Matchmaking: Distributed Resource Management for High Throughput Computing

Rajesh Raman
Miron Livny
Marvin Solomon
University of Wisconsin
1210 West Dayton Street
Madison, WI 53703
{raman,miron,solomon}@cs.wisc.edu

Abstract

Conventional resource management systems use a system model *to describe resources and a centralized scheduler to control their allocation. We argue that this paradigm does not adapt well to distributed systems, particularly those built to support* high-throughput computing. *Obstacles include heterogeneity of resources, which make uniform allocation algorithms difficult to formulate, and distributed ownership, leading to widely varying allocation policies.*

Faced with these problems, we developed and implemented the classified advertisement *(classad) matchmaking framework, a flexible and general approach to resource management in distributed environment with decentralized ownership of resources. Novel aspects of the framework include a semi-structured data model that combines schema, data, and query in a simple but powerful specification language, and a clean separation of the matching and claiming phases of resource allocation. The representation and protocols result in a robust, scalable and flexible framework that can evolve with changing resources.*

The framework was designed to solve real problems encountered in the deployment of Condor, a high throughput computing system developed at the University of Wisconsin—Madison. Condor is heavily used by scientists at numerous sites around the world. It derives much of its robustness and efficiency from the matchmaking architecture.

1. Introduction

A principal consideration of resource management systems is the efficient assignment of resources to customers. The problem of making such efficient assignments is referred to as the *resource allocation* or *scheduling* problem, and it is commonly formulated in the context of a *scheduling model* that includes a *system model*, which is an abstraction of the underlying resources. The system model provides information to the allocator regarding the availability and properties of resources at any point in time. The allocator uses this information to allocate resources to tasks so as to optimize a stated performance metric. This paradigm is useful for *high performance applications*, which have tight constraints. Efficient scheduling of resources is critical in meeting these constraints. However, an increasing number of organizations now have environments that are not amenable to this resource management paradigm; their customers are interested in *throughput* and their computing resources are *distributively owned*.

In a distributively owned environment, the owner of a resource has the right to define its usage policy, which may be very sophisticated. For example, the policy may state that a job can run on a workstation only if the it belongs to a particular research group, or if it is run between 6 p.m. and 6 a.m., or if the keyboard hasn't been touched for over fifteen minutes and the load average is less than 0.1. Distributed ownership makes it impossible to formulate a monolithic system model . There is therefore a need for a resource management paradigm that does not require such a model and that can operate in an environment where resource owners and customers dynamically define their own models. The Matchmaking resource management paradigm presented in this paper was designed to address this need.

Matchmaking uses a semi-structured data model—the *classified advertisements* data model—to represent the principals of the system and folds the query language into the data model, allowing entities to publish queries (i.e., requirements) as attributes. The paradigm also distinguishes between matching and claiming as two distinct operations in resource management: A match is an introduction between two compatible entities, whereas a claim is the establishment of a working relationship between the entities. This distinction has several benefits, which will be discussed in Section 3.

We found the matchmaking paradigm to work extremely well in an environment where a large number of dissimilar resources (such as workstations, tape drives, network links, application instances, and software licenses) transit between available and unavailable states without advance notice and where resources may be available to some entities, and unavailable to others. Such an environment has to employ *opportunistic scheduling*: Resources are used as soon as they become available and applications are migrated when resources need to be preempted. The applications that most benefit from opportunistic scheduling are those that require high *throughput* rather than high *performance*. Traditional high-performance applications measure their performance in instantaneous metrics like floating point operations per second, while high throughput applications usually use such application-specific metrics as weather simulations per week or crystal configurations per year. In other words, rather than MIPS (millions of instructions per second), the performance of high-throughput applications might measured in TIPYs (trillions of instructions per year).

The rest of this paper is structured as follows. Section 2 describes related work, and Section 3 describes the proposed matchmaking framework. In Section 4, we describe how the framework has been used in the Condor system. We conclude with a summary of our results and an outline of our plans for continuing research in Section 5.

2 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 [13], PBS [6], LSF [15] and LoadLeveler [1] process user submitted jobs by finding resources that have been identified either explicitly through a job control language, or implicitly by submitting the job to a particular queue that is associated with a set of resources. 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.

