SuperWeb: Research Issues in Java-Based Global Computing

Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{*berto,ibel,schauser,chriss*}*@cs.ucsb.edu*

Abstract

The Internet, in particular the World-Wide-Web, continues to expand at an amazing pace. We propose a new infrastructure, SuperWeb, to harness global resources, such as CPU cycles or disk storage, and make them available to every user on the Internet. SuperWeb has the potential for solving parallel supercomputing applications involving thousands of cooperating components on the Internet. However, we anticipate that initial implementations will be used inside large organizations with large heterogeneous Intranets. Our approach is based on recent advances in Internet connectivity and the implementation of safe distributed computing realized by languages such as Java.

Our SuperWeb prototype consists of *brokers*, *clients*, and *hosts*. Hosts register a fraction of their computing resources (CPU time, memory, bandwidth, disk space) with resource brokers. Clients submit tasks that need to be executed. The broker maps client computations onto the registered hosts. We examine an economic model for trading computing resources, and discuss several technical challenges associated with such a global computing environment.

Keywords: Global computing, Internet, Java, supercomputing, secure computing, electronic commerce, encrypted computing

1 Introduction

The Internet has been growing exponentially in recent years. It is estimated that by the end of 1997, more than 30 million hosts will be connected to the Internet, compared to the 10 million hosts in January 1996. Although the primary reason for connecting a machine to the Internet is to gain access to information, information is not the only resource available. Ideally, the combined resources of millions of hosts on the Internet could be harnessed to form a powerful global computing infrastructure. Recent advances made in Internet connectivity, and the implementation of safe distributed computing embodied in languages such as Java [GM95] make this a promising and realizable goal.

Architecture. We propose a design of a global computing infrastructure called *SuperWeb*. In our model there are three kinds of participants: brokers, clients and hosts. *Brokers* coordinate the supply and demand for computing resources. *Hosts* offer their resources (CPU time, memory, permanent storage) as a commodity by registering with a broker. For example, a host can register 50% of its CPU time from 5pm to 8am Monday through Friday, and 100% during the weekend. Another host may register 10 hours of 30-megaflops-per-second CPU time, while a third host is willing to offer 100% of its CPU time only while being completely idle. *Clients* are users or computers that need extra computational resources. For example, a client that wants to run a task requiring 500 billion floating point operations may register a request for CPU resources with a broker. It is the broker's responsibility to find host[s] capable of executing the client's task. In our model the role of a host or a client is not fixed. A machine may register as a host when being idle, and act as a SuperWeb client when its owner needs additional computing power. While we target the SuperWeb infrastructure to work across the whole Internet, it is also usable on a smaller scale. In fact we anticipate that its initial implementations will be inside large organizations with intranets consisting of a large number of heterogeneous computers.

Design. Our main goal is to allow everyone connected to the Internet to participate in SuperWeb. To this end, our design is based on widely used components such as WWW browsers like Netscape and Microsoft Internet Explorer and portable languages like Java. Although we use only Netscape and Java in this paper, the SuperWeb architecture is not dependent on them. It can be implemented using a different portable language or even using plain binary executables. The underlying key technology in SuperWeb is the safe execution of an untrusted piece of code sent over the Internet by an untrusted third party. For example, the Netscape Web browser can safely run arbitrary Java code[1] while software fault isolation [Sof95] allows for safe execution of untrusted binaries.

Economic model. The SuperWeb design makes computing resources freely exchangeable commodities which can be captured by a microeconomic model based on supply and demand. In this model, the brokers implement an accounting mechanism for billing and crediting users. Depending on the quality of its services, the host receives some kind of credits. These credits may be exchanged for free CPU time (e.g. if the user needs to run her own supercomputing application), information access, free backups, cheap access to the Internet, access to software licenses, or cyber money.

Applications. SuperWeb is useful for casual users, users that want to run parallel supercomputing applications, and users of network computers or web stations. Individual users that occasionally need extra computing power can rent CPU hours from a host when needed. Users with diskless web stations are able to rent disk storage from remote hosts or software and CPU time for temporary use. Corporations and researchers can use the SuperWeb brokers to obtain thousands of hosts at a time on which they can run large parallel and distributed applications. We expect this approach to work well for non-communication intensive applications such as prime number factorization, Monte-Carlo and coarse-grained simulations, and others. For these applications, we expect to obtain performance far superior to today's supercomputers, at a fraction of the cost.

Security and correctness. The fact that we allow interaction between unknown hosts and clients that do not necessarily trust one another raises many security concerns. For example, a host wants the anonymous jobs to run without crashing the machine or using up more resources than claimed. On the other hand, a client requesting a remote computation needs to be sure that the participating machines really perform the correct computation and do not fake results in order to overcommit themselves and get lots of credits. Moreover, the client wants to make other computers execute some algorithm on its data, but does not necessarily want to expose the data to an untrusted host or send them over insecure Internet channels. In order to preserve trust in the results and confidentiality of data, we propose features such as duplicated execution of code to detect cheating and encrypted computing to preserve confidentiality of the data passed and processed.

This project can be distinguished from other network based computing approaches in that it

[1] Assuming all bugs have been found and fixed.

is based on standard components such as Netscape browsers and Java. Because of the simplicity of use, we expect thousands of users registering their computer resources with SuperWeb brokers, giving us the ability to harness an extraordinary amount of computational power. Alternative approaches either require the user to have login access at all machines used for the computation, or require operating system modifications. As a consequence, these approaches are limited in size, as few individual users have access to more than a thousand workstations.

The rest of this paper is structured as follows: In Section 2, we propose a prototype resource broker architecture based on Netscape and Java. Section 3 discusses economic issues involved in a global computing scenario. In Section 4, we reflect on challenging technical problems, and sketch available solutions. Section 5 discusses encrypted computing and Section 6 discusses some preliminary performance results. We make a brief recapitulation of previous research and related work in Section 7, and conclude the paper in Section 8.

2 The Broker

We start with an overview of the broker architecture, then describe a very simple broker implementation to present the reader a concrete picture of this model. We then discuss the broker in more detail and examine responsibilities, components, and possible applications.

