Javelin: Internet-Based Parallel Computing Using Java

Bernd O. Christiansen, Peter Cappello, 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
{*bernd, cappello, mionescu, neary, schauser, danielw*}*@cs.ucsb.edu*

Abstract

Java offers the basic infrastructure needed to integrate computers connected to the Internet into a seamless distributed computational resource: an infrastructure for running coarse-grained parallel applications on numerous, anonymous machines. First, we sketch such a resource's essential technical properties. Then, we present *Javelin*, an infrastructure for global computing. The system is based on Internet software that is interoperable, increasingly secure, and ubiquitous: Java-enabled Web technology. 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. Javelin's architecture and implementation require participants to have access to only a Java-enabled Web browser. The security constraints implied by this, the resulting architecture, and the current implementation are presented. The Javelin architecture is intended to be a substrate on which various programming models may be implemented. Different computation servers, each corresponding to a different programming model, are supported by Javelin's architecture. Several are presented briefly. Experimental results are given in the form of micro-benchmarks, a Mersenne Prime application, and a ray-tracing application that run on a heterogeneous network of several parallel machines, workstations, and PCs.

Keywords: Distributed Computing, High Performance Computing, Java, Internet, World Wide Web.

1 Introduction

Our vision is to harness the combined resources of millions of computers connected to the Internet, forming a powerful heterogeneous computing environment for running coarse-grain parallel applications. Cluster computing and network of workstations in a LAN setting already are being used extensively to run compute-intensive applications. On a more global scale, Internet-based cluster computing involving thousands of computers, recently has demonstrated its usefulness for several important and news-breaking applications. In November 1996, a PC in France running for 88 hours found the 35th known Mersenne prime, the largest known prime discovered so far [Gil]. This computer was part of a world-wide distributed computing effort for finding large primes, involving a group of more than 700 workstations and PCs. Then, in January 1997, RSA announced a code breaking challenge. Within only 3.5 hours, a graduate student at Berkeley had cracked the 40-bit RSA code running on a 128-processor UltraSparc NOW cluster [Las97a]. Two weeks later, the 48-bit RSA code was cracked using 3,500 workstations spread across Europe [Las97b]. On June 17th, 1997, the 56-bit DES [RDS97] was cracked, using approximately 78,000 computers, as many as 14,000 per day. The press release [DES97] states:

> *This project demonstrates the kind of supercomputing power that can be harnessed on the Internet using nothing but "spare" CPU time. "Imagine what might be possible using millions of computers connected to the Internet!" Aside from cryptography and other obvious mathematical uses, supercomputers are used in many fields of science. "Perhaps a cure for cancer is lurking on the Internet?", said Verser, "Or perhaps the Internet will become Everyman's supercomputer."*

The above examples illustrate that large numbers of users are willing to participate in global computing efforts and that there are applications that can be run very efficiently in wide area networks. While dramatic and encouraging, they also show that of the tens of *millions* of computers connected to the Internet, only a small percentage participate. There are several reasons for this. Security is of foremost concern. Participation requires either downloading and running an executable, or downloading unknown source code and then compiling it. The code may contain bugs or viruses that can destroy or spy on local data. Secondly, participating in such a computational challenge is not easy. Each of the examples mentioned above, runs on a different set of architectures, and requires different mechanisms to download and participate. Finally, participants currently do so voluntarily, donating idle computing time that otherwise would have been wasted. There is no incentive provided beyond either curiosity, helping a worthy cause, or fame (if one is lucky enough to find the key). We believe that lack of security, lack of a unified framework, and lack of incentive are key reasons explaining the small participation.

The advent of a safe and portable language system such as Java has provided tools and mechanisms to address these issues and interconnect computer systems around the globe in cooperative work efforts. We contribute to this goal by developing a flexible, scalable, easily-installed infrastructure for running coarse-grained distributed applications on any Internet machine.

1.1 Concept and Architecture

We have implemented 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 with the broker their tasks to be run; hosts register with the broker their intention to run tasks. The broker assigns tasks to hosts that, then, run the tasks and 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.

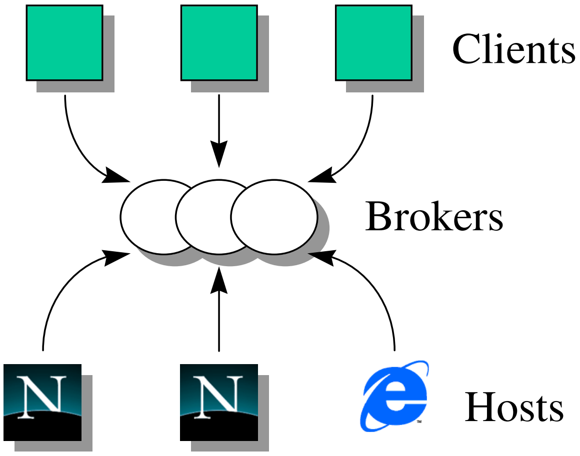


Figure 1: *The Javelin Architecture.*

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 interoperable language Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available to host part of distributed 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 to use for clients. When a client wants 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 do not even assume that a client has an HTTP server. All it needs is a web browser for uploading applets and spawning threads.

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 eliminate most of the interpretation overhead, allowing Java programs to execute both safely and efficiently. 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 tens of 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:

Ease of use: Ease of use is a key property of Javelin, since it relies on the participation of thousands of users. Our approach is *web-centric*, where a user needs only a Java-enabled web browser to participate.

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.

Scheduling and Load Balancing: Since brokers cannot know the semantics of particular applications, the client ultimately is responsible for *scheduling* tasks. *Load balancing* in its simplest form is achieved through the broker network either by contacting other brokers for help, if local hosts are busy, or by "work-stealing" from other brokers, if local resources are idle. Beyond this, we do not enforce any specific algorithm.

Fault Tolerance: It is desirable for the network of brokers to provide some form of *fault tolerance*. However, in this version we do not concentrate on this issue. The client can provide some degree of fault tolerance itself, e.g., by duplicating subtasks and rescheduling them if results do not arrive.

