Notice

Chapter 1

High Throughput Computing Resource Management

Historically, users of computing facilities have been mainly concerned with the response time of applications while system administrators have been concerned with throughput. As a paradigm, users judged the power of a system by the time taken to perform a fixed amount of work. Given this fixed amount of computing to perform, the question most users asked was: How long will I have to wait to get the results of this computation? Administrators, who were charged with the responsibility of managing scarce and expensive computing resources, were judged by the utilization and throughput of the facility. While the average response time and the throughput of a facility are related, they represent two very different viewpoints of the performance of a computing environment. In recent years, however, we have experienced a change in these traditional viewpoints.

The dramatic decrease in the cost-performance ratio of computing resources has effectively substituted "response time" for "utilization" as a primary concern of administrators. At the same time, a growing community of users are now concerned about the *throughput* of their applications. As more scientists, engineers, and decision makers use computers to generate behavioral data on complex phenomena, it is not uncommon to find users who ask the question: How much behavioral data can I generate by the time my report is due? This question represents a paradigm shift. In contrast to other users, these users measure the power of the system by the amount of work performed by the system in a fixed amount of time. For these *throughput oriented users*, the fixed time period is measured in the relatively coarse units of days or weeks, whereas the amount of work is seemingly unbounded — one can never have too much data when studying a biological system, testing the design of a new hardware component, or evaluating the risk of an investment.

The computing needs of these throughput oriented users are satisfied by High Throughput Computing (HTC) environments that can generate large amounts of behavioral data. These users are less concerned about the instantaneous performance of the environment (typically measured in Floating Point Operations per Second (FLOPS)) rather than the amount of computing they can harness over a month or a year. They measure the performance of a computing environment in units of scenarios per day, wind patterns per week, instructions sets per month, or crystal configurations per year. Given their unbounded need for computing resources, the HTC user community is closely watching activity in the computational grids area, and anxiously awaiting the moment when they can tap into the vast computational power of these grids.

In this chapter we present important lessons learned, promising directions and future challenges in the design and implementation of scalable and robust HTC environments. We present these issues as a result of our decade-long interaction with groups of high throughput computing users that include scientists and engineers who employ diverse computation techniques from a wide range of disciplines. These users have been using HTC resources to study a wide spectrum of phenomena including among others diesel engines, neural networks, high energy physics events, computer hardware and software, the structure of crystals, and optimization techniques [25]. Most of them have been customers of the Condor [24] environment that we have developed. While this chapter is based on our experience with Condor, our objective is by no means to present Condor or to evaluate its capabilities. Since one can view a Condor pool as a private computational grid of desk-top workstations that are managed for HTC use, it is our hope that builders of computational grids who would like to provide HTC services (e.g high throughput distributed supercomputing (Chapter **??**) will find our experience and frameworks useful.

We believe that the experience from these interactions in terms of lessons learned and promising future directions are applicable to all types of computational grids, regardless of whether they are private, virtual, organizational, or public grids. Furthermore, we expect that the size, scope, heterogeneity, and dynamics of computational grids will only strengthen the validity of these conclusions. The confidence in these beliefs stems from working with a wide range of customers with real-life computing needs, from maintaining and supporting Condor for more than a decade and from managing a large HTC production environment at the University of Wisconsin-Madison. (We currently manage a

Condor Flock [6] at the University of Wisconsin that consists of more than 500 desk-top UNIX workstations and serves users throughout the campus.)

The most important lesson our HTC experience has taught us is that in order to deliver and sustain high throughput over long time intervals, a computing environment must build its resource management services on an integrated collection of robust, scalable and portable mechanisms. Robustness minimizes down time whereas scalability and portability increases the size of the resource pool the environment can draw upon to serve its customers. As will be argued in the next section, a typical environment is physically distributed, its resource pool is heterogeneous and is owned by several entities, the availability of resources can change at any time, and new types of resources are continuously added to the pool as older technology is removed. Fragile mechanisms that depend on the unique characteristics of specific computing platforms are likely to have a negative rather than a positive impact on the long term throughput of the environment.

Four groups of users are served by the mechanisms provided by a HTC environment: resource owners, customers, system administrators and application writers. The needs and expectations of each of these groups and the role they play in the success of a HTC environment will be discussed in section 1.2. In the same way that an electric power grid is not just a collection of generators, lines, outlets, and trading policies, but a community that consists of power providers, customers, share holders and maintenance crews, a HTC computational grid is a community with its own culture and a unique set of rules. In section 1.3 we present and discuss a promising suite of matchmaking mechanisms that can bring providers of computational services and consumers of such services together, thus integrating the HTC community.

In the most general case, either party, the provider or the customer, can have the right to break an allocation at any time. A mechanism capable of preserving any partially completed work is thus needed. In section 1.4 we discuss a user level checkpointing mechanism by which a snapshot of an executing program can be stored away. The snapshot can be later used to restart the program from that state. A brief overview of commercial and public domain batch systems is made in section 1.5.

1.1 Salient Characteristics of HTC Environments

Given the seemingly infinite appetite for computing power of its customers, a HTC environment is continuously on the lookout for additional resources. HTC environments have the mentality of scavengers. The services of a provider of computing power are always accepted regardless of the resource's characteristics, degree of availability, or duration of service. As a result, the pools of resources HTC environments draw upon to serve their customers are large, dynamic and heterogeneous collections of hardware, middleware, and software.

As a result of the recent decrease in the cost/performance ratio of commodity hardware and the proliferation of software vendors, resources that meet the needs of HTC applications are plentiful and have several different characteristics, configurations and flavors. A large majority of these resources reside today on desk-tops, owned by interactive users, and are frequently upgraded, replaced or relocated.

The change in the cost/performance ratio of hardware not only improved the power of our desk-top machines but also rendered the concept of multi-user time-sharing obsolete. While in the early days of computing the idea of allocating a computer to a single person was not sensible, it has become common practice in recent years. In most organizations, each machine is usually allocated to one individual in the organization to support his/her daily duties. A small fraction of these machines will be grouped into small farms and allocated to groups who are considered to be heavy users of computing by management. We believe that the trend to distribute the ownership of resources within organizations will continue giving full control over powerful computing resources to individuals and small groups. As a result of this trend, while the absolute computing power of organizations has improved dramatically, only a small fraction of this computing power is accessible to HTC users due to the ever increasing fragmentation of computing resources. In order for a HTC environment to productively scavenge these distributively owned resources, the boundaries marked by owners around their computing resources must be crossed.

