A Guide to Transparency in Five Research Metasystems

Allen Chu, Marsha Eng, Kevin D. Munroe, and Shava Smallen

Abstract

Metasystems are geographically distributed networks of heterogeneous computers on which a user can run very large program sets and parallel programs. We explain the concept of *transparency* and how it is presented to a metasystem client in five active research metasystems. We also judge the extent to which this transparency will benefit the novice and the expert user. We consider transparency in terms of five general areas: resource discovery, resource management, security, user interface, and fault tolerance. In general, resource discovery has sufficient transparency for both expert and novice users; however, the research metasystems under consideration differ widely in their implementations of the other four areas. We discuss the differences in those areas. Based on our findings, we describe the metasystems we would build to suit our user profile.

1. Introduction

We take *transparency* to mean the extent to which the users are aware their programs are running on a metasystem as opposed to a single computer or workstation. Transparency can show or hide details and nuances, which can be desirable or unnecessarily complex depending upon the particular user. We are interested in seeing how the design of five different research metasystems affects transparency to the user. We will also consider the effects of this transparency on users and judge which forms of transparency are more beneficial in certain situations.

Transparency is one of many factors that should be considered when selecting a metasystem. However, transparency should not be the deciding factor. Nevertheless, we feel that it is significant because the level of transparency in a system's features will strongly affect how easily users will be able to accomplish their computing goals.

We define transparency in terms of five metrics: the means by which computing resources are discovered, the means by which computing resources are managed, the interface presented to the user, user authentication and runtime process monitoring, and the fault tolerance of the metasystem. How each of these five metrics is handled affects exactly what the user sees and how the user will run their program on the metasystem.

We take a *metasystem* to mean a geographically distributed network of heterogeneous computers on which a user wishes to run a particularly large program set. The motivation behind using a metasystem instead of a single local computing site is that the metasystem will most likely provide higher cost effectiveness. That is, a metasystem allows us to take advantage of existing networks of machines, reducing the cost of adding more computing power. The low overhead of modern networks makes the cost of using distributed computing sites less of a performance factor.

A "more transparent" metasystem hides more details of its implementation from the user and limits user interaction with the underlying system components. Furthermore, the user should be less aware of the effects of system details. That is, a more transparent metasystem should provide consistent quality of service in a uniform interface.

We distinguish between novice and expert users in this discussion. The terms "novice" and "expert" are used to represent the two opposite ends of the user spectrum. For the purposes of this paper, we assume that the more transparent the metasystem, the more desirable it is in general for novice users. We assume further that expert users may wish to take advantage of particular underlying features of the metasystem. Therefore, a less transparent metasystem would be more desirable for expert users in this case. Although in our model we describe our novice and expert users as disjoint entities, this is not necessarily the case in practice. Depending upon their particular computing needs, they may share some transparency requirements. Also, typical users are probably neither exclusively novices nor experts; instead, based on their individual computing requirements, they are likely to be somewhere in between.

The five metasystems we have chosen to evaluate are: Condor, at the University of Wisconsin-Madison; Globus, at the University of Southern California and Argonne National Laboratory; Java Market, at Johns Hopkins University; Legion, at the University of Virginia; and WebOS, at the University of California, Berkeley. Although these systems vary widely in terms of their maturity and purpose, they all address valid concerns a typical user of a metasystem might have. Condor is interesting in that its implementation addresses the issue of computing on metasystems in a batch processing environment. Globus and Legion are implemented in the context of a highly distributed, heterogeneous environment. WebOS and Java Market explore computing on a metasystem in the context of the Internet.

Section 2 gives working summaries of the metasystems we examined. In Section 3 we show how each of these research systems fits into our transparency metric. In Section 4, we present our suggestions to both the novice and the expert metasystems user. Section 5 presents our conclusions. Our future work is outlined in Section 6.

2. Summaries of Metasystems

2.1. A Worldwide Flock of Condors

Condor is a system designed to distribute batch jobs across a pool of workstations [8]. Its goal is to better utilize the pool by transferring queued jobs to one or more idle machines. A Condor pool consists of machines connected by a local area network. A Flock of Condors extends the system by incorporating multiple Condor pools connected by either a local or wide area network.

The Flock of Condors uses both centralized and distributed server models, as seen in Figure 1. Within each pool, a single Central Manager (CM) is responsible for matchmaking between the idle machines and the waiting jobs. To form the Flock, a dedicated gateway machine is installed within each pool. Each gateway maintains and represents a list of all the idle machines outside of the pool to the CM. The CM views the gateway machine as any other member inside the pool, but with many free resources. There is no central server in the Flock since the free resource list is distributed on all gateways.

Matchmaking between idle machines and waiting jobs is done with a "classified advertisement", or Classad, system [14]. Both the idle machine and waiting job will submit a Classad to the CM. The Classad contains constraints on what each idle machine is able to provide and what each waiting job needs. With the information supplied by both parties, the CM uses a predefined matchmaking algorithm to pair the request Classad with an idle Classad.

Once a match is found, the waiting job can be transferred to the free machine. This is done with the migration and checkpointing mechanism [13]. The checkpointing mechanism allows Condor to save important process states at regular time intervals. These two mechanisms not only allow the relocation of an unexecuted process but also the stopping and migration of a running process when necessary.

Users submit batch jobs in the same way they would to a regular batch job interface. The programs can be executed on the local machine or be distributed to other idle machines in the Flock without additional modification. Once the machine owner decides to participate in the pool, the owner should not see any degradation in performance. This is achieved because guest processes are relocated to a host machine only when it is idle. When the owner returns and becomes active, the guest processes are stopped and moved using the checkpoint and migration mechanisms described above.

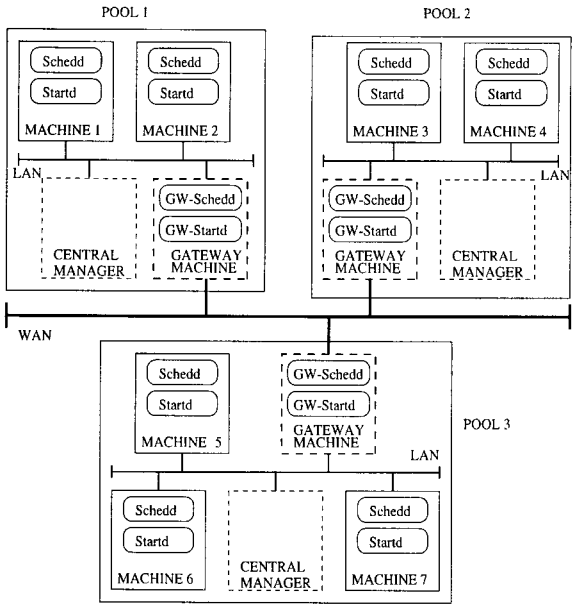

Figure 1. A Flock of Condors.
Source: D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. Journal on Future Generations of Computer Systems Volume 12, 1996.

