A COST-BENEFIT FRAMEWORK FOR ONLINE MANAGEMENT OF A METACOMPUTING SYSTEM

Yair Amir, Baruch Awerbuch, R. Sean Borgstrom
Department of Computer Science
The Johns Hopkins University
Baltimore MD 21218
{yairamir, baruch, rsean}@cs.jhu.edu

1. Abstract

Managing a large collection of networked machines, with a series of incoming jobs, requires that the jobs be assigned to machines wisely. A new approach to this problem is presented, inspired by economic principles: the Cost-Benefit Framework. This framework simplifies complex assignment and admission control decisions, and performs well in practice. We demonstrate this framework in the context of an Internet-wide market for computational services and verify its utility for a classic network of workstations.

1.1 Keywords

Networks, resource allocation, markets.

2. Introduction

Collections of networked machines are common in the modern world. Using each individual machine as a completely independent computer is obviously inefficient – one machine could be working on a dozen jobs while the others sit idle. A metacomputing system is a set of networked machines that can pool their computational resources to avoid this problem. Each machine has several computational resources associated with it. Intelligent management of a metacomputing system requires a good strategy for assigning computational resources to jobs the system must perform.

Computational resources are heterogeneous. A given task may require a certain amount of normalized CPU seconds, Megabytes of memory, and network bandwidth. Not only are these resources independent, they are not even directly comparable – they are measured in unrelated units. This can make it difficult to determine the optimal machine to which to assign a given computational task. This situation can be further complicated by tasks with different priority levels and different requirements in terms of completion time. More generally, tasks can have different rewards that the system will receive for completing them.

The Cost-Benefit framework is a unified approach to these two problems, inspired by economic principles. The approach is straightforward: we assign a homogeneous *cost* to each resource, which depends on that resource's current utilization. The total cost for all resources used is the cost of a given scenario. The *marginal cost* method for assigning jobs puts a job on the machine where its resource consumption has the minimum marginal cost. The resulting scenario has the minimum total cost.

This relatively simple concept turns out to have useful theoretical properties. In particular, when certain exponential cost functions are used, the marginal cost strategy is competitive – its consumption of resources is at most O(log n) greater than that of an optimal strategy that knows the future (where n is the number of machines with resources). This "system-friendly" strategy effectively predicts a system's future needs, regardless of the correlation between past and future.

In our framework, jobs with different priorities and demands are also assigned a homogeneous *benefit*, which depends on the time in which that job is completed. Benefits and costs are measured in the same units. Maximizing system benefit directly improves the system's value to its users. We cannot simultaneously minimize system cost and maximize received benefit. We have therefore developed additional strategies that attempt to strike a balance between the two.

We do not control the benefit functions. Therefore, in order to guarantee competitive acquisition of benefit we must perform admission control. We will only accept jobs with benefit larger than the minimum marginal cost. We assign them to maximize our "profit" - the benefit minus the marginal cost. This "miserly" strategy is competitive in benefit achieved.

It is important to emphasize that the Cost-Benefit framework is not intended to help a metacomputing system reach a steady state. Instead, it is designed to maximize the system's transient performance. The framework does not deal with "offline" optimization problems where all the variables are known in advance; it must manage the system "online," making decisions based on the system's current state and online demands at any given time.

In this paper, we study how the Cost-Benefit framework helps manage online metacomputing systems. Our test case is the *Java Market*, a computational market designed for use with this framework. The Java Market "buys" spare cycles from interested machines anywhere on the Internet and "sells" their computational services to anyone on the Internet with a job written in Java. The concepts of costs, benefits, and negotiations between the two are fundamental to its design.

The Java Market has been shown to produce good performance in practice. Using a handful of machines connected only by the Internet, the Java Market produced a 3.6x speedup in our sample application. Because these machines used only Web browsers to connect to the Market, they could have been anywhere in the world, running any operating system -- and the machines would still have been secure. The Cost-Benefit framework has also been shown efficient in a more general context. Applying the Cost-Benefit framework to the more traditional problem of resource allocation on a cluster of workstations, we improved the performance of naïve strategies by 38% and intelligent, optimized strategies by about 13%.

3. RELATED WORK

The LYDIA project [1] studies single-resource resource allocation on a system where there are many kinds of jobs, and each "class" of jobs has different performance expectations. The performance of each class is given a homogeneous cost called the "performance index", much like our job benefit functions. The LYDIA project does not yet address the complex issues involved in balancing several of the diverse system resources simultaneously.