Economic Incentive: Participation in such a global computing infrastructure must 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 deployment.

1.4 Contributions and Outline

We present a system that makes it easy for computers all over the world to cooperate in distributed computations. This paper is organized as follows. Section 2 identifies some architectural implications of using a heterogeneous, secure, Internet execution environment, in the form of Java applets running under standard Web browsers. Section 3 presents a set of abstract behaviors that must be implemented to realize the architecture, and discusses some implementation tradeoffs. We also present the design and implementation of 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. Section 4 presents measurements of Javelin's current performance on two popular distributed applications: Mersenne Primes and ray-tracing. These measurements shed light on those aspects of the system that would most benefit from greater efficiency. Section 5 discusses related work and Section 6 concludes.

2 Design

2.1 Basic Design Issues

One of the most important goals of our design is to minimize the administrative overhead associated with installing a broker, registering host resources and client tasks. Since Web browsers are available on almost every platform and architecture, and common to almost every user, our framework facilitates these operations. 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 encode tasks as applets. We do not assume that clients have to run their own HTTP server. In fact, similar to hosts, the simplest form of a client is just a Java-enabled web browser that uploads an HTML page containing an applet to a server, and submits the URL of this page to a broker.

Broker: The broker manages a pool of available hosts and a ready queue of tasks, each represented by a URL. It schedules tasks to be run on hosts, possibly taking into account meta-level information such as machine characteristics or the availability of a JIT compiler.

Server: The server is an HTTP server that performs two separate functions. First, it stores clients' applets and serves them to hosts; secondly, it provides point-to-point communication among hosts by serving as a smart router.

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 running their own HTTP server may skip the first step.

2. The client registers the corresponding URL with a broker.

3. The host registers with the broker its intention to execute tasks, and retrieves the URL of a task.

4. The host downloads the HTML page from the HTTP server, and executes its embedded applet.

5. The host stores the result at the server site. If communication between cooperating hosts is required, messages are stored at, and retrieved from, the server site.

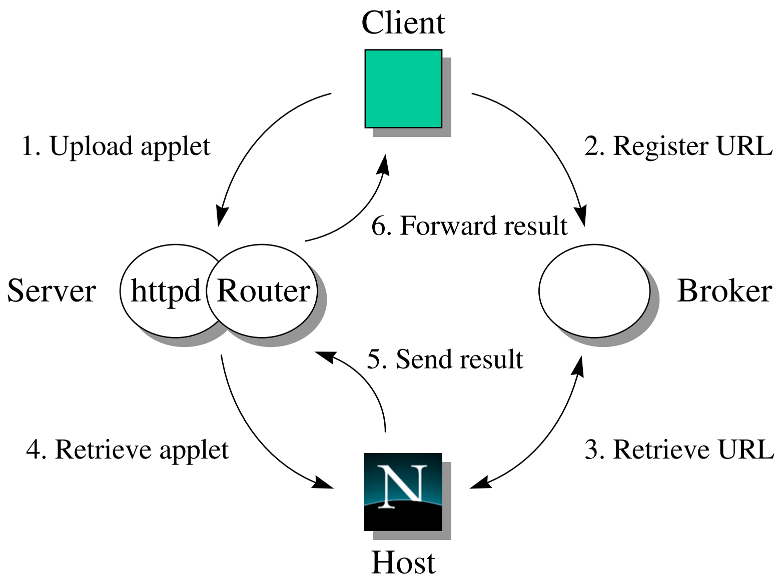


Figure 2: *Steps involved in the remote execution of an applet.*

6. The client retrieves the result.

Our model supports "subcontracting" where hosts simultaneously act as clients: a host can register part of its work with a broker to be done by other hosts.

2.2 Design Implications

Using plain web browsers enables everyone to easily participate in Javelin. However, for purposes of security, a Java-enabled web browser disallows applets from: opening a network connection to any computer other than the server from which it was downloaded; listening for network connections; and accessing the local file system. These restrictions imply that: communication among hosted applets must be routed through their associated servers, and that file access implies network access. Furthermore, 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). Therefore, a daemon must ask the underlying web browser to open a new location[2], and so initiate the downloading of the embedded applet with one step of indirection. Unfortunately, the corresponding Java API call is restricted to applets that are currently displayed within the Web browser. Thus, the daemon must live in a different browser window 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 are hidden in libraries, as discussed in Section 3.5.

In the future, we may use signed applets to alleviate some of these restrictions without compromising host security. The appletviewer already allows applets in Java archives that are signed by a trusted entity to run with the same full rights as local applications. That is, such applets are not subject to the sandbox restrictions of the Java security model. A signed daemon applet may implement a less restrictive security manager, for example allowing an applet to communicate with its peer applets using native Java network classes (instead of an abstraction layer hiding the routing through servers). This reduces the server load substantially and opens our infrastructure to new applications, such as a distributed web crawler. It is an interesting research challenge to design a security model for Javelin that is powerful enough to support a wide range of distributed applications without compromising host security.

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

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

3 Javelin Prototype

This section presents our prototype system. It describes its core components, runtime library, and API.

3.1 Components

The Javelin system requires at least a regular HTTP server. File upload, routing, and the broker itself are implemented as Java applications that are accessed via RMI as well as servlets that are compatible with the API defined by JavaSoft [Sun96c]. Servlets allow for the smooth integration of Javelin into server technology that is becoming more and more common in enterprises. Optional servlets provide additional services such as form-based file upload (as defined in *RFC 1867*), semaphores, and a Linda tuple space. In our prototype implementation, we even allow applications to upload their 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.

When the host registers with the broker, a daemon applet is downloaded that connects to the broker and blocks until the broker schedules an applet to be run at this particular host. On receipt of a URL, the daemon asks the underlying Web browser to open it in a second frame. Since 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. However, these restrictions can be circumvented by running the daemon as an application.

3.2 Load-Balancing Runtime Library

In the current release, a client explicitly stores its applet at one or more servers and registers it with at least one broker as shown in Figure 2. Load-balancing can be achieved by propagating applets between servers, and their URLs between hosts. Currently, both applets and URLs are propagated only if some broker has insufficient applets to keep its hosts busy.