Since a single Condor pool is not a true metasystem as given by our definition, we will only consider a Flock of Condors in this paper. For the sake of simplicity, we will simply refer to a Flock of Condors as "Condor".

2.2. Globus

Globus is a metasystem that provides a toolkit or "bag of services" to its users [9]. This toolkit provides a "translucent" interface to users by providing services that layer upon existing services. The idea behind this is to provide high level services which can still exploit a resource's low level services. Each service provided by Globus is meant to be distinct so that users can incrementally build up their programs by adding more services. Likewise, this gives users a choice of which services they want to use (i.e. users are not required to use all of the toolkit in order to run under Globus). The following are services provided by the Globus Metacomputing Toolkit: resource management, information, security, communication, health and status, remote data access, and executable management. These services are described below.

In terms of resource management, Globus provides the Globus Resource Allocation Manager (GRAM) [7] which layers on top of existing local resource management systems, as shown in Figure 2. One component of GRAM is the gatekeeper, which provides authentication of the Globus user to the local resource and starts up a job manager. The job manager creates and manages the user request. GRAM also provides a client library, which gives users the ability to submit job requests to remote sites

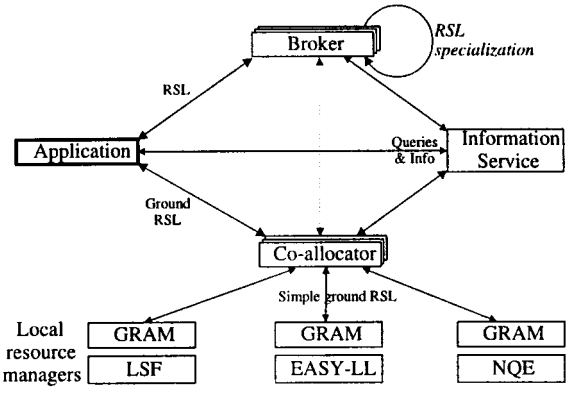

Figure 2. Globus resource management architecture.
Source: I. Foster and C. Kesselman. The Globus Project: A Status Report. *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pp. 4-18, 1998.

and authenticate with the remote site. In addition, a reporter communicates descriptive information about the host and its current status to the information service.

Globus' information service is called the Metacomputing Directory Service (MDS), which utilizes the Lightweight Directory Access Protocol (LDAP) [10]. The MDS organizes the Globus name space into a tree that allows the user to query for available resources. One type of query is a "white pages" lookup in which, given a host name, contact information about the gatekeeper would be returned (i.e. IP address and port number). A "yellow pages" query would submit attributes about the type of hosts it was looking for and receives in return a set of machines that matched those attributes. Therefore, users can find resources to execute their program provided they have the necessary authentication to do so.

The Globus Security Infrastucture (GSI) [9] layers on top of a resource's underlying security system such as plaintext passwords or Kerberos. The GSI performs authentication using the Secure Socket Library (SSL) protocol. Once authentication is performed, the gatekeeper of the machine checks to see if the user has permission to use the machine. In order to execute on a remote machine, a Globus user must have a local user account on that machine and be permitted to execute as a Globus user on the local machine.

Health and status in Globus are provided by the Heartbeat Monitor (HBM) [9], which is described further in Section 3.5. Briefly, Globus provides a unicast and multicast communication service called Nexus that has been used to implement a version of the Message Passing Interface (MPI). Remote data access is provided by Global Access to Secondary Storage (GASS), and Globus Executable Management (GEM) provides executable management. These last three services will not be discussed in the context of this paper.

2.3. Java Market

Java Market is a metasystem that allows user jobs to be run on any machine that wants to participate without any installation or platform-dependency issues [4]. Java Market jobs must be written in Java, and the job's source code must be submitted to the Market for inspection. In theory, the Market would act as a "broker", matching user jobs (consumers) with machines willing to sell CPU cycles to the Market (producers or "hosts"). Figure 3 shows a

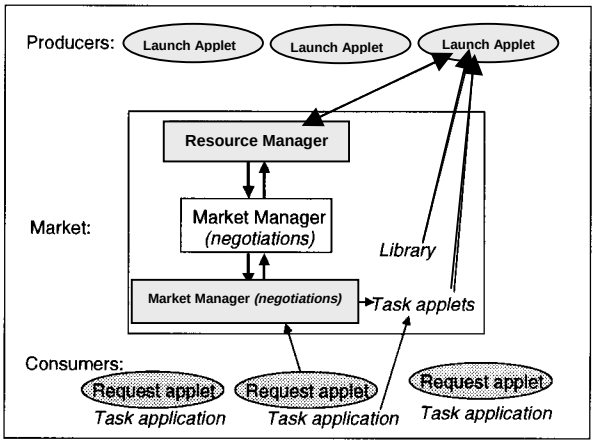

Figure 3. The Java Market architecture.
Source: Y. Amir, B. Awerbuch, and R. S. Borgstrom. The Java Market: Transforming the Internet into a Metacomputer. Department of Computer Science, The Johns Hopkins University, Technical Report CNDS-98-1.

block diagram of the Java Market architecture. The consumer jobs are written in Java so that they can be executed on any Java-enabled host. The Market is centralized; one Market accepts user jobs and host machines, and matches jobs with hosts. To register a job, a user would point their Java-enabled browser to a Java Market web page, and follow the instructions; likewise for a host system.

The Java Market takes a capitalistic approach to its scheduling policies. Unlike other metasystems in which hosts offer wasted CPU cycles for free, in Java Market, hosts are "rewarded" for participation. Users register their jobs and pay a "fee"; hosts register their services with a price tag attached. A host with a higher price tag would have desirable characteristics such as a fast CPU, high reliability, and high availability. Java Market matches user jobs with hosts and collects the difference between the host price and the user fee, with the goal of maximizing its profit. To prevent the Market from being an unscrupulous matchmaker (i.e. matching a high-paying user with a cheap host and pocketing the profits), the user states a job deadline that the Market must meet in order to receive the user's fee. The "fee" and "reward" need not be monetary; a Market operating within a corporate intranet might use an arbitrary token system to impose different cycle utilization limits on different divisions.