2.1 Architecture

The central idea in the SuperWeb architecture is a computing broker, which collects and monitors resources, and services requests for these resources. A user interacts with the broker in two ways, by registering or requesting resources. The broker matches clients with host machines for running computations. Figure 1 depicts the interaction between three hosts and the broker after a client has submitted three tasks. The broker sends the tasks to the three hosts which had previously registered their CPUs. To avoid contention at the broker, it does not ship actual client tasks, but rather sends a URL pointer to the tasks. The hosts then directly retrieve the chosen task from the client.

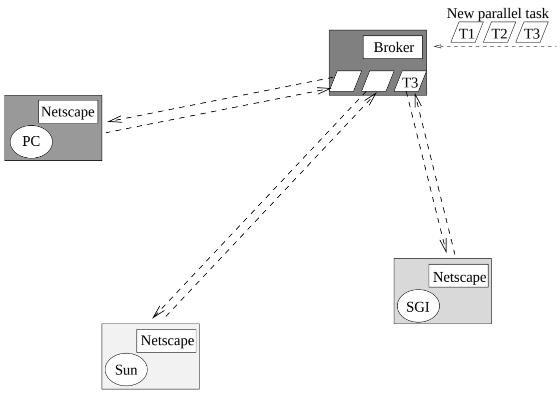

Figure 1: *Three host machines receive client tasks from the broker.*

A key design feature of this architecture is that the host software runs completely at the user-level, and only requires that registered machines have a WWW browser, such as Netscape, which can run untrusted code, such as Java applets. We do not require clients to actually have access (e.g., login privileges) to the host machines running their code.

2.2 Prototype Implementation

We exemplify the idea of brokers with the very first prototype of SuperWeb that we implemented using Netscape browsers and Java applets. This was a rudimentary implementation based on HORB [HOR]. It had poor performance and none of the enhancements discussed later. Nevertheless, it is a good practical illustration of how SuperWeb can be implemented.

In the prototype, hosts register their CPU by visiting a broker web page. This causes the host to download a Java applet, which stays in contact with the broker and downloads client's Java/HORB objects for execution until the user leaves that broker web page. Clients submit tasks by visiting a different web page and submitting a form to request resources. On this form, they specify the Java program (also a URL) they wish to have executed.

As clients request resources, the broker maps these to the next available host, using a simple round-robin algorithm. Once a host is selected, the broker retrieves the client applet (URL) and submits it to the host. The host then starts executing it and delivers the result directly to the client. Finally, after finishing a task, the host contacts the broker again and signals its readiness for new tasks.

2.3 Broker Responsibilities

The broker is responsible for three main activities: registering hosts, accepting client requests, and overseeing the interaction between the client and host.

Registering hosts: Owners of machines can register with the broker, allowing their computer to become part of the SuperWeb. They specify when their machines can be used and what percentage of resources are available.

Accepting client requests: Users make requests to the broker for the use of resources. A typical request would ask for a particular quality of service. This includes such properties as a time limit (how soon the results should be returned), required bandwidth, memory, floating point accuracy, etc. It could even include a request for a particular type of machine, or group of machines.

Overseeing client/host interaction: The broker must match host machines with client tasks. Conceptually, the broker itself sends the tasks to the hosts (and is responsible for sending the results back to the client). In practice, this would make the broker a bottleneck. Therefore, in the actual implementation the broker only "introduces" the client and the host to one another and is not involved in further communication. In this scheme the broker is only responsible for keeping accounting and other bookkeeping information for the host and the client and is not involved in the transfer of tasks, data, or results between them.

Client applications are not limited to sequential computations. Users can submit computations that involve parallel or distributed subtasks, and the broker can map a request onto multiple computers: either a tightly-coupled network of workstations, or a number of scattered resources, depending on the communication requirements. Furthermore, the brokers themselves may be distributed processes, allowing this model to scale very well as thousands of machines register.

2.4 Broker Modules

The broker consists of three modules: the interface, the scheduler, and the accounting module.

Broker Interface: This is what users see when registering their machines or requesting resources. The initial version of this module is the most straightforward to build, as it is just a web page with forms for the client or hosts to fill in.

Scheduler: The scheduler has the job of assigning incoming tasks to available resources, based on the specified task requirements and hosts capabilities. The scheduling algorithm is heuristic, since mapping resources optimally is computationally infeasible, especially in the scale and with the range of variables envisioned. Moreover, the broker has only incomplete information, since hosts might become unreachable or might make false claims about their capabilities, like CPU power, memory and disk space, network bandwidth, or deadlines. To estimate the host capabilities, the broker does not trust the host completely, but instead runs small micro-benchmarks to characterize the resources and determine the machine's capabilities. The results are used to heuristically find a good match between task requirements and host machines.

Accounting Module: This module keeps track of the charges incurred by the client, and credits gained by the hosts. This can be viewed as part of the scheduler, since the cost of the services rendered is related to what host is chosen.

2.5 Examples and Uses

We envision three types of users for the broker: the typical user looking for some extra computational power; supercomputer users, looking to acquire hundreds or thousands of machines; and Internet station owners, who otherwise have no usable computing resources of their own.

Typical User: The average user already has a complete workstation, but occasionally needs extra resources. He is willing to trade some of his CPU time (by registering his machine), in return for cycles from other machines. This is a win-win situation since most machines have idle cycles. For example, many users would not hesitate to register their machine after hours, or during their coffee break or meetings.

An important prospective source of clients and hosts is the multi-player game software market. There already exist chess, bridge, flight-simulator and other game servers that allow real-time gaming over the Internet among subscribers (users) or with a computer as opponent. A single centralized server is hardly powerful enough for an interactive virtual reality game, or computer opponent for strategy games, which essentially are distributed computations. Under the SuperWeb model, the server can play the role of a broker and users may register their machines as hosts during idle time. The registered hosts can be used by the server itself or by other currently active subscribers that have slow machines. In exchange, the owner of a host may get a reduced subscription fee or other bonuses.

Another possibility are software *parasites*, which we define as useful computations that run in the background. A user downloading a game, for example, can pay for it by willingly allowing a parasite to use part of her computer resources while she plays the game.

Supercomputing: Large scale supercomputing is another potential application of our system. Since SuperWeb requires no operating system changes, and clients need no direct access to all hosts, the potential pool of participating machines is very large. Therefore, for course-grained parallel tasks with little communication, we could feasibly support the worlds largest supercomputer.