Initially, the client cannot know how many — if any — hosts will be scheduled to run its applets. Thus, it owns the whole computation. If a new host becomes available or a host finishes and becomes idle, work must be requested from another host (where, here, the client is a special host). As illustrated in Figure 3, the Javelin runtime system implements a method to request work from other hosts. This involves locating other hosts that run the same applet, selecting a prospective host with work, and interrupting its ongoing computation. If no work can be extracted from this host another host must be selected or the task request must be terminated. The I_WANT_WORK and HERE_IS_WORK messages are asynchronous.

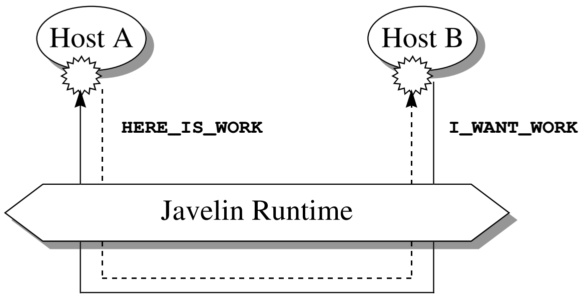


Figure 3: *The Javelin Runtime Library implementing work stealing.*

Currently, a prospective host is selected randomly from a local table that is updated continuously (similar to Cilk [BL94]). Figure 4 illustrates the implemented scheme. Each processor keeps a local table of hosts which it believes have work. Initially, this table contains the client only. In Figure 4(a), host A issues a request for work to the client, because it is not aware of other hosts. The client cannot satisfy the request, but forwards it to host B. It also adds host A to its table of hosts that might have work, although there is no guarantee that host A will actually get work. We have experienced that this optimistic update usually results in better performance compared to more pessimistic schemes. Host B splits its work, sends a portion to host A, and updates its local table, because it is likely that A will have work in the near future. Finally, host A adds host B to its local table, because B might have more work. The next time host A wants more work it randomly sends a request to either the client or host B. The random forwarding of requests terminates unsuccessfully after a given number of hops. There also is a maximum number of retries. The time between two retries is increased using exponential back-off.

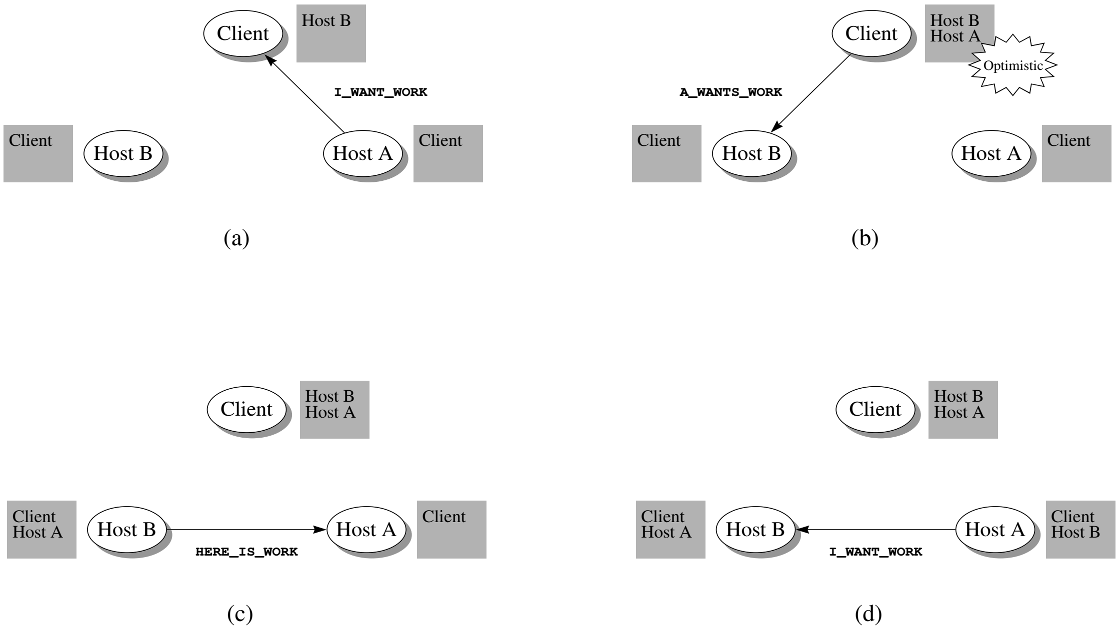


Figure 4: *Random task request (work stealing) with optimistic propagation. Local tables of hosts that are assumed to have work are shown in gray. The client is idle, but host B has more work than it can handle.*

The Javelin runtime system is implemented as a class of which each host must create an instance. The host programmer cannot subclass it, since a host running within a browser must already subclass `java.applet.Applet` and the Java language does not support multiple inheritance.

3.3 High-Level API

Using the low-level API presented in the previous section, the Javelin runtime system calls a handler implemented by the host when a request for work arrives. The handler thread must synchronize with the ongoing computation and, if possible, extract work. While flexible, this mechanism requires the programmer to deal with complex synchronization problems.

To make programming the Javelin system easier, we have implemented a library class that realizes a distributed task queue. It automatically manages load balancing. The programmer simply pushes tasks that may be executed remotely on its local stack and pops the topmost task from the stack when it finishes its current task. The stack itself is implemented as a deque (double-ended queue) as shown in Figure 5. A pop on an empty deque triggers a task request being sent to another host. When a task request arrives, a task is removed from the tail of the deque, if it is not empty; otherwise the request is forwarded. An example of an application that is based on our library is the raytracing program presented in Section 4.2. It is transparent to the application whether a task is retrieved from the local deque or a remote host. The programmer is freed from any synchronization issues.