A unique feature of the Java Market is its user job placement strategy, also known as the "Winner Picking" strategy [5]. Given a user job and a set of compatible hosts, Java Market determines with a high degree of probability which host will stay alive for the duration of the job, and then places the job on that host. Specifically, suppose a user job takes d steps (where a "step" is an arbitrary uniform time unit) to complete, and at least one of the n hosts will be available for at least $D \geq (3\ d \log n)$ steps. Then, a host which will be alive for at least d steps can be selected with probability of at least $1 - O((d \log n) / D)$ [5]. This strategy can be employed even when D is not known, although in that case performance bound would be maintained over a larger set of jobs, and the number of jobs that may have to be killed and restarted increases to order ($\log n$).

2.4. Legion

Legion is an object-oriented metasystem that presents to the user a single virtual machine with the following objectives:

> site autonomy; an extensible core; scalability; easy-to-use seamless, computational, environment; high performance via parallelism; a single persistent object space; security for users and resource providers; resource management and exploitation of heterogeneity; multi-language support and interoperability; and fault tolerance [11].

Legion is designed to accommodate workstations, vector supercomputers, and parallel supercomputers connected by either local or wide area networks. It is built on a uniform programming model based on a core set of objects or mechanisms which users and resource providers can customize to meet their needs.

The Legion environment is viewed as a system comprised of objects that communicate with one another using function calls [11]. All objects belong to a class that is also considered to be an object. Furthermore, each class is responsible for implementing its own security and object placement policies using Legion-provided mechanisms. In addition, objects must create instances of their objects and maintain the location of these instances in case others would like to contact them. So, given that users have access to a set of objects within what is called a *context domain*, they can possibly add more objects provided they have the necessary authentication. However, in order for users to add a new object into their context domain, they would need to be able to locate it within the global name space.

In Legion, objects are identified by unique Legion Object Identifiers (LOIDs) [11]. A LOID can

contain up to $2^{16} - 1$ bytes of information such as its public key, class identifier, and an instance number of the class. In order to communicate with another object, an object binds the LOID to a low level object address that gives the location of the object (i.e. an IP address and port number). In order to get a binding, an object either contacts the object's class itself or contacts a Binding Agent who knows exactly where certain instances of objects are located. Each object is given a standard set of bindings when it is created so it can seek out all objects from there. Additionally, since a machine has limited resources, it is not possible to keep all instances of objects as active processes at all times. Therefore, Legion provides a mechanism to deactivate (i.e. put in secondary storage) and then reactivate an object.

By using a given set of core objects, users can build up their own environments and develop applications. One example of a core object is the host object, which basically guards access to a host machine and performs resource management. Another example is the vault object, which is a storage area for objects to go when they are deactivated. The user's context space is also a core object. Figure 4 shows an example of the relationship between core objects.

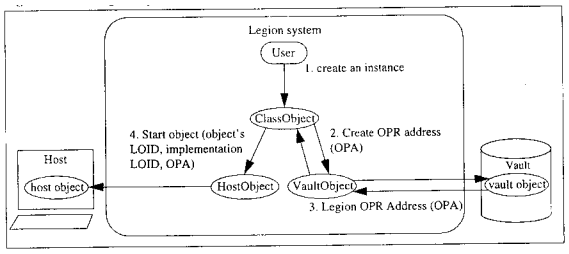

Figure 4. Legion object creation.
Source: Legion 1.4 System Administrator Manual, p. 31, November 9, 1998.

2.5. WebOS

WebOS seeks to exploit the large number of computers connected to the Internet as its distributed network of heterogeneous machines [15]. Client-side Java applets enable any authorized client with Internet access and a Java-capable browser to make use of the services provided by a variety of remote servers.

WebOS is designed to support a variety of different services over the Internet, such as chat services, news and weather updates, remote computing engines, and distributed web servers. These services range from the highly application specific, such as the chat services and the news and weather updates, to the more general, such as the remote computing engines and the distributed web servers. The types of services we consider here are similar to the WebOS services "Remote Compute Engine" and "Rent-A-Server" [15]. The "Remote Compute Engine" allows computing centers to permit access to their machines via the Internet. "Rent-A-Server" allows overloaded web servers to offload some of their web page and script requests temporarily onto other machines.

It is significant to note that in all the other metasystems considered here, there is a single set of developers writing the metasystem operating code. These developers are separate from the end users, which create their own programs on the metasystem. However, WebOS has separate developers writing the server system and the service code. Therefore there is a difference between the WebOS developers, who write the general architecture; the developers of the services that run on WebOS; and the end users, who run their own programs using these provided services. Because the service developer does not exist as a separate entity in the other projects, we will consider "users" to be the end users who run their own programs in the metasystem environment.

With this in mind, we summarize the user's view of the WebOS environment regardless of the application being run. Users first identify via a web browser a service they wish to use. The user then downloads a Smart Client from the location pointed to by the service hypertext.

Figure 5 depicts the Smart Client model. The Smart Client, when downloaded to the local machine, copies a client interface applet, a director applet, and a preliminary listing of servers that can possibly provide the requested service. The applets run entirely on the client's machine, though not

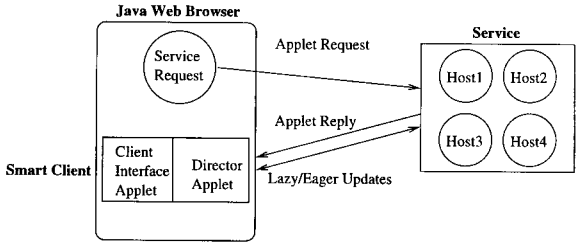

Figure 5. WebOS Smart Client service access model.
Source: C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. Proceedings of USENIX '97, January 1997.

necessarily in a web browser. The client interface applet provides the user interface via the web browser; the user issues commands and requests via this interface. The client interface applet provides a means of interaction specific to the particular service being provided. The client interface applet also makes requests to the server similar to remote procedure calls. The director applet handles all other communication to provide authentication, dynamically update its resources list, and process responses from the server. The primary listing of servers allows the director applet to "bootstrap" by identifying a potentially free server and beginning execution [16]. Because servers are accessed via the Internet, servers are simply named using their URLs and IP addresses.

The client-requested process runs in its own virtual machine on a remote server. The resource manager on the server is responsible for managing its own resources and turning away or warning new clients of possibly poor response time.

File accesses on remote servers are done using WebFS [15]. This is essentially a file system that uses the URL name space to identify files on remote servers. Ultimately the designers hope to provide IP multicast support in WebFS to allow for easier file updates and cache invalidations.

Authentication is done via the CRISIS security protocol. This is described in more detail in Section 3.3.1.