However, crossing ownership boundaries for HTC requires that the rights and needs of individual resource owners be honored. Resource owners are generally unwilling to donate their machines for HTC use at the cost of degraded performance or availability. The restrictions placed by owners on resource usage for HTC can be complex and dynamic, involving parameters such as recent "idleness" of the resource and characteristics of the customer. These restrictions constrain when and which customers can be allocated to the resource.

The constraints attached by owners to their resources prevent the HTC environment from planning future allocations. All the resource manager knows is the current state of the resources. It therefore has to treat them as sources of computing power that should be exploited *opportunistically*. Available resources can be reclaimed at any time and resources occupied by their owners can become available without any advance notice. The resource pool is also continuously evolving as the mean time between hardware and software upgrades of desk-top machines is steadily decreasing. Owners are likely to replace their hardware as faster CPUs and larger memories become affordable, and will install new versions of operating systems or switch to a new ones all together soon after announcement.

In addition to ownership boundaries, HTC environments must cross administrative domains as well. The main obstacle to inter-domain execution is access to the environment from which the application was submitted, such as input and output files. The HTC computing environment has to provide means by which an application executing in a foreign domain can access its input and output files that are stored at its home domain. The ability to cross administrative domains not only contributes to the processing capabilities of the environment, but also broadens the "customer circle" of the environment. It makes it very easy to connect the computer of a potential customer to the environment. In a way, the HTC environment appears to the user as a huge increase in the processing power of her personal computer since almost everything looks the same except for the throughput. As in the case of an electric grid where one does not know who generated the power that cooks one's meal, a user of a HTC environment does not know who executed the program that transformed the parameters stored in the input file to the time-series that has "miraculously" appeared in the output file.

The applications that perform these transformations usually follow the master-worker computing paradigm, where a list of tasks is executed by a group of workers under the supervision of a master. The realization of the master and the workers and their interaction may take different forms. The workers may be independent jobs submitted by a shell script that acts as the master and may collect their outputs and summarize them, or they can be implemented as a collection of PVM processes which receive their work orders in the form of messages from a master process that expects to receive the results back in messages [18]. Regardless of the granularity of the work-units and the extent to which the master regulates the workers, the overall picture is the same — a heap of work is handed over to a master who is responsible for distributing its elements among a group of workers.

Since workers in one tier can act as masters to workers in a lower tier, hierarchies of master-workers can be easily formed. These hierarchies may span more than one HTC environment. For example, a group of researchers from the University of Amsterdam has been running its HTC application in six Condor pools located in three different countries and spanning two continents. Over the last three years they have used more than one hundred and fifty CPU years to search for global potential energy minima of a N-particle system consisting of Lennard-Jones particles on a spherical surface [26].

At any given time, hundreds of workers of the above application could have been found scattered over the different pools. Some of these workers consumed more than 100 days of CPU over a life time of 4–5 months. In many cases these workers were left unattended as members of the group were away from their desks attending meetings, or on vacation. The group expected the HTC environment not to lose any of these workers before or during their execution phase. Any such loss may have had a significant impact on their throughput. Like most other HTC users we have worked with, they counted on the robustness of the mechanisms used by the environment to successfully take their workers from submission to completion, and were much less concerned about the efficiency of the mechanisms or the policies that control them.

While losing workers prematurely is clearly what HTC customers worry about most, they obviously have some basic expectations regarding wasted resources or fairness in resource allocation. Given the rapid changes in hardware and operating systems, the biggest potential source of throughput inefficiency is exclusion of resources due to the inability of the mechanisms of a HTC environment to operate on new computers. There is nothing more frustrating for a HTC customer than new resources that are likely to be the biggest and fastest being excluded from the HTC environment due to porting difficulties.

Simplicity clearly holds the key to the robustness and portability of HTC mechanisms. As will be discussed in the next section, these mechanisms serve not only the customers, but also owners of resources, administrators of the environment, and programmers who write HTC applications. Since computational grids are likely to be large, physically distributed, distributively owned, dynamic and evolving just like HTC environments, we believe that the same principles hold for the mechanisms that will support these grids. While users with tight time constraints or very demanding quality of service needs will expect computational grids to employ sophisticated resource allocation policies, most of them and the entire community of HTC users will expect robust services that run anywhere.

1.2 Resource Management Layers of a HTC Environment

The cornerstone of a HTC environment is the Resource Management System (RMS) that manages its pool of distributively owned resources. The RMS provides Resource Management (RM) services to its user community, which consists of four groups of people: owners, system administrators, application writers and customers. The ordering in this list is significant, because we believe that owners are the most important group of people in a distributively owned environment. Without resources which have been "donated" by owners for HTC, the RMS ceases to exist. The distinguishing aspect of distributed ownership is that this donation is not unconditional — the RMS must ensure that owners have unhindered access to their resources, that there is no perceived degradation in the availability or performance of the resource during personal use, and that the resource access policy specified by the owner is honored. Next, system administrators must feel confident that the RMS is robust and can run continuously without frequent intervention. If the RMS fails to win the trust of system administrators, its installation and use at a site is not possible. Even in the presence of a robust and reliable system, inflexible and obscure application programmers' interfaces (API's) to the services provided by the RMS nullify the generality and power of the RMS since the available features cannot be effectively harnessed for productive computation. Thus, it is important to address the requirements of application writers during the design and implementation of RMS's. Customers are in many ways the easiest group to please because they are the ultimate beneficiaries of the RMS. However, if the system is not flexible enough to adapt to the requirements of the customer, it will fail to effectively address their concerns and fall into disuse.

Thus, it can be seen that the requirements of an effective RMS are quite demanding, and building a successful RMS is a complex task. It is important to note here that in addition to the performance related requirements of the users community, they also have security concerns that must be addressed by the HTC environment. In a computational grid, the RMS will relay on the services of other components of grid (e.g. network services Chapter **??** and security, accounting and assurance services Chapter **??**) to satisfy all the needs of its users. The success of an RMS can only be assessed when it runs continuously and reliably in "production mode," with owners and customers who are satisfied by the delivered quality of service and reliability, and with system administrators and application writers who can rely on the robustness and flexibility of the system. These requirements suggest that the system must be built using a layered approach with close interaction, monitoring and control of resources at bottom level, and abstractions and interfaces for application developers and customers at the topmost level.

An important point to note is that each layer is defined by its responsibility and the protocols with which it interacts with other layers. Actual implementations of components in layers may vary greatly across the RMS. Thus, for example, it is possible to have different implementations and paradigms of access control at the owner level for different resources, as long as each implementation is compliant to the behavioral specification of owner layer components. The same argument extends to all layers of the RMS. These layers therefore define the architecture of the system, the granularity of inter-operability, and domains of fault containment.

