Javelin: Internet-Based Parallel Computing Using Java

Peter Cappello, Bernd Christiansen, Mihai F. Ionescu
Michael O. Neary, Klaus E. Schauser, and Daniel Wu

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{*cappello, bernd, mionescu, neary, schauser, danielw*}*@cs.ucsb.edu*

Abstract

Java offers the basic infrastructure needed to integrate computers connected to the Internet into a seamless parallel computational resource: a flexible, easily-installed infrastructure for running coarse-grained parallel applications on numerous, anonymous machines. Ease of participation is seen as a key property for such a resource to realize the vision of a multiprocessing environment comprising thousands of computers.

We present *Javelin*, a Java-based infrastructure for global computing. The system is based on Internet software technology that is essentially ubiquitous: Web technology. Its architecture and implementation require participants to have access only to a Java-enabled Web browser. The security constraints implied by this, the resulting architecture, and current implementation are presented. The Javelin architecture is intended to be a substrate on which various programming models may be implemented. Several such models are presented: A Linda Tuple Space, an SPMD programming model with barriers, as well as support for message passing. Experimental results are given in the form of micro-benchmarks and a Mersenne Prime application that runs on a heterogeneous network of several parallel machines, workstations, and PCs.

Keywords: Global computing, Internet, Java, just-in-time compilation, World-Wide-Web.

1 Introduction

Of late, Global Computing has been a very active area of research. By the end of 1997, we expect to see 30 million hosts connected to the Internet. Ideally, the combined resources of thousand of computers could be harnessed to form a powerful global computing infrastructure. This is difficult to achieve, however, since most of the machines run different CPUs and different operating systems. Also, many machines are administered individually, making it extremely difficult to obtain login access and view file resources across a large set of machines. As a result, each individual or small group within an organization usually purchases its CPU time according to its maximum intended usage requirements, resulting in resource fragmentation and poor utilization.

The advent of a portable language system such as Java has provided tools and mechanisms to interconnect computer systems around the globe in cooperative work efforts. We work toward this goal by developing a flexible, scalable, easily-installed infrastructure that permits us to run coarse-grained parallel applications on numerous, anonymous machines on the Internet.

1.1 Concept and Architecture

We have designed a Java based global computing infrastructure called *Javelin*. In our model, there are three kinds of participating entities: brokers, clients and hosts.

A *client* is a process seeking computing resources; a *host* is a process offering computing resources. A *broker* is a process that coordinates the supply and demand for computing resources. Figure 1 illustrates our architecture. Clients register tasks to be run with the broker; hosts register their intention to run tasks with the broker. The broker assigns tasks to hosts who then run the assigned tasks and, when done, send results back to the clients. The role of a host or a client is not fixed. A machine may serve as a Javelin host when it is idle (e.g., during night hours), while being a client when its owner wants additional computing resources.

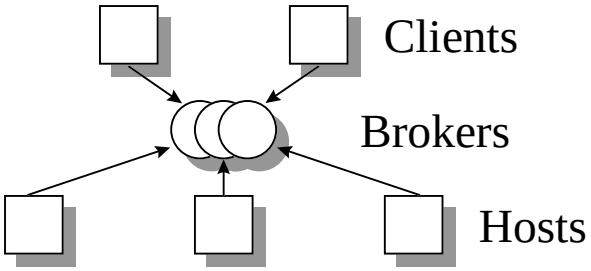

Figure 1: *The Javelin Architecture.*

1.2 Goals and Philosophy

Our most important goal is *simplicity*, i.e., to enable everyone connected to the Internet or an intranet to *easily* participate in Javelin. To this end, our design is based on widely used components: Web browsers and the portable language Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available to host part of parallel computations. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks from the broker. The simplicity of this approach makes it easy for a host to participate—all that is needed is a Java-capable web browser and the URL of the broker.

The infrastructure is also simple for clients to use. When a client needs to run a job, it first contacts the broker, again by pointing the web browser to a specific URL. It then provides the broker with the necessary program code in the form of a Java applet. The broker, in turn, schedules the jobs, distributing the program code among the available hosts. In our basic implementation, we don't even assume that a client has a HTTP server. All it needs is a web browser for uploading applets and spawning threads.

Additionally, since dial-up accounts and mobile computing are common, clients can disconnect from the network after submitting a task, and retrieve the result later. Similarly, hosts can receive a task, compute it off-line, and then re-establish the network connection to return the results.

The key technology underlying our approach is the ability to safely execute an untrusted piece of Java code sent over the Internet, something most WWW browsers can do (assuming all security bugs have been fixed). While existing Java virtual machines currently have a high interpretation overhead, new technologies such as just-in-time compilation, dynamic compilation, and software fault isolation are becoming commercially available. These will eventually eliminate most of the interpretation overhead to allow efficient execution of Java, while still maintaining safety. Since a standard WWW browser can register CPU time at the user-level, no changes to the operating system have to be made. We therefore expect thousands of hosts to participate.