3. Analysis of Transparency

3.1. Resource Discovery

All the systems under consideration are in a state of flux. New machines come into the pool of known resources all the time; likewise, existing machines crash or become otherwise unusable. Because the machines attached to the network are in this state of flux, it is important to be able to discover whether resources have been added to or deleted from the known pool. The process of updating the status of the resource pool we call *resource discovery*.

The metasystem is responsible for knowing about all available resources as they are added to and deleted from the known pool. Whether or not the user is aware of resource pool updates is entirely system-dependent. For the most part, resource discovery mechanisms vary in user transparency via its side effects; that is, the user is aware that resource discovery is going on only by noticing changes in metasystem performance.

WebOS is unique from the other projects in that it can potentially affect processes on the local machine. All resource discovery is handled via interaction between the client-side director applet and the servers providing the requested service [15, 16]. If updating the local known resources list is unusually CPU-intensive, the client will be adversely affected by the resource discovery mechanism because the director thread will use a large fraction of the CPU time. However, the discovery of new resources allows migration of computation to less heavily loaded servers between calls to the server.

Java Market, like all of the other projects, benefits most from resource discovery when more machines can handle more work. Since processes in general cannot be migrated during execution, users will likely benefit the most if they frequently run processes on the Java Market. If more machines are available on subsequent runs, the user will probably see a performance increase as well as a decrease in cost for services. The computational cost of adding new resources has no bearing on the user as this is handled via the resource manager.

The Java Market resource manager itself has little overhead in discovering new resources, as resources are handled by a central database. Administrators wishing to add their machines to the market simply enter the necessary naming and resource information on a web page. The central database is updated automatically. The user does not see extra overhead incurred by having the resource manager handle the computational cost of adding a database entry.

Condor is similar to the Java Market in that currently running processes will not be able to take advantage of newly added resources until they are migrated. That is, currently running computations will not be shifted to the new machines unless local activity pushes the computations off the current machines. Users will therefore see the most improvement in performance if they run many processes or their processes migrate frequently.

Legion and Globus use a metasystem-side mechanism for handling resource discovery [11, 9]. Therefore the computation cost of maintaining the free resources is not imposed on the local machines. Unlike the rest of the projects, Legion and Globus can benefit directly from the addition of new resources; that is, both systems allow programs to

redistribute their workload dynamically when better resources are available.

3.2. Resource Management

We refer to the placement of user jobs on specific machines as *resource management*. The resource manager's action will directly affect the quality of service to the users. In general, a smart resource manager will schedule processes in such a way as to maximize use of all the resources at all times. In doing so, a good resource manager will both improve turnaround time of processes and increase the total number of possible users. A metasystem is most cost-effective when it can handle as many users and processes as possible as quickly as possible.

All the projects under consideration offer different levels of resource management transparency to the user. These levels range from total transparency, where the user has no involvement, to almost no transparency, where the user is allowed significant control of resource management. The involvement can include more than the initial matchmaking process, but also maintenance of running processes.

In WebOS, resource management does not come from a centralized resource manager. Instead, resource management arises from the interaction between the director applet and the servers providing the desired services [15, 16]. Therefore, in a good resource manager for WebOS, the director applet will use dynamic information to determine the least loaded server and pass this name up to the client interface applet. The servers will provide enough information about their current loads and capabilities for the director applets to make a good choice. Furthermore, the servers will be able to determine when their resources are fully loaded and be careful not to take on any more processes. Transparency in resource management comes from the idea that the director applet and the server working synchronously will always provide the client interface applet with the highest quality of service. In the case of WebOS, the client interface applet never knows what is being done to find the best machine; it only knows that the director applet somehow passes it the best machine name. Nevertheless, the client cannot otherwise choose which servers on which to run the remote program. It should also be noted that if the director applet is wasteful of CPU cycles, the resource manager is not transparent at all to the client.

Like WebOS, Java Market clients cannot explicitly choose the machine on which to run their processes. The resource manager selects the machine on which to run the process using the Winner algorithm [5], which gives a reasonably high probability that the selected machine will be sufficient to finish the process within the given time and without crashing. The design of the Java Market assumes the typical user will not care which machine is used as long as the job finishes in the specified running time.

The Java Market user can only affect which machine to use by changing the running time parameter and the amount of money paid to the market. If the client had some prior knowledge as to what machines may be in the market and running at the time of process submission, it would be possible to skew the input parameters to target a specific machine. However, the Java Market does not give clients information about the exact machines in the market and does not divulge the machine on which the process actually ran. There is no guarantee that skewing the input parameters will cause a client to receive a particular machine; the client can only guess as to the outcome.

Thus the Java Market is not completely transparent to the client. The client has a degree of control over how resources are allocated to their process. Nevertheless, the exact management mechanism remains invisible to the client.

In Condor, users have more flexibility in controlling how their jobs are matched to resources. When the user submits a job to the queue, a Classad will be generated for the user using parameters from the job submit script. The Classad includes important information such as memory, disk requirements, and architecture type [14]. The idle machines also send Classads detailing what each machine is capable of doing. The Central Manager uses a simple matchmaking algorithm in which the first fitting pair is considered. Both the requester and provider are notified after the initial match and negotiation is done between them without CM intervention [8]. If the negotiation fails due to circumstances such as the resources no longer being available, the request is resent to the CM and another match is attempted.

Condor not only shields the user from the matchmaking algorithm, it also prevents the user from participating in the maintenance of the process. When the process is required to move because the owner needs the resource, migration is done without intervention from the user and the owner. The guest process simply sends out another request Classad to the CM and another match is executed. The users

can, however, kill their jobs from the queue and query their jobs' status.

Both Legion and Globus are aware that a single algorithm can not effectively do matchmaking. Therefore both provide the means for the user to have control over how resources are managed. In Globus, resource management is done by the "broker" [7]; in Legion, this is done by the "mapper" [12].

Instead of a general permanent matchmaking algorithm, Globus and Legion have the ability to replace the broker or mapper, respectively. In Legion, the user is allowed to design and use a completely new algorithm in place of the default mapper. The user also defines an algorithm in Globus; however, Globus does not provide a default broker. This replaceable design is based on the assumption that an all-inclusive algorithm is not necessarily the most efficient way of doing resource allocation. The user writing the mapper or broker will probably have a clearer understanding of the details of the resources needed and thus be able to make smarter decisions.

Globus uses the Resource Specification Language (RSL) [7] to communicate the needs of the program to the broker. RSL is a simple language which combines parameters and conditions using operators such as **&** (conjunction) or **|** (disjunction). Legion also has a query language that communicates resource requests to the resource manager. Globus and Legion are more flexible than Condor because they are actively involved in the refinement of resource queries, whereas in Condor the user only interacts with the query once. By refinement, we mean taking a general query and breaking it into more specific requests to the information server.