SPAWN [2] and other systems like it provide computational markets where tasks bid competitively for resources. This approach uses economic principles conceptually similar to ours. The Cost-Benefit framework, however, integrates the computer science notions of competitive algorithms and analysis with the economics concepts of marginal costs and markets for services.

The Condor system [3] is similar to the Java Market, our testbed. The Condor team has created a set of software tools for utilizing the wasted CPU cycles in a cluster of workstations. Condor provides a checkpointing mechanism for the jobs it schedules, allowing interrupted jobs to be resumed later when a machine of the appropriate architecture is available. Although Condor is a mature system, proven to work efficiently with hundreds of machines at a time, it has very limited support for heterogeneous machine architectures. The Java Market, while limited by the speed of the virtual Java machines available, is 100% cross-platform.

The Popcorn project [4], independently developed at the Hebrew University in Israel, is an online market for computational services that shares many features with the Java Market. The project differs from the Java Market primarily in that it provides a new programming model. Its users must write their applications with the Popcorn project in mind -- Popcorn cannot be used with ordinary Java applications. A key Java Market design decision is that users submit their jobs as regular Java applications, and the Market itself does all necessary modifications.

The Milan project [5], like the Popcorn project, provides a programming model that can take advantage of heterogeneous Internet-connected machines. Also like the Popcorn project, Milan is not designed for use with standard Java applications.

4. THE COST-BENEFIT FRAMEWORK

We will use the same conceptual structure for the computational market and the metacomputing system that we will study. In both cases, we will examine the system in terms of *benefits* and *costs*, where the system tries to maximize its benefit and minimize its cost.

4.1 Cost

The key to our Cost-Benefit framework is that the cost of a resource is an exponential function of its utilization. Each time a certain amount (e.g. 10%) of the resource is used, the cost for that resource doubles. In its simplest form, this could be used to perform admission control: if the benefit of a job is higher than its cost, the job is admitted. (See Figure 4.1).

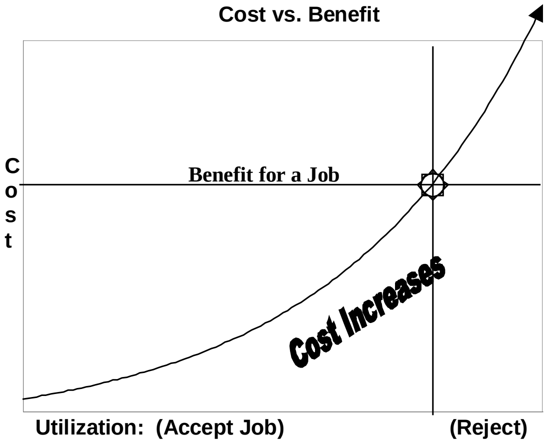

Figure 4.1: Exponential Costs

The picture becomes more interesting when we have multiple resources that can be exhausted. Normally, decision-making in this circumstance is difficult – but with the Cost-Benefit model, we can simply add these costs together. This makes decision-making very easy. For example, suppose we have two machines. A job comes in for one of these machines requiring *m* Megabytes of memory and *c* normalized seconds of CPU time. The cost for this job's memory usage will be different on each machine, based on how much memory is already used. Similarly, the cost to share the CPU will be different on each machine, based on how many jobs are already using that machine. We place the job on the machine where the sum of these marginal costs is minimized. (See Figure 4.2).

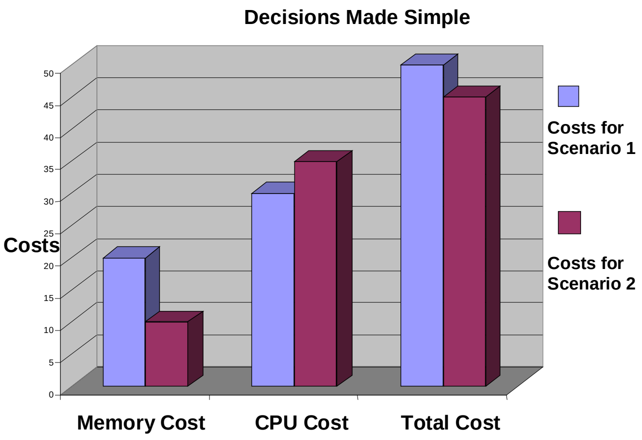

Figure 4.2 Our Cost Function Makes Choosing the Right Machine Simple!