1.3 Scope of our Work

The purpose of this work is to provide a simple yet efficient infrastructure that supports as many different programming models as possible without compromising portability and flexibility. As discussed in [AISS97], there are several important issues such an infrastructure must provide to be successful. In this paper, we do not attempt to study these issues, but instead focus on a particular design, discussing the constraints imposed by standard, Java-enabled web browsers. Future versions however will take these aspects into account. These issues include:

Security: Hosts must be able to trust the programs being executed on their machines. We therefore assume the usual Java security restrictions that apply to applets. For example, applets downloaded from the broker are permitted to communicate with the broker only. Additional Java security issues are discussed in [DFW96]. In this version, we do not provide levels of security beyond what Java-enabled web browsers offer.

Scalability: We envision extending our broker to a *scalable* network of brokers. The local broker then can contact other brokers if it does not have enough resources to run a job. This is a high priority enhancement.

Fault tolerance: The Broker is responsible for *scheduling*, *load balancing* and *fault tolerance*. However, in this version we do not concentrate on any of these issues.

Economic aspects: Participation in such a global computing infrastructure can be encouraged by providing hosts with benefits (e.g., digital cash, or bartering computer resources). The construction of such exchange mechanisms can be undertaken with the tools provided by Javelin.

Result verification: Clients must have some level of assurance that the results received from anonymous hosts are correct. Assurance methods exist for some kinds of computations. For general computations, it appears to be an interesting research topic.

Privacy: Clients may want to secure their program and data from spying by hosts.

The initial implementation of Javelin can be used by organizations with intranets consisting of a large number of heterogeneous computers. Running within a restricted intranet simplifies many security, privacy, and network performance issues that are present in full Internet applications.

1.4 Contributions

We present a system that makes it easy for computers all over the world to cooperate in parallel computations. In doing so, we:

- identify some architectural implications of using a heterogeneous, secure, Internet execution environment, in the form of Java applets running under standard Web browsers;
- present a set of abstract behaviors that must be implemented to realize the architecture, and discuss some implementation tradeoffs;
- design and implement a portable and flexible infrastructure, called Javelin, and illustrate how it can be used to implement several programming models, including the Linda tuple space, and SPMD programming;
- provide measurements of Javelin's current performance on a popular parallel application: Mersenne Primes. These measurements shed light on those aspects of the system that would most benefit from greater efficiency.

2 Design and Implementation

2.1 Basic Design Issues

One of the most important goals of our design is to minimize the administrative overheads associated with operations such as installing a broker, registering resources at a host and submitting tasks at the client. Since Web browsers are available on almost every platform and architecture and very common to almost every user, we expect our framework to allow for the registration of hosts and submission of tasks from clients using a web browser. Similarly, in implementing the broker we want to leverage existing HTTP server technology and ways of extending server functionality using CGI scripts [Rob] or servlets [Sun96c]. In other words, we want our architecture to be build on top of the existing Internet infrastructure.

Consequently, our design is based on the following premises:

Tasks: A task on which a host can operate is represented as an applet embedded in an HTML page. It resides on an HTTP server and is accessible through a URL.

Hosts: The simplest form of a host is just a Java-enabled web browser. To work on a task the host opens the corresponding URL.

Clients: Clients create tasks by producing the equivalent applet. We do not assume that clients have to run their own HTTP server. In fact, just like a host, the simplest form of a client is just a Java-enabled web browser that uploads an applet to a broker.

Broker: The broker is an HTTP server which performs two separate functions (which may be implemented by two separate servers). First it stores the applet and secondly it matches the client task with a host.

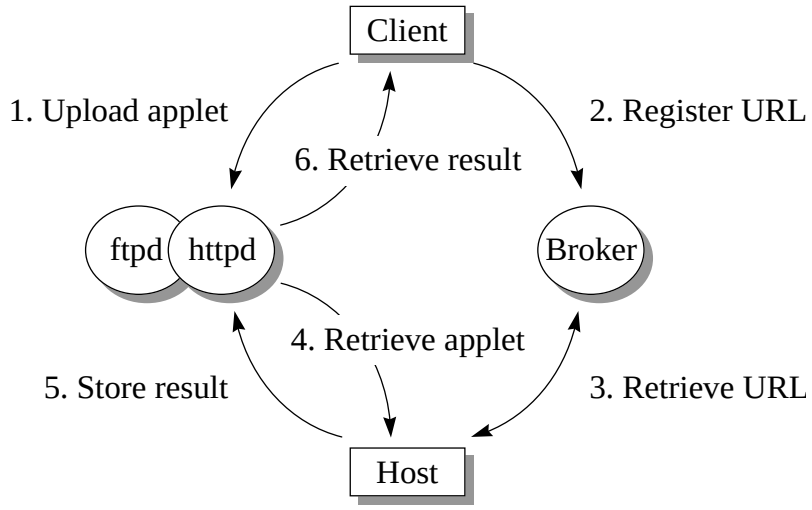

Figure 2: *Steps involved in the remote execution of an applet.*

Figure 2 shows the steps involved in the remote execution of an applet. These steps are:

1. The client uploads the applet and an embedding HTML page on an HTTP server. Clients that run their own HTTP server may skip the first step.
2. The client registers the corresponding URL with a broker.
3. The host or a daemon acting on behalf of it asks the broker for work and retrieves the URL.
4. The host downloads the applet from the HTTP server and executes it.
5. The host stores the result at the server site. If communication between hosts is required, messages are stored at and retrieved from the server site.
6. The client retrieves the result.

2.2 Design Implications

Our approach of using plain web browsers enables everyone to easily participate in Javelin. To ensure security, a Java-enabled web browser introduces certain limitations. For example, an applet cannot open a network connection to any computer other than the server from which it was downloaded. As a consequence, the server has to function as a gateway for any communication to the client or to other applets (e.g., in the case of a parallel program running on multiple hosts). Additionally, the Java API only provides an interface to the TCP and UDP protocols. Since applets are not allowed to listen for or accept network connections on any port of the local system, the broker cannot send datagrams to an applet. Although communication can still be established through a shared namespace by exchanging files, it is highly inefficient. The next

section discusses alternative communication channels. Furthermore, applets are not allowed to access the local file system. Thus data can only be stored persistently at the broker or the client, and every file access involves communication.

All communication must be routed through the server, which additionally may serve as a file system to large numbers of clients and hosts. This makes a single server a bottleneck. We propose to use a network of servers instead. Then clients and hosts as well as files are migrated between servers that are connected by high bandwidth communication channels.

A host can be migrated between brokers by making the corresponding daemon download a new daemon from the broker it is moving to, and then terminating itself. Similarly, registration information is migrated between brokers by asking other brokers for URLs. The physical migration of applets is supposed to reduce the number of requests at the original site, but must respect synchronization dependencies between applets in terms of sharing memory at some site all applets can connect to. Since communication stubs do not reduce the number of hits we propose to perform physical balancing of applets only at the time of submission by redirecting clients' requests to upload an applet. Especially, a client's request can be redirected multiple times. Additionally, we propose to distribute large numbers of applets that need to be synchronized to several brokers by providing a library to the application programmer implementing virtual shared memory.

Since an applet is not allowed to create instances of classes that are given by a stream of bytes[1] (that might have been received from the network) a daemon must ask the underlying web browser to open a new location[2] and thus initiate the downloading of the embedded applet with one step of indirection. Since, unfortunately, the corresponding Java API call is restricted to applets that are currently displayed within the Web browser, the daemon must live in a different web browser or a different frame than applets executed on behalf of clients.

Even though these considerations seem to complicate applications programming, the programmer using our architecture does not have to concern herself with any of these restrictions; they can be hidden in libraries as discussed in Section 3.

2.3 Broker/Server Services

In order to submit applets, clients need to upload them to a server. The clients clients are provided a home directory (a private namespace), and a mechanism to upload files on the HTTP server. Furthermore, the broker allows clients to register and unregister URLs, and allows hosts to retrieve URLs.

Although abstractions of storage and communication channels as well as synchronization constructs can be implemented by hosts and clients on top of a file system that can be accessed by both, dedicated *services* at the broker site can provide better performance as well as convenience.

Recently developed HTTP servers can be extended in two ways, either by using the Common Gateway Interface [Rob] and spawning a new process, or by installing servlets [Sun96c]. Servlets are small Java programs like Applets, but are executed at the server side and, like CGI scripts, allow for the dynamic creation of Web pages. They can either be embedded into Web pages or invoked like CGI scripts. Currently, Servlets are supported by JavaSoft's Jeeves[Sun96c] and W3C's Jigsaw[Con]. However, we feel that Servlets will be supported by most HTTP servers released in the near future.

We have found the servlet approach to be a very convenient way of extending the functionality of HTTP servers with application-specific services. Actually, in our prototype implementation, we even allow the application to upload its own new servlets. Unfortunately, ensuring security is not easy. Thus, we envision that in the final deployed system, only a fixed set of servlets will be provided. One of the most interesting open research questions is how much resources (CPU time, memory, disk space) an individual user should be able to get on such servers.

We propose to extend the HTTP server by services that allow an efficient implementation of parallel programming paradigms, efficient communication and convenience functions such as wrapping an applet around compiled Java classes or an HTML page around an applet.