Legion's and Globus' replaceable matchmaking algorithm, in conjunction with a flexible request language, gives the user control of basic resource management if it is desired. However, a consequence of this flexibility is the user loses some transparency. The user will need to know the interface to the resource database in order to write a mapper or broker. Even if the user decides to use the pre-made mapper or broker, the user still needs to learn a request language such as RSL; this is more complex than how requests are submitted to the other metasystems.

3.3. Security

There are several security issues in metasystems that affect the transparency of the system to the user. Some of the more important security issues are authentication, monitoring the user's job on host machines to prevent the job from operating outside its prescribed bounds, and the privacy of user data on foreign hosts. In this paper, we focus on the transparency level of our metasystems in two security issues: authentication, and monitoring user jobs on host systems. We omit discussion of the other security issues because they are not sufficiently addressed in current publications to be of use here.

3.3.1. Authentication

Authentication is described in [9] as "the process by which one entity verifies the identity of another". We define *authentication transparency* as the level of awareness a user has of a metasystem's underlying security mechanism. Authentication transparency varies from metasystem to metasystem. In some environments, a client may have to verify itself explicitly with two or more entities: the metasystem itself, and potentially one or more underlying host machines being used by the metasystem. In other environments where all user jobs are trusted, or where security breaches are almost impossible ("everything-is-bolted-down" environments), the user may not need to do any sort of explicit authentication at all. In this section, we highlight the features of metasystems, which promote authentication transparency, and those features that diminish it.

Java Market is designed to be used by anyone with access to the web and a Java-enabled web browser, so within Java Market there is no concept of "user authentication". All users are considered "trusted", because all user jobs must be written in Java, which is considered to be inherently secure. Also, because Java applets run only on Java Virtual Machines (JVMs), there are no authentication issues with the host machines. Therefore, Java Market's reliance on implicit Java security instead of some external mechanism promotes authentication-transparency. Note that the Java Market architects do not mention any secure credit transaction scheme for payment. Future addition of an authentication scheme for payment might affect transparency to the user; we do not consider the possible effects here.

Condor has a similar policy to Java Market in that there is no metasystem-wide "user authentication" performed; it is based solely on a user's local login to a machine. In other words, in order to submit jobs to a pool, the user only needs to be locally logged on to a machine that is a member of a Condor pool [8]. The idea behind this is that it is assumed the user will be the owner of that machine

and thus will be a mutually benevolent contributor to the Condor pool. Since the user does not need to do anything special to perform authentication with the Condor pool, we can describe Condor's interface as one that promotes authentication transparency.

The authentication interface in Globus needs the user to login just once per session with any one node of the metasystem [9]. This login is done using public key cryptography. From that point on, the metasystem handles authenticating the user on host systems. Once the user has logged on, the system creates a proxy, which negotiates access to resources on the users' behalf. The proxies can be mapped onto access credentials that the host machines use, e.g. Kerberos. If there are several security systems in place on a host machine, Globus also includes a negotiation algorithm, which lets the proxy interact with the host machine to settle upon a mutually agreeable security mechanism. Although not as authentication-transparent as Condor or Java Market (at least one metasystem login is required), Globus does encapsulate all the authentication interactions with underlying resources and host systems.

Authentication transparency in Legion is on a level similar to that of Globus. The Legion user is only required to do a single login to the metasystem [11]. Once the user's password is matched with valid user ID, a certificate is generated for the user by a local authentication entity (an object). This certificate is implicitly passed in all interactions with other objects in the system.

WebOS' authentication is done via its own CRISIS security architecture [6]. As in Legion, in WebOS a user will type in a password at a login domain. The login domain will authenticate the user through the user's home domain. The user's home domain will pass back the user's identification certificate and transfer certificates. A transfer certificate would give a user a subset of privileges from another user or machine. Users keep their identity and transfer certificates in a "purse". This similarity in login mechanism means that WebOS has a level of authentication transparency similar to that of Legion.

3.3.2. Behavior Monitoring

Metasystems are composed of multiple machines, usually operating in different administrative domains and not under the metasystem's direct control. An ideal metasystem would take steps to ensure that the user job respects the host, for example by preventing the user job from accessing private host resources without permission. We call this form of runtime checking *behavior monitoring*. Generally speaking, behavior monitoring and fault tolerance is similar because both are concerned with managing errors that might occur as a job executes on a metasystem. However, they are different on a lower level of abstraction because behavior monitoring policies try to prevent bad things from happening, while fault tolerance policies try to recover gracefully when bad things happen. This notion in hand, we describe *behavior-monitoring transparency* as the degree to which a user is aware of and/or affected by a metasystem's behavior monitoring policy. In a metasystem that encourages behavior-monitoring transparency, the user has very little (if any) awareness of the metasystem's efforts to keep a job within its bounds.

First, we consider Condor. Once a job is submitted and has been accepted by the idle machine, it can execute in any matter it wants subject to the restrictions of the Standard Universe, as described in Section 3.4. [2]. However, by the protocol of non-interference, a job would not be able to do anything malicious to the host. For example, if a program contained an infinite loop, it would not be able to consume CPU cycles indefinitely because the program would eventually be migrated when the owners came back to use their machines. On the other hand, the program would most likely be migrated to another machine and do the same thing there. Thus, protection to the host machine is actually a subset of the Condor's policy of non-interference, which is hidden from the user. We say that Condor's behavior monitoring policies promote behavior-monitoring transparency.

Java Market takes the more authoritarian approach of editing the user's source code to prevent the recompiled executable from violating its boundaries [4]. It also edits out potentially dangerous function calls, such as the ability of an applet to resurrect itself after it is killed, and forces the application to adhere to strict Java applet coding standards. This is done because the limitations of Java applets preclude a user process from doing a subset of security breaches: Java applets cannot write to a host's disk, nor can they establish network connections with other machines behind the host system's firewall [4]. Java Market's behavior monitoring policy requires no user intervention, so it encourages behavior-monitoring transparency. However, it is unclear whether the modifications to the user's code always produce results that the user originally expected. If these silent modifications to the user's job produce unintended results, then the

policy discourages behavior-monitoring transparency.

Globus monitors user jobs executing on host systems [9]. Specifically, when queried, the GRAM job manager will return the current state of a user job. However, Globus does not use these status reports to verify that the user job is executing within its prescribed bounds. Although it is possible for users to take the GRAM status reports and implement their own behavior monitoring policy, Globus by itself has no policy that encourages behavior-monitoring transparency.