Globus [4, 2] 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. The RSL allows customers to provide very sophisticated resource requirements, but no analogous mechanism for resources exists.

Legion [5] takes an object-oriented approach to resource management, formulating the matching problem as an *object placement problem* [7]. 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 [7] that is similar to classads, although it uses an object-oriented type system with inheritance to define resources [8], 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 requests that haven't been offered that it would rather serve.

3 The Matchmaking Framework

The basic idea of matchmaking is simple: Entities which provide or require a service advertise their characteristics and requirements in *classified advertisements* (classads). A designated matchmaking service (*matchmaker*) matches classads in a manner that satisfies the constraints specified in the respective advertisements and informs the relevant entities of the match. The responsibility of the matchmaker then ceases with respect to the match. The matched entities establish contact, possibly negotiate further terms, and then cooperate to perform the desired service.

The matchmaking framework may be decomposed into five components:

1. the *classad specification*, which defines a language for expressing characteristics and constraints, and a semantics of evaluating these attributes,

2. the *advertising protocol*, which defines basic conventions regarding what a matchmaker expects to find in a classad if the ad is to be included in the matchmaking process, and how the matchmaker expects to receive the ad from the advertiser,

3. the *matchmaking algorithm*, which defines how the contents of ads and the state of system relate to the outcome of the matchmaking process,

4. the *matchmaking protocol*, which defines how matched entities are notified and what information they are given in case of a match, and

5. the *claiming protocol*, which defines what actions the matched entities take to enable discharge of service.

There are two noteworthy aspects of this approach that distinguish it from conventional resource allocation models.

- Conventional resource management systems only allow customers to impose constraints on the type of services they require. Our mechanism also allows service providers to express constraints on the customers they are willing to serve.

- The semantics of a matchmaker identifying a match between entities A and B is not "allocating A to B." Rather, a match results in a mutual introduction of the two entities, who may activate a separate claiming protocol, not involving the matchmaker, to complete the allocation. Either entity may choose to not proceed further and reject the introduction altogether. Thus, a match is to be construed as a "hint." A beneficial consequence of this approach is that the matchmaker is a stateless service, which simplifies recovery in case of failure.

We discuss important aspects of the paradigm in further detail below.

3.1 Classified Advertisements

A classad is a highly flexible and extensible data model that can be used to represent arbitrary services and constraints on their allocation. The model has several novel aspects.

- Classads use a semi-structured data model [11], so no specific schema is required by the matchmaker, allowing the matchmaker to work naturally in a heterogeneous environment.

- The classad language folds the query language into the data model. Constraints (i.e., queries) may be expressed as attributes of the classad.

- Classads are first-class objects in the model. They can be arbitrarily nested, leading to a natural language for expressing resource aggregates or co-allocation requests.

```
[
  Type         = "Machine";
  Activity     = "Idle";
  DayTime      = 36107 // current time
    // in seconds since midnight
  KeyboardIdle = 1432;    // seconds
  Disk         = 323496;  // kbytes
  Memory       = 64;      // megabytes
  State        = "Unclaimed";
  LoadAvg      = 0.042969;
  Mips         = 104;
  Arch         = "INTEL";
  OpSys        = "SOLARIS251";
  KFlops       = 21893;
  Name         = "leonardo.cs.wisc.edu";
  ResearchGroup = { "raman", "miron",
                    "solomon", "jbasney" };
  Friends      = { "tannenba", "wright" };
  Untrusted    = { "rival", "riffraff" };
  Rank         =
    member(other.Owner, ResearchGroup) * 10
    + member(other.Owner, Friends);
  Constraint   =
    !member(other.Owner, Untrusted)
      && Rank >= 10
        ? true
    : Rank > 0
        ? LoadAvg<0.3 && KeyboardIdle>15*60
    : DayTime < 8*60*60
        || DayTime > 18*60*60;
]
```

Figure 1. A classad describing a workstation

A classad is a mapping from *attribute names* to expressions.[1] For example, Figure 1 shows a classad that describes a workstation in a Condor [10, 3] pool at the University of Wisconsin,[2] and Figure 2 shows a classad that describes a job submitted for execution.

