JaWS: An Open Market-Based Framework for Distributed Computing over the Internet

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. Harnessing the power of idle personal workstations remains a challenge for large scale distributed computing. In this paper, we present the Java Web-computing System (JaWS), which simplifies the connection of heterogeneous machines in a global computing grid as well as the development of applications that exploit this computing capacity. Machines are assigned to applications via a dynamic market-based mechanism that allows machine owners and clients to change their requirements even in the midst of a computation. The system takes care of the main communication issues offering basic programming primitives that can be extended to develop class hierarchies which in turn support distributed computing paradigms. Due to the object-oriented structuring of code, development becomes as simple as implementing a few methods.

1 Introduction

The large growth of the Internet, both in the number of connected devices as well as in bandwidth, constitutes the distribution of applications over the Internet appealing. However, in order to exploit the thousands of processors that may be available at a given point in time, several issues must be addressed.

First of all, there exists the problem of machine heterogeneity, both in terms of hardware and operating system. Secondly, security considerations arise from the execution of code from untrusted parties. Programming support must also be offered through platforms that make development of distributed computations simple enough to be attractive even for inexperienced programmers. And then of course, these platforms need to be installed and maintained, a task that if proven awkward or time consuming may limit their wide deployment in practice.

Large scale distribution of computations over the Internet goes beyond dedicated server machines that are specifically set up for this purpose. It also means involving powerful yet often under-exploited machines, notably PCs, which may switch from being available to unavailable anytime. This requires a way of dynamically allocating machines to applications even in the midst of their execution. In turn, applications must be able to deal with this dynamic environment via appropriate resource allocation strategies, which may depend not only on the form of the computation to be performed but also on various parameters that are supplied by the user interactively at runtime.

In this paper we present the Java Web-computing System (JaWS), which enables users to effortlessly and safely export their machines in a global market of processing capacity. Allocation of host machines to applications is dynamic and tasks can even migrate among hosts without disrupting the ongoing computation. A framework that promotes code reuse and incremental development through object-oriented extensions is offered to the 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 is of importance to the scientific community as most of the programming is done by scientists themselves with little support from computer experts.

The rest of the paper is organized as follows. Section 2 gives an overview of the system architecture. Details about the resource allocation mechanism are given in Sect. 3. In Sect. 4 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. 5. Section 6 discusses the advantages of our approach. Finally, future directions of this work are mentioned in the last section.

2 System Architecture

In order to guarantee maximal cross-platform operability the system is implemented in Java. Due to Java's large scale deployment, the system can span across many architectures and operating systems. Low administrative cost and an acceptable degree of security for the providers of host machines is achieved by having the host runtime system implemented as an applet that is downloaded over the Internet via a Java enabled web browser.

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

2.1 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.

The *Market Server* is the meeting place for trading processing power. Based on the orders issued by hosts (sellers) and distributed computations (buyers), the market server produces matches 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 that takes care of the communication with the rest of system in a transparent way. To install the host

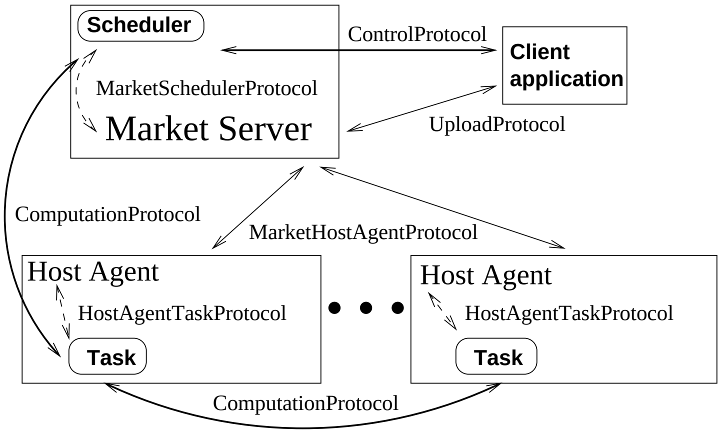

Fig. 1. Overview of architecture

agent, the owner of a host machine simply visits a URL with a Java enabled web browser. From that point onwards, the host agent computes the benchmark scores needed for the host's profile and takes care of placing orders in the market server on behalf of the user. The host agent also downloads, initializes and runs tasks assigned to the machine, thus acting as a runtime kernel for task execution.