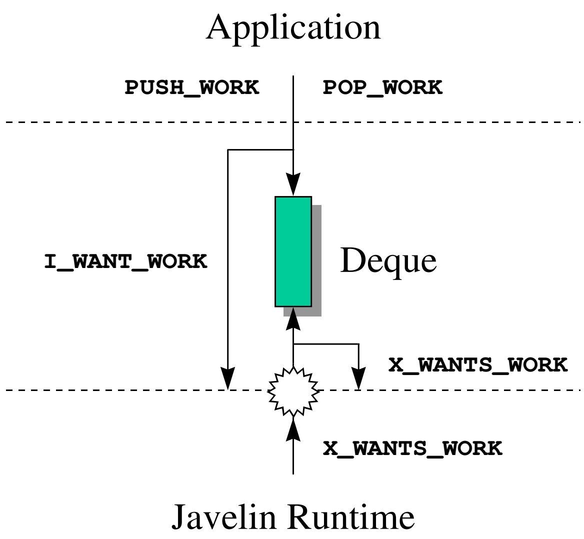


Figure 5: *Abstraction of a local task stack used to implement a distributed task queue.*

3.4 Communication

As discussed in Section 2.2, any communication between applets must be routed through their associated servers. Javelin defines a communication layer that provides the abstraction of a direct connection between applets. It implements the classes `JDatagramPacket`, `JDatagramSocket`, and `JnetAddress` that mimic the native Java UDP classes in `java.net`, but may be implemented on top of an arbitrary communication mechanism such as TCP sockets, UDP sockets, RMI, NexusJava [FT96], or CORBA.

A `JnetAddress` is built from the IP addresses of the host and the server it is attached to. Figure 6 illustrates how a message is routed from the sending to the receiving applet. When sending a packet to a `JnetAddress` it is *always* first send to the server the sender is associated with. Then, the server extracts the IP address of the destination host's server from its `JnetAddress`, and forwards the packet to that server. Finally, the packet is sent from the destination server to the destination host.

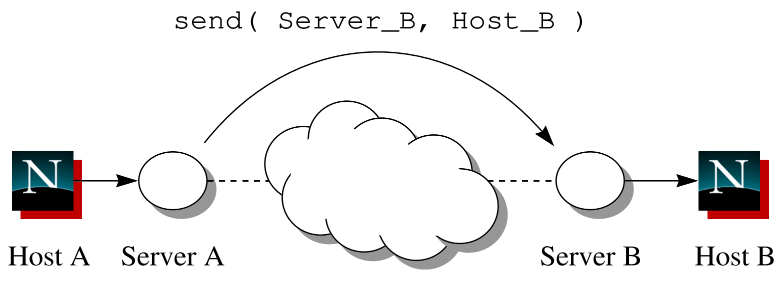


Figure 6: *Communication between two peer applets.*

Since the number of socket connections that can be kept open at the same time is usually limited, not all servers can be connected by *permanent* socket connections at any time. However, any implementation of `JDatagramSocket` on top of a communication mechanism that requires a permanent socket connection must be able to time multiplex between different connections.

The current release provides implementations of `JDatagramSocket` on top of TCP and UDP sockets.

3.5 Language Support for Javelin

The Javelin system presented in the previous sections provides a low-level layer of code distribution, communication, and process execution. Though it provides a very flexible layer for communication and process creation, programming at this low-level layer can be difficult. For this reason, we propose to extend the HTTP server with services that enable the efficient implementation of parallel programming paradigms and constructs.

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 implementation of a Linda tuple space as an example.

The Linda programming model [WL88] was originally developed by Gelernter and Carriero of Yale University. 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 the Linda tuple space onto the Javelin applet layer is fairly straightforward. The server can act 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 server, they then invoke a high-level Linda primitive to exchange data and information.

4 Experimental Results

In this section, we present performance numbers for our initial prototype implementation. We conducted our experiments in a heterogeneous environment: Pentium PCs, Sparc-5s, Sparc-10s, and single and dual processor UltraSparcs connected by 10 and 100 Mbit Ethernet. Although all machines run under Solaris, heterogeneity was introduced by using Java VMs with and without JIT. Furthermore, all experiments were conducted under these machines' typical workloads.

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 infrastructure. Table 1 summarizes the micro benchmarks of our infrastructure in a LAN.

4.1.1 Remote Execution of Applets

To be executed by a host, an applet is uploaded to a server, and its URL is registered with a broker. Then, the broker schedules it by passing its URL to a host. The host opens this URL and thereby downloads the embedded applet, and pipes it to a Java VM. Finally, any results are sent back to the client via the server. We implemented a micro-benchmark applet that just returns a single byte to the client. This measures the overhead of downloading the applet and starting it.

On a standard 10 Mbit Ethernet network the remote execution of our micro-benchmark applet within Netscape 3.01 takes about two seconds. However, a substantial part of this overhead is caused by the underlying browser starting a new Java thread.

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.

Overhead	ms
Applet Start-up Time	2000
Send/Receive	6-7

Table 1: *Micro benchmarks for a 10 Mbit Ethernet LAN using TCP/IP sockets.*

4.1.2 Message Passing

As discussed in Section 2.2, messages between two applets must be routed through the servers they were downloaded from. Furthermore, the Java VM requires all communication to go through the Java network layer (implemented by the `java.net` package) and the complete TCP or UDP stack of the underlying OS. This causes a substantial software overhead compared to communication libraries of parallel machines.

Using Javelin with TCP/IP sockets, a single message can be sent from one applet to another within 6-7 milliseconds on a standard 10 Mbit Ethernet LAN. As long as applications are coarse-grained, the overhead of opening a socket connection can be ignored.

Since message passing using Java TCP and UDP sockets is slow compared to dedicated parallel machines, and large networks of heterogeneous machines cause unpredictable communication delays, only computation-intensive parallel applications benefit from the proposed infrastructure.

4.2 Raytracing

We have ported a sequential raytracer written in Java to Javelin as an example of a real-world application that benefits from additional processors even if communication is relatively slow. The raytracer was written by Frederico Inacio de Moraes, Universidade Estadual de Campinas, [dM] and was first parallelized by Laurence Vanhelsuwe [Van97].