WebOS also employs a runtime security checking scheme [6]. Whenever a request is made to a host, the request is either granted or denied based on the user's access level. Whenever users make requests to a remote site, their "purses" are implicitly transferred there. First the user and remote site mutually authenticate themselves via a trusted certificate authority (CA) and then the remote site checks the user's transfer certificates to ensure that the user has the appropriate permission to invoke the request. Basically, anyone who is considered to be trusted by the remote machine via his or her certificate may execute on that remote machine (i.e. no local login required). This "purse" scheme promotes behavior-monitoring transparency, because user processes are kept in check, and the user is unaware of the underlying mechanism that does so. Further, the user's job runs within a virtual machine that provides protection to the remote machine and is also transparent to the user.

Legion is similar to WebOS in that a job's access to the rest of the metasystem is controlled. Unlike WebOS, security in Legion is provided on an object basis [3]. Each object is guarded by a **`MayI()`** function which basically provided access control. The **`MayI()`** function is implicitly invoked whenever a user invokes a procedure of an object. The **`MayI()`** function can be implemented for each object and/or for each procedure within that object. The default implementation of **`MayI()`** is to only allow procedure calls from itself and other objects within its class. However, it can be further refined by using deny and allow access control lists. Because the **`MayI()`** call is implicit, we can say that Legion promotes behavior-monitoring transparency. An alternative security schema would be to use certificates, in which case the object would verify the signature of the certificate and attempt to match the name of the procedure being invoked with that contained within the certificate.

3.4. User Interface

In the context of metasystems, the term "user interface" is more inclusive than its common interpretation. An "interface" in this context can include a set of libraries, a coding convention, a programming language, a script, or the standard "GUI" and command-line interface. We define *UI-transparency* as the relative amount of modifications that a job must undergo in order to run on a particular metasystem. We posit that the less work a user must do to prepare a job for a metasystem, the less awareness the user has of the metasystem's underlying interface complexity, hence the greater UI-transparency of the system.

Java Market has a relatively simple user interface: users point their web browser to a Java Market web site, submit a job's Java source code to the Market "broker", then await the results via email. There are no explicit provisions for parallel execution. At best, a user can submit several jobs to the Market at the same time, and depending on resource availability, the jobs will execute in parallel. Java Market promotes UI-transparency because no effort is required from the user to prepare a job to run in the system. However, if the user's goal is parallel execution, Java Market's lack of an explicit parallel execution policy diminishes transparency.

In WebOS, the interface presented to the user is simply the client interface applet. This interface should abstract away the details of service invocation from the user.

For Condor, little interaction is needed to submit a job to the pool other than choosing the environment to execute in and writing a script to submit the job [2]. There are three types of execution environments in Condor. The first is the Standard Universe in which case Condor will provide automatic checkpointing and migrating of the user's job (discussed further in Section 3.5). This only requires that programs be relinked with Condor libraries. However, there are restrictions imposed on the type of jobs submitted such as disallowing multi-process jobs and IPC calls. The second environment, Vanilla, requires no modification of the executable, however neither automatic checkpointing nor migration is performed (i.e. the job's current state will be lost). Finally, the third type of environment allows execution of programs written in PVM. Therefore, all of these execution environments promote UI-transparency.

Condor uses a submit-description script to submit a job to the queue, therefore it is similar to using any other batch processing system [2]. The script specifies the job name to execute and its requirements and preferences. This information is then used to generate the job's Classad. Once submitted to the queue, a user can monitor or remove the job. Optionally, users can specify whether or not they want to be notified when their processed get migrated. When the job is completed, Condor will notify the user and give it statistics on how much CPU and wall clock time it took.

The Globus user interface is essentially the Globus Metacomputing Toolkit, a set of components, which provide security, resource location/management, communication, and other services necessary to make a job Globus-aware [9]. Developers can select services from the Toolkit in whichever combination that fulfills their needs. Globus claims that because its Toolkit consists of a modularly disjoint set of services, Globus customers need not rewrite their entire application to make it Globus-aware; incremental augmentation is possible [7]. While the toolkit "bag of services" interface aids in incrementally building Globus applications, the interface diminishes UI transparency by exposing underlying system components to the user. When building a Globus application, a user must explicitly include GRAM, MDS, HBM, GASS, and other packages.

Legion's user interface provides a similar level of transparency as Globus. In terms of creating an application, a Legion user can write a program in a high level language which is then compiled with a Legion-targeted compiler and linked with the Legion Runtime Library (LRTL) [11]. One high level language that Legion provides is the Mentat Programming Language (MPL) which is described as a parallel C++ language. The idea behind MPL is to simplify the construction of parallel programs by having the programmer specify the parallelism and letting the compiler take care of the details of implementing the parallelism. Additionally, the parallel computing tools Message Passing Interface (MPI) Parallel Virtual Machine (PVM) are available in C, C++, and Fortran. Also, Basic Fortran Support and Java are supported by Legion. It is also possible to execute existing "legacy" codes written in C, Fortran, and Ada with minimal changes. Users wishing to learn little about Legion may choose to use Legion's MPI or PVM to create their applications and thus not need to learn the details of the Legion environment. This promotes UI-transparency. On the other hand, more advanced users may want to learn MPL and become familiar with Legion's object-oriented framework. In this case, UI-transparency is diminished.

Legion's environment is different from those of the other systems in that once users have logged on, they are allowed to interact only within their context spaces [1]. This is organized as a Unix directory hierarchy. Contained within this hierarchy are all user-related host objects, files, executables, other context spaces, etc. The user can traverse their context space using Unix-like commands (e.g. **`legion_ls`**, **`legion_mv`**). Optionally, the user may navigate through their context space using a Java GUI called the Context Manager. In order to start up a job on a remote machine, the executable must first be registered. This will create an object to manage the program (i.e. if there are different executables for each environment on which the program will run). Even though this interface bears similarity to existing environments the user will still need to spend time learning how to use it, which diminishes UI-transparency.

3.5. Fault Tolerance

Metasystems are most useful to people who need to execute jobs that are too large to run on one machine, or jobs for which parallel execution is a natural choice. In general, executing a large, long-lived job brings up fault tolerance and recovery issues, even more so if the machine executing the job has multiple points of failure. If the completed sub-tasks of a large job cannot be saved, the sub-tasks can be lost in the event that their host system crashes. If this were to happen, the time saved by parallel execution would be lost by having to re-execute sub-tasks. In this section we focus on our metasystems' policies on managing failure among one or more host machines, and their features which promote or diminish *fault-tolerance transparency*, which we define as the degree to which the user is made aware of failures relating to their job.