The principal layers of a RMS are illustrated in figure 1.1 and enumerated below:

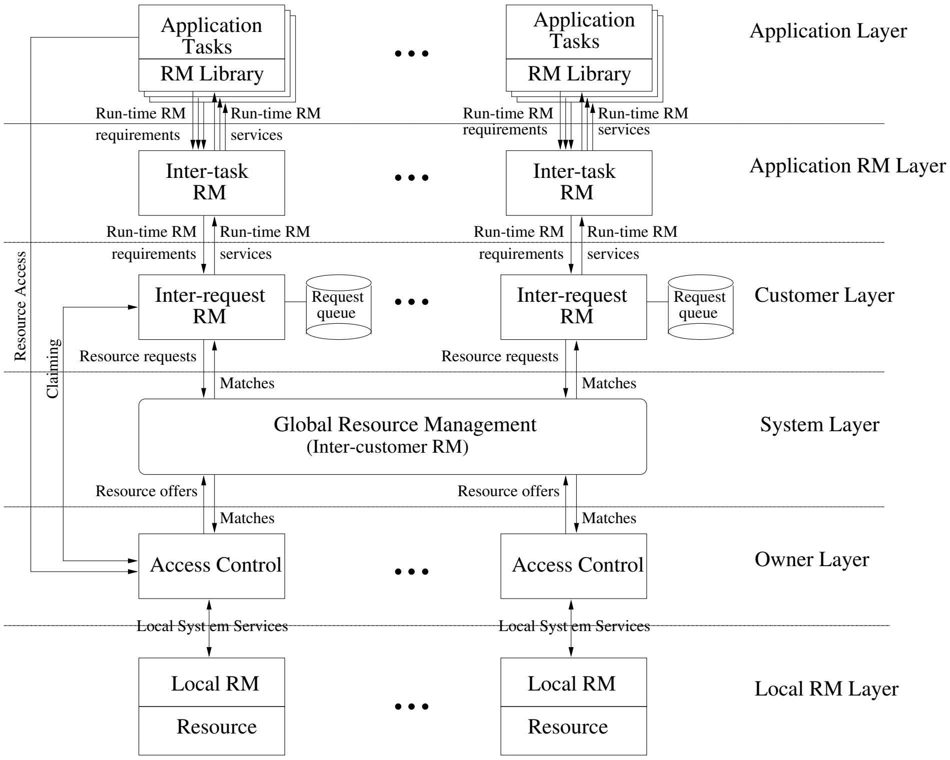

Figure 1.1: Layers of a Resource Management System

1. *Local Resource Management Layer.* The first layer of the RMS is not really part of the system but rather is logically part of the resource. It is a software layer (e.g operating system, batch system or even another computational grid) that provides basic RM services for processes executing in the domain of that resource. Since we are principally interested in the higher order problem of managing distributed resources rather than local RM, we do not discuss this layer further. Nevertheless, it is a fundamental and important component of a robust HTC because unreliable hardware

and local services can seriously affect the sustained operation of a HTC system.

2. *Owner Layer.* The owner layer of the RMS represents the interests of the resource owner. A fundamental purpose of the owner layer is to provide *access control* mechanisms to the resource, which interacts with and enforces the owner's policy. It is important to note that these policies are beyond those otherwise imposed by the RMS itself, and imply the necessity of constraints which determine when and to whom the resource may be allocated for HTC. Within the constraints of the owner's policy, the owner layer also informs the *system layer* of the characteristics and availability of the resource.

3. *System Layer.* The system layer may be thought of as the global resource allocation layer. Its principal function is *matchmaking*, i.e., matching resource offers and requests so that the constraints of both are satisfied. This matchmaking occurs in the context of high level policies which implement *inter-customer scheduling policy.* Although this policy is not directly relevant to the architecture of the RMS, the policy may dictate when and with whom matchmaking may take place. For example, these policies may enforce fair-share [12], stable marriage [8], or economic-based [13] matching policies.

4. *Customer Layer.* The customer layer is the layer which represents the customer's interests in the RMS. This layer provides the abstraction of a "user" as a queue of resource requests. The primary goals of this layer are to maintain this queue in a persistent and fault tolerant manner and to interact with the system layer by injecting resource requests for matchmaking, claiming matched resources for the requests and handing these resources off to the application RM layer. The injection of requests takes place in the context of an *inter-request resource management policy* which may dictate, for example, which requests have priority over others or which requests are dependent on others so that certain requests are satisfied before others. Another important function of this layer is to provide an interface to the HTC environment for both human users, and importantly, also applications.

5. *Application RM Layer.* Once a resource has been claimed by the customer layer, it is passed on to the application resource management layer which implements per-application RM services. The application RM is responsible for communicating with the resource's access control module

to establish the run-time environment for the application. Importantly, it also provides run-time services for querying, utilizing and requesting more resources. This functionality is extremely useful as it provides a framework for the development of applications which are adaptive and can grow and exploit resources as and when these resources become available. This functionality is afforded by close interaction of the application RM layer with both the application itself and the customer layer through well defined interfaces. New requests for resources made by applications appear in the queue of the customer layer, which then proceeds to negotiate for a resource in the usual way. An additional responsibility of application RM is the implementation of *inter-task resource management* which determines which resource will be used in fulfilling which task's request.

6. *Application Layer.* The application layer represents instances (or tasks) of the customer's application. These tasks accomplish pieces of the end-result of the customer's computation by utilizing resources handed to the application's resource manager. Run-time RM services required by the application are forwarded to the application RM layer, which may service these requests directly, or indirectly by acting as an intermediary to the customer layer.

The effectiveness of an HTC environment depends on how well the four different groups that constitute its user's community are interwoven. What brings them together are the services provided by the six layers of the RMS. All the layers have to operate in harmony in order to establish an atmosphere of collaboration among the members of the community. Such an atmosphere, which can not be built unless there is goodwill and mutual respect, holds the key to the success of an HTC environment. Like in any community, matchmaking plays a pivotal role in the nature of the relationships developed between owners, system administrators, application writers, and customers in a HTC environment. We therefore start our discussion of HTC resource management mechanisms with a presentation of a matchmaking suite in the next section.