[1] `Object java.lang.Class.newInstance()`

[2] `void java.applet.AppletContext.showDocument( URL url )`

2.4 Communication

Since fast communication is crucial for parallel programs, services allowing for efficient *message passing* should be provided. In general, messages between applets must be routed through the broker, because an applet cannot open a network connection to any site other than the one from which it was loaded. Additionally, the Java API only provides an interface to the TCP and UDP protocols. Since applets are not allowed to listen for or accept network connections on any port of the local system, the broker cannot send datagrams to an applet.

Basically, there are two possibilities to enable message passing. A routing service at the broker site can either link two *permanent* or two *temporary* socket connections possibly running a connection-less protocol such as HTTP (which is directly supported by the Java API) on top of TCP. These alternative approaches present a trade-off between performance and the maximum number of hosts that can be served by a single broker, since the number of socket connections that can be kept open at the same time is usually limited.

Obviously, if no routing service is in place, one must be requested before establishing a socket connection to the broker site. Figure 3 shows the steps involved in establishing a logical communication channel between two applets.

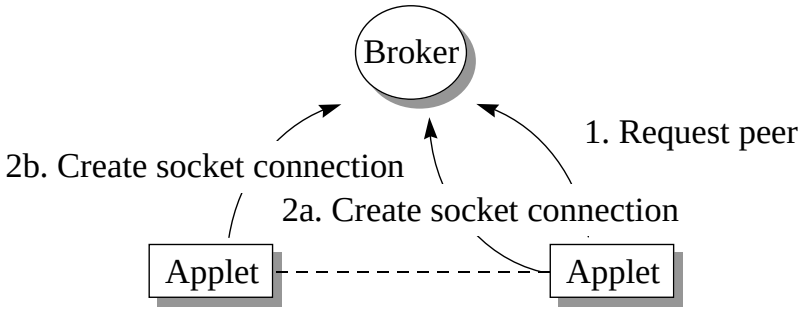

Figure 3: *Steps involved in establishing a communication channel between two applets.*

When HTTP is used, the receiver must continuously invoke its routing service to pick up messages, since applets cannot listen for HTTP requests from the router. Unfortunately, clients and hosts continuously connecting to the HTTP server increase contention as well as its work load, and thus affect the overall performance. This can be avoided by delaying responses to HTTP requests, and returning tokens sporadically to avoid time-outs at the requesting site.

Additionally, since every request increases the work load, the number of requests polling for some event must be minimized without introducing gaps to keep the overall overhead low. We propose the use of *adaptive polling*, i.e., a client or host polling the broker is returned a new polling interval that depends on the current work load, and all recently assigned intervals. As long as all clients and hosts obey these intervals, the overall overhead is reduced, although it may increase for individual applications.

2.5 Implementation

Our implementation is based on an HTTP server entirely written in Java that runs servlets compatible with the API used by JavaSoft's Jeeves[Sun96c]. The broker is implemented as a servlet and schedules applets using a FIFO queue. Furthermore, servlets provide additional services such as form-based file upload (as described in *RFC 1867*), semaphores, a Linda tuple space and communication channels (as described in the previous section). Since form-based file upload is supported by a broker servlet, clients can submit an applet by visiting a broker web page and filling out a form as shown in Figure 4. The form data is posted to the servlet which creates a new directory for the task, wraps the applet into an HTML page residing in the task directory, and enqueues the corresponding URL in the task queue.

Our daemon continuously polls a broker servlet until it receives an URL from the broker's queue. On receipt of the URL, the daemon asks the underlying Web browser to open it in one of two frames. Since

Figure 4: *Submitting an applet to Javelin*

the URL points to a web page embedding an applet, the web browser is indirectly asked to download and execute an applet. Unfortunately, in the current implementation of Netscape and Internet Explorer, applet security does not prevent hostile applets from killing the daemon.

3 The Parallel Programming Language Layer

The Javelin services presented in the previous sections provide a low-level layer of code distribution, communication and process execution. Though they provide a very flexible layer for communication and process creation, programming at this low-level layer can be a daunting exercise. For this reason, we describe simple programming models that enable programmers to express many parallel programming constructs in their client applet code. These programming models are realized by executing specialized servlets on the broker.

3.1 Language Support for Javelin

Language support for a distributed and parallel programming environment usually involves a suite of programming tools such as libraries, compilers, pre-processors, and stub generators. The Javelin programming environment provides a number of library support classes and tools to map different programming models onto the current applet infrastructure. To illustrate how these language constructs can be mapped, we consider the following programming models:

3.2 SPMD Programming Model

The Single Program Multiple Data programming model is perhaps the most widely supported form of parallel programming. Each node on a network executes a common copy of the same program code. Parallelism is obtained by providing different data inputs to each node, and combining their computations into a common result.