Using such a large number of machines is probably too expensive, since most people would register their machines in return for something more than curiosity. Corporations may be willing to pay for the use of thousands of machines in order to run large parallel applications, but only if their work is secure and the results can be trusted. This issue is discussed later in Section 5.

Internet Stations: Another emerging technology is the Internet station (also called *thin client*, *web station* or *network computer*): a relatively cheap box that allows connection to the Internet. Internet stations lack a general purpose operating system and can only run programs that are downloaded. The broker model would provided such clients with the ability to choose an arbitrary host for all sorts of computing services.

2.6 Broker Extensions

We plan to incorporate the following features in future implementations of the SuperWeb broker.

Other languages support: Currently we support only Java for client tasks. We can extend the implementation to allow arbitrary executables and languages. Also, we might make use of other mobile languages, like the "E" programming language [Com96] or the *limbo* language from the *inferno* operating system [LT], which promise better security features or faster execution.

Verifying results: One important concern is the integrity of the result returned from an anonymous host to the client. The broker could assist the client in verifying that the output from client computations is correct and can be trusted.

Service requests: Instead of just providing requests for hosts, the broker could provide requests for particular services, such as "Run Fast Fourier Transform on this input", or "produce a graph of this data using Excel 4.0". (There already exist projects of this nature [CD95, NEO].) This notion could also be embraced by sites serving the network computers mentioned above.

3 Economic Model

The SuperWeb architecture makes computational resources a freely tradeable commodity that can be captured by a microeconomic model of supply and demand. Like other commodity exchanges, fluctuations in supply and demand result in a changed costs for the services provided. As an example, suppose the SuperWeb is serving a number of supercomputing researchers. Just before a supercomputing conference deadline, the demand for resources is likely to grow. As a result, the cost of requesting a hundred CPU hours will increase, as will the demand to have results returned in a timely fashion. Conversely, after the deadline has passed, the cost of such resources will drop. When the host registers its computational resources with a broker, and the broker can give it tasks to execute, the broker and the host establish a *contract*. Part of this contract is the *quality of service* which the host promises. This quality might be measured in reliability promises or time and availability constraints. The contract also specifies what the host will receive in exchange for its services and what is the penalty for the host in case it fails to complete the task properly.

A similar approach has been suggested by the Mariposa project and is discussed in detail in [SDK+94]. Mariposa is a WAN distributed database management system that bases its query and storage optimization on a microeconomic model. Mariposa's model includes brokers, advertising and bidding for requests.

SuperWeb's model will most likely be based on currency-based trading. A micro-currency as described in [Tan95, Man95] might be used as the cost of a short computation is very small. A currency-based SuperWeb economy strikingly resembles the economy of the electric utility industry. The supercomputer centers play the part of power plant operators, whereas network computer owners stand for pure consumers, and private hosts for consumers with a wind or solar powered generator feeding surplus power into the power grid.

Alternatively, it may be more convenient to view broker economics in terms of barter. Depending on what the broker has to offer, services rendered might be paid for with free CPU time (e.g., to run another supercomputing application), information access (e.g., free access to air-travel information systems), free backups, cheap access to the Internet, access to software licenses, etc. A user can trade extra CPU time now for CPU tonight. This model would provide the incentive to use the system; since most machines spend a large percentage of their time idle, many people may be willing to trade this "free time" for something useful. Barter works well if we have only a few commodities to trade (e.g., CPU time, disk space, bandwidth). Note that the perceived value of resources might be higher than the actual value (e.g., CPU time for free games).

An economic model based on the broker needs to incorporate more than just the cost of the requested computation and expected quality of service. There are also the "middle-man" issues such as the broker's commission, and the cost to ship the computation. The latter overhead could have a severe impact on some types of service: the cost to transmit a large program, large data input, and large data output (e.g., a graphics program that processes large images) might be prohibitive: it then would be cheaper to perform the computation locally. The extra cost not only consists of the extra time involved to send the data, but also the direct cost of using the network, assuming that is the only way to access the host computer. Currently, this cost is often hidden from users in academia, since free Internet connections are provided. Unsubsidized users are charged a hefty price by the telecommunication companies, although bandwidth continues to increase and prices to drop.

An interesting aspect of the anticipated cyber-economy is the extremely high depreciation of computer hardware. Since our computing infrastructure facilitates renting of resources, it might be better to rent half of a computer on a nearby computer center than to buy a whole computer now, and leave most of its CPU cycles unused. Service centers will have to continuously adapt prices to remain competitive. One could also envision trading derivatives, such as CPU time futures.

4 Technical Issues

The computing environment we envision has several unique properties that, taken together, distinguish it from other distributed systems and pose challenging technical problems that need to be addressed.

Interoperability: The participating computers are *heterogeneous*, meaning that they have different CPUs and run different operating systems. This complicates the transfer of programs between clients and hosts. Different solutions are available: For instance, portable, interpreted bytecode (e.g., Java bytecode) might be used. Alternatively, source code or precompiled binaries might be sent to hosts capable of executing such tasks.

Execution speed: When executing portable code the host has to translate them to its native machine language. Whether done by interpretation or by compilation, translation presents execution overhead. Furthermore, providing host security incurs additional run-time overhead.

Security: Except for the broker, all participants in SuperWeb are untrusted: the client and the host need not trust, or even know of, one another. As a consequence their interaction raises a variety of security problems for hosts as well as clients.

Communication: SuperWeb is a *global* environment in which the participating computers can be thousands of miles apart. Communication between the client and the host, and between cooperating hosts of a distributed task can be quite slow or even unreliable, unlike in a tightly coupled parallel machine or in a network of workstations. This limits the class of parallel applications that we can run efficiently.

Although these are difficult problems, a lot of research has focused on them. As we show in the following paragraphs, solutions to most of the individual problems are already available.

4.1 Interoperability

The key operation in SuperWeb is *task shipping*: submitting tasks from *any* client to *any* host. This is easy to achieve in a homogeneous system where all machines are compatible, since then binary executables can be sent. Unfortunately, in a heterogeneous environment like the Internet, hosts and clients may have different instruction sets, integer/word sizes, or operating systems. Generally, there are two different ways to ship tasks: using multiple binary executables or using a machine independent language.

