An Open Market-Based Architecture for Distributed Computing

Spyros Lalis and Alexandros Karipidis

Computer Science Dept.,
University of Crete, Hellas
{lalis,karipid}@csd.uoc.gr
Institute of Computer Science,
Foundation for Research and Technology, Hellas
{lalis,karipid}@ics.forth.gr

Abstract. One of the challenges in large scale distributed computing is to utilize the thousands of idle personal computers. In this paper, we present a system that enables users to effortlessly and safely export their machines in a global market of processing capacity. Efficient resource allocation is performed based on statistical machine profiles and leases are used to promote dynamic task placement. The basic programming primitives of the system can be extended to develop class hierarchies which support different distributed computing paradigms. Due to the object-oriented structuring of code, developing a distributed computation can be as simple as implementing a few methods.

1 Introduction

The growth of the Internet has provided us with the largest network of interconnected computers in history. As off-the-shelf hardware becomes faster and gains Internet access, the network's processing capacity will continue increasing. Many of these systems are often under-utilized, a fact accentuated by the globe's geography since "busy" hours in one time-zone tend to be "idle" hours in another. Distributing computations over the Internet is thus very appealing.

However, several issues must be resolved for this to be feasible. The obstacle of platform heterogeneity must be overcome and security problems arising from the execution of code from untrusted parties must be confronted. Further inconveniences arise when installing and maintaining the corresponding programming environments. And then, distributed computations must be designed and implemented on top of them, a challenging task even for experienced programmers.

In this paper we present a system that addresses these problems, simplifying distributed computing over the Internet considerably. Through a maintenance-free, web-based user interface any machine can be safely connected to the system to act as a host for remote computations. A framework that promotes code reuse and incremental development through object-oriented extensions is offered to the application programmer. Writing computations for the system can be as trivial as implementing a few routines. We feel that the ease of deploying the system and developing applications for it is of importance to the scientific community since most of the programming is done by scientists themselves with little or no support from computer experts.

The rest of the paper is organized as follows. Section 2 summarizes the general properties of the system. Details about the resource allocation mechanism are given in Sect. 3. In Sect. 4 we look into the system architecture, giving a description of components and communication mechanisms. In Sect. 5 we show how our system can be used to develop distributed computations in a straightforward way. A comparison with related work is given in Sect. 6. Section 7 discusses the advantages of our approach. Finally, future directions of this work are mentioned in the last section.

2 System Properties

When designing the system, the most important goal was to achieve a level of simplicity that would make it popular both to programmers and owners of lightweight host machines, most notably PCs. Ease of host registration was thus considered a key issue. Safety barriers to shield hosts from malicious behavior of foreign code were also required. Portability and inter-operability was needed to maximize the number of host platforms that can be utilized. A simple yet powerful programming environment was called for to facilitate the distribution of computations over the Internet. All these features had to be accompanied by a dynamic and efficient mechanism for allocating resources to applications without requiring significant effort from the programmer.

In order to guarantee maximal cross-platform operability the system was implemented in Java. Due to Java's large scale deployment, the system can span across many architectures and operating systems. Host participation is encouraged via a web based interface, which installs a Java applet on the host machine. This accommodates the need for a user friendly interface, as users are accustomed to using web browsers. Furthermore, the security manager installed in Java enabled browsers is a widely trusted firewall, protecting hosts from downloaded programs. Finally, due to the applet mechanism, no administration nor maintenance is required at the host – the majority of users already has a recent version of a web browser installed on their machines.

On the client side we provide an open, extensible architecture for developing distributed applications. Basic primitives are provided which can in turn be used to implement diverse, specialized processing models. Through such models it is possible to hide the internals of the system and/or provide advanced programming support in order to simplify application development.

3 Resource Allocation

Host allocation is based on profiles, which are created by periodically benchmarking each host. A credit based [1] mechanism is used for charging. Credit can be translated into anything that makes sense in the context where the system is deployed. Within a non-profit institution, it may represent time units to facilitate quotas. Service-oriented organizations could charge clients for using hosts by converting credit to actual currency.