Our framework's "costs" and real-world costs like poor machine performance are related as follows. Using our cost function when performing admission control and job assignment provides an upper bound on these real-world costs. In practice, this upper bound has been shown to be pessimistic - the Cost-Benefit framework performs much better than the theoretical bound requires. (See section 4.3.)

4.2 Benefit

The other half of the Cost-Benefit framework is the benefit earned from jobs. This benefit can be compared to the marginal cost for running a job to see whether running the job on a given machine is worthwhile. If the job cannot make a "profit" on any machine, then the job is rejected or delayed, depending on the system.

In the simplest case, the benefit for completing a job is equal to its priority, much as in Figure 4.1. Jobs with low benefit will be rejected or delayed when the system is heavily loaded. Jobs with high benefit are more likely to be admitted immediately. A job whose benefit is greater than the highest possible cost will always be accepted and placed on a machine.

Things become more interesting when jobs have benefit *functions*. For example, a job might give no benefit at all unless it is completed within a given time frame, or its benefit might reduce linearly over time. (See Figure 4.3.) The benefit for a job might even become negative, if enough time passes, meaning that the party that submitted the job must be compensated for lost time.

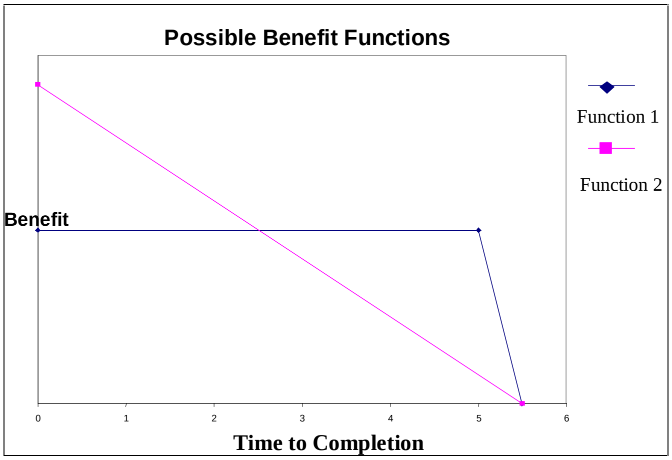

Figure 4.3: Benefit as a Function of Time Taken

When benefit functions are variable, as seen above, the system has a more complicated decision to make. Faster machines will complete jobs more rapidly, earning more benefit, but the more jobs are running on a given machine, the higher its cost will be. In this context, the job and the system will have to "negotiate" to determine where to place the job.

4.3 Cost Function

One particular cost function, used with our Cost-Benefit framework, shows very nice theoretical properties. This cost function, in a cluster of *n* machines, charges:

$$n^{(usage\ /\ maximum\ usage)}$$

for each resource.

Using the marginal cost strategy with this particular cost function has a beneficial theoretical property. Over the system's continuously operating lifetime, the maximum usage of each resource is within O(log *n*) of the *optimal* assignment strategy's maximum usage [6]. Further, this holds even when the optimal strategy knows the future.

This theoretical guarantee is weak, but most job assignment strategies have no theoretical guarantees at all. Further, this particular strategy has been shown in tests to perform extremely well in practice. (See Sections 5 and 6.)