Machine independent code is the straight-forward way to allow arbitrary machines to execute one another's code. With this approach, the client compiles its task to the machine independent language and then sends it to the host. The host executes the task either by interpreting it, or by compiling and then running it. Java [GM95] employs this approach. It defines both a high-level programming language for writing programs and a machine-independent virtual machine language, Java bytecode. Although it provides additional compatibility and security guarantees, it is not necessary to invent a new high-level language in order to ensure portability. For example, Omniware [Sof95] only defines a machine-independent task shipping language and any high-level language can be compiled to it.

Another feasible solution is to submit tasks as machine-specific binaries. This offers high execution speed, but limits the choice of hosts on which the task can run in a heterogeneous environment. In an organization's mini-SuperWeb, encompassing only homogeneous machines, this might be the most effective solution. In a heterogeneous environment, the broker is responsible to map tasks to compatible hosts only.

4.2 Execution speed

One of the main reason people would submit tasks for execution on remote computers is that their own machine alone does not provide enough performance. Therefore, if SuperWeb is to succeed, it must provide fast task execution. This can be achieved either by using binary executables for task shipping or by using a portable language in combination with modern compilation techniques and optimizations.

Currently there is a misconception among many users that portable languages like Java are inherently slow. This perception is due to high interpretation overhead present in current Java implementations which results in a 5-50 times slowdown compared to compiled C code. Fortunately this is not a fundamental problem with portable code, but rather a shortcoming of current implementations. For example, in the Omniware [Sof95] system the portable code is compiled to machine code at the host before execution. As a result Omniware tasks run within 9.3% of native machine code. Even for Java, modern techniques such as just-in-time (JIT) compilation and dynamic compilation can eliminate most of the interpretation overhead by compiling pieces of the application to machine code. Using these approaches, Java bytecode code can be executed at a speed close to that of compiled C code. Commercial just-in-time compilers from a number of vendors are readily available, and the ubiquitous Netscape browser uses JIT techniques for some platforms.

If the ultimate in execution speed is necessary, tasks can be shipped as plain machine binaries. Theoretically this should ensure execution speed equivalent to that of native machine code. This is indeed possible if the host trusts the client and is willing to run the executable without any security precautions. In practice there is always extra overhead that comes from the safety checks needed to guarantee the security of the host.

4.3 Host Security

Executing an anonymous piece of code poses integrity and security threats to the host. Consider the dangers of downloading a binary executable from some anonymous FTP site and using it. A buggy binary executable can interfere with other running applications or, in some operating systems, crash the whole machine. A malicious executable (e.g., one containing a virus or a Trojan horse) can delete important files or secretly transfer sensitive data of the unsuspecting user. Observe that these dangers are not limited to binary executables. Rather, they come from the fact that the anonymous program has unlimited access to system resources. An interpreted program may cause as much damage as a binary executable if the interpreter does not limit its access to memory, files, network, CPU usage, and other resources.[2] To deal with these dangers, the host should run untrusted code inside a confined environment that allows only limited access to system resources.

Confining untrusted programs is technically simple to achieve when the code is being interpreted. Since every single instruction is executed under the control of the interpreter, the interpreter can check the instruction arguments to ensure that all accesses are legal before executing the instruction. For example, two existing Java-enabled web browsers guard file and network accesses, as well as system calls. HotJava allows the user to select the strictness of the policy while Netscape prohibits all local file accesses and allows network connections only to the server from which the applet was downloaded. The "E" programming language [Com96], which basically is a spiced-up version of Java, provides for a more general and flexible framework to define the capabilities of downloaded code.

Ensuring protection for the host for compiled code or plain binary executables is more complicated since there is no controlling entity that can check every instruction for safety. The recently developed technique of *Software Fault Isolation* (SFI) [Sof95] inserts the protection checks in the machine code itself. SFI allows the safe execution of arbitrary untrusted binaries. Before running a piece of machine code, it is instrumented by placing safety checks in front of every potentially dangerous instruction. Later, at run time, if a safety check determines that the executable is about to access a prohibited resource, the perpetrating instruction is aborted. Other variations of SFI [GWTB96] do not modify the executable at all, but instead use operating system specific mechanisms to intercept and screen dangerous system calls.

The methods discussed so far provide security for the host, using user-level methods that do not rely on mechanisms provided by the operating system. One can imagine alternative approaches where the operating system's mechanisms for authentication, memory protection and access control are used (e.g., using *chroot*.) This approach is less flexible since it relies on OS functionality that may not be present, or be implemented differently, on various systems. Also, this approach

[2] A typical example is the ghostview interpreter for postscript. Numerous security bugs were discovered in this interpreter.

is cumbersome, since it requires system administrator privileges from the user running the host server software. Nevertheless it appears to be a viable alternative for machines used as dedicated SuperWeb hosts.

One general concern for the host is whether the implementation of the method used to provide security can be trusted. For example the current Java interpreters have not been formally verified for correctness and therefore a user can not be absolutely certain that the Java interpreter has no bugs that expose security holes. We recognize that this concern will stop certain users from participating in SuperWeb. Nevertheless the potential danger is apparently perceived as acceptable by the large number of people who are currently using Java-enabled web browsers on a daily basis.

4.4 Correctness

The economic incentive in SuperWeb raises the very real possibility of incorrect behavior (fraud). To be useful, SuperWeb must provide at least some mechanism to guarantee correct execution. How does a client know that the anonymous host that volunteered to execute a task produced the correct result? In an attempt to earn easy cyber-money the host might have faked a computation and returned a wrong result. Or, maybe the host had a cheap faulty processor (e.g., an old Pentium) producing false floating point results.

In some cases it may be easy and computationally cheap for the client to verify the correctness of the result. For example, if the task was to solve a system of linear equations, the client can check that the result is correct by performing a matrix by vector multiplication (which is much cheaper than finding the solution). In other cases it is possible to statistically verify that the result is correct with high probability. For example, if the task is to render a 1000x1000 image using ray-tracing, the client itself can personally compute a few hundred random pixels of the image and compare them with the pixels from the image computed by the host. A more general technique is for the client (or broker) to insert a checksum computation into its program; this checksum is returned with the results and checked to assure that the program was actually run. Finally the broker or client can replicate the computation by submitting multiple identical tasks to independent hosts and comparing the returned results. This at least doubles the amount of work done by the computation, but gives a reasonable guarantee that the results are correct. Some work in the area of fault-tolerance with *malicious* failures can be found in [YCW92].