The SPMD model is easily mapped onto Javelin. Each node on the network executes a common applet, representing the single program code. Data can be read in at run-time through the broker, or can be coded into the applet and replicated among all the hosts, each node working on its portion of the data. Synchronization mechanisms conducted through the broker is provided to coordinate the tasks involved. We explore two commons forms of synchronization primitives:

3.2.1 Support for Send & Receive

Message passing is often required in a parallel program to pass information between hosts and direct the flow of computation. Toward this end, we illustrate how we map a Send and Receive onto Javelin.

The following steps are performed:

- Each pair of host passes *send* and *receive* requests to the broker.
- If a *send* arrives before the *receive* request, the broker stores the data and acknowledges the *sending* process. The *sending* process can continue with its computation. When the *receive* request arrives, the broker passes the data to the *receiving* process, so the *receiving* process can also continue processing.
- If a *receive* request arrives before a *send*, the broker must reply to the *receiving* process that the *send* data is not yet available. The *receiving* process, in turn, blocks and proceeds to poll the broker until the corresponding *send* request arrives with the data. If no *send* request does arrive, eventually, the *receive* request times out.

As indicated above, no polling is required in a send-and-receive request unless a *receive* arrives before the *send*. Such polling is an expensive process involving issuing repeated HTTP GET requests to contact the broker and check if the *sending* data has arrived.

In the mapping to Javelin, we can perform a modest optimization by keeping the HTTP GET connection active until the *send* data has arrived. The broker periodically sends NOOP data to the host in order to prevent the GET connection from timing out. The host ignores the NOOP message until the intended *send* data finally arrives. As a further optimization, we can employ socket connections between host and broker to avoid polling altogether, at the expense of maintaining these connections.

3.2.2 Support for Barriers

One other common form of synchronization that we must support in SPMD program is a barrier. Each processor executing a barrier must wait until all processors have reached that point in computation; only then can the processor be fully synchronized and continue with subsequent execution. As with send-and-receive requests, it is fairly simple to implement a barrier in Javelin by extending the broker with a barrier servlet.

- Each host executes a barrier operation, by sending a *barrier* request to the broker, indicating the number of hosts, n, participating in the barrier synchronization. The host then polls the broker waiting for the broker to acknowledge with a *release* reply.
- The first *barrier* request that arrives at the broker indicates to the broker the number of pending *barrier* requests, $n - 1$, to expect before releasing the barrier. When the broker receives subsequent *barrier* requests, it decrements the number of pending requests by 1.
- If the number of pending requests is 0, the broker acknowledges each host poll with a *release* reply; otherwise if replies with *no-release*.

Just as with send-and-receive requests, a barrier can time out if a polling host does not receive a *release* reply from the broker within an allotted amount of time. We can further perform the same optimizations of keeping the HTTP GET connection alive when a host polls a broker, or of employing socket connections.

3.3 Linda Tuple Space

To illustrate the flexibility of the Javelin design, we will also show how SPMD programming can be achieved by building upon Javelin applet layer a programming abstraction known as Linda tuple space.

The Linda programming model[WL88] was originally developed by Gelernter and Carriero of Yale University in 1988. This Linda model provides a construct known as tuple space that concurrent processes can access to insert, delete, and update data known as *tuples*. Atomic operations known as *Out*, *Rd*, and *In* provide synchronization for these accesses. The tuple space is viewed as a central pool of storage that hosts contact to share data. Programming is vastly simplified because each parallel program need only invoke these three primitives to pass and update information.

Mapping Linda tuple space onto the Javelin applet layer is fairly straightforward. The broker can serve as a tuple space manager by executing a dedicated servlet that implements the response handler for the *Out*, *Rd*, and *In* operations. When host machines running applet programs contact the broker, they now invoke a high-level Linda primitive to exchange data and information. We show how the two SPMD synchronization mechanisms can be implemented through the tuple space.

3.3.1 Linda Support for Send & Receive

Earlier, we saw how send-and-receive required a broker executing a dedicated servlet to handle synchronization requests. Though the broker must once again execute a servlet to provide tuple space management facilities, programming a send-and-receive can be vastly simplified using this tuple space abstraction: The host processor performing a *send* inserts a message into the tuple space by executing an *Out* command, while the *receiving* processor executes a blocking *In*. The *In* command deletes the message from tuple space and delivers it to the *receiving* processor.

3.3.2 Linda Support for Barriers

To provide a barrier we extend the functionality of the tuple space to include a general semaphore. When each processor reaches the barrier, it signals the tuple space manager to decrement the count of the semaphore by 1, then queries for the semaphore's current value. While in the barrier, the host processor performs a busy wait, polling the tuple space manager until the semaphore value reaches 0.

A Linda tuple space thus provides a simple programming model to implement message passing and shared memory abstractions. By extending the tuple space with additional synchronization primitives, such as a barrier, we can fully realize the benefits of SPMD programming in Javelin.