Condor's fault tolerance policies promote fault tolerance transparency because errors are handled solely by the system (given that the user executes under the Standard Universe). The CM detects failures when it no longer receives updated status messages from the host machine. As is the case of security, fault detection and recovery are handled by the mechanisms provided for the non-interference policy with the owner, namely checkpointing and migration. Checkpointing a process involves periodic saves of the process state by writing to a file the process's data and stack

segments, file information, pending signals, and CPU state [13]. For Condors, the checkpoint file is sent back to the host that submitted it where it once again may be rescheduled. To resume a process on another machine, a new process is created and manipulated such that it attempts to restart at the same place as where it left off.

Similarly, in WebOS, error recovery would be provided by the service (i.e. via the director applet). The director applet could be implemented to send keep alive messages or use timeouts to determine failure of servers [15]. Additionally, the director applet could restart the job on another machine without the user being aware of it. Maintaining the statefulness or statelessness of a process would be dependent upon the director applet (i.e. it would depend upon the type of service being offered).

There are no provisions within Java Market to alert the user of host system failures. The Market stays in constant contact with all host systems, and if a host system disappears/crashes while executing a user's job, the job is restarted on a new machine. If no such machine is available, after some time the Java Market will return the job to the user and indicate that the job request failed.

Unlike its metasystem peers, Java Market claims that its job scheduling policy is clever enough to place jobs on hosts that will remain functional for the lifetime of the user's job, with a high degree of statistical probability [5]. Specifically, if at least k of n host systems will be available for D units of time, then Java Market claims that with probability at least $1 - O(1/n)$, they can schedule $(k \log n)$ jobs with lower bound of $\Omega (D / \log n)$, and those $(k \log n)$ jobs will complete. While this is a good statistical bound for performance, it is unclear how performance scales in the "real world".

Java Market attempts to reduce the number of faults which occur on active hosts by maximizing the probability that a job is assigned to host that will remain available long enough to complete the job. This scheduling policy promotes fault-tolerance transparency by minimizing the number of times Java Market would need to restart the user's job, an action which would not be performance-wise transparent to the user.

The Globus system monitors the status of its components using the Heartbeat Monitor (HBM) service. The HBM is designed to monitor the "health and status of a set of distributed processes" [9]. It consists of a client interface, and a data collection "status" set of services. The HBM client uses the interface to register with the object it wants to monitor, and the object in turn sends the client a regular "heartbeat". If too many heartbeats are missed, the HBM service attempts to determine the cause. The cause of error is returned to the process that established the HBM. Recovery is completely user-dependent: once the user is notified, it is up to the user to dictate what happens next. If the user chooses to implement a checkpoint policy, the process can be resumed from the last checkpoint; otherwise the user can restart the job. While Globus detect errors on host systems without user participation, it diminishes fault tolerance transparency by expecting the user to implement their own fault recovery policy.

Like Globus, Legion provides a less transparent fault tolerance interface to the user. It has a fault detection method, but not a fault recovery mechanism. In Legion, host faults would be detected through the regular checking for stale objects. An object is considered to be stale after a number of repeated failed communications [11]. Thus if a program attempts to access a host object and fails, the programmer would receive an error and would need to appropriately recover from it.

4. Recommendations

We have so far seen how the features in our considered systems promote or diminish transparency to the user. We now consider these features in the context of the novice and the expert user.

Our novice user in general requires a greater level of transparency. That is, the novice user would like simply to drop a program into the metasystem and collect the results in the end with as little modification of the existing code and as little understanding of the underlying system as possible.

On the other hand, the expert user will probably want less transparency than the novice user in order to manipulate the metasystem with finer granularity. The expert user is more likely to have a better understanding of the underlying system and has probably optimized their program to take advantage of various low-level features offered by components of the metasystem. Thus the expert user might want to be able to control program behavior in the system beneath the higher level abstractions presented to the novice user.

4.1. Resource Discovery

Resource discovery in the metasystems under consideration tends to happen in the background. As mentioned in Section 3.1, the user generally only sees side effects of the resource discovery algorithm. The resource discovery algorithm is likely to have two major side effects: first, program execution time will improve or degrade as resources are added or deleted from the known pool; second, program execution time will degrade if the resource discovery algorithm takes away substantial computing time from user programs. We have seen that all of the considered metasystems run resource discovery in the background. WebOS is the only metasystem that does not conceptually isolate the resource discovery mechanism from the computing resources used to execute the user programs. Thus we feel that for all of the metasystems being considered, the level of transparency in resource discovery should accommodate most novice and expert users.

4.2. Resource Management

As opposed to the case of resource discovery, expert users may wish to make resource management decisions. Expert users will want less transparency in order to run processes targeting certain machine properties. They would probably prefer to use either Globus or Legion, as these systems allow users to write their own matchmaking algorithm to decide where processes will run. On the hand, novice users might not necessarily want to target a specific environment and would therefore want more transparency. Novice users would prefer Condor, WebOS, or Java Market as these systems do not allow user-defined matchmaking. Condor simply matches a job request to the first resource that satisfies the job's constraints. WebOS selects machines based on which will give the fastest or highest-quality response. Java Market bases its decision based on profitability and approximate runtime.

4.3. Security

Both types of users would like to see a high level of transparency as far as user authentication is concerned. Authentication overhead generally comes during setup time and is not incurred continuously throughout program execution. The expert user probably would not optimize this stage considerably. The novice user sees this overhead as part of the setup time, which only slightly increases the total execution time. Thus neither user would benefit from seeing a multi-stage authentication process. We conclude that both users would not favor any one system in particular.

Because expert users would like fast recovery from program error, they would most likely appreciate a lower level of transparency in behavior monitoring. Therefore we feel that expert users might like Legion best because all behavior monitoring is done via a user-defined **`MayI()`** function; this scenario would give users the greatest control over how their program is being monitored. On the other hand, novice users might want a high level of transparency because they would not wish to be aware of the underlying security mechanisms of the host machines. Novice users might prefer Java Market or WebOS. The former guarantees a user program will not violate security on the host machine insofar as the Java Virtual Machine makes the host machine secure. The latter pushes any behavior monitoring duties to the director applet, away from the users' sight.

4.4. User Interface

The user interface has the potential to abstract away from or emphasize the underlying complexity of the metasystem. The more the user sees of the underlying system, the more variables that must be managed. We assume the expert user would like a more configurable and therefore less transparent user interface as it will provide a finer grain of control over program execution. We conclude that the expert user would prefer either Globus or Legion. Legion provides a lower-level interface if desired, whereas Globus provides the option of using a detailed toolkit or a higher level API around which to build a program.