A related concern is fault-tolerance caused by non-malicious failures: clients or hosts crash or lose connectivity, networks go down, etc. This related issue is also important, though easier to handle, and covered sufficiently in the literature.

4.5 Communication speed

For many parallel applications the communication network is a bottleneck. For those applications, a high-performance network is essential for efficiency. At present, neither the latency nor the bandwidth across the Internet are satisfactory. While emerging technologies will eventually increase the bandwidth to satisfy any requirement, the improvement of the latency between geographically separated computers is limited by the speed of light. This severely affects the efficiency of applications requiring extremely low latency that run globally on our infrastructure.

It seems that for now, only applications that require relatively little communication will benefit from our *global* computing infrastructure. However, the communication capabilities of networks of workstations (NOW's) or parallel computers can be exploited. The broker can be instructed to find and map parallel tasks that require very fast communication to a local NOW or parallel computer.

5 Encrypted Computing

A big concern in our global computing approach is that untrusted hosts get access to client data or client algorithms. In this section, we present a methodology for computing on encrypted data and encrypting the computation itself. Although we do not present a complete solution to this challenging problem, we want to show why encrypted computing is important in a commercial world with mobile or portable code. To this end, we begin with an example scenario.

Assume the fictitious company Lemon Computers, Inc. wants to run simulations for their new supercomputer chip, using our proposed broker architecture. Since the required computational power is so large, the only sensible choice for the broker might be to assign the task to Lemon's largest competitor, Prune Corp., which has enough spare cycles to offer right now. Lemon can not be sure that Prune's employees do not eavesdrop on the data sent to their computers and make a copy of the data, using it to develop their own clone of Lemon's new chip. So Lemon wants to encrypt the data before shipping it to the broker to prevent eavesdroppers on the net from snooping on the data. More importantly, the data should remain encrypted on the machines performing the calculation, and should never be decrypted during the calculation. Likewise, the result of the computation should also be produced in encrypted form and not be decrypted before reaching Lemon's headquarters.

Ideally, we would like to hide three aspects of the computation: the input, the output, and the actual algorithm. Encryption is one solution. Another way to achieve this is to split the computation into small segments, such that no part by itself reveals any useful information about the complete computation. This can be generalized to a division scheme that guarantees that even k out of a total of n parts of the computation do not reveal meaningful information.

5.1 Processing Encrypted Data

The idea that clients let service providers work with their private information without having access to it might look utopian at first glance, but a closer look reveals that the idea of encrypted computing is neither completely new nor impractical. Indeed, Feigenbaum [Fei85] showed that there are many problems that are encryptable, i.e., allow for an encryption scheme sketched above. The main problems encountered are making the encryption/decryption process on the client side computationally inexpensive with respect to the actual computation, and to make malicious decryption on the host side computationally infeasible.

We can formalize the processing of encrypted data, as shown in Figure 2. Assume a client wants to perform a function f on input x and output $f(x)$. For some reasons, e.g., lack of resources, the client cannot compute $f(x)$ directly. However, a second party, the host, is willing and able to compute functions for the client. We define f as fully encryptable if there exists an encoding function E, a decoding function D, and a function f_E that satisfies the equation $f(x) = D(f_E(E(x))$ such that the host, knowing $E(x)$ and f_E cannot derive x, f or $f(x)$. This means that the input x, the output $f(x)$ and the algorithm f are hidden from the host. The client can now send the encrypted input $E(x)$ and the encrypted program f_E to the host, which computes $f_E(E(x))$ and sends it back to the client, which then decodes $f(x) = D(f_E(E(x))$. The properties of E, D, and f_E determine what part of the computation is actually encrypted. Ideally, the host should not gain knowledge about the input x, the output $f(x)$, or the algorithm f_E.

Note the fundamental difference between traditional cryptography and encrypted computing: For a traditional encrypted transmission between two users, they first have to set up key(s). This can be done using the well known Diffie-Hellman protocol for public-key encryption. Alternatively, the two users might agree on keys using a secure channel provided by other means. This problem

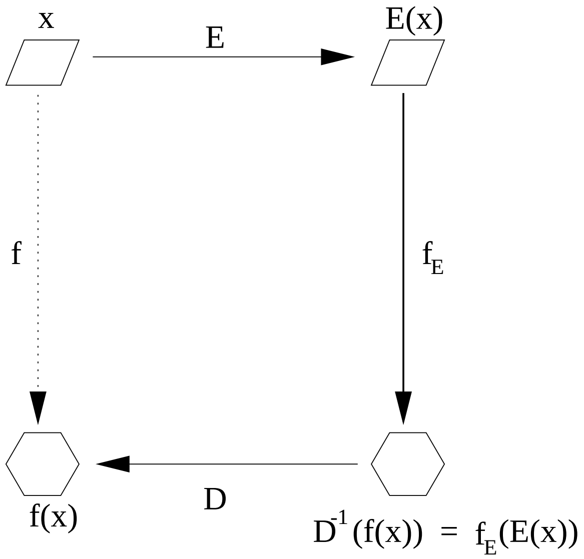

Figure 2: A general scheme for encrypted computing: A client, unwilling to compute $f(x)$ gives the encrypted input $E(x)$ to the host, which computes an encrypted version of the output and returns it back to the client which can then decode it.

is not present in our encrypted computing example, since the encryption key is not transmitted at all but rather resides only at the client. The true challenge in encrypted computing is to send and receive encrypted data only, and never decrypt the data for the computation, but directly operate on the encrypted data.

5.2 Simple Examples

Let us start with a trivial example: The client wants to add up two integers x and y, assuming in the following that no overflow occurs. The client encrypts x and y by adding a random number h_1 and h_2, respectively and sends the resulting integers $x + h_1$ and $x + h_2$ to the host. The host then computes $(x + h_1) + (y + h_2) = (x + y) + (h_1 + h_2)$, and sends this result back to the client, who then can retrieve the desired result $x + y$ by subtracting $h_1 + h_2$ from the host's reply. Since h_1 and h_2 act as a one-time pad, this encryption scheme is provably secure for the input and output. On the other hand, it is not practical, since the client has to make 4 arithmetic operations to compute the sum as opposed to just one. However, other, more complicated problems can be encrypted and decrypted with very small overhead. Feigenbaum gives an example for the hard problem of finding a discrete logarithm [Fei85].