Raytracing is a method that creates photo-realistic synthetic images from a mathematical description of the scene to be generated. A scene is defined by the camera, light sources, and objects built from textured primitives such as cubes, spheres, and planes. Raytracing software mathematically traces the light rays as they bounce around the scene, reflecting and refracting until they end up in the lens of the imaginary camera. This can involve millions of floating-point operations and take anything from a few minutes to many days. Since each imaginary beam of light going through a pixel of the image is traced independently, raytracing can be parallelized easily by assigning each processor a subregion of the image.

4.2.1 Parallel Raytracing

We have implemented our parallel raytracer using the distributed "queue" library discussed in Section 3.3. Figure 7 shows the main loop of the raytracer. Initially, the client pushes a rectangle of the size of the image onto its local stack. In their main loop, hosts (where the client is a special host) pop a rectangle from the stack and either split it or compute this region of the image. A pop on an empty local stack triggers a task request being issued by the library and routed through the network by the Javelin runtime system. The image is split if the area of the rectangle is larger than a given size, the *granularity*. If it is split, both resulting rectangles are pushed back onto the stack. In the next iteration, the rectangle pushed second may be split further.

The library gives remote hosts tasks from the bottom of the local double-ended queue. Thus, most of the time large rectangles, usually implying a compute-intensive task, are given to requesting hosts. The basic idea of this scheme is to reduce communication by returning a remote host a long lasting computation so that it will be busy for a while and not send another request over the network. This scheme was first used for distributed scheduling by Cilk. Figure 8 illustrates how this scheme applies to raytracing an image on three processors.

```
JStack stack = new JStack();
JRectangle rect = null;
while ( ( rect = stack.pop() ) != null )
{
  if ( rect.tooBig( JAVELIN_DEFAULT_GRANULARITY ) )
  {
    rect.split_and_push( stack );
  }
  else
  {
    raytrace( rect );
  }
}
```

Figure 7: *The main loop of our raytracer.*

Initially, processor 1 is assigned the whole image and splits it according to the algorithm in Figure 7. When a second host becomes available it downloads our raytracing applet from a server and requests a rectangle of the image that has not been raytraced yet from processor 1. This request is satisfied from the bottom of processor 1's stack. Thus, processor 2 is assigned half of the image. When a third processor becomes available, it randomly requests work from either processor 1 or 2. In 8(c), processor 3 is assigned half of the work of processor 1, i.e., a quarter of the image. In 8(e), processor 1 requests work back from processor 2.

4.2.2 Performance

To evaluate the performance and dynamic behavior of our infrastructure we have raytraced images of the size 1024x1024 pixels for the two scenes shown in Figure 9, and a randomly generated scene. `simple` is built from a plane and two spheres; `cone435` is built from a plane and 435 spheres arranged as a cone[3]; `random` is built from 12,288 randomly chosen primitives.

Figure 10(a) shows the speedup curve for our parallel raytracer running on `simple` and `cone435` in a cluster of two Sparc-5s, five UltraSparcs, and one dual-processor UltraSparc connected by 10 and 100 Mbit Ethernet. Figure 10(b) gives the speedup curve for raytracing `random` on the 64 Sparc-10 nodes of our Meiko CS-2. We used a simple Web "browser" written in Java that executes applets and applications by spawning Sun's JVM with JIT. The graphs illustrate that the speedup that can be achieved depends on the computation to communication ratio. The more complex the scene, the longer it takes to compute the color of a pixel, and the less frequently communication is required. In general, coarse-grained applications with a high computation to communication ratio are well suited to Javelin.

Figure 11 illustrates the dynamic assignment of work to hosts through the Javelin runtime system. The performance of the processors differs within a factor of ten. The figure shows a high correlation between the relative performance of a host and the fraction of work it does in a parallel run. We have observed that subcontracting (called work-stealing in Cilk parlance) with optimistic propagation and exponential back-off leads to task distribution that is close to optimal.

4.3 The Mersenne Prime Application

As our second application for our Javelin infrastructure we implemented a parallel primality test which is used to search for Mersenne prime numbers. As we shall see, this type of application is well suited to

[3] In favor of a better illustration Figure 9 shows only 89 spheres arranged as a cone.

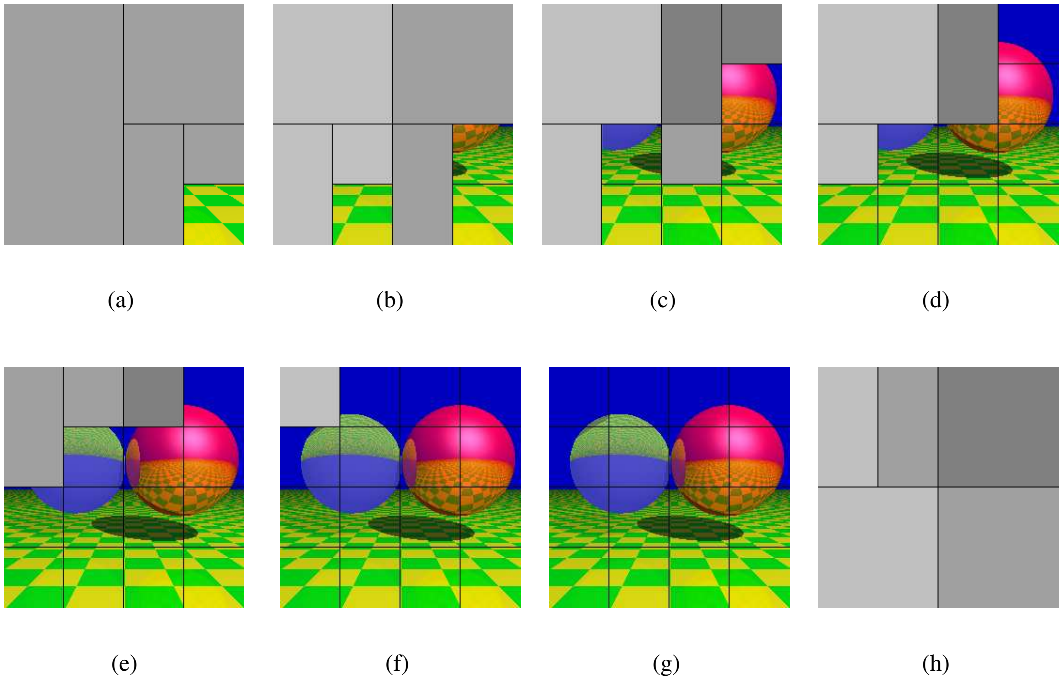


Figure 8: *Snapshots of a simple scene being raytraced by three processors. The background coloring illustrates which processor currently owns which part of the computation. The black lines show how the image is split into smaller rectangles over time. The final picture shows which parts are actually raytraced by which processor.*

