Baglets: Adding Hierarchical Scheduling to Aglets

Arvind Gopalan, Sajid Saleem, Matthias Martin, and Daniel Andresen
Department of Computing and Information Sciences
234 Nichols Hall
Kansas State University
Manhattan, KS 66506
{*gxa2007,sajid,mattmar,dan*}*@cis.ksu.edu*

Abstract

A significant number of new Java-based technologies for mobile code (aka agents) have recently emerged. The 'Aglets' system, from IBM's research labs, provides an elegant mechanism for creating mobile code, but lacks a native scheduling mechanism for determining where code should be executed. In this paper we present the results of an investigation into adding sophisticated scheduling capabilities to Aglets (which we refer to as brilliant Aglets, or Baglets) from the H-SWEB project, which provides hierarchical scheduling across sets of WWW server clusters. H-SWEB uses scheduling techniques in monitoring and adapting to workload variation at distributed server clusters for supporting distributed computation. We show how the two systems can be integrated, and present several algorithms indicating a major advantage (over 350%) can be achieved through the use of dynamic scheduling information. We provide a detailed discussion of our system architecture and implementation, and briefly summarize the experimental results which have been achieved.

1. Introduction

The Aglets Framework provides a mechanism for creating mobile agents. Mobile agents are "programs that can be dispatched from one computer and transported to a remote computer for execution" [10]. H-SWEB is a hierarchical scheduler for distributed HTTP server clusters [2]. H-SWEB allows dynamic scheduling of HTTP requests on resources distributed across the Internet utilizing a distributed set of servers, set up in hierarchical clusters. We propose to combine the H-SWEB and Aglets technologies so that Aglets may be scheduled in an intelligent and flexible fashion. This scheduling ability is absent in the current Aglet system. Each Aglet is only aware of its own state; it has no information with which to make scheduling decisions.

The paper is organized as follows: Section 1 presents Aglets and H-SWEB. Section 2 discusses the design and integration of H-SWEB and Aglets. Section 3 presents the experimental results, and Section 4 discusses related work and conclusions.

1.1 Aglets

Aglets are Java objects that can move from one host to another. It is possible for them to halt execution, dispatch to a remote host, and re-start executing again by presenting their credentials and obtaining access to local services and data. Aglets provide "a uniform paradigm for distributed object computing" [10].

Characteristics of Aglets include:

- **Object-Passing**: When a mobile agent moves, the whole object is passed; that is, its code, data, state, and travel itinerary are passed together.
- **Autonomous Execution**: The mobile agent contains sufficient information to decide what to do, where to go, and when to go.
- **Asynchronous**: The mobile agent has its own thread of execution and can execute asynchronously.
- **Local Interaction**: The mobile agent interacts with other mobile agents or stationary objects locally. If needed, it can dispatch messenger agents or surrogate agents, which are all mobile agents, to facilitate remote interaction.
- **Disconnected Operation**: The mobile agent can perform its tasks whether the network connection is open

or closed. If the network connection is closed and it needs to move, it can wait until the connection is reopened.

- **Parallel Execution**: More than one mobile agent can be dispatched to different sites to perform tasks in parallel [10].

Using Aglets can ease the development of a system for distributed computationally intensive processes through the automation of code and data transport. Aglets utilize a protocol called ATP (Aglet Transfer Protocol), which is an application-level standard protocol for distributed agent information systems which operates on top of TCP.

Aglets, on being dispatched, serialize their internal state and byte code into the standard form, and then are transported to the destination. On the receiver side, the Java object is reconstructed according to the data received from the origin, and a new thread is allocated and executed.

Aglets API The Aglets API defines methods to control mobility, life cycle, travel, itinerary, and security.

- *onCreation()* Initializes the new Aglet. This method is called only once in the life cycle of an Aglet.
- *onDispatching()* Called before an Aglet is actually dispatched.
- *onArrival()* Called after the arrival at the destination.
- *onReverting()* Called just before an Aglet is retracted.
- *dispose()* Disposes of the Aglet.
- *dispatch(URL)* Dispatch Aglet to a destination specified by the URL.
- *getAgletInfo()* Get information on the Aglet.