A more detailed treatise for addition and multiplication of integers is given by Ahituv et al. [ALN87] and Yu and Yu [YY91]. The encryption presented by Yu and Yu [YY91] is linear and thus allows for easy repeated addition and multiplication of encrypted integers. The algorithm is based on a time-reversal diffusion process. The algorithms in [ALN87] is based on linear homomorphic functions. Unfortunately, no proper security analysis is provided for any of those algorithms. Furthermore, it is not clear how one can devise a scheme for larger calculations that do not require frequent interaction between the client and the host. In particular, we would like to find a generic method of translating any code sequence into a code sequence that works on encrypted data. This involves the realization of a comparison between two encrypted numbers and other primitives. We would also like to identify the amount of leaked information. Abadi et al. [AFK89] showed that it is unlikely hard problems can be encrypted without leaking at least some properties of the input data.

5.3 Real-Life Examples

So far, the examples given covered only basic operations like summations, multiplications or (discrete) logarithms. We do not (yet) know of a feasible universal method of converting an arbitrary algorithm into a safe algorithm working on encrypted data. However, we show a few more realistic examples that might actually arise in real life applications. This leaves room for hope that there might be efficient encryptable algorithms for a larger variety of applications.

Let us consider a Fourier transform $\hat{f}(.)$ of a function $f(.)$. The Fourier Transform is a linear transformation, i.e., for two scalar values s and r and two functions $f(.)$ and $g(.)$, the following relation holds: $(r\widehat{f+s}g)(.) = r\hat{f}(.) + s\hat{g}(.)$. Assume the client has a given function $f(.)$ and wants to compute $\hat{f}(.)$. Instead of sending $f(.)$ to the host, the client chooses a random *hiding function* h. Since a function is usually given by its values on a number of points, h is nothing but a vector of values that can be chosen randomly. Now the client sends out the two functions $h(.)$ and $g(.) = f(.) + h(.)$ to two independent hosts. Since neither host knows of the other's transactions (some feature the broker architecture has to provide), they cannot recompute f from g and h. Instead, the hosts compute $\hat{h}$ and $\hat{g}$ respectively. After receiving the results, the client combines the results into $\hat{f}$ using the equality $\hat{f}(.) = \hat{g}(.) - \hat{h}(.)$. If the client cannot be assured that its requests are being watched and used to reconstruct $f(.)$, it might have a large list of sample hiding functions h together with the corresponding precomputed Fourier transform $\hat{h}$, such that h need not be transmitted.

The above example illustrates a feature that can be found in most examples of encrypted computing: the original data is being perturbed by a function (key) not known to the host, which is still able to work on the data and come up with a solution for the perturbed problem. The client can then remove the effect of the perturbation from the solution to obtain the correct solution. However, introducing arbitrary perturbations might upset numerical stability of the used algorithms. The admissible range of perturbations not only depends on the algorithm, but also on the data, since a perturbation numerically stable for one data set might be unstable for another one. It is up to the application to ensure the perturbation does not affect numeric stability and sufficiently blurs the original data.

Other examples in scientific computing include matrix calculations. Instead of asking the host for a solution to the linear equation $Ax = b$ for an $n \times n$ matrix A and n-vectors x and b, the client might multiply both A and b by a regular matrix T and then ask the host for the solution of the equation $(TA)x = (Tb)$, which leaves the host in the dark about the original input A and b. Note that our remarks about numerical stability apply for matrix calculations.

5.4 Encrypting Algorithms

Finally, we want to discuss the following problem: Assume the client wants to perform a certain algorithm on the host's machine. The client can prevent the host from knowing what data it worked on by using one of the previous techniques, but the host still can discover the algorithm of the computation. Knowing the algorithm, the host might deduce some information (e.g., that the client solves a lot of differential equations used for shock wave modeling). Moreover, the host might find the program sent to it very useful, and decide to keep it and use it on its own data.

While preventing reverse-engineering is hard, preventing the reuse of a given program on a different input is even harder. We currently lack a framework to classify the information-theoretic security of algorithm encryptions. Therefore, we will just describe one way of preventing code reuse without giving a security analysis. We modify the algorithm to depend on the encrypted input. This in effect would make it useless for any other inputs, leaving the host without any advantage. One first step in this direction would be to use dynamic compilation and specialization techniques, which enables the client to send the host a modified version of the program that is optimized for the specific input and will not produce correct output for other inputs. For instance, consider the assignment $B := 3 * A + 2 * d * C$ with matrices A, B, C and scalar d. Assume A is a (constant) matrix provided as input with $A[1,3] = 2$ and $A[2,3] = 3$. Then, the above assignment can be replaced by $B := A[1,3] * A + A[2,3] * d * C$ for the specific input matrix A. The transformed program will then run only if the input does have the same entries in $A[1,3]$ and $A[2,3]$. If these specializations are employed throughout a program, it will be hard, if not impossible to run it on other input or reverse-engineer it.

6 Measurements

In this section we cite and briefly discuss performance numbers from a second prototype implementation of SuperWeb, called Javelin [CCI+97]. For a more detailed presentation of the design issues, the performance results and the application used the reader is referred to [CCI+97]. The experiments were conducted on a heterogeneous network of computers consisting of a 64-node Meiko CS-2, Sun UltraSparcs, and Pentium PCs connected by an SCI and 10 Mbit Ethernet networks. The tests were done with the Mersenne prime application. This application tests whether or not a large number is a Mersenne prime, i.e., a prime of the form $2^p - 1$. This is a very coarse grained application and thus is well suited for executing on the SuperWeb architecture.

The Netscape Navigator browser was used on all platforms except the Pentium PCs. The Pentium PCs were running the Windows 95 version of Microsoft Internet Explorer which has a just-in-time compiler.

The first set of measurements were performed on clusters of identical machines. This test shows that one can obtain good speedup for the Mersenne Prime Application. Figure 3 presents the speedup curves for test runs on a 64-node Meiko CS-2.