Both hosts (sellers) and clients (buyers) submit orders to a market, specifying their actual and desired machine profile respectively. The parameters of an order are listed in table 1. The *performance vectors* include the host's mean score and variance for a set of benchmarks over key performance characteristics such as integer and floating point arithmetic, network connection speed to the market server etc. The host *abort ratio* is the ratio of computations killed by the host versus computations initiated on that host (a "kill" happens when a host abruptly leaves the market). The host performance vectors and abort ratio are automatically produced by the system. Host profiles can easily be extended to include additional information that could be of importance for host selection.

Table 1. Parameters specified in orders

	Description	
Parameter	**Sell Orders**	**Buy Orders**
price/sec	The minimum amount of credit required per second of use of the host.	The maximum amount of credit offered per second of use of the host.
lease duration	The maximum amount of usage time without renegotiation.	The minimum amount of usage time without renegotiation.
granted/demanded compensation	Credit granted/demanded for not honoring the lease duration.	
performance statistics vectors	The host's average score and variance for each of the benchmarks (measured).	The average performance score and variance a buyer is willing to accept.
abort ratio	The host's measured abort ratio.	The abort ratio a buyer is willing to accept.

An economy-based mechanism is employed to match the orders that are put in the market. For each match, the market produces a lease, which is a contract between a host and a client containing their respective orders and the price of use agreed upon. Leases are produced periodically using continuous double auction [8]. A lease entitles the client to utilize the host for a specific amount of time. If the client's task completes within the lease duration, then the buyer transfers an amount of credit to the seller as a reward, calculated by multiplying actual duration with the lease's price per second. If the lease duration is not honored, an amount of credit is transfered from the dishonoring party to the other.

4 System Architecture

4.1 Overview of System Components

An overview of the system's architecture is depicted in Fig. 1. The basic components of our system are the market server, hosts, the host agent, schedulers, tasks and client applications.

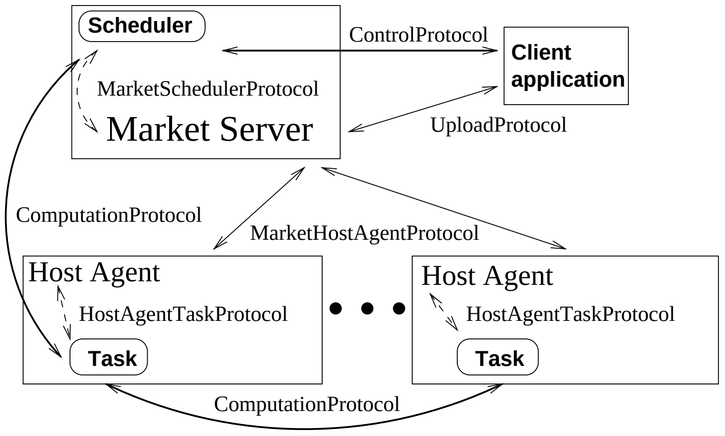

Fig. 1. Overview of architecture

The *Client Application* is a program which needs to perform computations that require considerable processing power. Through the system, it may either distribute a computation across a number of machines or just delegate the execution of an entire computation to a faster machine to speed up execution.

The *Market Server* is the meeting place for buyers and sellers of processing power. It collects orders from clients and hosts. Using the host profiles, it then matches buy with sell orders and thus allocates resources.

A *Host* is a machine made available to be used by clients. A host participates in the market through the *Host Agent*, a Java applet. The user visits a URL with a Java enabled web browser and the agent is downloaded to his system. The agent communicates with the market server, takes care of placing orders on behalf of the user and executes tasks assigned to the host. It also provides the market server with the benchmark scores needed for the host's profile.

A computation in our system consists of a *Scheduler* and one or more *Tasks*. The application installs the scheduler on the market server. The scheduler then places orders in the market for acquiring machines to complete the computation. New orders can be issued at any time in order to adapt to fluid market conditions. When a lease is accepted by the scheduler, a task is launched in the host machine to assist in completing the computation.

4.2 Basic System Services and Communication

There are six protocols used for communication by the system.