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. These are traditionally the largest known primes. 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. The current record holder is $2^{1398269} - 1$ and was discovered in November 1996 by a coordinated effort of over 700 PCs and workstations worldwide. With larger and larger prime exponents, the search for Mersenne primes becomes progressively more difficult[4].

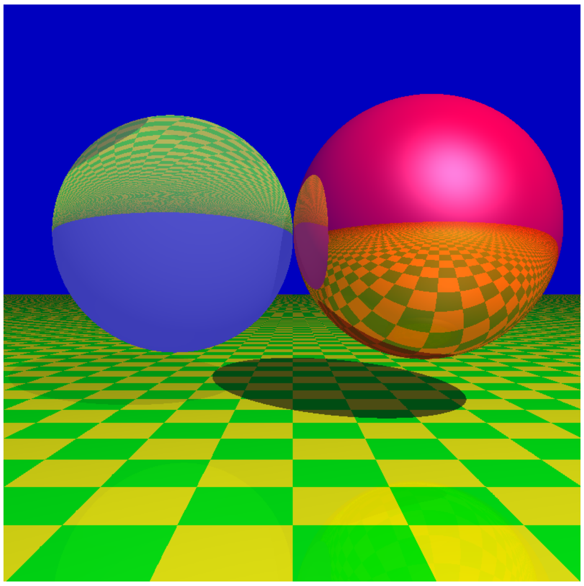
In our current implementation, we developed a Java application to test for Mersenne primality, given a range of prime numbers. Our application works as follows:

1. First, the client submits an applet to the broker containing the Mersenne primality test.

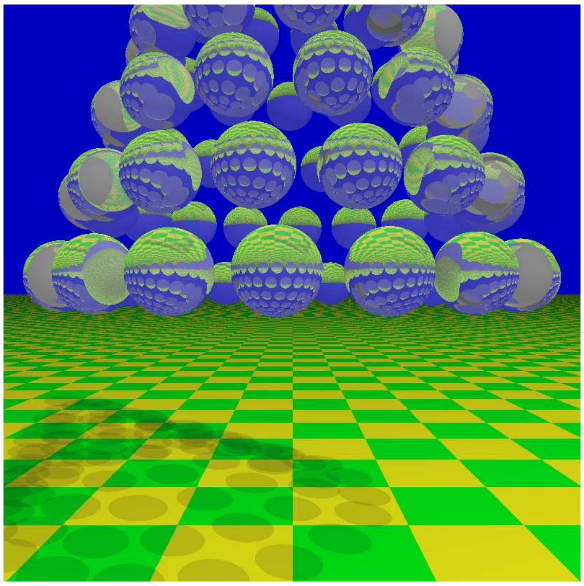
2. The client then waits for hosts to participate in the Mersenne prime test. It maintains a FIFO of primes to be checked, dealing them out one at a time among the hosts that join in the computation.

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 client with the broker serving as a router to obtain a prime exponent p. The applet then checks if the resulting Mersenne $2^p - 1$ is prime.

[4] For more information on Mersenne primes see `http://www.utm.edu/research/primes/mersenne.shtml`

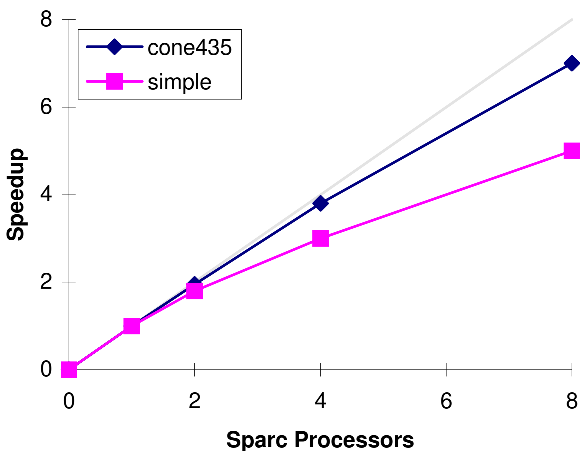


(a) `simple`

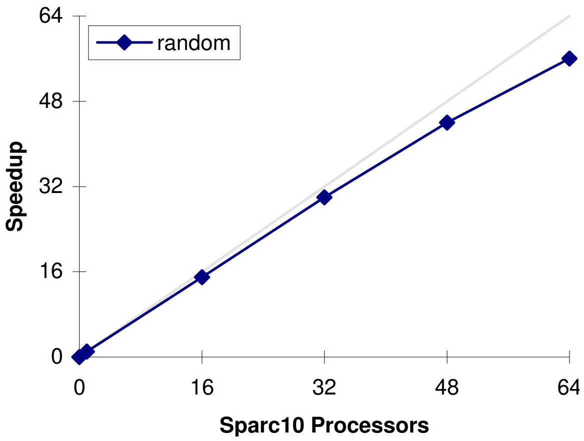


(b) `cone89`

Figure 9: *Ray traced images.*



(a)



(b)

Figure 10: *Speedup curves for* `simple` *and* `cone435` *on a SparcStation cluster and* `random` *on a 64 Sparc-10 processor Meiko CS-2.*