We assume the novice user would prefer a less complex and therefore more transparent user interface to hide many of the details of program execution. Fewer details to manage mean less room for error and a flatter learning curve. In terms of user interface, the novice user would prefer Java Market, Globus MPI, Legion PVM or MPI, WebOS, or Condor. Java Market, Globus MPI, and Legion PVM or MPI allow the user to develop in a non-system specific high-level language, which eases the learning curve and allows the use of existing code and programming skills with few or no modifications. The client interface applet on WebOS, if written properly, will greatly simplify the user's point of view by hiding many of the underlying details of the metasystem. Likewise, once users have submitted their jobs to the Condor queue, execution is completely independent of user input.

4.5. Fault Tolerance

Transparency in fault tolerance determines the extent to which the user is aware that the system is attempting to detect and recover from error. An expert user would prefer a "medium" level of transparency. That is, if users feel that checkpointing is a worthwhile feature to include in their programs, then it should be provided in a highly transparent manner. Ideally, all code checkpointing should be done automatically by the metasystem if so desired; that is, the expert user would not have to think about exactly where to checkpoint code and think about how to resume programs precisely at the checkpoints. On the other hand, as far as process migration between machines is concerned, experts most likely would like to dictate where a stalled process moves. It is possible that resuming stalled processes on a fewer number of fast machines is more beneficial than separating these processes onto many slower machines. Thus we conclude that a system like Condor would be ideal for the expert user as regards to checkpointing, whereas Globus and Legion would be ideal as regards allowing user interaction in migrating processes.

On the other hand, novice users would like to see a high level of transparency overall. The novice is more likely concerned about program completion and correctness instead of the exact semantics of optimizing program execution. Condor would be best for the novice user as it can handle all checkpointing and migration independently of user input.

5. Conclusions

We have evaluated five research metasystems for user transparency. We have attempted to identify the concepts that are perhaps most significant to transparency as seen by the user, and evaluated how features of these metasystems fit these concepts. Furthermore, we have considered the effects of user transparency on both the expert and the novice metasystem user.

After evaluating the aforementioned metasystems, if we were to build a metasystem, we would adjust some of the features to match the profiles of our hypothetical users. The system we would build for the expert user would extend Globus or Legion to include an automatic checkpointing mechanism like the one in Condor. In Globus, this would mean adding a new service to the toolkit. In Legion, this would mean adding a new core object.

The metasystem we would build for the novice user would incorporate high-level, easy-to-use features, such as the features found in the metasystems which promote transparency. Java Market and WebOS provide the simplest interface without embellishment, while Globus and Legion would be appealing to the novice only if they are used together with their high-level tools, namely MPI and PVM. Additionally, we would give users the option of using Condor's automatic checkpointing and automatic migration features.

Creating a single system to accommodate both types of users would require extra work to provide both the higher-level and the lower-level abstractions for each user type. The idea is we would want to keep the model simple for the novice user without compromising functionality, yet allow a way for the expert user to access lower-level abstractions easily. We feel that Legion and Globus move in the right direction with regard to this goal; however, creating and executing a job is ultimately not as simple as we would like for either user profile.

6. Future Work

This paper is only intended as an introductory look at transparency in five research metasystem environments. The next step would be to conduct a more comprehensive analysis of the metasystems. Ultimately, the evaluations would be presented in such a way as to create a more complete comparison guide for all metasystems users in real-world environments.

A more comprehensive analysis would look at varying degrees of transparency as opposed to the extremes presented here. The systems under consideration in fact have *windows* of transparency, as depicted in Figure 6. That is, the systems accommodate varying subsets of the user population. Systems like Globus and Legion have wide windows of transparency as they can conform to fit the requirements of many types of users. Systems like Java Market have narrow windows of transparency

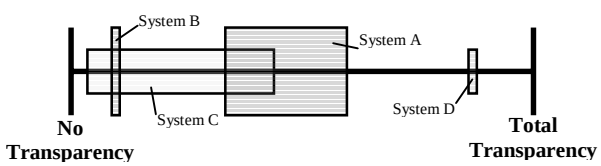

Figure 6. Windows of Transparency.

because they offer limited ways to take advantage of the underlying resources.

A more comprehensive analysis would also compare the degree of transparency with the performance extracted from the system. As previously discussed, transparency is not the only metric for determining the usability of a metasystem. A more comprehensive comparison would evaluate the tradeoffs involved in using a less or more transparent system.

7. Acknowledgements

We wish to thank Amin Vahdat of the University of California, Berkeley, for his help with the WebOS project; Ryan Sean Borgstrom of the Johns Hopkins University for his insight regarding Java Market; Walfredo Cirne for reviewing our drafts and suggesting the window of transparency idea; and, last but not least, Chez Bob for helping us through our spirited discussions.

8. References

[1] "Getting Started". Legion 1.4 Basic User Manual, pp. 7-9 and 10-20, November 9, 1998.

[2] "Job Preparation". Condor Version 6.0.3 Manual, Section 2.5. Found at http://www.cs.wisc.edu/condor/manual/ref1/node16.html#SECTION00350000000000000000, May 22, 1998.

[3] "Legion Security". Legion 1.4 System Administrator Manual, pp. 21-28, November 9, 1998.

[4] Y. Amir, B. Awerbuch, and R. S. Borgstrom. The Java Market: Transforming the Internet into a Metacomputer. Department of Computer Science, The Johns Hopkins University, Technical Report CNDS-98-1.

[5] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* , pp. 519-530, 1996.

[6] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS Wide Area Security Architecture. Proceedings of the 1998 USENIX Security Symposium, January 1998.

[7] K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, S. Tuecke. A Resource Management Architecture for Metacomputing Systems. *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,*1998

[8] 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 Volume 12, 1996.

[9] I. Foster and C. Kesselman. The Globus Project: A Status Report**.** *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pp. 4-18, 1998.

[10] I. Foster, G. von Laszewski. Usage of LDAP in Globus. Unpublished work, found at http://www-fp.globus.org/documentation/papers.html.

[11] A. S. Grimshaw, M. J. Lewis, A. J. Ferrari, and J. F. Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems. UVa CS Technical Report CS-98-12, June 3, 1998.

[12] J. F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. UVa CS Technical Report CS-96-03, January 16, 1996.

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

[14] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28-31, 1998.

[15] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. To appear in the *Seventh Symposium on High Performance Distributed Computing*, July 1998.

[16] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. Proceedings of USENIX ’97, January 1997.