1.2 H-SWEB

H-SWEB provides a means by which requests can be dynamically scheduled across clusters of servers, optimizing the use of client resources as well as the scattered server nodes [2]. H-SWEB uses a cluster of servers, and can dynamically schedule a request across a group of nodes and between clusters of nodes. The H-SWEB Cluster Server (HCS) is an entity that runs as a separate process, and may reside on one of the cluster nodes, or on a separate machine. It gathers load and cache information about the entire cluster, as well as the information about the bandwidth and latency between it and other clusters. It then uses this information to make scheduling decisions.

H-SWEB API The following are some of the API functions that were utilized for our project:

- *(HcsRmi)Naming.lookup("rmi://"+hcs+"/" +"HSWEB")*: Looks up the appropriate HCS server
- *Request(Vector cpuCycles, Vector outBytes, Vector timePenalty, float clientCpu, float clientDisk, Hashtable clientLatency, Hashtable clientNet, double localTime, String url)*

  cpuCycles Predicted CPU requirements for each split point

  outBytes Predicted output bytes for each split point

  timePenalty Predicted time penalty for each split point

  clientCpu Client CPU available in Hz

  clientDisk Client disk available in bytes/sec

  clientLatency Hashtable where the keys are cluster names and the values are the corresponding latencies between the clusters and the client in milliseconds

  clientNet Hashtable where the keys are cluster names and the values are the corresponding throughput between the clusters and the client in bytes/sec

  localTime - The time it will take to complete the request locally

  uri Requested URI

- *rmiChooseCluster(Request req)*: Choose the best cluster in the supercluster of server clusters
- *RemoteClusterInfo(String,URL,int,int,String,String, File,Hcs)* : Retrieves information from a remote cluster regarding the load information on the remote cluster. The information is received in the form of a compressed file to minimize overhead, and is used for load calculations and comparisons.

2 Integrating Aglets and H-SWEB

In Figure 1, we see two clusters which are exchanging their load and cache information. This exchange takes place between HCS of respective clusters. There is also a flow of load and cache information between the nodes within the cluster and the HCS for that particular cluster.

We utilize H-SWEB to improve the decision making of Aglets so that they become capable of executing code on the most favorable node that is part of the super-cluster of machines. Every node which is part of this configuration will have an Aglets daemon running on it. This will enable it to accept/dispatch Aglets that might be coming in, or

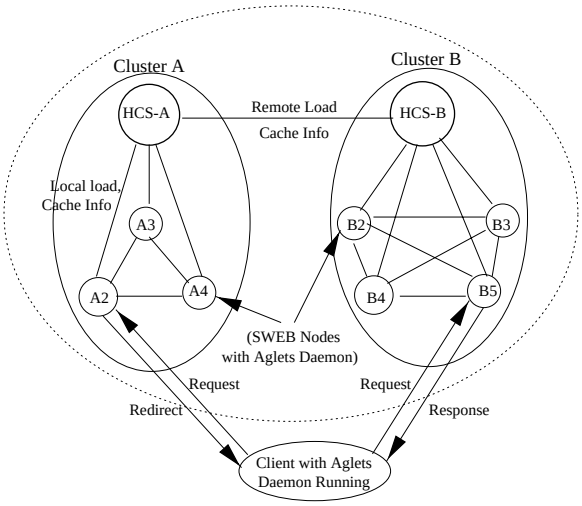

Figure 1. H-SWEB Architecture

need to be sent out. The API of Aglets is well suited for such a situation. It provides easy to use mechanisms for serializing and moving objects from place to place.

One primary design goal was to make the system as easy to use as possible. Thus we wanted to hide as many details from a user of the system as possible. To that end, we constructed a base class called **SwebAglet** which takes care of the details of making the RMI calls to the HCS as well as moving the Aglet from place to place. This class extends from Aglet, and contains the methods *onCreation()* and *signalCheckPoint()* and the abstract methods *estimateCpu()* and *exec()*. A user simply need extend from **SwebAglet**, provide implementations for the two abstract methods and call *signalCheckpoint()* when appropriate to utilize the full scheduling functionality.