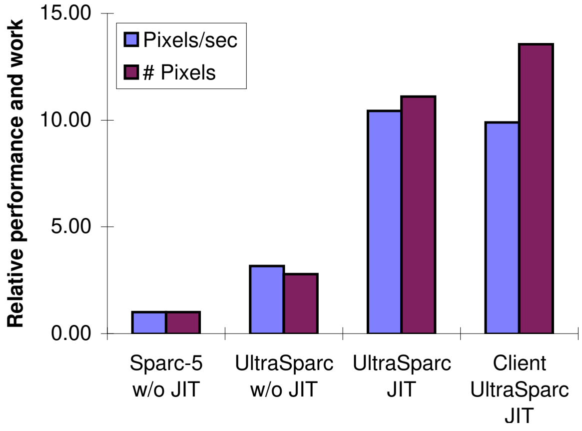


Figure 11: *The relative performance for rendering* `simple` *locally and the relative number of pixels computed by each host when raytracing* `simple` *in parallel on all four processors. All numbers are normalized to the Sparc-5.*

4. After the host applet has completed its computation, it reports the result back to the client. The client 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 client dispenses subsequent primes until the queue is empty.

4.3.1 Measurements

For our measurements, we chose to test the Mersenne primality for all 119 prime exponents between 4000 and 5000. The reason for selecting this range is that on the one hand, we tried to make numbers large enough to simulate the true working conditions of the application, and on the other hand, we wanted to keep them small enough to be able to complete our set of measurements in a reasonable amount of time.

We conducted the experiments on a heterogeneous computing environment consisting of various architectures: Sun UltraSparc (both single and multiprocessor), Sparc5, Pentium PC, and a 64-node Meiko CS-2. We used the JDK (versions 1.0.2 and 1.1.2 without the just-in-time compiler[5]) on all platforms.

The first set of measurements was performed on clusters of identical machines. Figure 12 presents the speedup curves for test runs on a 64-node Meiko CS-2[6] (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. For large numbers of processors communication through a single router becomes a bottleneck. In a more realistic setting where huge primes are checked we do not expect this to be a problem since the computation will be extremely coarse-grained. For our tests, we chose a strategy where the biggest tasks (large amount of computation) were enqueued at the beginning of the task queue, thus ensuring that the hosts that participate first get the largest amount of work. This led to an even task distribution.

Next, we tested our prototype on a heterogeneous environment consisting of 16 PCs, 8 UltraSparcs and one Sparc5. All machines were running Solaris 2.5.1. The execution time for 32 Meiko nodes is given for comparison. 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 should distribute the tasks such that resource utilization is close to optimal. Figure 13 shows total execution times for different combinations of the previously mentioned architectures. In our case, the PCs are slightly faster than the UltraSparcs. The combined resources of all machines showed an improvement, but not linear, due to the saturation of the router and the big difference in computing speed of the various hosts. There are three basic approaches to

[5]The Solaris JIT compiler from Sun was available to us in a beta version. However, at present the Mersenne application will not run on this compiler due to a numerical bug which we hope will be fixed in the near future.

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

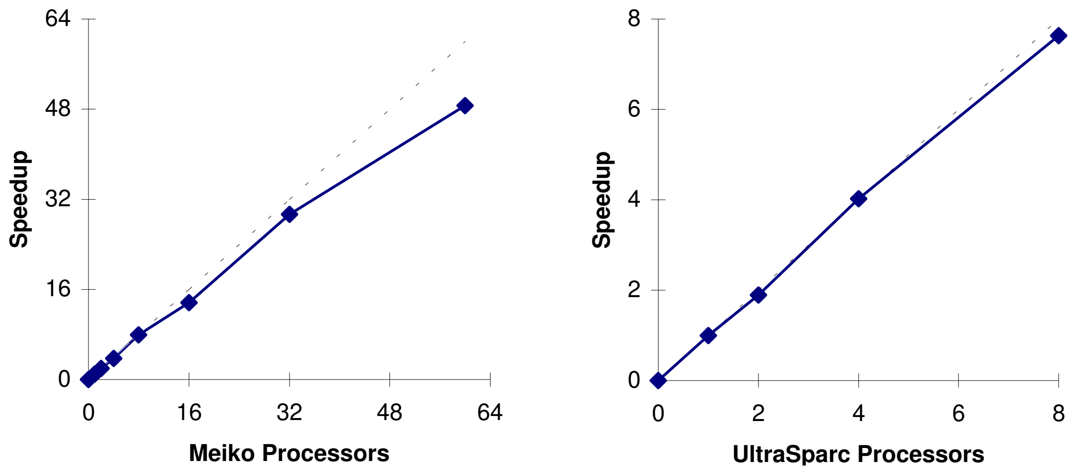


Figure 12: *Speedup curves for primality test on a 64 node Meiko CS-2 (Sparc 10 processors) and on a cluster of 8 UltraSparcs.*

solve this problem: the client could give out more than one number at a time, thus reducing the number of messages, or more than one router could be used, or hosts themselves could become clients for other hosts, leading to a work-stealing approach as in our previous raytracer application.

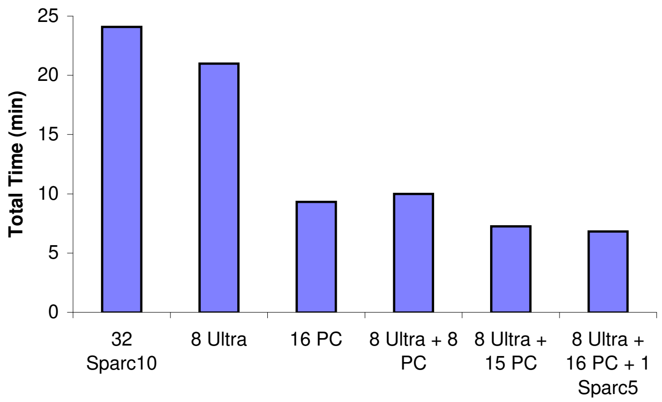


Figure 13: *Primality test on a heterogeneous platform consisting of Pentiums, Sparcs and UltraSparcs.*

We also measured the difference between the C and Java versions of our Mersenne primality test program. Figure 14 shows the execution times to test a Mersenne prime ($2^{4253} - 1$) for a 166 MHz Pentium and an UltraSparc 2. Here, the Pentium PC was running Windows 95, and we used Microsoft Internet Explorer as the test platform. The Explorer already has a just-in-time compiler. We tested the C version of the program and the Java version with and without just-in-time compilation. As we can see, the performance of just-in-time compilers becomes much closer to C (in the case of the Pentium processor, the C version is only 50% faster than the Java compiled version). Newly released JIT compilers for Java are already having a dramatic impact on the execution time (here, a factor 8 improvement in the case of the 166 MHz Pentium).