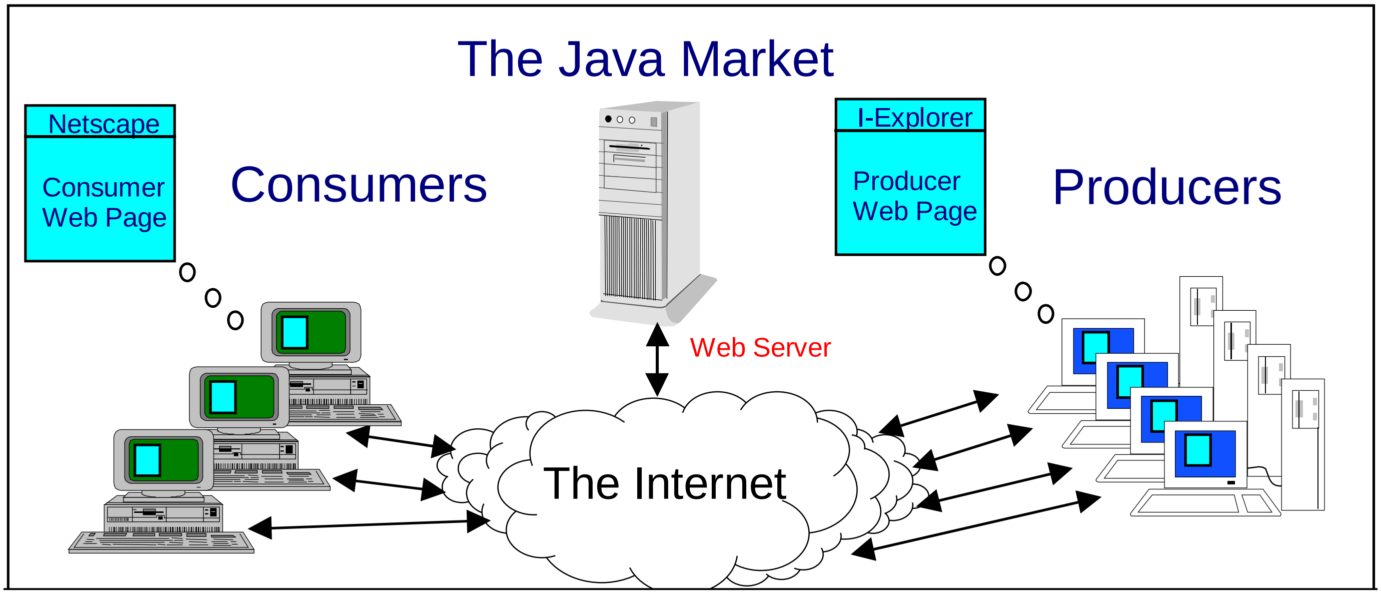

Figure 5.1: An Internet-Wide Metacomputer

Note that the maximum usage of a resource is not defined in terms of our abstract costs. Restricting these maxima translates directly into bounded CPU loads, bounded memory usage, limited network congestion, *etc*.

4.4 A Decentralized Approach

The Cost-Benefit framework uses a centralized scheduler in both of our test-beds. Experience with the Condor system [3] shows that one scheduler can manage hundreds of producer machines, as long as it acts only as a decision-maker.

For a more distributed approach, our framework can be extended as follows. Assign each scheduler a subset of the producer machines, which it must choose from when assigning jobs. These subsets can intersect, and they can be changed on-line. Clients can deliver their job to any scheduler, or even shop among schedulers for the lowest price. Assuming that the optimal assignment strategy is restricted to the same clients and the same producer subsets, the theoretical guarantee above will still apply.

5. THE JAVA MARKET

The Java Market is an Internet-wide market for computational services that is being developed at the Johns Hopkins University. It is the first testbed built for our Cost-Benefit framework. (For more details, see http://www.cnds.jhu.edu/publications, technical report CNDS-98-1.)

5.1 Basic Concepts

The Java Market’s world is defined by two entities: machines and jobs. Both machines and jobs contract with the Java Market: one sells computational services to the Market and one buys such services from the Market.

The Java Market "pays" producer machines for their services and "charges" consumers for each job it runs. These payments and charges might be measured in terms of virtual money, usable only to buy Java Market services, or real currency. For example, a consumer might be willing to pay $20 to complete a large simulation in 6 hours or less. Similarly, a machine owner might charge the Java Market $10 for eight hours of their machine's services.

5.2 The System

The Java Market brokers the distribution of computational resources among machines scattered across the world. As depicted in Figure 5.1, the Java Market is designed to transfer jobs from *any* machine on the Internet *to* any machine on the Internet that wishes to participate. There is no installation or platform-dependent code – the only requirement is that the jobs be written in Java. Further, the Java job does not need to be written especially for the Market -- the Market can rewrite Java applications into Applets automatically, and provides services that can overcome some of the inherent Applet restrictions. These applications can then be ported automatically to any producer machine. Using the Market is only slightly more difficult than clicking on a browser bookmark.

The Java Market has no dependence on any given architecture. Producers and consumers can be running any kind of machine, and any operating system, that has a Java-capable Web browser. In other words, it can handle heterogeneous machines as easily as the Cost-Benefit framework handles heterogeneous resources. The program-transfer technology associated with the Market has already been implemented. Further development will implement several variants of the Cost-Benefit decision strategy.