The *UploadProtocol* is a fixed, published Remote Method Invocation (RMI) interface used by the client application to upload a computation to the market server and to instantiate it's scheduler. A client application may instantiate multiple schedulers to simultaneously launch the same code with multiple data.

The *ControlProtocol* is a published RMI interface for the client application to control a scheduler. Through this interface the application performs tasks such as starting a computation with new parameters, altering the computation's budget for acquiring hosts, instructing the scheduler to kill all tasks and exit, etc. The basic functions are implemented in the system classes. The programmer can introduce computation specific control functions by extending this interface.

The *ComputationProtocol* is used within the bounds of a single computation for communication among tasks and their scheduler. It is application dependent and thus unknown to the system. We do, however, provide message passing support (not further discussed in this paper) that can be used by application developers to implement flexible, safe and efficient data exchange.

The *MarketSchedulerProtocol* is used for local communication between the market server and schedulers. The market server implements a standard published interface for servicing requests from schedulers such as placing orders and retrieving host and market status information. Respectively, schedulers provide methods for being notified by the market of events such as the opportunity to acquire a new lease, a change in the client's account balance, the completion of a task's work and the failure of a host that was leased to them. Similarly, the *HostAgentTaskProtocol* provides local communication among a host agent and the task it is hosting. The agent implements a published interface for servicing requests from tasks, such as retrieving information about a host's performance.

The *MarketHostAgentProtocol* is a proprietary protocol used by the market server and the host agent. It allows orders to be placed in the market by the host. It is also used to retrieve tasks from the market, ask for "payment" when tasks complete and to post benchmarking data to the market server.

5 Supporting Distributed Computing Paradigms

Through the set of primitives offered by the system, it is possible to develop a wide range of applications. More importantly generic support can be provided for entire classes of distributed computations. Applications can then be developed by extending these classes to introduce specific functionality. This incremental development can greatly simplify programming. As an example, in the following we describe this process for embarrassingly parallel computations requiring no communication between tasks. Other distributed computation paradigms can be supported in similar fashion.

5.1 The Generic Master – Slave Model

In this model work is distributed among many processors by a distinguished processor referred to as the "master". The other processors, referred to as "slaves", complete the work assigned to them and return the results to the master. In order to process its workload a slave does not need to communicate with any other slave. This model is used in image processing, genetics algorithms, brute force search and game tree evaluation. One possible implementation of this model is sketched below. For brevity, only the methods a programmer has to be aware of are shown.

```
public interface MS_Control extends Control {
  void start(Object pars); // inherited by superclass
  void stop();             // inherited by superclass
  Object[] getResults(boolean all, boolean keep);
}
public abstract class MS_Scheduler
extends Scheduler implements MS_Control {
  public abstract Object[] doPartitions(Object pars);
  public void receiveResult(Object result);
}
public abstract class MS_Task extends Task {
  public abstract Object processPartition(Object partition);
}
```

The *MS_Control.start* method starts a new computation. *MS_Control.start* triggers *MS_Scheduler.doPartitions* to produce the various partitions of the computation. These are forwarded to instances of*MS_Task* residing on hosts allocated to the computation and *MS_Task.processPartition* is invoked to process them. The results are returned to the scheduler where post-processing is performed via calls to the *MS_Scheduler.receiveResult* method.

It is important to notice that programmers need to implement just three methods in order to complete a computation following this model. All other implementation issues, including the resource allocation strategy of the scheduler, remain hidden. The *MS_Control* interface, which defines the primitives for controlling and retrieving the results of the computation, is implemented by the base *MS_Scheduler* class and thus does not concern the programmer. This master/slave model could be further extended to introduce additional functionality such as check-pointing and restarting of tasks for fault tolerance. Programmers would exploit this functionality without effort.

5.2 A Sample Client Application

Based on this model, we show how a specific application, e.g. for computing the Mandelbrot set, can be implemented. We assume that the area to be calculated is partitioned in bands, processed in parallel to speed up execution. The user selects an area and the computation is started to zoom into the selected area.

