technical Report CMU-RI-TR-98-22, Carnegie Mellon University, 1998.

[52] Uddi.com. UDDI: Technical White Paper. http://www.uddi.com/pubs/Iru_UDDI_Technical_White_Paper.pdf.

[53] J. Waldo. *Jini Architecture Review: White Paper*. Sun Microsystems Inc., 1998.

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

Appendix A

ClassAd Language: Built-in Functions

The classad language provides a rich set of built-in functions. User-defined functions may not be defined. However, additional functions may easily be added if access to the library source code is available. The syntax of a function call is

$$name(arg_0, arg_1, \ldots, arg_n)$$

As with operators, most functions are strict with respect to **undefined** and **error** on all arguments. However, some functions are non-strict, and these exceptions are noted. The name of the function is not case-sensitive.

A.1 Type predicates (Non-Strict)

IsUndefined(V)	True iff V is the **undefined** value.
IsError(V)	True iff V is the **error** value.
IsString(V)	True iff V is a string value.
IsList(V)	True iff V is a list value.

IsClassad(V)	True iff V is a classad value.
IsBoolean(V)	True iff V is a boolean value.
IsAbsTime(V)	True iff V is an absolute time value.
IsRelTime(V)	True iff V is a relative time value.

A.2 List Membership

Member(V,L)	True iff scalar value V is a member of the list L.
IsMember(V,L)	Like Member, but uses `is` for comparison instead of `==`. Not strict on first argument.

A.3 Time Queries

CurrentTime()	Get current time (absolute time)
TimeZoneOffset()	Get time zone offset as a relative time
DayTime()	Get current time as relative time since midnight.

A.4 Time Construction

MakeDate(M,D,Y)	Create an absolute time value of midnight for the given day. M can be either numeric or string (e.g., ”jan”).
MakeAbsTime(N)	Convert numeric value N into an absolute time (number of seconds past UNIX epoch).
MakeRelTime(N)	Convert numeric value N into a relative time (number of seconds in interval).

A.5 Absolute Time Component Extraction

GetYear(A)	Get integer year.
GetMonth(A)	$0 = jan, \ldots, 11 = dec$
GetDayOfYear(A)	$0 \ldots 365$ (for leap year)
GetDayOfMonth(A)	$1 \ldots 31$
GetDayOfWeek(A)	$0 \ldots 6$
GetHours(A)	$0 \ldots 23$
GetMinutes(A)	$0 \ldots 59$
GetSeconds(A)	$0 \ldots 61$ (for leap seconds)

A.6 Relative Time Component Extraction

GetDays(R)	Get days component in the interval
GetHours(R)	0 ... 23
GetHours(R)	0 . . . 23
GetMinutes(R)	0 . . . 59
GetSeconds(R)	0 . . . 59

A.7 Time Conversion

InDays(T)	Convert time value into number of days
InHours(T)	Convert time value into number of hours
InMinutes(T)	Convert time value into number of minutes
InSeconds(T)	Convert time value into number of seconds

A.8 String Functions

StrCat(V1, . . . , Vn)	Concatenates string representations of values V1 through Vn
ToUpper(S)	Upcases string S
ToLower(S)	Downcases string S

SubStr(S,offset [,len])	Returns substring of S. Negative offsets and lengths count from the end of the string.
RegExp(P,S)	Checks if S matches pattern P (both args must be strings). See `regexec(3C)` for details on patterns.

A.9 Type Conversion Functions

Int(V)	Converts V to an integer. Time values are converted to number of seconds, strings are parsed, bools are mapped to 0 or 1. Other values result in **error**
Real(V)	Similar to Int(V), but to a real value.
String(V)	Converts V to its string representation
Bool(V)	Converts V to a boolean value. Empty strings, and zero values converted to **false**; non-empty strings and non-zero values converted to **true**.
AbsTime(V)	Converts V to an absolute time. Numeric values treated as seconds past UNIX epoch, strings parsed as necessary.
RelTime(V)	Converts V to an relative time. Numeric values treated as number of seconds, strings parsed as necessary.

A.10 Mathematical Functions

Floor(N)	Floor of numeric value N
Ceil(N)	Ceiling of numeric value N
Round(N)	Rounded value of numeric value N