The Java Market uses the Web and the Java language as its primary tools. A producer makes their machine available as a resource by directing its browser to one of the Market web pages. A consumer registers its request for computational resources by contacting another of the Market's web pages, and posting its program (written in Java) in a Web-accessible location. The Java Market is a metacomputing system that performs admission control for tasks, signs contracts with and manages producer machines, and places tasks on producer machines based on advanced resource allocation algorithms.

The Java Market is composed of three main entities, as depicted in Figure 5.2:

- **The Resource Manager** keeps track of the available machines – those that have registered themselves as producers.
- **The Task Manager** keeps track of the consumer-submitted tasks.
- **The Market Manager** mediates between the Resource Manager and the Task Manager.

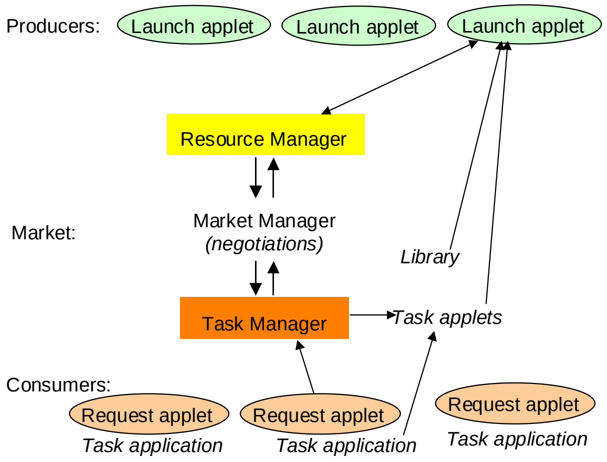

Figure 5.2: Java Market Components

5.2.1. The Resource Manager

The Resource Manager is run on the Market machine. When a producer registers with the Market web pages, they automatically run a special applet, the *Launch Applet*, that tells the Resource Manager about the producer's machine's power. The Resource Manager stores this information, as well as the state of the machine (in this case, 'available,') the IP address, and so on. This information forms a machine profile.

The Launch Applet is the part of the Resource Manager executed on the producer's machine. This gives it two responsibilities. The first is resource discovery, that is, assessing the machine's computational and networking capabilities. The second is directing the producer's browser to a page containing whatever task is assigned to it.

5.2.2. The Task Manager

The Task Manager is run on the Market machine. When a consumer registers their task with the Market web pages, they run another special applet, the *Request Applet*, that gathers information about the task they want the Market to perform. Once this information is gathered, it is sent to the Task Manager. The Task Manager then gathers all the Java files and input files associated with the task from the Web, edits the Java files as necessary, compiles them, and then passes the entire task to the Market Manager.

The Request applet is the part of the Task Manager that is executed on the consumer's machine. Its primary responsibility is to wait. First, it waits while the consumer types in the necessary data about their task. It sends this information to the Task Manager proper and then waits again. As it waits, the Task Manager downloads, edits, and compiles all of the Java code associated with the task. When this is complete, it tells the Request Applet that the task has been accepted or rejected by the Market. The Request Applet displays this information and ceases computation.

5.2.3. The Market Manager

The Market Manager acts as an overseer, performing resource allocation and admission control. There are two issues that must be addressed: the uncertain *availability* of a producer machine for the length of time a given task requires, and the on-line decision-making necessary to maximize the profit of the market.