The skeleton of the **SwebAglet.java** Java class is as follows:

SwebAglet extends Aglet

- *onCreation()* An **ExecMobilityListener** is added to listen for arrival events (other mobility events are ignored).
- *signalCheckpoint()* First, **java.rmi.Naming**.*Lookup()* is called to obtain a reference to the HCS for this cluster. Then a **Request** object is built and sent to the HCS. The HCS responds with the best cluster and node on which to perform the computation. The Aglet them moves itself if necessary.
- *exec()* Abstract method. Should be overridden by subclasses to contain the code that should be executed by the Aglet. This is similar to the *run()* method within **java.lang.Thread**.
- *estimateCpu()* Abstract method. Should be overridden by subclasses to contain code that estimates the amount of cpu cycles necessary to complete the computation specified by *exec*.

The base class by itself, however, is not sufficient. The base Aglet class has no way of detecting when it is moved from place to place. Instead the Aglet API provides a **MobilityListener** interface. The **MobilityListener** interface specifies that implementing classes provide methods to handle **MobilityEvent**s. **MobilityEvent**s occur whenever an Aglet is dispatched, reverted or arrives at a destination. The implementing objects are attached to Aglet objects at runtime (in our case in the *onCreation()* method). Whenever so called **MobilityEvent**s occur, the appropriate method is called in the attached **MobilityListener**.

In this case, we are only interested in detecting arrival events. We constructed the **ExecMobilityListener** class to detect when the Aglet arrives at a new node. The skeleton of the **ExecMobilityListener** Java class is as follows:

SwebMobilityListener extends MobilityListener

- *onArrival(MobilityEvent)* Call the *exec()* method of the **SwebAglet** to which it is attached to start or restart the computation.
- *onDispatching(MobilityEvent)* Empty. Provided only to satisfy the interface.
- *onReverting(MobilityEvent)* Empty. Provided only to satisfy the interface.

In Figure 2 we see the flow of information between the components in the system. For our experiments the Network Weather Service [12] was not used since all clusters were local to our location; bandwidth and latency were relatively constant over our switched, 100Base-T network. Significantly varying bandwidth and/or latencies would tend to increase the value of dynamic scheduling [2, 3]. We can see that the load, network, and request information flow into the HCS, allowing it to make intelligent scheduling suggestions regarding optimal load placement. This information is then passed on to the Aglets so that they may move themselves accordingly. The name of the initial cluster server is initially supplied by an external mechanism (e.g., command-line option).

For a full description of the H-SWEB scheduling algorithm, see [2]. In brief, for each request, H-SWEB predicts

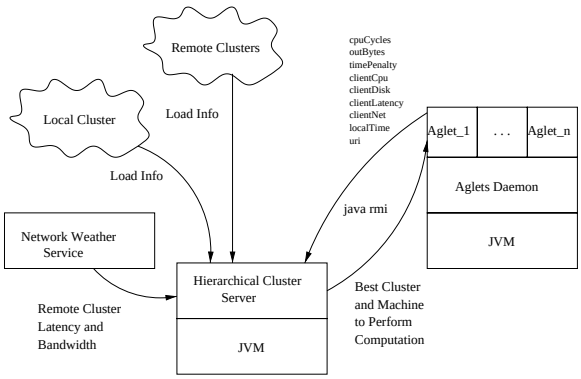

Figure 2. Block Diagram of Interaction between System Components

the processing time and assigns this request to an appropriate processor within our set of clusters. Our cost model (Equation 2) for a request takes into account the cost to move the Aglet from one cluster to another, the relative loads, and the costs to fetch any data required.

Fault tolerance is not supplied by the H-SWEB system, which only provides scheduling services.

$$t_s = t_{aglet\_movement} + t_{data} + t_{server} + t_{net} + t_{client}.$$

The Aglet requesting the scheduling services is expected to supply the necessary parameters to the H-SWEB processing model. It is assumed there is no actual 'client', so all terms reduce to zero related to client processing time. $t_{aglet\_movement}$ is the cost to move the Aglet to another processor, if required. t_{data} is the server time to transfer the required data from a server disk drive. Typically this would be 0 under current security models. t_{server} is the time for required server computation. In the implementation, we need to collect three types of dynamic load information: CPU, disk, and network. CPU and disk activity can be derived from the Unix *rstat* utility, as well as some network information. Latency and bandwidth information between clusters is provided by the Network Weather Service, while a custom daemon is used within each cluster. A daemon ($loadd$) periodically updates the above information between the server nodes. An incorrect estimate of Aglet CPU needs, for example, might cause a temporary error in scheduling. Since the system relies on actual system load information for its scheduling of other Aglets, however, they would be affected for only a short time while the effects of the rogue Aglet were being detected and accounted for.