4 Experimental Results

In this section we present performance results for our initial prototype implementation. We conducted our experiments on a heterogeneous network of computers consisting of a 64 node Meiko CS-2, Sun Ultrasparcs, SGIs and PCs connected by SCI and 10 Mbit Ethernet networks.

4.1 Micro Benchmarks

From a client's point of view, the overhead of executing applets remotely, and the overhead of passing messages between remote applets are of great interest. We present experimental results for our implementation, and discuss their implications for using the proposed architecture.

Overhead	ms
Applet Start-up	2000
Send using permanent sockets	4
Send using HTTP	135

Table 1: Micro benchmark results between three workstations connected by Ethernet

4.1.1 Remote Execution of Applets

To be executed by a host, an applet must be sent to the broker, and downloaded by a host. Finally, its results must be sent back to the client via the broker.

The client posts its applet to the broker, where a servlet enqueues it. When a daemon polls the broker, the URL of a web page embedding the applet is sent to it. Then, the daemon opens the URL, and thus invokes the embedded applet. Our micro-benchmark applet immediately invokes a servlet at the broker to indicate that it is done. Finally, the client polls the corresponding servlet at the broker site for results.

On a standard 10 Mbit Ethernet network, downloading and invoking an applet takes about 2 seconds for an idle broker. This basic overhead increases as the work load at the broker increases. However, a substantial part of it is caused by the underlying browser starting a new Java thread. Notice that when running identical applets on different parameter sets this initial overhead can sometimes be reduced. Applets that have finished may query the client or broker for new parameters before quitting the host to avoid the overhead of sending the same applet to a different host. However, reloading it to the same host might take advantage of a cache provided by the used web browser.

This seems to imply that only coarse-grained applications can take advantage of the proposed architecture. On the other hand, the reader should not forget that most parallel machines have similarly high start-up costs when spawning parallel work. Having paid this start-up cost, communication can now be implemented more efficiently, as discussed next.

4.1.2 Message Passing

As discussed in Section 2.4 message passing can either be implemented by linking two permanent or two temporary socket connections (possibly running a connection-less protocol such as HTTP).

With an established permanent socket connection a single message can be sent within 4 milliseconds on a standard 10 Mbit Ethernet network. As long as applications are coarse-grained, the overhead of opening a socket connection can be ignored.

When HTTP is used, the receiver must either poll the broker site continuously or the response to its request for the message must be delayed until it arrives. Applying the latter technique, a single message takes about 135 milliseconds.

Since message passing on top of TCP is slow compared to dedicated parallel machines, and networks of heterogeneous machines produce unpredictable applet runtimes, only computation-intensive parallel applications take advantage of the proposed architecture.

Table 4.1.2 summarizes the micro benchmarks of our architecture.

4.2 The Mersenne Prime Application

As an interesting application for our Javelin infrastructure, we implement a parallel primality test which is used to search for prime numbers. As we shall see, this type of application is well suited for Javelin, since it is very coarse-grained, with a high computation-to-communication ratio when testing large Mersenne primes.

A Mersenne prime is a prime number of the form $2^p - 1$, where the exponent p itself is prime. Discovery of prime numbers was at one time relegated to the domain of mathematicians working in number theory, but

the advent of the computer provided a powerful tool to search and verify Mersenne primes. Beginning in the 1952, with the discovery of the 13th Mersenne prime by Raphael M. Robinson[Rob54], all subsequent Mersenne primes have been found by computer. To date only 35 Mersenne primes have been discovered. With larger and larger prime exponents, the search for Mersenee primes becomes progressively more difficult.

In our current implementation, we developed a Java application to test for Mersenne primality, given a range of prime numbers. Our application works as follows:

1. First, the client submits to the broker an applet containing the Mersenne primality test and a servlet that coordinates the range of numbers to be checked.

2. The broker running the servlet waits for hosts to participate in the Mersenne prime test. It maintains a FIFO of primes to be checked, parceling them out among the hosts that join in the computation.

3. When a user registers with a broker, the host daemon downloads the Mersenne Applet to the host site and executes the applet. The applet then contacts the broker to obtain a prime exponent p to check if the resulting Mersenne $2^p - 1$ is prime.

4. After the host applet has completed its computation, it reports the result back to the broker. The broker marks whether the prime exponent p the host has tested corresponds to a Mersenne prime. The host then asks for the next prime to test, and the broker dispenses subsequent primes until the FIFO is empty.