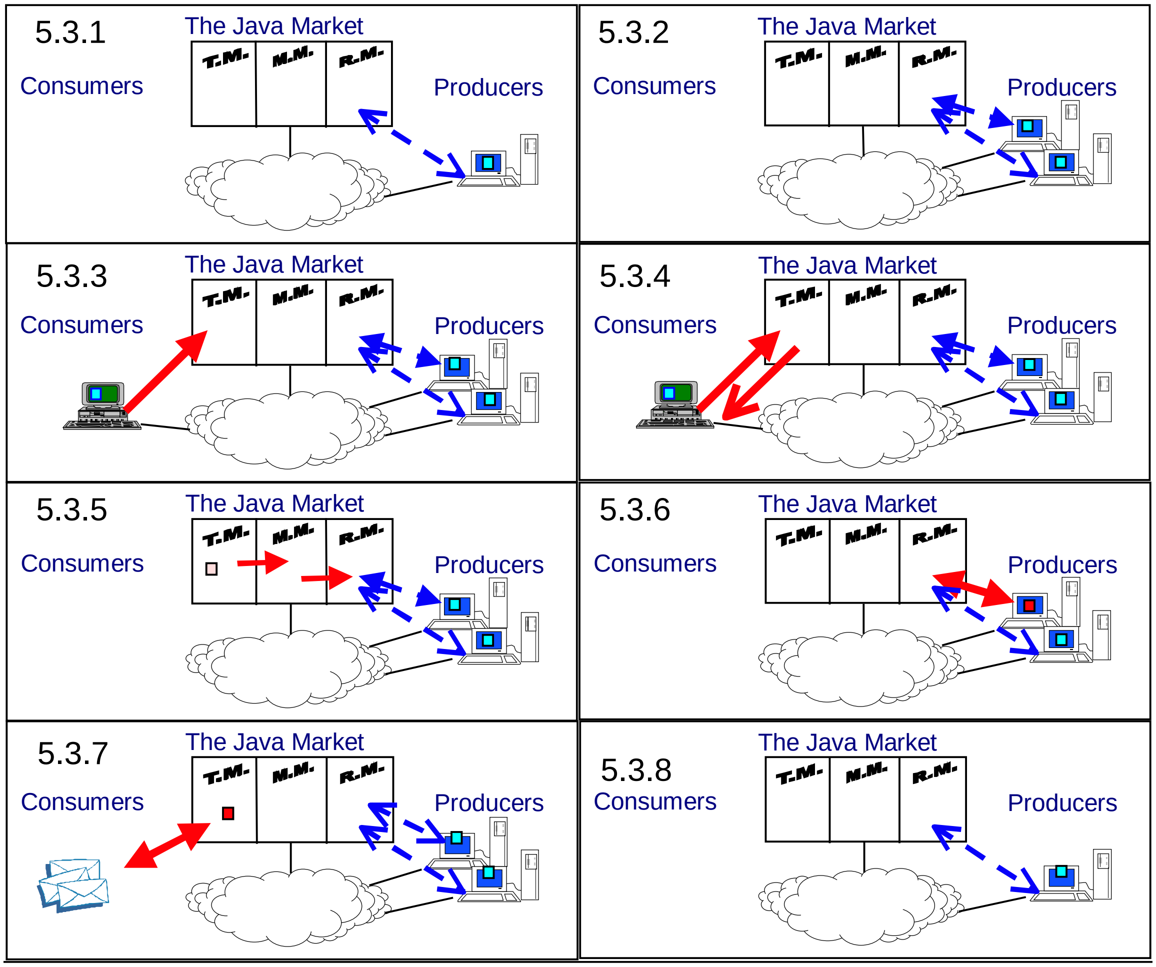

Figure 5.3: A Sample Scenario

5.2.4. An Example Scenario

The scenario presented in Figure 5.3 demonstrates how the Java Market metacomputing system works.

1. A producer machine registers its availability over the network with the Java Market.

2. A second producer machine registers its availability with the Java Market.

3. A consumer connects to the Java Market and registers a task.

4. The Task Manager downloads the task information from the Web and modifies and compiles the code. It then notifies the consumer of the consumer's success at launching the task.

5. The Market Manager mediates between the Task Manager and the Resource Manager in order to find an appropriate producer to execute the task.

6. The selected producer's browser automatically begins executing the task.

7. The task completes and its results are mailed to the consumer.

8. A producer leaves the Java Market.

5.3 Implementation Status & Experimental Results

Current Implementation Status

The Java Market metacomputing system is fully implemented and is publicly accessible at http://www.cnds.jhu.edu/projects/metacomputing. The publicly accessible version differs from the description in this paper only in that it uses simpler decision algorithms within the Market Manager. Optimizing and refining these resource allocation and admission control algorithms is a subject of continuing research within our group. Future releases will incorporate refined versions of the decision making mechanisms.

The complete software package is roughly 5000 lines of Java code. This includes about 2000 lines implementing the Task Manager, the Resource Manager and the Market Manager. About 2000 lines implement the Market Library.

The additional 1000 lines implement the Request, Launch, and Monitor applets.

Experimental Results

The first major test of the Java Market was performed in the Johns Hopkins Center for Networking and Distributed Systems (CNDS). A CPU-intensive simulation (about 1000 lines of Java code) evaluating five different job scheduling policies for the Mosix system [7] and a stream of jobs was run on the CNDS lab machines. One hundred simulations, each of them representing 10,000 real-time seconds, were run in the following two ways:

1. Running on a standalone Pentium II machine using the Java Developer's Kit.

2. Using the Java Market with six producer machines (two Pentium IIs and four Pentium Pros) using Netscape.

The execution time of the complete simulation on the standalone machine, without compilation and, of course, without remote I/O, was approximately 127 minutes. The execution time of the complete simulation using the Java Market, including downloading, compilation, and remote I/O was approximately 35 minutes. This shows a speedup of about 3.6 on a system with a combined power roughly 4.7 times greater than that of the standalone machine.