2.1 Primes Example

We present an example of how to turn a plain Aglet into a H-SWEB-aware Aglet. In this case the Aglet computes prime numbers. The bulk of the functionality is obtained by extending the SwebAglet class (itself a subclass of Aglet). The following are the steps necessary to convert a plain Aglet to a H-SWEB-aware Aglet

1. In a normal Aglet, the code that is to be executed is placed inside a *run()* method similar to what is done with Java Threads. To avoid conflict with that name, code to be executed in this case is placed in an *exec()* method.

2. Provide an implementation of the abstract method *estimateCpu()*. This should be an estimation of how long a given computation should take. The more accurate this estimate is, the better the scheduling will be.

3. Add periodic calls to *signalCheckpoint()*. We found that making this call at somewhat random intervals helped prevent flooding the HCS with Java RMI calls. An alternative, currently unimplemented method would allow HCS to asynchronously notify Aglets when conditions are favorable for them to move.

4. Since Java serialization is the method used for transferring the Aglet from place to place, care must be taken in the handling of variables. All variables must be class (global) variables because only class variables are saved during the serialization process. In addition, the exec method must be structured in such a way that when *signalCheckpoint()* is called, the object is in a consistent state. Here this is done by calling *signalCheckpoint* at the top of the loop, thus preserving the atomicity of each loop iteration.

3 Experimental results

We utilized the small (10KB) "Primes" Aglet to compare different scheduling mechanisms. The aim was to find out which of the different scheduling mechanisms for these Aglets actually resulted in the maximum advantage, as far as response time was concerned. The different scheduling mechanisms chosen for evaluation were the following:

- **Static Scheduling**
  In this type of scheduling, all Aglets are sent to the same node within a cluster. Upon arrival at this node, the HCS is queried to determine the best node on which to perform the computation. If necessary, the

Aglet moves to that location. No further scheduling is done, and the *exec* method runs to completion.

- **Dynamic Scheduling**
  Under Dynamic Scheduling, the Aglets are again all sent to the same node with the cluster and initially scheduled. Then, the HCS is periodically queried at *checkpoints* in the code to determine if the Aglet needs to move. If so, the Aglet is then dispatched to the new location. The potential problem with such an approach is that a "wave" of Aglets could migrate off to a remote site if all of them query the HCS and find that a lightly loaded machine is available. To combat this problem, the program which contains the computational code was written in such a way that, the Aglets perform checkpointing at random intervals between twenty and forty seconds.

- **Round Robin Scheduling**
  Here Aglets are sent one after the other in a round robin fashion to the nodes that constitute the cluster. Once these Aglets are sent, they do not query the HCS to find out if any further scheduling is required. This scheduling method was only included for comparison purposes, and is equivalent to running the Aglets without HCS.

- **Round Robin Dynamic Scheduling**
  This is a combination of round-robin and dynamic scheduling. The client sends off the Aglets to all the machines in the cluster in a round-robin fashion. Then the Aglets query the HCS at the code at *checkpoints* specified by the user.

The configuration for testing was one cluster with three nodes. The nodes consisted of one lightly loaded 85 MHz SPARC 5, one heavily loaded 140 MHz UltraSPARC, and one lightly loaded 269 MHz UltraSPARC 5. Each Aglet (or "request") consists of the program to calculate the first n prime numbers, where n is passed in as a parameter. Figure 3 shows the number of homogeneous requests for round-robin and round-robin dynamic scheduling vs. response time. Each Aglet calculated the first 50,000 primes. It can be seen that the response times for round-robin scheduling become larger at a much faster rate than for round-robin dynamic scheduling. The percentage improvements ranged from 12% to 89%, averaging 46%.