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. Some of the more recent network computing approaches, include *CONDOR* [LLM88], *Linda* [WL88], *PVM* [Sun90], *Piranha* [GK92], *MPI*

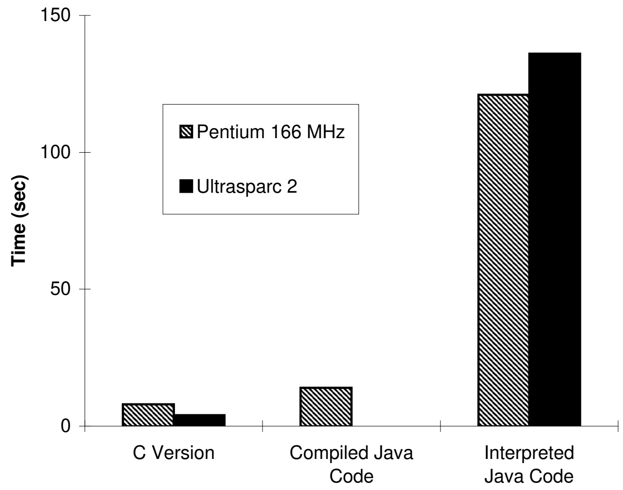


Figure 14: *Comparison of C and Java versions of the Mersenne primality test program for one large prime:* $2^{4253} - 1$.

[MPI94], *Network of Workstations (NOW)* [ACP95], *Legion* [GWtLt97], *GLOBUS* [FK97], and *WebOS* [VEY$^+$97]. The goal of the Legion research project is to provide secure shared object and name spaces, application-controlled fault-tolerance, improved response time, and greater throughput. Multiple language support is another of its goals. Globus is viewed as a networked virtual supercomputer also known as metacomputer: an execution environment in which high-speed networks are used to connect supercomputers, databases, scientific instruments, and advanced display devices. The project aims to build a substrate of low-level services—such as communication, resource location and scheduling, authentication, and data access—on which higher-level metacomputing software can be built. WebOS is being developed with an eye towards providing operating system services for wide area applications such as resource discovery and management, a global namespace, remote process execution, authentication, and security.

Most of these 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. Java helps to address these issues.

The flexibility of Java for Internet computing has 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], *ParaWeb* [BSST96], and *Popcorn* [CLNR97]. All these 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. 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.

Popcorn provides a Java API for writing parallel programs. Popcorn applications are decomposed by the programmer into small, self-contained subcomputations, called *computelets*. The API facilitates something like RMI, but where the Popcorn application does not specify a destination on which the computelet is to execute. Rather, a "market", which brings together buyers and sellers of CPU, determines which seller will run the computelet. Market incentives are supported.

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

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, while at other times utilize resources from other machines when more computational power is needed.

In order to be successful, an infrastructure for transforming the Web into an immense parallel computing resource must be flexible and 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.

We analyzed the limitations imposed by the security restrictions of most Web browsers and showed how they influenced our design and implementation. We also discussed possible extensions of HTTP servers which allow efficient implementations of parallel programming paradigms. Finally, we presented micro-benchmarks and experimental results for two coarse-grain, compute-intensive applications: a raytracer and a Mersenne primality test. For both applications we achieved good speedups on Javelin. We expect that future versions of Javelin will see a performance boost due to optimized JIT compilers.

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

Acknowledgements

Bernd Christiansen was supported by the German Academic Exchange Service (Deutscher Akademischer Austauschdienst). Klaus E. Schauser was supported by the National Science Foundation NSF CAREER Award CCR-9502661. Computational resources were provided by the NSF Instrumentation Grant CDA-9529418, Sun Microsystems, and NSF Infrastructure Grant CDA-9216202.

References

[ACP95] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), February 1995.

[AISS97] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, June 1997.

[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.

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

[BL94] R. D. Blumofe and C. E. Leierson. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, Santa Fe, New Mexico, November 1994.

[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 *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 *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[CLNR97] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN Project: Distributed Computation over the Internet in Java. In *6th International World Wide Web Conference*, April 1997.

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

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

[DES97] DESCHALL. Internet-Linked Computers Challenge Data Encryption Standard. Press Release, 1997.

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

[dM] Frederico Inacio de Moraes. Java RayTrace. http://www.dcc.unicamp.br/ chico/raytracing/raytrace.html.

[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 *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.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997.

[FT96] I. Foster and S. Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, 1996.

[Gil] D. Gillmor. Move over, supercomputers. San Jose Mercury News. http://www.sjmercury.com/business/compute/prime1122.htm.

[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/.

[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.

[GWtLt97] A. S. Grimshaw, Wm. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

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

[Las97a] A. Lash. 40-bit crypto proves no problem. CNET: The Computer Network, January 1997. http://www.news.com/News/Item/0,4,7483,00.html.

[Las97b] A. Lash. 48-bit crypto latest to crack. CNET: The Computer Network, February 1997. http://www.news.com/News/Item/0,4,7849,4000.html.

[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.

[RDS97] Inc. RSA Data Security. The RSA Data Security Secret-Key Challenge, 1997. http://www.rsa.com/rsalabs/97challenge.

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

[Rob54] R. M. Robinson. Mersenne and Fermat numbers. 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.

[Van97] L. Vanhelsuwe. Create Your Own Supercomputer With Java. *JavaWorld*, 2(1), January 1997.

[VEY+97] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services For Wide Area Applications. Technical Report CSD-97-938, UC Berkeley, 1997.

[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.