1.3 Matchmaking and Claiming

In this section, we present the basic requirements of a robust and effective matchmaking suite (or framework). A primary concern of distributed resource management in large computational grids is the scalability, flexibility and robustness of the matchmaking mechanism. We introduce *classified advertisements* (classads), an approach for representing and matching requests and offers for services in a distributed matchmaking framework.

The paradigm of entities advertising their attributes and requirements to a matchmaker is a promising one. This scheme has several advantages:

1. The details of formulating and managing requirements and constraints is the responsibility of the advertisers themselves. This facilitates an end-to-end approach for resource management. The specific entities matched themselves control the claiming, usage and control of resources and services without subsequent intervention of the matchmaker, whose responsibility ends after identifying the match. This enhances the generality and scalability of the system.

2. The paradigm does not imply an architecture for the matchmaker. The implementation of the abstract matchmaking service can be parallelized and distributed for better reliability, availability and performance.

3. The paradigm is extremely flexible as it is not tied down to any specific type of resource. Indeed, the matchmaker may be used for more abstract services than finding resources, such as finding other matchmakers.

Thus classads are more of a negotiation based approach to resource management, where the advertising entities (and not the matchmaker) assume full responsibility for advertising, claiming and managing resources and services. We discuss this mechanism in further detail below.

1.3.1 Advertising Offers and Requests

The fundamental problem of an HTC system is resource management. As such, its purpose is to bring resources and customers together to enable productive computation. To perform this matchmaking, the system must first have a method of representing resources and customers. The flexibility and expressiveness of the representation is extremely important as it directly affects the functionality of the resource management system. For example, in a general system of resources, not all resources of interest are compute nodes. Other possibilities include software licenses and network links. A representation that assumes that every resource is a single compute node would be unable to effectively represent other entities such as storage media, network links and multi-processor parallel machines. Thus, a representation must avoid any assumptions about the nature and characteristics of resources. This requirement would allow a HTC environment to represent several heterogeneous resources. This flexibility along with the implied dynamic implementation would facilitate inclusion and exclusion of resources from the pool at run time.

The distributed ownership of resources implies that owners of resources must be able to restrict the usage of their resources. The mechanism which implements these access policies must be flexible enough to account for both the technical concerns and sociological idiosyncrasies of owners' policies. For example, the owner of a workstation may demand an access policy which states that the resource is available for HTC only if the keyboard has been idle for over fifteen minutes and the background load average is less than 0.3; customer requests made by the owner have higher priority than those made by members of the owner's research group, which in turn have a higher priority than other requests. Finally, no requests made by members of a competing research group are to be serviced, and customers requiring less than 100 MB of virtual memory are preferred. These restrictions may be thought of as the conditions under which the owner grants the resource to a resource request, or the *requirements* of the resource offer, which must be honored by the matchmaker.

Similar restrictions may be placed by requests on offers too. Malleable parallel applications and large jobs with task dependencies can significantly affect the requirements of applications during a run. These applications place both qualitative and quantitative constraints on required resources as the task set in question grows and shrinks with time. For example, a customer to the system may state a requirement of at least five machines, and at most fifty. Of these, four machines with over 64 MB of memory are required, and the rest should have at least 32 MB of memory and a MIPS rating of over 80. Furthermore, since the application was compiled for a particular architecture it *requires* compute nodes which are of the same architecture.

Thus, with respect to matchmaking, a symmetry exists in the structure of offers and requests: both need to express their attributes and requirements. This allows us to formulate the basic unit of the matchmaking mechanism as the encapsulation of the attributes and requirements of an entity that requires matchmaking services. The similarity between this mechanism and the classified advertisements one finds in newspapers prompted us to define this unit of encapsulation as a *classified advertisement* or *classad*.

Before discussing classads in further detail, one must note that despite the similarity of our formulation to newspaper classads, important differences exist.

- Requests vs. offers. A key feature of newspaper classads is that it is trivial to determine if an ad represents a request or an offer either purely by the contents or by the context of the ad. Although this differentiation is often useful when implementing non-trivial matching policies (e.g., fair matching), our mechanism does not require it. In matching two classads, identifying and distinguishing the entity offering the service is not generally necessary because, as we shall shortly see, the matching process is operational and does not intrinsically depend on any implied semantic content of the ads. However, in the interest of clarity of discussion, we continue to make this distinction.

- Advertisers vs. matchmakers. Unlike newspaper classads, our framework clearly differentiates between advertisers and matchmakers. This allows one to designate specialized matchmakers which match offers and requests using criteria such as priority, fairness and preferences. Optionally, these designated matchmakers may be treated as trusted authorities that can grant capabilities or tickets to the matched entities.

1.3.2 Desired features of Matchmaking Mechanisms

The assumed model of the HTC environment is that it is an *open environment*. By this we mean that services and customers of different types (including completely new ones) can be added in or removed at run time. There is no inherent necessity for a central authority that determines which entities may advertise, and what they should advertise for. The following constraints are immediately imposed:

- Portability. Since the specific entities involved in matchmaking cannot be assumed to be of a fixed type or architecture, the mechanism must be portable and architecture-independent.

- Self describing. The advertised services in the environment may vary from compute nodes to software licenses to storage space to network bandwidth. Each resource requires a different description, but the matchmaker must be able to function correctly in a manner that is independent of the specific descriptions.

- Well-defined and robust semantics. Due to the inherent uncertainties in large heterogeneous open environments, the mechanism must have well-defined and robust semantics to handle situations such as when characteristics required by an entity are not correctly represented in the candidate match ad, or if such information is completely absent.

- Decoupled protocols. For maximum robustness and scalability, the mechanism must carefully distinguish and decouple the protocols for advertising, matchmaking and claiming. Decoupling the advertising and matchmaking protocols allows the matchmaker the freedom of matching asynchronously with respect to advertising clients. Decoupling the claiming and matchmaking protocols relaxes the required degree of information consistency in disseminated advertisements.

The classad mechanism has been designed to address all of the above issues.

1.3.3 The ClassAd Mechanism

A classad based matchmaking framework consists of five logically independent components: (1) the evaluation mechanisms, (2) the claiming protocol, (3) the advertising protocol, (4) the matchmaking protocol and (5) the matchmaking algorithm. Of these components, the evaluation mecahnisms are *absolute*, i.e., they are standard and remain fixed across all matchmaking frameworks. In contrast, a given matchmaker defines its own matchmaking algorithm, advertising protocol and matchmaking protocol, and advertising entities employ a claiming protocol to connect with each other. Thus, the definitions of these components are *relative* to a given framework.