The parameters, partitions and results of the fractal application must be extensions of the *Object* class. The classes must implement the *Serializable* interface in order to be successfully transported across machine boundaries.

```
class FractalParameters extends Object implements Serializable {
  // ... fractal computation parameters
}
class FractalPartition extends Object implements Serializable {
  // ... parameters for calculating a slice
}
class FractalResult extends Object implements Serializable {
  // ... results of a slice calculation
}
```

Assuming the parameter and result objects have been appropriately defined, a *FractalScheduler* class must be programmed as a subclass of *MS_Scheduler* to produce partitions via the *doPartitions* method. The *MS_Scheduler.receiveResult* method is not overridden because individual results are not merged by the scheduler. Also, the basic *MS_Control* interface needs no extension since it already offers the necessary routines for controlling and monitoring the computation. Analogously, a *FractalTask* class must be provided that implements the *MS_Task.processPartition* method to perform the calculation of slices.

```
class FractalScheduler extends MS_Scheduler {
  Object[] doPartitions(Object comp_pars) {
    FractalPartition partitions[];
    FractalParameters pars=(FractalParameters)comp_pars;
    // ... split calculation and produce partitions
    return (partitions);
  }
}
class FractalTask extends MS_Task {
  Object processPartition(Object partition) {
    FractalResult result;
    FractalPartition pars=(FractalPartition)partition;
    // ... perform the computation
    return(result);
  }
}
```

Finally, to run the application, the computation's classes must be uploaded to the market server using the *UploadProtocol* and a scheduler instance must be created. The *MS_Control* interface is used to control the scheduler and periodically retrieve the computation's results.

6 Related Work

Popular distributed programming environments such as PVM [9] and MPI [9] lack advanced resource allocation support. PVM allows applications to be notified when machines join/leave the system, but the programmer must provide code that investigates hosts' properties and decides on proper allocation. MPI, using a static node setup, prohibits dynamic host allocation: the programmer must make a priori such decisions. Both systems require explicit installation of their runtime system on participating hosts. A user must therefore have access to all participating machines, as she must be able to login to them in order to spawn tasks. This is impractical and may result in only a few number of hosts being utilized, even within a single organization. Finally, the choice of C as the main programming language, compared to Java, is an advantage when speed is concerned. But to be able to exploit different architectures, the user must provide and compile code for each one of them, adding to the complexity and increasing development time due to porting considerations. The maturation of Java technology ("just in time" compilation, Java processors, etc.) could soon bridge the performance gap with C. Notably, a Java PVM implementation is underway [6], which will positively impact the portability of the PVM platform.

Condor is a system that has been around for several years. It provides a comparative "matchmaking" process for resource allocation through its "classified advertisment" matchmaking framework [11]. A credit-based mechanism could be implemented using this framework, but is currently unavailable. Condor too requires extensive administration and lacks support for easy development.

Newer systems such as Legion [10] and Globus [7] address the issues of resource allocation and security. They provide mechanisms for locating hosts and signing code. However, both require administration such as compiling and installing the system as well as access to the host computer. They do not support the widely popular Windows platform (though Legion supports NT) and do little to facilitate application development for non-experts. Globus merely offers an MPI implementation whereas Legion provides the "Mentat" language extensions. Legion's solution is more complete but also complicated for inexperienced programmers. It requires using a preprocessor, an "XDR" style serialization process and introduces error-prone situations since virtual method calls will not work as expected in all cases. Stateful and stateless objects are also handled differently. Finally, adding hosts to a running computation is done from the command line and additional hosts are assigned to the computation at random – no matching of criteria is performed.

Several other systems using Java as the "native" programming language have been designed for supporting globally distributed computations, such as Charlotte [3], Javelin [4] and Challenger [5]. These systems automatically distribute computations over machines. However, they do not employ market-based principles to allocate hosts and do not maintain information about hosts' performance.

The market paradigm has received considerable attention in distributed systems aiming for flexible and efficient resource allocation. A system operating on the same principles as ours is Popcorn [12]. Popcorn also uses auction mechanisms to allocate hosts to client computations and exploits Java applet technology to achieve portability, inter-operability and safety. However it does not provide "host profiling", nor promotes incremental development.