Accessing our web site, the interested reader is able to become either a producer or a consumer. Moreover, demo applications, including the above simulation, are available on our web site and ready to be launched.

5.4 Key Features

Producers and consumers can use the Java Market with minimal effort. The only programs that they need to run are secure Applets, which can be executed automatically by their Web browsers when they connect to the Market web pages. There can be no deleterious effects to the producer's machine.

As mentioned above, the Java Market is platform-independent. It can handle machines and task requests from anywhere on the Internet. A single Market can handle potentially hundreds of machines at a time -- almost all of the Market's work is done on the producer machines.

5.5 Implementation of the Cost-Benefit Framework

The Java Market uses the Cost-Benefit framework to determine what it is willing to pay for a resource and what it requires to accept a job. This built-in economic framework makes it easy to create limited Markets for exchanging resources between specific companies (using real money) or within a single organization (using virtual money.) It can also be used to make an Internet-wide computational market.

The Java Market can convert the abstract cost functions associated with the marginal cost assignment strategy into real costs. This means that the Java Market's decisions are inherently comparable to those of the optimal off-line prescient Market.

5.5.1 Jobs

Each job *j* submitted to the system is defined by the following properties:

- Its arrival time, *a(j),*
- Its resource vector, *r(j),*
- The benefit function for this job, *b(j,t),* where *t* is the time the job takes to complete.

The resource vector *r(j)* represents the various system resources the job requires to complete. In the current implementation of the Java Market, these resources are CPU speed and network connectivity. As the power of the Java language grows, it will become feasible to add resources such as memory and disk I/O to this model.

The benefit *b(j,t)* is the reward the system receives for completing job *j* in *t* time. In the Java Market, this is literally the amount the consumer of computational resources will pay to have their job completed in that time.

Only one job can be run on a given machine, which requires thinking about resource allocation in a new way. Instead of treating each producer as an independent entity, we group them together into a pool of resources on a single conceptual machine. This machine has two resources: messages and computations per second.

This conceptual division gives us a normalized unit cost. Observed trends in what people want to sell their machines for and buy resources for can be used to convert this to real money (if necessary). We then use the "miserly" strategy for assigning resources.

5.5.2 Machines

Machines can be *dedicated machines*, which guarantee their availability to the system for a certain length of time, or *opportunistic machines*, which offer unreliable service for an indefinite period of time.

Each machine *m* that is made available to the system is defined by the following properties:

- Its arrival time, *at(m),*
- Its resource vector, *r(m),*

- Its departure time, *dp(m)*,
- (For dedicated machines) its total cost, *c(m)*, and
- (For opportunistic machines) its cost per second of use, *cps(m)*.

The arrival time *at(m)* is the time that the machine signals its availability to the Market. The departure time *dp(m)*, similarly, is the time when the machine will signal that it is no longer available or breaks off its connection with the Market. (e.g. when the machine's owner returns and removes the machine from the Market.)

The resource vector *r(m)* describes the relevant resources associated with the machine. These are, naturally, the same resources that are associated with jobs.

When a dedicated machine *d* is offered to the system, all of these properties are known. This is not a bidding system, and *c(d)* is assumed to be the "true" cost for *d*. The system must either accept *d* and pay the cost *c(d)* or reject *d* and pay nothing. If it accepts that machine, it can use *d*'s computational resources until time *dp(d)*.

When an opportunistic machine *o* is offered to the system, all of these properties except the departure time *dp(o)* are known. No immediate decision is necessary. At any point before the (unknown) departure time, the system can use *o*'s computational resources. If it completes a *t*-second job on machine *o*, it must pay the cost *t* * *cps(o)*. In other words, it must pay for each second of successful use.

In the Java Market, the costs represented by *c(m)* and *cps(m)* are literally paid to the machines used.

The Java Market cannot expect its providers to charge based on the cost model in Section 4. It can, however, determine whether to buy a dedicated machine's time and whether to use an opportunistic machine's resources using the value calculations above. Deterministic machines should be purchased based on the price the system would put on that resource immediately after accepting it.