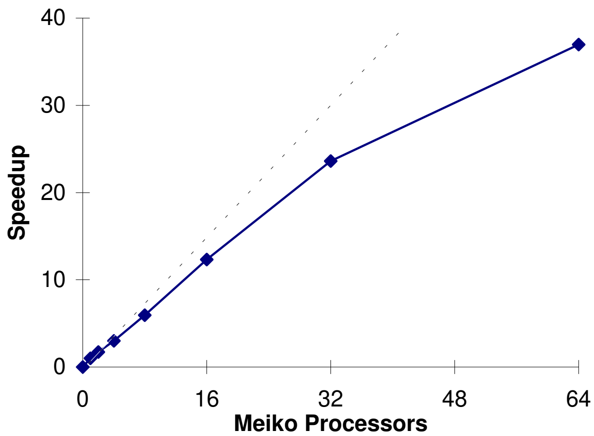

Figure 3: *Speedup curves for the Mersenne primality test of all prime exponents less than 3000. Tests were run on a 64-node Meiko CS-2 (Sparc 10 processors running Solaris 2.3).*

The second test was on a heterogeneous environment consisting of 2 PCs running Windows 95, 8 UltraSparcs and 32 Meiko nodes (Figure 4). The main purpose of this test was to show that the Javelin prototype is capable of running parallel tasks on a variety of computing platforms. This reflects a possible real situation of computing resources registered at a broker at some moment. Figure 4 also shows the importance of advanced compilation techniques. Note that the two 166MHz PCs using a just-in-time Java compiler perform just as fast as the 32 nodes of the Meiko CS-2 running at 40MHz using the Netscape Java interpreter. Combining the resources of all machines resulted in an improvement, albeit not a substantial one, due to the big difference in speed between the compiled and interpreted versions.

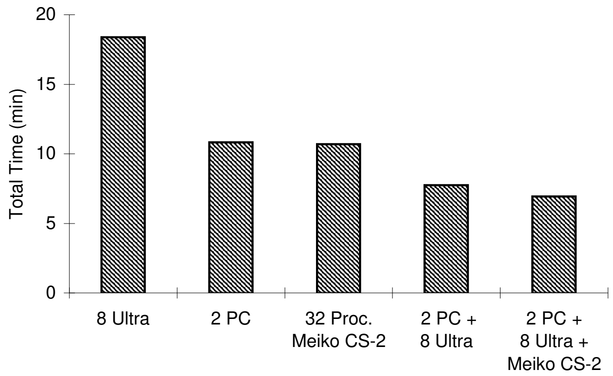

Figure 4: *Mersenne primality test of all prime exponents less than 3000. Tests were run on a heterogeneous platform consisting of Pentiums, Meiko CS-2 nodes (Sparc 10s) and UltraSparcs.*

7 Related Work

Several approaches have been proposed recently to provide a Java-based metacomputing infrastructure for large scale networks. These include *ATLAS* [BBB96], *Charlotte* [BKKW96], and *ParaWeb* [BSST96]. All three projects are designed explicitly to run parallel applications and provide a specific programming model. None of the three include an economic model or way of providing incentives for participation.

ATLAS provides a global computing model based on Java and on the Cilk programming model [BJK+95] which is restricted to tree-based computations. ATLAS ensures scalability using a hierarchy of managers (their equivalent of SuperWeb brokers). The current implementation uses native libraries which may raise some portability problems. Charlotte provides distributed shared memory and uses a fork-join model for parallel programming. A distinctive feature of this project is the eager scheduling of tasks where a task may be submitted to several servers to provide for fault tolerance and ensure timely execution. Both Charlotte and ATLAS provide fault tolerance based on the fact that each task is atomic. In Charlotte, the changes to the shared memory become visible only after a task completes successfully. This allows a task to be resubmitted to a different server, in case the original server fails. In ATLAS each subtask is computed by a subtree in the hierarchy of servers. Any subtask that does not complete times out and is recomputed from a checkpoint file local to its subtree.

ParaWeb provides two separate implementations of a global computing infrastructure each with a different programming model. Their Java Parallel Class Library implementation provides new Java classes that provide a message passing framework for spawning threads on remote machines and sending and receiving messages. ParaWeb's Java Parallel Runtime System is implemented by modifying the Java interpreter to provide global shared memory and allow transparent instantiations of threads on remote machines.

Earlier network computing approaches, such as *Network of Workstations (NOW)*, *PVM* [Sun90], *MPI* [MPI94], *Linda* [WL88], *CORBA* [YD96], *Piranha* [GK92], and *Legion* [GWF+94] can be distinguished from this project in that they either require the user to have login access to all machines used in the computation, require operating system modifications, or require the maintenance of binaries for all architectures used in the computation. Thus, they are limited in size and have a high administrative overhead. Although more recent systems, such as *JPVM* [Fer] and *Java-MPI* [Tay], overcome the heterogeneity of today's networks, they are not intended to execute on anonymous machines. In contrast, our project aims at allowing any individual user to register its computing resources with a broker using an arbitrary Java-capable browser such as the Netscape Navigator and without violating any local security policies. Because of the simplicity of use, thousands of users could easily register their computing resources with a broker, giving us the ability to harness an extraordinary amount of computational power.

The use of Java as a means for building distributed systems that execute throughout the Internet has also been recently proposed by Chandy et al. [CDL+96] and Fox et al. [FF96] and studied in [Sar96]. Some of the mechanisms needed to implement the proposed framework may eventually be realized through recently released standard Java components such as *Remote Method Invocation (RMI)* [Sun96b] and *Object Serialization* [Sun96a], or have already been provided by other research groups (for example [SC96], [Ros] and [Gut]). Recently, a large variety of Java performance boosters have become available [FJa]. Most commercial compiler vendors offer JIT Java compilers, and new web browsers use JIT techniques, too. Other languages worth considering, besides Java, include the "E" programming language [Com96] which is very similar to but offers a more flexible security framework than Java, and the Limbo programming language in the Inferno operating system [LT], which allows, among other features, for easier and more efficient just-in-time compilation.

The secure execution of arbitrary binaries has been recently addressed by at least two techniques. First, *software-based fault isolation* techniques [Sof95] guard against insecure system calls of programs by patching their binaries. Second, *secure remote helper applications* [GWTB96] use operating system tracing facilities to limit the use of resources that could violate system integrity.