7 Discussion

Besides the fact that the allocation strategies used in most systems don't take into account "behavioral patterns" of hosts, there is also virtually no support for leasing. We argue that both are invaluable for efficient resource allocation in open computational environments.

Providing information about the statistical behavior of participating hosts can assist schedulers in taking task placement decisions, avoiding hosts that will degrade performance (and waste credit). For example, assume a scheduler has two tasks to allocate. Blind allocation on two hosts is not a good idea; unless two machines exhibit comparable performance, the faster machine will be wasted since the computation will be delayed by the slower one. Similarly, using the abort ratio, schedulers can avoid unstable hosts for placing critical parts of a computation. Those can be assigned to perhaps more "expensive" but stable hosts. Computations implementing check-pointing and crash-recovery could utilize less credible hosts.

The lack of leasing is also a drawback in open environments: a client could obtain many processors when there is no contention and continue to hold them when demand rises. This is unacceptable in a real world scenario where credit reflects priorities or money. This would imply that prioritized or wealthy computations can be blocked by "lesser" ones. To guarantee quality of service, some form of leasing or preemption must be adopted. Leases are also practical in non-competitive environments. The lease duration allows users to indicate the time during which hosts are under-utilized. Based on this knowledge, tasks can be placed on hosts that will be idle for enough time, and checkpoints can be accurately scheduled, right before a host is about to become unavailable.

Finally, it is generally acknowledged that incremental development increases productivity by separation of concerns and modular design. Distributed computing can benefit from such an approach. Modern object-oriented programming environments are a step towards this direction, but significant programming experience and discipline are still required. We feel that with our system's design, it is possible even for inexperienced programmers to write computations rapidly.

8 Future Directions

New versions of the Java platform will offer more fine grained control in the security system. Using the new mechanisms we expect to be able to provide more efficient services, such as access to local storage for task checkpoints, invocation of native calls to exploit local, tuned libraries such as [2] [13]. Logging mechanisms along with the signing of classes, will further increase the security of the system.

We also wish to experiment with schedulers capable of recording the performance of previous allocations. Accumulated information can perhaps be converted into "experience", leading towards more efficient allocation strategies.

Lastly the issue of scalability needs to be addressed. The current architecture is limited by the market server. A single server could not handle the millions or billions of hosts connecting to a truly world-wide version of this service. It would also be impossible to have all schedulers running on the machine. We intend to overcome this problem by introducing multiple market servers that will allow traffic to be shared among several geographically distributed servers.

References

[1] Y. Amir, B. Awerbuch, and R. S. Borgstrom. A cost-benefit framework for online management of a metacomputing system. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 140–147, October 1998.

[2] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava: An Object-Oriented Java Interface to MPI. Presented at International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, April 1999.

[3] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Ninth International Conference on Parallel and Distributed Computing Systems*, September 1996.

[4] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. In *Proceedings of the ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[5] A. Chavez, A. Moukas, and P. Maes. Challenger: A multiagent system for distributed resource allocation. In *Proceedings of the First International Conference on Autonomous Agents '97*, 1997.

[6] A. Ferrari. JPVM – The Java Parallel Virtual Machine. *Journal of Concurrency: Practice and Experience*, 10(11), November 1998.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2), 1997.

[8] D. Friedman. The double auction market institution: A survey. In D. Friedman and J. Rust, editors, *Proceedings of the Workshop in Double Auction Markets, Theories and Evidence*, June 1991.

[9] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: a Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, June 1996.

[10] A. S. Grimshaw and W. A. Wulf. The legion vision of a worldwide computer. *CACM*, 40(1):39–45, 1997.

[11] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[12] O. Regev and N. Nisan. The POPCORN Market – an Online Market for Computational Resources. In *Proceedings of the First International Conference on Information and Computation Economies*, pages 148–157, October 1998.

[13] The Java Grande Working Group. Recent Progress of the Java Grande Numerics Working Group. http://math.nist.gov/javanumerics/ reports/jgfnwg-02.html.