Figure 4 graphs heterogeneous requests. Round-robin scheduling performed fairly well initially, because the number of requests was quite low. As the number of requests increases, then the performance of round-robin scheduling becomes very poor because the Aglets can never get rescheduled to take advantage of less loaded nodes. This difference was especially pronounced in this data set because one of the machines in the cluster was heavily loaded during testing. The static scheduling line only extends to 30 because we were unable to successfully get numbers for forty Aglets under that configuration. The Aglets daemon always core dumped. Again dynamic round-robin showed its worth, peaking at a 354% improvement over round robin, indicating a slowdown under light loads due to the overhead of rescheduling, and averaging a 128% improvement.

The following points were observed during the implementation of the different scheduling mechanisms:

1. Round-robin dynamic scheduling did very well for both homogeneous and heterogeneous requests.
2. The use of Java 1.2 for running HCS helped to a great extent. This was because HCS was actually suffering from severe memory leaks while running with Java 1.1.6. The memory requirements for HCS, as observed with *top*, grew from about 12 MB (9 MB resident) when the process is started to 47 MB (20 MB resident) over the course of ten minutes. Under Java 1.2, the figure was constant at 10 MB (9 MB resident). We feel this was an artifact of the JVM being used.
3. RMI calls to the HCS are a bottleneck to the performance of the system, as well as a source of scalability problems. If a large number of Aglets query the HCS at the same time to find out which node they need to move to, it results in a degradation of performance, or the JVM on which the HCS is running crashes with a segmentation fault. We believe the segmentation fault to be occurring somewhere in the underlying communications structure.

4 Conclusion and Related Work

In this paper we have presented a design for integrating a dynamic multi-faceted hierarchical scheduling system with the Aglets mobile code system. We have implemented this design, and give experimental results indicating a substantial speedup (over 375% in some cases) is achieved over blind scheduling mechanisms such as round-robin in a heterogeneous environment. We have also demonstrated hierarchical multiple-cluster operation, and feel this substantially improves the ultimate scalability of our approach. In the future we plan to characterize the system and its bottlenecks more thoroughly, and explore its applicability to differing problems. To the best of our knowledge, this is the first hierarchical dynamic scheduling system across multiple server clusters for Aglets.

In the future we hope to further characterize the costs of moving Aglets (currently sitting at approximately 1,500