The matchmaker of a framework defines:

- its *advertising protocol* which describes both the expected contents of ads and the means by which it obtains these ads,

- its *matchmaking protocol* through which it communicates the outcome of the matchmaking process to the entities involved, and

- the *matchmaking algorithm* which semantically relates the the contents of classads to the matchmaking process.

The claiming protocol is executed by the entities matched by the matchmaker to connect to each other and perform productive computation. The protocol may also involve a verification phase by the entities involved when the match is validated with respect to their current state, which may have changed since the advertisement from which the match was made.

The distinction between absolute and relative components is noteworthy, because a classad is defined as an attribute list that has been constructed in conformance to a given matchmaker's advertising protocol. Thus an attribute list may be a classad with respect to matchmaker A, but just an arbitrary attribute list to another matchmaker B which defines its relative components differently. Although this distinction is useful for the purposes of design and discussion, one must note that:

- The evaluation mechanisms, which define the semantics of expression evaluation, do not depend on this distinction. The repercussions of non-conformance to an advertising protocol is defined by the matchmaker's matchmaking protocol and is completely independent of the absolute components, which have well-defined behavior regardless. Specifically, the correctness and performance of a matchmaker is not compromised by an entity which advertises "non-conforming classads" (which are construed to be arbitrary attribute lists).
- In the interest of simplicity, the different aspects of matchmaking are explained with respect to a single matchmaker. Thus, in this discussion, "classad" and "attribute list" are used interchangeably.

The Evaluation Mechanisms

An entity that requires matchmaking services expresses its characteristics and requirements as a set of *attributes* called an *attribute list*. Each attribute is a binding of an identifier with an expression. The expressions are structurally similar to arithmetic expression constructs in common programming languages, and are composed of constants, attribute references (which can refer to attributes in candidate match ads), calls to primitive inbuilt functions and other subexpressions combined with operators and parentheses. Figure 1.1 illustrates two example attribute lists.

The key feature of attribute lists that makes it an attractive mechanism for open environments is the semantics of expression evaluation, which are defined so that the uncertainties of an open environment can be handled in a graceful manner. Specifically, the evaluation of an expression is well-defined even if required expressions are not available, or do not yield values of expected types. In these cases, the evaluation results in the distinguished `UNDEFINED` and `ERROR` values respectively, which can be explicitly tested for.

The evaluation of expressions is usually (but not required to be) carried out by the matchmaker when testing two classads for mutual constraint satisfaction. This test is usually performed by evaluating expressions from well known attributes (specified by the advertisement protocol) from the two ads and ensuring that they evaluate to `TRUE`. The matchmaker evaluates these expressions from the classad in an "environment" which contains the two classads being tested. The two classads involved in the match are expected to conform to the advertising protocol, and the number, contents and scope names of the other classads (which contain default attributes and other match related information) are fixed by the matchmaking algorithm. This entire set of classads serves as an environment from which attributes can be looked up. Expressions in classads

Example 1	
Type	⇒ “Machine”
OpSys	⇒ “OSF/1”
Arch	⇒ “Alpha”
Memory	⇒ 32
Disk	⇒ 782
KbdIdle	⇒ 17
LoadAvg	⇒ 0.1
ReplyTo	⇒ ”<chestnut.cs.wisc.edu:5964>”
Requirement	⇒ (self.LoadAvg < 0.3) && (self.KbdIdle > 15) && (other.Owner != “foo”)
Example 2	
Owner	⇒ “bar”
Group	⇒ “condor team”
Executable	⇒ “a.out”
ImageSize	⇒ 10
State	⇒ “Idle”
RemoteCPU	⇒ 0
ReplyTo	⇒ ”<perdita.cs.wisc.edu:3748>”
Requirement	⇒ (Type == “Machine”) && (OpSys == “OSF/1”) && (Arch == “Alpha”)

Table 1.1: Examples of classads. The expressions in these classads may be arbitrarily complex. Attribute references of the form *self.X* and *other.X* force lookup of attribute X in the same ad and the candidate match ad respectively. If the references do not have these prefixes, natural default lookup rules are used.

can refer to any attribute in the environment, including attributes from other classads. This lookup in other ads may be performed explicitly by prefixing attribute references by *scope resolution prefixes* such as “self,” “other” and “env” in the example, which explicitly name classads from where the attributes will be looked up. The semantics of attributes references without scope resolution is also defined. Interested readers are referred to [20] which details the structure and evaluation semantics of classad expressions.

1.3.4 Matchmaking

The model of matchmaking in the classad framework is that entities that require matchmaking services post classads to a matchmaker which matches the ads and notifies the advertisers concerned in the event of a successful match. The framework may contain several matchmakers, each of which may be distinguished by one or more features such as the domain in which it matches (e.g., automobile, furniture), the matching algorithm used, the semantics of a match and its communication protocol.

Notably absent in the responsibilities of a matchmaker is any notion of allocation. This intentional omission is due to the following reasons:

1. In highly dynamic environments, the status of advertising entities may

have changed since their last advertisement. The matchmaker may therefore make some invalid matches with regard to the current state of the advertising entities. Entities that receive notification of a match must activate a *claiming protocol* which both validates the match with regard to their current state and establishes a relationship between the matched entities if the match is deemed valid. This protocol, which involves only the two matched entities and not the matchmaker, is required irrespective of whether one considers the match as an "allocation."

In such dynamic environments a match is a substantially weaker operation than an allocation. This operation may be strengthened by having the matchmaker generate a capability as a part of the match. In this case, the match may be considered as "permission" rather than a "hint," but the operation still remains considerably weaker than allocation.

2. The details of allocation can greatly vary depending on the type of the entity being allocated. These details are best left out of the matchmaker, which may be involved in matching (a possibly unknown) number of types of heterogeneous entities.

3. Allocation is by nature an asymmetric operation where one entity is allocated to another. Matchmaking is more symmetric, which allows more general interactions. Any desired asymmetry can be introduced by the claiming protocol in the context of the matched entities, and should not be imposed by the matchmaker itself.

1.3.5 Claiming

Claiming is the process by which the two parties agree to use the services of each other: the provider which serves the request, and the consumer which requests that it be served. Claiming has two important roles to play in the matching of offers and requests:

1. Since the matchmaker does not constrain or verify the contents of advertisements, it is possible for an entity to incorrectly represent itself or its characteristics and origin to obtain a match. Verifying the correctness of advertisements is an issue that requires further investigation. A promising approach is that of "licensing and assurance," which is discussed in chapter **??**. Regardless, the claiming protocol is an extremely useful interaction in this regard because it can be designed to include a challenge-response protocol for mutual authentication.