The *Client Application* is an application which needs to perform computations that require considerable processing power. Through the system, a computation may either be distributed as a set of mobile tasks across many machines or come as a single task to be scheduled on fast machines speeding up its execution.

A computation consists of a *Scheduler* and one or more *Tasks*. The scheduler essentially implements the resource allocation strategy for a given application by placing respective 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 host is allocated to the application scheduler, a task is launched in that machine to assist in completing the computation. Notably, the number of tasks to be created need not be specified a priori; it can be determined at runtime, as a function of available machines in the system and/or parameters supplied to the scheduler. Tasks can move between different machines (host agents) while preserving their state.

2.2 Basic System Protocols

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 it is possible to start a computation with new parameters, alter the computation's budget for acquiring hosts, instruct 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 the scheduler. It is application dependent and thus unknown to the system. We do, however, provide message passing support, which is not further discussed in this paper.

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.

3 Resource Allocation

Host allocation is based on machine profiles. Both hosts (sellers) and clients (buyers) submit orders to the market server, 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.

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.

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

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.

[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 Supporting Distributed Computing Paradigms

Through the set of protocols 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. Figure 2 shows schematically how the various issues of distributed computing are dealt with, in a layered fashion, in JaWS. Different implementations may exist for each layer, depending on the desired functionality. Application programmers may either rely on existing implementations or choose to develop their own solutions based on more basic primitives. They can also make their implementations available to others by registering the corresponding archives on the public JaWS web site.

One of our goals in creating the system was to provide APIs allowing for quick development of classes of computations. As an example, in the following we briefly indicate how support for embarrassingly parallel computations can be implemented in JaWS. Other distributed computation paradigms, or even different approaches to the same problem, can be supported in similar way.

4.1 The Generic Master – Slave Model

In the "Master - Slave"model work is distributed among many processors by a distinguished processor referred to as the "master". The other processors,