Attributes may be simple integer, real, or string constants, or they may be more complicated expressions constructed with arithmetic and logical operators and record and list constructors. Expressions can also refer to other attributes, as in "`Rank >= 10`." Expressions and attribute references are discussed in greater detail below.

[1]Similar structures have been called *records*, *dictionaries*, and *frames* in other contexts.

[2]Examples in this paper are adapted from actual ads in use in a working Condor installation. They have been edited slightly for clarity and to illustrate features of the classad mechanism.

```
[
    Type           = "Job";
    QDate          = 886799469;
      // Submit time secs. past 1/1/1970
    CompletionDate = 0;
    Owner          = "raman";
    Cmd            = "run_sim";
    WantRemoteSyscalls = 1;
    WantCheckpoint = 1;
    Iwd            = "/usr/raman/sim2";
    Args           = "-Q 17 3200 10";
    Memory         = 31;
    Rank           =
      KFlops/1E3 + other.Memory/32;
    Constraint     =
      other.Type == "Machine"
        && Arch == "INTEL"
        && OpSys == "SOLARIS251"
        && Disk >= 10000
        && other.Memory >= self.Memory;
]
```

Figure 2. A classad describing a submitted job

3.2 Matching and Claiming

We now describe the specific actions taken by entities which require matchmaking services. Providers and customers construct classads describing themselves and send them to the Matchmaker (Step 1 in Figure 3). These classads must be constructed to conform to the *advertising protocol* specified by the matchmaker, which attaches a meaning to some attributes. For example, the advertising protocol may specify that the attribute `Constraint` indicates compatibility and the attribute `Rank` measures the desirability of a match (see Figures 1 and 2). The advertising protocol also specifies how the entities send the classads to the matchmaker.

The matchmaker then invokes a *matchmaking algorithm* by which matches are identified (Step 2). To perform the match, the matchmaker evaluates expressions in an environment that allows each classad to access attributes of the other: An attribute reference of the form "`self.`*attribute-name*" refers to another attribute of classad containing the reference, while "`other.`*attribute-name*" refers to an attribute of the other ad. 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.Memory >= self.Memory`" expresses the requirement that the server advertise an amount of `Memory` sufficient to meet the memory needs of this job (the expression could also have been written "`other.Memory >= Memory`").

The classads in Figures 1 and 2 assume a matchmaking algorithm that considers a pair of ads to be incompatible unless their `Constraint` expressions both evaluate to **true**. The `Rank` attributes is then used to choose among compatible matches: Among provider ads matching a given customer ad, the matchmaker chooses the one with the highest `Rank` value (non-integer values are treated as zero), breaking ties according to the provider's `Rank` value.

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

```
Mips >= 10 || 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.
```

The matchmaking algorithm effectively treats **undefined** as **false**—the match fails if the `Constraint` evaluates to **undefined**—but the three-valued logic used in Boolean expressions supports natural expression of constraints on objects whose types are only partially known *a priori*.

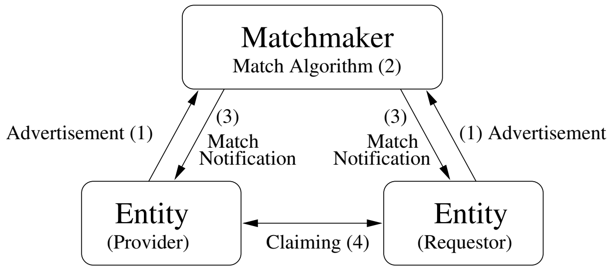

Figure 3. Actions involved in the Matchmaking process

After the matching phase, the matchmaker invokes a *matchmaking protocol* to notify the two parties that were matched (Step 3) and sends them the matching ads. The matchmaking protocol could also include the generation and hand-off of a session key for authentication and security purposes. The customer then contacts the server directly, using a *claiming protocol* to establish a working relationship with the provider (Step 4). It is important to note that identifying a match and invoking the matchmaking protocol does not immediately grant service to a customer. Rather, the match is a mutual introduction to the advertising entities.

The separation of matching and claiming has several beneficial properties.

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 not possible 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 which really need to interact with specific kinds of providers and customers. The matchmaker may be written as a general service which does not depend on the kinds of services and resources that are being matched.