2. In addition to authentication, both entities use the claiming protocol to verify that their respective constraints are indeed satisfied with respect to their current states. Thus, the claiming protocol forms the first phase of implementing the constraints imposed by entities involved in the match. If these constraints are not satisfied, the match is rejected, and the entities restart the advertise-match-claim cycle.

Example. We now furnish an example which illustrates matchmaking and claiming in a simple classad based framework. The example is necessarily informal about the specification of the relative components.

- *Advertising protocol.* Every classad sent to the matchmaker must include an attribute named `Requirement` which represents the constraints of the advertiser. The classad must also contain an attribute named `ReplyTo` which is a communication endpoint at which it can be contacted. (Communication protocols regarding sending the classad to the matchmaker are omitted.)

- *Matchmaking algorithm.* Two classads A and B are said to match if A's `Requirement` evaluates to `TRUE` *and* B's `Requirement` evaluates to `TRUE` in the environment constructed by the matchmaker.

- *Matchmaking protocol.* In the event of a match, the matchmaker will contact the two matched entities at their `ReplyTo` addresses, and pass the `ReplyTo` attribute of the other entity involved in the match. Additionally exactly one of the two entities is passed a tag, which denotes it to be the active entity. The other entity is said to be the passive entity.

- *Claiming protocol.* In the event of a match, the active entity contacts the passive entity (i.e., its match) at the `ReplyTo` address sent by the matchmaker. The passive entity makes sure the network connection comes from the `ReplyTo` address specified by the matchmaker. The connection is then made, and the matched entities jointly perform their computation.

In the context of the above relative components, one can see that the examples of attribute lists in figure 1.1 are classads in this matchmaking framework, and may be potentially matched by the matchmaker.

It is important to note that the matchmaking algorithm does not contain any references to specific resources or services, or what it takes to match them — all this information is contained in the classads themselves which are created by the entities requiring matchmaking services. The algorithm also makes no distinction between the offer and the request for the service. If any of the ads sent to the matchmaker do not have a `Requirement` expression, the match would fail because the evaluation of the "Requirement" attribute would result to `UNDEFINED` and not `TRUE`.

In the most general and flexible case, either party, the provider or the requester, has the right to break an allocation at any time. The provider may have to give the resource back to the owner, or may have an offer from a more important or profitable customer, while the consumer may have obtained access to a cheaper or more powerful resource. In many of these cases, it would be very unfortunate if the work accomplished so far is lost. It is therefore in the interest of both sides to have access to checkpointing mechanism that can save the current state of the computation so that another provider can resume execution at a later stage. While the traditional view of checkpointing is that it is a means to improve the reliability of a computing environment, for a HTC environment it is a basic resource management tool with a mean inter-usage time that is much smaller than the mean inter-failure time of the hardware or the software. In the next section we discuss the different aspects of the checkpointing problem and provide an overview of a checkpointing mechanism we developed for UNIX systems.

1.4 Checkpointing

A checkpoint of an executing program is a snapshot of its state which can be used to restart the program from that state at a later time. Computing systems have traditionally employed checkpointing to provide reliability: when a compute node fails, the program running on that node can be restarted from its most recent checkpoint, either on that same node once it is restored or potentially on another available node. Checkpointing also enables preemptive-resume scheduling. All parties involved in an allocation can break the allocation at any time without losing the work already accomplished by simply checkpointing the application. Thus, a long running application can make progress even when allocations last for relatively short periods of time. Due to the opportunistic nature of resources in a distributively owned environment, any attempt to deliver HTC has to rely on a checkpointing mechanism.

Checkpointing services provide an interface both to the application and the surrounding environment. At the least, an application should be able to request that its state be checkpointed at any time during its run and be able to request that no checkpoints be performed during specified critical sections. The POSIX P1003.10 draft standard on checkpoint and restart additionally provides an interface for an application to specify pre- and post-checkpoint processing.

Researchers have also developed user-directed checkpointing services [16], which rely on user hints about memory usage to significantly increase the performance of checkpointing. In addition to the application interface, a checkpointing service provides an external interface to schedulers and users to trigger an application checkpoint due to external events (preemption, system shutdown, etc.). Often this is done by sending a signal to the application (in the case of user-level checkpointing) or by making a system call (in the case of kernel-level checkpointing).

Checkpointing can be an expensive and time-consuming operation, since the (potentially large) checkpoint must be written (possibly over the network) to disk. Checkpoints of parallel applications can be particularly huge, since the state of a parallel program includes the state of the interconnection network in addition to the state of each process. Also, performing a checkpoint requires that the process's address space be read, which can involve swapping virtual memory pages in from disk. In an opportunistic environment, it is imperative that a preempted process vacate the machine quickly, so if the scheduler can not write a checkpoint quickly, the work accomplished since the last checkpoint will be lost at preemption time. In a system where checkpoints are written periodically and very fast preemption is required, preemption sans checkpointing may be desirable. User-directed checkpointing is one method for writing potentially fast checkpoints. Another method is to deploy specialized checkpoint file storage servers throughout the computational grid and direct checkpoints to the nearest or least loaded server at preemption time [19]. A third method is to migrate the process immediately to another machine by writing the checkpoint to a network stream and reading it directly off the network on the new machine. In this method, the checkpoint does not need to be written to disk. This requires, of course, that a new machine be available at checkpoint time.

The decision of where to send a checkpoint can have a significant impact on performance and reliability, and can impact other scheduling decisions. Migration requires that a new compute node be allocated for the task at the time of preemption. Disk space must be available for checkpoints not being used for immediate migration. Network bandwidth will affect the speed with which a checkpoint can be written. A checkpointing mechanism should, therefore, provide an interface to allow a scheduler to direct checkpoints to the appropriate network endpoint or disk.

Since most workstation operating systems do not provide kernel-level checkpointing services, an HTC environment must often rely on user-level checkpointing. In our experience developing and maintaining a user-level checkpointing library [15], we have found portability to be a significant challenge. For example, after porting our library to a new version of a popular Unix operating system, we had reports from a user that his simulation was exiting prematurely after restarting from checkpoint. After much investigation, we discovered that we needed to reset a flag to tell the operating system to save floating point registers on context switches. Small differences like this between operating systems and operating system versions add up to make maintaining a portable, robust user-level checkpointing mechanism a significant challenge to the HTC environment developer. Silicon Graphics has included kernel-level checkpointing services in a recent version of the Irix operating system [21]. We hope this is the start of a trend among operating system vendors.