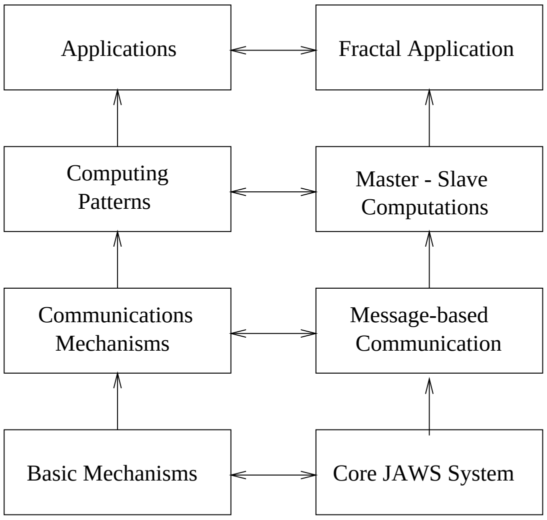

Fig. 2. Overview of architecture

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 MSControl extends gr.jaws.SchedulerControl {
  public void startComputation( Object partitionParams,
    Object orderParams ) throws java.rmi.RemoteException,
    AlreadyStartedException, ExitingException;
  public void stopComputation() throws java.rmi.RemoteException,
    NotStartedException, AlreadyStoppingException;
  public Object[] getResults() throws java.rmi.RemoteException;
  public void stopScheduler() throws java.rmi.RemoteException;
}
public abstract class MSScheduler
extends gr.jaws.models.msg.MsgScheduler implements MSControl {
  public abstract Object[]
    createPartitions( Object partitionParams );
  public abstract void placeOrder( Object orderParams );
}
public abstract class MSTask
extends gr.jaws.models.msg.MsgTask {
  protected abstract Object processPartition( Object partition );
}
```

A computation is initiated using the *MS_Control.startComputation* method. The scheduler can execute several computations during its lifetime. An ongoing computation can be aborted at any time by calling *MS_Control.stopComputation*. If there is no more need for the scheduler, i.e. the application does not need to perform any other computations on JaWS, a call to *MS_Control.stopScheduler* will cause the scheduler to exit releasing all associated resources.

When a computation is started, *MS_Scheduler.createPartitions* produces the partitions of the computation. These are forwarded to *MS_Task* instances, residing on hosts allocated to the computation. Once the data arrives an invocation to *MS_Task.processPartition* is triggered for processing the data. The results are returned to the scheduler to be retrieved by *MS_Scheduler.getResults*.

The resource allocation strategy employed in this case is simple. The scheduler places a bid in the market using *MS_Scheduler.placeOrder*, as long as there are still partitions to be processed and all of the hosts allocated so far are busy. Bid parameters are provided by the programmer to take into account application specific allocation parameters and may also be supplied by the user at runtime.

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.

4.2 A Sample Client Application

On top of this framework, we have developed a fractal generator application, which calculates the Mandelbrot, and Julia sets. It was adopted, within a few days, from a fractal generation program created by Peter Walser [14].

The fractal generation application consists of two major parts: the application interface and the computation. The computation is further constituted of two modules, the scheduler and the task. A scheduler is instantiated for each fractal requested and tasks are installed on hosts acquired, carrying out the actual work.

For each fractal a distinct control thread is launched in the client application along with a new scheduler on the market server's side. The control thread instructs the scheduler to create the desired fractal and retrieves the results in order to draw them on screen. Evidentely, several computations may be running concurrently and competing with each other for resources.

When launching the application, the scheduler and task classes are uploaded and registered to the JaWS server using the corresponding *Upload* protocol call (a RMI invocation):

```
upIf = (gr.jaws.UploadInterface)
  java.rmi.Naming.lookup( "//host.domain.gr:1200/" +
```

```
    gr.jaws.UploadInterface.upIfName );
  upIf.uploadCode( user, passwd, "fractalApp", "fractalApp.jar",
    jarContents, "FractalScheduler", false );
```

After uploading the appropriate classes, it is possible to instantiate several schedulers in a similar manner, again through the JaWS *Upload* protocol:

```
  controlInterface = (FractalControl)
    upIf.instantiateScheduler( user, passwd, "fractalApp" );
```

Once a scheduler is instantiated and a handle to its remote interface is received, computations can be started and controled as already described via calls to methods *startComputation*, *stopComputation*, and *getResults*.

The process of implementing the scheduler was straightforward. Only two methods were necessary: one for creating partitions and one for bidding in the market. The number of partitions created, as well as the bid, are determined by parameters passed to each method that stem from user input from the client application. Analogously, the task implementation merely required the programming of a single method. An implementation sketch is given below:

```
public class FractalScheduler
extends gr.jaws.models.ms.MSScheduler {
  public Object[] createPartitions( Object pars ) {
    FractalPartitionsDef fp = (FractalPartitionsDef) pars;
    Object[] partitions =
      new FractalPartition[ pars.totalPartitions];
    for( int i=0; i<partitions.length; i++ )
      partitions[i] = /* Code for calculating partition i */
    return partitions;
  }
  public void placeOrder( Object orderPars ) {
    FractalOrderParams p = (FractalOrderParams) orderPars;
    placeBuyOrder( p.units, p.price, p.duration,
      0,0, null,null, 0.0 );
  }
}
public class FractalTask extends gr.jaws.models.ms.MSTask {
  public Object processPartition( Object partition ) {
    FractalPartition p = (FractalPartition) partition;
    /* Code for calculating fractal partition p */
    return myPart;
  }
}
```

It is evident that the amount of work required for creating the computation was minimal. Using the "Master - Slave" framework, one can develop any computation that falls under this computation pattern by implementing just 3 methods and thus the goal of ease of programmability is therefore achieved.

5 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 decide on proper allocation. MPI, using a static node setup, prohibits dynamic host allocation: such decisions are taken a priori. Both systems require explicit installation of their runtime system on hosts. A user must therefore have access to them, as she must be able to login in order to spawn tasks. This is impractical and may result in only a few 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 advertisement" 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.

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 and thus require administrative 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 – there is no matching of criteria.

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.

6 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 task placement decisions, avoiding hosts that 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, unstable hosts can be avoided when placing critical parts of a computation. Those can be assigned to 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.

7 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 security.

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.

[14] P. Walser. idx_fract home page. http://www2.active.ch/~pro xima/java/-idxfract/idxfract.htm.