The concept of 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 [14]. The principals involved in a match are themselves responsible for establishing, maintaining and servicing a match. The matchmaker does not need to retain any state about the match, a fact that simplifies recovery in case of failure and makes the system more scalable.

4 An Example of Matchmaking at Work: Condor

Condor [9, 10] 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 [12]. This section briefly describes aspects of the Condor system that are relevant to the problem of matchmaking.[3]

Resources in the Condor system are represented by *Resource-owner Agents* (RAs), which are responsible for enforcing the policies stipulated by resource owners. An RA periodically probes the resource to determine its current state, and encapsulates this information in a classad along with the owner's usage policy. Figure 1 is an example of a classad that encapsulates a fairly sophisticated owner policy, demonstrating the flexibility of the mechanism. 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.

Customers of Condor are represented by *Customer Agents* (CAs), which maintain per-customer queues of submitted jobs, represented as lists of classads. RAs and CAs periodically send classads to a Condor *pool manager*, describing the resources and job queues respectively. The resource classads and the request classads conform to an advertising protocol that states that every classad should include expressions named `Constraint` and `Rank`, as discussed previously. The protocol also requires the advertising parties to include "contact addresses" with their ads, and allows an RA to include an "authorization ticket" with its ad.

Periodically, the pool manager enters a *negotiation cycle*. This phase invokes the matchmaking algorithm, which determines which CAs require matchmaking services, obtains

[3]More information about the project and the system may be found at `http://www.cs.wisc.edu/condor/`.

requests from these CAs, and matches them with compatible RA ads. 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 matchmaking algorithm also uses past resource usage information to enforce a fair matching policy.

When the pool manager determines that two classads match, it 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 RA.

The CA then performs the claiming protocol by contacting the RA and sending the authorization ticket. The RA accepts the resource request only if the ticket matches the one that it gave the pool manager, and the request matches the RA'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. When the CA finishes using the resource, it relinquishes the claim, and the RA advertises itself as unclaimed. The RA 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 RA.

Classads are used for other purposes in Condor as well. All entities are represented with classads, as are queries submitted by various administrative and user tools. "One-way matching" protocols are used to find all objects matching a given pattern. For example, there are tools to check on the status of job queues and browse existing resources.

5 Conclusions and Future Research

The classad matchmaking framework is a flexible and general method of resource management in pools of resources which exhibit physical and ownership distribution. Novel aspects of the framework include a semi-structured data model to represent entities, folding the query language into the data model, and a clean separation of the matching and claiming phases of resource allocation. The representation and protocols facilitate both static and dynamic heterogeneity of resources, which results in a robust, scalable and flexible framework that can evolve with changing resources.

The framework has been developed in response to real problems that have been encountered in the design, development and deployment of Condor, a high throughput computing system developed at the University of Wisconsin–Madison, which finds constant use by scientists at the university and around the world. The success of the framework in a real system demonstrates the validity of our approach.

Although the classad mechanism has no intrinsic notion of a schema, lists of classads representing resources and customers exhibit a high degree of regularity, which is manifest in two ways: structural regularity and value regularity. The former occurs when entities tend to publish attributes with the same names, and the latter occurs when groups of entities publish attributes with similar values. We are currently investigating techniques for exploiting this regularity, and automatically aggregating classads so that matches may be performed in groups. Group matching may be used to both boost matchmaking throughput and service co-allocation requests.

The complexity of constraints imposed by resources and customers may hinder the diagnostic capability of administrators and customers who may wonder why certain requests are unable to find resources with particular characteristics. To alleviate this problem, we are researching methods for identifying constraints which can never be satisfied by the pool. In addition to diagnostic utilities, this tool may help discovering hidden characteristics of a pool.

References

[1] I. B. M. Corporation. *IBM Load Leveler: User's Guide*, Sept. 1993.

[2] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. Available from `ftp://ftp.globus.org/pub/globus/papers/gram.ps.Z`.

[3] D. 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.

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

[5] 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*, Aug. 1996.

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

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

[8] 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*, Aug. 1996.

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

[10] 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.

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

[12] 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.

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

[14] 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, Aug. 1984.

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