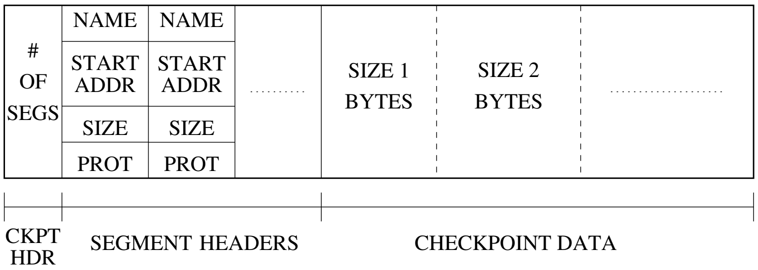

Figure 1.2: Structure of a checkpoint file

Process checkpointing is implemented in our user-level checkpoint library as a signal handler. When a process linked with this library receives a checkpoint signal, the provided signal handler writes the state of the process out to a file or a network socket. To determine where to write the checkpoint, the signal handler either uses a file location provided on the command line or sends a message to a controlling process asking for a file location or a network address to connect to. The checkpoint includes the contents of the process's stack and data segments, all shared library code and data mapped into the process's address space, all CPU state including register values, the state of all open files, and any signal handlers and pending signals. On restart, the process reads the checkpoint from the file or network socket, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. Again, the location from which to read the checkpoint is either determined by a command line option or the response to a query of a controlling process. The checkpoint signal handler then restores the CPU state and returns to the user code, which continues from where it left off when the checkpoint signal arrived. A program can request that a checkpoint be performed by sending itself the checkpoint signal and can disable checkpointing in critical sections by blocking the checkpointing signal.

Other details in implementing a practical checkpointing mechanism, such as handling dynamic libraries and checkpointing state that is not accessible directly from user-level (such as the open file table), have also been overcome with indirect solutions. The interested user is referred to [15].

Since our checkpointing support is implemented in a static library, applications which use it must be linked with this library. This unfortunately means that applications for which source or object files are not available can not make use of our checkpointing support. Some Unix variants include a method for injecting a dynamic library into an executable at startup time. This method could potentially be used to provide a shared library implementation of checkpointing support which could be injected into unmodified programs at startup time.

Checkpointing processes which use network communication requires that the state of the network be checkpointed and restored. Our checkpointing library has been enhanced to support applications which use PVM or MPI [3]. To checkpoint the state of the network, this library synchronizes communicating processes by flushing all communication channels prior to checkpoint. At restart time, the library restores the communication channels. Programs which communicate with processes that can not be checkpointed also pose an interesting problem. Programs which communicate with X servers or license managers fall into this category. We have developed a solution which places a switchboard process between the two endpoints. Instead of connecting directly, these processes connect through the switchboard. When the program is checkpointed, it notifies the switchboard and closes its connection. The switchboard, however, keeps the connection to the other endpoint open and buffers any communication from this endpoint until the checkpointed program is restarted and the connection is restored. Protocol specific knowledge is required in the switchboard if the non-checkpointable endpoint expects prompt replies.

1.5 An overview of Batch Systems

Since the days of the first mainframes, batch systems have played a crucial role in providing computing resources to HTC applications. Equipped with queuing mechanisms, scheduling policies, priority schemes, and resource classifications, these systems have been running batch jobs on dedicated resources. In recent years the mechanisms employed by batch systems have been extended to deal with large multi-processor computers and clusters of workstations. Their policies were also adapted to meet the needs of workloads that consists of a mix of sequential and parallel applications.

The resources controlled by a batch system are typically owned by one organization and managed by a single administrative domain. System administrators have full control over all resources and are in charge of the scheduling policies.

Jobs are placed in queues classified according to their resource requirements and the customer who submitted them. Each queue is assigned computing resources to process the class of jobs it serves. Designed and built to operate as a production tool, batch systems are known for their robustness and reliability. These qualities will be extremely valuable assets to any computational grid that provides HTC services and wants to exploit the resources managed by such a system.

It is beyond the scope of this chapter to provide an in depth discussion and evaluation of the currently available commercial and public-domain batch systems. An excellent review of batch systems can be found in the Cluster Computing Review by M. Baker et al. [2]. The results of a very detailed and systematic evaluation of six job management systems (JMSs) was recently published in the latest NASA Job Management System (Batch/Queuing Software) evaluation report [11]. Three of the systems evaluated (CODINE [7], DQS [23] and LSF [17]) emphasize heterogeneous environments, whereas the other three systems (LL [10], NQE [5], and PBS [9]) focus their efforts mainly on supercomputers.

A recent trend of "do-it-yourself" has been adopted by administrators of large production systems who design and implement their own batch schedulers (e.g. EASY from ANL [14], and the Maui Scheduler from MHPCC [4]) and make them available to the community. These schedulers reflect the unique needs and resource allocation philosophy of their implementors and utilize the API's of an underling batch system that provides the scheduler with queuing and process management mechanisms. The Nimrod system [1] is an example of another recent trend to build "tailored" batch schedulers designed to support customers who are engaged in large multi-job computing efforts in a specific domain.

1.6 Challenges

There are several challenges on the way to large scale HTC computational grids. Indeed, every chapter in this book identifies and examines technologies that must be revisited or created to achieve this goal. In the interest of brevity, we identify issues that immediately challenge the very large scale deployment of the technology of high throughput computing.

1. *Understanding the sociology of a very large and diverse community of providers and consumers of computing resources.* Unlike electrical grids, in a HTC computational grid, every consumer can also be a provider and every provider can also be a consumer of services. In electrical grids, a small

number of providers serves a much larger community of consumers. If every consumer in an electrical grid had her own generator and some consumers were always looking for more power, electrical grids would have looked and behaved much more like HTC environments. The National Technology Grid [22] of the NCSA alliance will provide us with a laboratory to study such a community.

2. *Semantic free matchmaking services.* In the interest of flexibility and expressiveness, semantic free matchmaking services for providers and customers of complex services with constraints must be developed. These constraints define who they are willing to serve or by whom they are willing to be served by, respectively. These mechanisms are required to not only be expressive but also efficient, as matchmakers would have to check extremely large numbers of candidate matches in a grid of even moderate size.

3. *Tools to develop and maintain robust and portable resource management mechanisms for large, dynamic and distributed environments.* Current approaches to developing such complex resource management frameworks usually involve a new implementation of large fractions of the framework for each instance of a marginally different RMS. An established framework with tools and API's would allow the construction of inter-operable components. This would greatly enhance both the functionality and development time of complex RMS's. Many of these concerns are addressed in chapter **??** in some detail.