4.2.1 Measurements

For our measurements, we chose to test the Mersenne primality for all the prime exponents less than 3000. We conducted the experiments on a heterogeneous computing environment consisting of various architectures: Sun Ultrasparc (both single and multiprocessor), Pentium PC, PowerMac, SGI MIPS (both single and multiprocessor) and a 64-node Meiko CS-2. We used the Netscape Navigator browser on all platforms except the Pentium PCs (running Windows 95) where we used Microsoft Internet Explorer (the explorer has a just-in-time compiler, and we performed measurements for both compiled and interpreted versions of our test program). We started by measuring the total execution time on each architecture separately. Figure 5, shows the total time needed to test all the prime numbers less than 3000 on a single machine. The tests were performed using the Javelin infrastructure, therefore the reported times include all overheads. As we can see from Figure 5 newly released compilers for Java (such as the ones for PCs) have a dramatic impact on the execution time (for example a 7.4 times improvement in the case of the 166 MHz Pentium).

The second set of measurements were performed on clusters of identical machines. Figure 6 presents the speedup curves for test runs on a 64-node Meiko CS-2[3] (left) and for a cluster of 8 Sun Ultrasparcs (right). In both cases, the speedup was close to linear as long as the ratio of job size to number of processors was big enough, to achieve a good distribution of tasks to processors. For our tests, we chose a worst-case scenario, when the biggest tasks (large amount of computation) were enqueued at the end of the task queue, which strongly affected the speedup because full parallelism was not exploited for the last tasks. This shows that clever task distribution is an important issue that has to be taken into consideration.

Finally, we tested our prototype on a heterogeneous environment consisting of 2 PCs running Windows 95, 8 Ultrasparcs and 32 Meiko nodes running Solaris. We chose an uneven distribution of platforms to reflect a possible real situation of computing resources registered at a broker at some moment. The broker's job would be to distribute the tasks such that it makes a close to optimal utilization of its resources. Figure 7 shows total execution times for different combinations of the previously mentioned architectures. Note that the two PCs using a just-in-time Java compiler perform just as fast as 32 Sparc 10 running the Java interpreter! The combined resources of all machines showed an improvement, but not substantial due to the big difference in computing speed of the compiled versus interpreted versions. This shows that tasks for

[3]The nodes on a CS-2 are Sparc 10 processors running an enhanced version of Solaris.

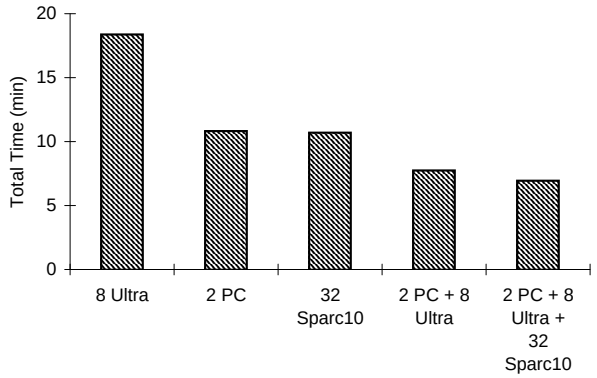

Figure 7: *Primality test on a heterogeneous platform consisting of Pentiums, Sparc 10 and Ultrasparcs.*

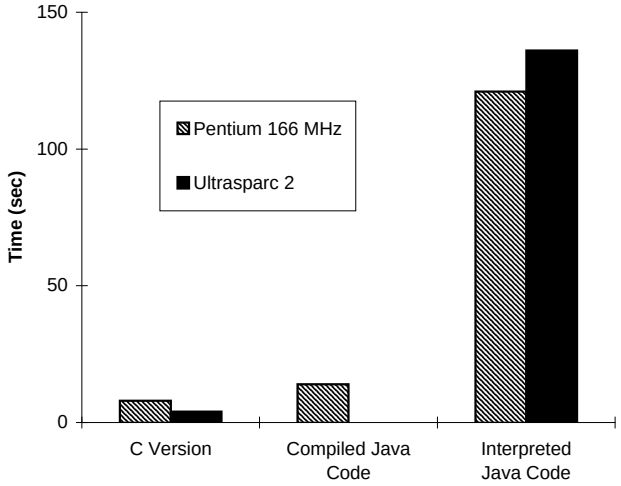

Figure 8: *Comparison of C and Java versions of the Mersenne primality test program for one large prime:* $2^{4253} - 1$.

5 Related Work

There is a rapidly expanding body of work based on the vision of seamlessly integrating networked computers into a global computing resource. This vision is as old as the Internet. More recent network computing approaches, include *CONDOR* [LLM88], *Linda* [WL88], *PVM* [Sun90], *Piranha* [GK92], *MPI* [MPI94], *Legion* [GWF+94], and *Network of Workstations (NOW)* [ACPtNt94]. Legion is probably the most powerful of these models and provides both greater throughput and improved response time via parallel execution. With the exception of Legion, all the systems require the user to have login access to all machines used in the computation. All of systems require the maintenance of binaries for all architectures used in the computation.