8 Conclusion

In this paper, we have proposed a global computing architecture. It allows any machine connected to the Internet to make a portion of their computing resources available to remote clients and also utilize resources offered by other machines. A broker is used to register and claim resources, and to take care of accounting. We also presented a rudimentary Netscape-based broker architecture using Java. We discussed economic aspects of our proposal and gave scenarios of possible developments and various uses of the proposed infrastructure, from coarse-grained supercomputing applications running on thousands of nodes, to Internet stations.

We anticipate several challenging technical problems when implementing a complete version of the proposed broker architecture. In particular, we examined the high interpretation overhead of Java byte code, and solutions like just-in-time compilation and dynamic compilation. We also examined security and correctness problems arising when executing untrusted code. Finally, we raised the problem of encrypted computing, a technique to hide programs or data from the remote hosts executing them. Although we currently are unaware of any attempt to give a precise definition of algorithm encryption, we believe encrypted computing to be a major issue in large-scale implementation of SuperWeb-like architectures.

While the presented infrastructure is primarily intended to work across the whole Internet, it is also usable on a smaller scale. In fact, we anticipate that its initial implementations will be inside large organizations administering their own Intranets. This would allow such organizations to use their complete hardware base more efficiently. At the same time this simplifies several of the technical problems that are not yet solved completely, in particular, the security and privacy issues.

In our further research, we plan to make our rudimentary broker architecture more usable and user-friendly, improve its performance, and include security, accounting, and authentication mechanisms.

9 Acknowledgments

This work was supported by the National Science Foundation under NSF CAREER Award CCR-9502661 and NSF Postdoctoral Award ASC-9504291. Computational resources were provided by NSF Instrumentation Grant CDA-9216202, NSF Infrastructure Grant CDA-9529418 and Sun Microsystems. We would like to thank Peter Cappello, Bernd Christiansen, Mihai Ionescu, and Michael Neary for their work on Javelin and Bjorn Birnir for insightful discussions on encrypted computing.

References

[AFK89] M. Abadi, J. Feigenbaum, and J. Kilian. On Hiding Information from an Oracle. *Journal of Computer and System Sciences*, 39, 1989.

[ALN87] N. Ahituv, Y. Lapid, and S. Neumann. Processing Encrypted Data. *Communications of the ACM*, 30(9), 1987.

[BBB96] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[BKKW96] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *In Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[BSST96] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[CCI+97] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Computing Using Java. Technical report, University of California, Santa Barbara, 1997.

[CD95] H. Casanova and J. Dongarra. Netsolve: A Network Server for Solving Computational Science Problems. Technical Report CS-95-313, University of Tennesee, November 1995.

[CDL+96] K. M. Chandy, B. Dimitrov, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P. A. G. Sivilotti, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *In Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[Com96] Electric Communities. The E programming language, 1996. http://www.communities.com/e/epl.html.

[Fei85] J. Feigenbaum. Encrypting Problem Instances — Or, ..., Can You Take Advantage of Someone Without Having to Trust Him? In *Proceedings of the CRYPTO'85 Conference*, 85.

[Fer] A. Ferrari. JPVM – The Java Parallel Virtual Machine. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[FF96] G. Fox and W. Furmanski. Towards Web/Java based High Performance Distributed Computing – An Evolving Virtual Machine. In *In Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[FJa] Free Java compilers. http://webhackers.cygnus.com/ webhackers/projects/java.html.

[GK92] D. Gelernter and D. Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, July 1992.

[GM95] J. Gosling and H. McGilton. The Java Language Environment – A Whitepaper. Technical report, Sun Microsystems, October 1995.

[Gut] Y. S. Gutfreund. The WWWinda Orchestrator. http://info.gte.com/ftp/circus/Orchestrator/.

[GWF+94] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds, Jr. A Synopsis of the Legion Project. Technical Report CS-94-20, Department of Computer Science, University of Virginia, June 1994.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications — Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[HOR] HORB - Hirano's Object Request Broker. http://ring.etl.go.jp/openlab/horb/.

[LT] Inc. Lucent Technologies. inferno. http://inferno.bell-labs.com/inferno/.

[Man95] M. S. Manasse. The Millicent Protocols for Electronic Commerce. In *First USENIX Workshop on Electronic Commerce*, New York, NY, July 1995.

[MPI94] MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3), 1994.

[NEO] The NEOS Server. http://www.mcs.anl.gov/home/otc/Server/.

[Ros] D. Rossi. Jada. http://www.cs.unibo.it/˜rossi/jada/.

[Sar96] L. F. G. Sarmenta. Volunteer Computing. Draft Preliminary Concept Paper and Project Proposal, October 1996.

[SC96] P. A. G. Sivilotti and K. M. Chandy. Reliable Synchronization Primitives for Java Threads. Technical Report CS-TR-96-11, California Institute of Technology, June 1996.

[SDK+94] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin, TX, September 1994.

[Sof95] Colusa Software. *Omniware Technical Overview*. http://www.colusa.com, 1995.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. Technical Report ORNL/TM-11375, Dept. of Math and Computer Science, Emory University, Atlanta, GA, USA, February 1990.

[Sun96a] Sun Microsystems, Inc. *Java Object Serialization Specification*, May 1996. Revision 0.9.

[Sun96b] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, May 1996. Revision 0.9.

[Tan95] L. Tang. A Set of Protocols for Micropayments in Distributed Systems. In *First USENIX Workshop on Electronic Commerce*, New York, NY, July 1995.

[Tay] S. Taylor. Prototype Java-MPI Package. http://cisr.anu.edu.au/˜sam/java/ java_mpi_prototype.html.

[WL88] R. A. Whiteside and J. S. Leichter. Using Linda for Supercomputing on a Local Area Network. Technical Report YALEU/DCS/TR-638, Department of Computer Science, Yale University, New Haven, Connecticut, 1988.

[YCW92] K. Q. Yan, Y. H. Chin, and S. C. Wang. Optimal agreement protocol in malicious faulty processors and faulty links. *IEEE Transactions on Knowledge and Data Engineering*, 4(3), June 1992.

[YD96] Z. Yang and K. Duddy. CORBA: A Platform for Distributed Object Computing. *OSR*, 30(2), April 1996.

[YY91] K. W. Yu and T. L. Yu. Superimposing Encrypted Data. *Communications of the ACM*, 34(2), 1991.