4. *Universally available checkpoint services for sequential and parallel applications.* This goal is perhaps one of the most difficult ones to achieve for purely practical considerations. Differences in vendor implementations of operating systems, varied architectures and inadequate user-level support for checkpointing makes providing ubiquitous checkpointing services a difficult goal to achieve. However, as described in section 1.4, recent activity in the field makes this goal more tenable. The availability of a ubiquitous checkpointing mechanism would greatly increase the percentage of available cycles productively harnessed by applications.

5. *Understanding the economics (relationship between supply and demand) of a HTC computational grid.* While basic priority schemes for guaranteeing fairness in the allocation of resources are well understood, mechanisms and policies for equitable dispersion of services across large computational grids are not. A major aspect in the development of such policies involves

understanding supply and demand for services in the grid. Clearly, this is a topic that warrants further investigation.

6. *Data staging — moving data to and from the computation site.* When the problem of providing vast amounts of CPU power to applications becomes better understood, the next hurdle that must be crossed will be that of providing sufficiently sophisticated RM services to individual throughput oriented applications. A major aspect of this problem is that of integrated staging, transport and storage media management mechanisms and policies for high throughput.

1.7 Summary

The need for high computing power over sustained intervals has increased in the scientific and engineering community. In contrast to other users who are concerned with response time and use interactive computing services, these users are primarily concerned with the throughput of their applications over relatively long periods of time.

In this chapter we argue that resources of computational grids can be productively scavenged to service these applications. By doing so, both the HTC community and the HPC community will benefit as HTC applications will migrate to opportunistically managed commodity resources, freeing the high-end resources to HPC applications. In the general case, resources of the grid are distributively owned, which presents several technical difficulties if these resources are to be scavenged for productive computation. The most important aspect of using distributively owned resources is that the RMS must honor the policies of resource owners at all times. Other requirements for satisfying system administrators, application authors and customers must also be addressed.

The varied requirements of such an RMS requires careful decomposition of the system into manageable modules whose responsibilities and interactions are well defined. To this end we present a flexible, scalable and robust six layered architecture for distributed resource management systems, and discuss the specific responsibilities and interactions of each layer in the system.

A primary interaction in the system is between that of customers and resources who are brought together by a matchmaking service. The flexibility and robustness of the matchmaking service is extremely important as it directly impacts the usability and quality of service provided by the HTC system. We present the classad matchmaking framework as a promising matchmaking mechanism to be used in computational grids.

For maximum flexibility, the matchmaking service must have the ability to preempt and rematch resources which were matched previously. To guarantee that applications make progress in the face of such dynamic policies, it is important to have an application checkpointing mechanism. Such a mechanism is also important when an owner's access control policy revokes the resource for personal use, or other more preferred requests.

We claim that these mechanisms, although originally developed in the context of a cluster of workstations, are also applicable to computational grids. In addition to the required flexibility of services in these grids, a very important concern is that the system be robust enough to run in "production mode" continuously even in the face of component failures. A layered architecture with dynamic matchmaking frameworks can be used to address both concerns, and with the help of the other services provided in the grid, provide reliable and sophisticated high throughput computing resource management services in these grids.

1.8 Acknowledgements

We would like to thank Ian Foster and Carl Kesselman for their invaluable comments and suggestions which greatly improved this chapter. We also thank Jim Basney for his contribution to the Condor checkpointing section. In addition, we offer many thanks to Mike Litzkow for his work on the early implementations of Condor and Todd Tannenbaum, Derek Wright and Adiel Yoaz for their patient and skilful work in Condor Team.

Bibliography

[1] D. Abramson, R. Sosic, J. Giddy, and Hall B. Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[2] M. Baker, G. Fox, and H. Yau. Cluster Computing Review. Technical Report NPAC TR SCCS-748, Northeast Parallel Architectures Center, Syracuse University, New York, November 1995.

[3] A. Bode, G. Stellner, and J. C. Pruyne. CoCheck: Consistent Checkpoints. `http://wwwbode.informatik.tu-muenchen.de/CoCheck/`.

[4] Maui High Performance Computing Center. Maui Scheduler Overview. `http://www.mhpcc.edu/doc/uku/ms_overview.html`.

[5] CraySoft, Cray Research Inc. *NQE User's Guide*, 1997.

[6] 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.

[7] GENIAS Software GmbH, Neutraubling, Germany. *CODINE: Installation and Administration Guide 4.0*, March 1997.

[8] D. Gusfield and R. W. Irving. *The stable marriage problem :Structure and algorithms.* M.I.T. Press, 1989.

[9] R. Henderson and D. Tweten. Portable Batch System: External Reference Specification. Technical report, NAS, NASA Ames Research Center, December 1996.

[10] IBM. *Using and Administering IBM LoadLeveler, Release 3.0*, August 1996. SC23-3989.

[11] J Jones and C. Brickell. Second Evaluation of Job Queuing/Scheduling Software. Technical Report NAS-97-013, NASA, Numerical Aerospace Simulation (NAS), June 1997.

[12] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 32(1):44–45, January 1988.

[13] J. Kurose and R. Simha. A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[14] D. Lifka, M. Henderson, and K. Rayl. *Users Guide to the Argonne SP Scheduling System*. Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison, WI, USA, April 1997.

[16] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under UNIX. *Conference Proceedings, Usenix Winter Technical Conference*, pages 213–223, January 1995.

[17] Platform Computing. *LSF User Guide*, December 1996.

[18] 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.

[19] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS)*, 1996.

[20] R. Raman, M. Livny, and M. Solomon. The ClassAd Mechanism: Official Specification. Technical report, University of Wisconsin-Madison, WI, USA, 1997. (In preparation).

[21] Silicon Graphics International. *Hibernator II User's Guide*.

[22] R. Stevens, P. Woodward, T. DeFanti, and C. Catlett. From the I-WAY to the National Technology Grid. *Communications of the ACM*, 40(11):51–60, November 1997.

[23] Suptercomputer Computations Research Institute, Florida State University. *DQS 3.1.3 User Guide*, March 1996.

[24] Condor Team. The Condor High Throughput Computing Environment. `http://www.cs.wisc.edu/condor/`.

[25] Condor Users. Condor User Testimonials. `http://cs.wisc.edu/condor/stories.html`.

[26] J. M. Voogd, P. M. A. Sloot, and R. van Dantzig. Crystallization on a Sphere. *Future Generation Computer Systems*, 10:359–361, June 1994.