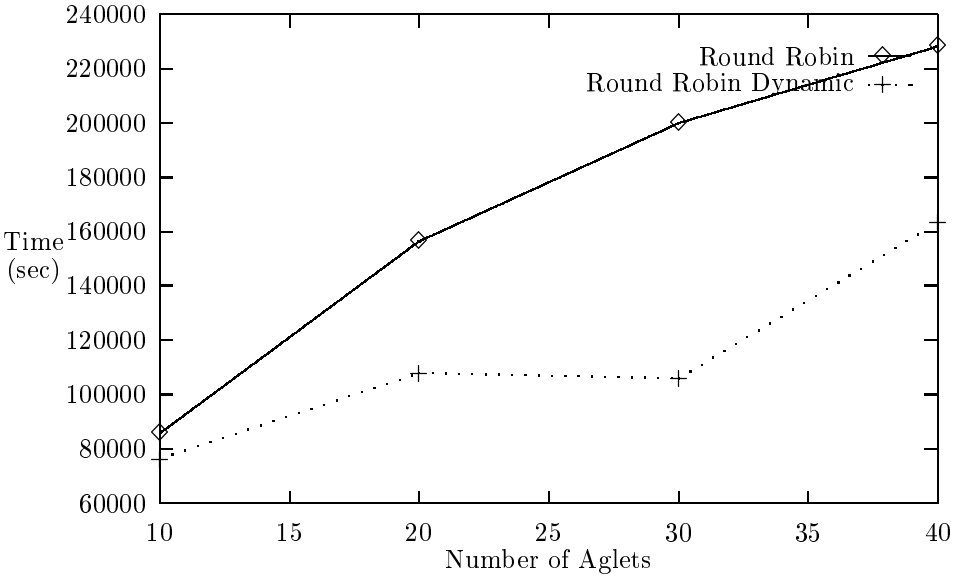

Figure 3. Large Homogeneous Requests. $n = 50,000$

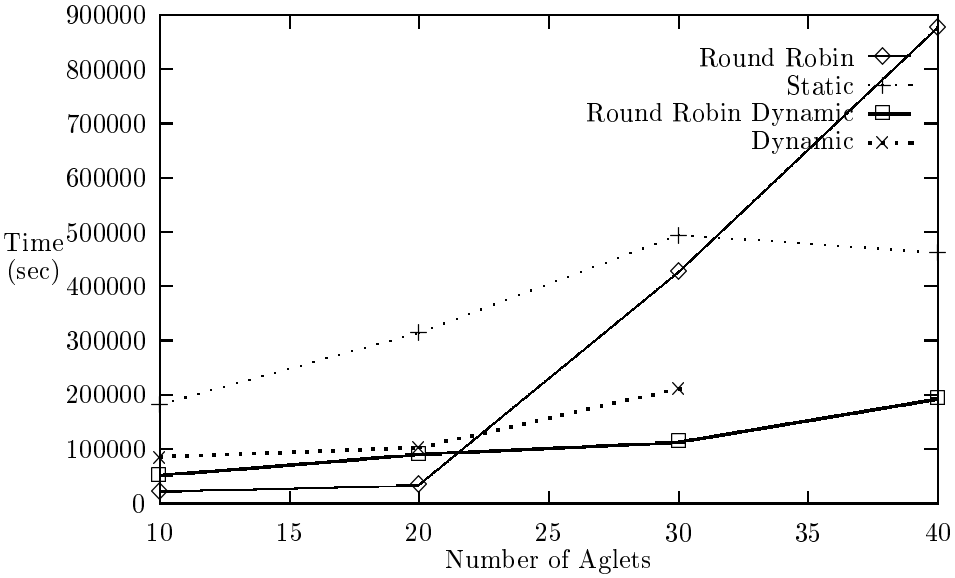

Figure 4. Large Heterogeneous Requests. $n = 10,000 - 50,000$

ms. for the prime number generator, primarily due to Java object serialization costs) factored by Aglet size and complexity. We have also been experimenting with multi-cluster installations, and hope to provide significantly more experimental data in this area. We are also examining ways to reduce the overhead imposed by the HCS system (currently 10-20ms. per call) and explore ways to avoid HCS-induced 'hotspots.' A technique to reduce the possibility of Aglets overloading the HCS scheduler through asynchronously notifying Aglets when conditions have changed is also under consideration.

Other systems have explored runtime load balancing within a distributed system, although few within the context of Java-based mobile computation environments. The Plangent project adds scheduling capabilities to Aglets, but puts the intelligence into each Aglet instance rather than having a single instance of the scheduler per cluster [11]. The Legion project provides a distributed load-balancing environment, at the cost of rewriting applications to fit the project libraries and languages [9]. The Globus project also aims at providing a global metacomputing object-oriented environment, but is oriented towards a different set of problems than Aglets + H-SWEB and is non-Java based [8]. Projects in [5, 7, 6] are working on global computing software infrastructures, but are generally still in the design stage. Scheduling issues in heterogeneous computing for large-scale scientific applications using network bandwidth and load information are addressed in the non-Java-based AppLeS project [4]. Recent work on mobile computing (e.g. [1]) also uses bandwidth and data locality information to dynamically migrate computation, but typically require greater much greater effort to integrate with the client code than the changes we require to an Aglet. We also provide a hierarchical scheduling environment, aiding scalability.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–??, 1997.

[2] D. Andresen and T. McCune. Towards a hierarchical scheduling system for distributed www server clusters. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing(HPDC7)*, pages 301–309, Chicago, Illinois, July 1998.

[3] D. Andresen and T. Yang. Multiprocessor scheduling with client resources to improve the response time of WWW applications. In *Proceedings of the 11th ACM/SIGARCH Conference on Supercomputing (ICS'97)*, Vienna, Austria, July 1997.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[5] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[6] B. Christiansen, P. Cappello, M. F. Ionescu, M. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 1997. to appear.

[7] K. Dincer and G. Fox. Building a world-wide virtual machine based on Web and HPCC technologies. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, June 08 1994.

[10] D. B. Lange and D. T. Chang. Programming mobile agents in java–a white paper. Technical report, IBM, Sept. 1997.

[11] A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. PLANGENT: An approach to making mobile agents intelligent. *IEEE Internet Computing*, 1(4):50–57, 1997.

[12] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, April 1998.