5.6 Discussion

A metacomputing environment can be evaluated generally based on its performance in the following areas:

- Awareness of changing resource availability;
- Ability to handle resource heterogeneity;
- Guarantees regarding QoS (Quality of Service);
- Security;
- Scalability;
- Ability to impose a desirable scheduling policy;
- Transparency -- how much extra work the user must do; and
- Speed.

Our two specific goals for the Java Market are:

- It should be able to apply the Cost-Benefit framework, and its associated algorithms, and
- It should be successful as a metacomputing system, able to use the power of the machines available to it to improve every user's performance.

How does the Java Market fare, regarding these measures?

Our system is continuously aware of all of its resources; when one of them disconnects or crashes, the Market will detect the loss of this resource and remove the relevant machine from its list of producers.

The Java Market can handle machines and task requests from anywhere on the Internet, and can impose its Cost-Benefit-based scheduling policy on them.

For these reasons, the Java Market is able to meet its first specific goal -- it can use the resource allocation algorithms associated with our framework.

The Java Market is optimized for security, at the expense of QoS guarantees. Rather than using the producer machines to the full extent possible, it operates within the Java "sandbox". This is a set of restrictions typically applied to Java applets that (in this case) protect the producer machines fully from hostile consumers. The Java Market also implements additional security to protect the integrity of the Market itself from hostile users.

The Market operates on the Java "virtual machine," which is an integral part of all Java-capable Web browsers. Such browsers are available on the vast majority of platforms.

The basic Market design does not allow transparency, but the Market has the next best thing -- ease of use. Using the Market is not the same as running a job on the local system, but it is a matter of a minute to upload a job or make one's machine available.

To measure speed, we performed 100 executions of a complex simulation, once distributing it via the Java Market and once doing it on a single machine. As mentioned above, the Java Market was able to complete this 127-minute calculation in 35 minutes.

The Market's "resources" are machines throughout the Internet. It has the technical ability to manage these machines -- regardless of their architecture or location. The security of the Java sandbox makes offering one's machine to the Market in its off-hours a reasonable course of action. The Market has proven ability to harness this vast computational power; therefore, we believe it has met both of its specific goals.

6. Enhancing Local Networks

We also studied the Cost-Benefit framework in an environment where standard resource allocation methods applied -- a network of workstations. In this work, found in [8], we created two new policies for resource allocation and compared them to standard methods. These policies were based on the "system-friendly" strategy and did not use admission control.

PVM is a popular resource allocation system with a naïve default strategy. Designed for networks of workstations where jobs can only be assigned to a machine once, it distributes jobs using a straightforward round robin policy. (Programmers can override this policy.) In comparison, our strategy for an identical network of workstations completed the average job 38% faster.

Mosix [7] is a set of kernel enhancements to the BSDI Unix-like operating system [9] that allows jobs to be moved from machine to machine without interrupting their execution. Mosix also has an experimentally tuned resource allocation strategy based on load balancing. A Cost-Benefit-based strategy for Mosix networks, also able to move jobs around, improved over the current Mosix strategy by about 13%.

References

[1] The LYDIA Project (goal-oriented scheduling). http://www.ics.forth.gr/pleiades/projects/LYDIA/.

[2] C. Waldspurger. A distributed computational economy for utilizing idle resources. Master's thesis, MIT, Dept. of Electrical Engineering and Computer Science, May 1989.

[3] Condor. http://www.cs.wisc.edu/condor/.

[4] N. Camiel, S. London, N. Nisan, O. Regev. The Popcorn Project -- An Interim Report, Distributed Computation over the Internet in Java. Sixth International World Wide Web Conference, April 1997

[5] MILAN: http://www.cs.nyu.edu/milan/milan/index.html

[6] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin and O. Waarts. On-Line Machine Scheduling with Applications to Load Balancing and Virtual Circuit Routing. In *Proceedings of the ACM Symposium on Theory Of Computing (STOC)*, May 1993.

[7] A. Barak, S. Guday and R. Wheeler. The Mosix distributed operating system, load balancing for Unix, Volume 672, May 1993.

[8] Y. Amir, B. Awerbuch, A. Barak, R. Borgstrom, A. Keren. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, October 1998. Available as tech report CNDS-98-2 at http://www.cnds.jhu.edu/publications.

[9] Berkeley Software Design, Inc. http://www.bsdi.com.