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, without violating any local security policies. Because of its 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 flexibility of Java has also been observed by several other researchers. A new, Java-based, generation of projects is aimed at establishing a software infrastructure on which a global computing vision can be

implemented. These projects include *ATLAS* [BBB96], *Charlotte* [BKKW96], and *ParaWeb* [BSST96]. All three projects are designed explicitly to run parallel applications and provide a specific programming model. Other recent systems such as *JPVM* [Fer] and *Java-MPI* [Tay] use Java to overcome heterogeneity, but are not intended to execute on anonymous machines. 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].

ATLAS provides a global computing model, based on Java and on the Cilk programming model [Blu95], that is best suited 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 supports distributed shared memory, and uses a fork-join model for parallel programming. A distinctive feature of this project is its eager scheduling of tasks, where a task may be submitted to several servers, providing fault-tolerance and ensuring 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 to allow transparent instantiations of threads on remote machines.

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 already have 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. Languages worth considering, besides Java, include the "E" programming language [Com96] which is very similar, but offers a more flexible security framework than Java, and the Limbo programming language in the Inferno operating system [Inc], which allows, among other features, for easier and more efficient just-in-time compilation.

The secure execution of arbitrary binaries recently has been 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.

Another important goal of our project is to allow a natural integration of our infrastructure with other currently existing models for distributed systems. There has been substantial work in the last decade geared towards heterogeneity and open systems. However, promising standards for interoperability and portability between different combinations of software and hardware components emerged only a couple of years ago. Java Beans [Sun96d] is the latest in the world of Web-based distributed computation, but we also have widely used technologies such as CORBA [Gro95] or DCOM [Cor96]. We expect to see a shift in Web technology towards systems based on distributed objects and IIOP.

6 Conclusions

In this paper, we have designed and implemented *Javelin*, a prototype infrastructure for Internet-based parallel computing using Java. Javelin allows machines connected to the Internet to make a portion of their idle resources available to remote clients and at other times utilize resources from other machines when more computational power is needed.

We believe that an infrastructure for transforming the Web into an immense parallel computing resource, to be successful, must be flexible and, first and foremost, easy to install and use. By requiring clients

and hosts to have access to only a Java-enabled Web browser, Javelin achieves ease of participation: No OS or compiler modifications are needed and system administrative costs are zero. Furthermore, the whole system implementation is based on an HTTP server entirely written in Java and current Web technology. We discussed possible extensions of HTTP servers which allow efficient implementations of parallel programming paradigms. We also analyzed the limitations imposed by the security restrictions of most web browsers and showed how they influenced our design and implementation.

To extend the functionality of the Javelin system, we provide a parallel programming language layer which offers support for the SPMD programming model and Linda Tuple Space from within an applet.

Finally, we presented experimental results consisting of micro-benchmarks measurements and Mersenne primality tests, a coarse-grain compute-intensive application, running on a heterogeneous environment consisting of a 64-node Meiko CS-2, Sun Ultrasparcs, SGIs and PCs connected by an SCI and 10 Mbit Ethernet networks. Noticeably, the just-in-time compilers make the differences between Java and C versions to become less of a performance issue. We expect that future versions of Javelin will see a performance boost due to optimized compilers.

Overall, we believe that Javelin can be used successfully for Internet-based parallel computation, and future versions will solve the remaining challenges, such as result verification, fault tolerance, and client privacy.

Acknowledgements

Bernd Christiansen is supported by an HSPIII scholarship for Ph.D. research studies of the German Academic Exchange Service (Deutscher Akademischer Austauschdienst).

References

[ACPtNt94] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations), December 1994.

[AISS97] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. SuperWeb: Towards a Web-Based Global Computing Infrastructure. To appear in International Parallel Processing Symposium, May 1997.

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

[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.

[Blu95] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[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.

[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.

[Con] World Wide Wed Consortium. Jigsaw HTTP Server. `http://www.w3.org/pub/WWW/Jigsaw/`.

[Cor96] Microsoft Corporation. DCOM – The Distributed Component Object Model, 1996.

[DFW96] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *In Proceedings of the 1996 IEEE Sumposium on Security and Privacy*, Oakland, CA, May 1996.

[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.

[Gro95] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1995. 2.0 ed.

[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.

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

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

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

[Rob] D. Robinson. *The WWW Common Gateway Interface*. Internet Draft. Version 1.1.

[Rob54] Raphael M. Robinson. Mersenne and Fermat numbers. In *In Proceedings of the American Mathematical Society*, volume 5, 1954.

[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.

[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.

[Sun96c] Sun Microsystems, Inc. *Java Servlet Application Programming Interface*, November 1996. Draft version 2.

[Sun96d] Sun Microsystems, Inc. *JavaBeans 1.0 API Specification*, December 1996. Revision A.

[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.