Una Infraestructura Genérica para Computación Distribuida en la Web
A Generic Infrastructure for Web Computing

Enrique V. Carrera Pablo Bustamante Fausto Pasmay
Systems Engineering Department
University San Francisco of Quito
PO Box 17-12-841, Quito, Ecuador
{vinicioc,pablo,fausto}@usfq.edu.ec

Abstract

This paper proposes a generic infrastructure for the development of parallel and distributed applications on the Web. The infrastructure is oriented to allow that every single host in the Internet can participate in the execution of distributed applications using a very simple configuration with rigid guarantees of security. Our proposal is based on the use of World Wide Web protocols and Java applets, exclusively. Thus, users willing to participate in the execution of applications only require a conventional Web browser with the Java Runtime Environment enabled. This work differs from previous proposals in its simplicity and improved performance. In addition, our study includes a detailed evaluation of real parallel applications.

1. Introduction

The Internet has been growing rapidly over the last few years interconnecting billions of computers all over the world. A significant fraction of these computers is idle most of the time. For instance, home computers have had a tremendous processing power increase but most of their resources remain practically unused. Considering this huge aggregated processing capacity, the Internet can be seen as a convenient platform for the execution of parallel and distributed applications.

This singular view of the Internet has already been exploited by Enterprise [16], Grid [1] and Peer-to-Peer [14] Computing. However, using the Internet as a metacomputing resource introduces new difficulties and problems. The most important of these problems are the heterogeneity of the participant systems, difficulties in administering the dynamic execution environment, critical users' security concerns, and high communication delays.

In order to attenuate some of the problems mentioned above, we propose a generic infrastructure for the development of parallel and distributed applications on the Web. This infrastructure is oriented to allow that every single host in the Internet can participate on the execution of distributed applications using a very simple configuration with rigid guarantees of security. Our proposal is based on the use of World Wide Web protocols [11] and Java applets [18], exclusively. Thus, users willing to participate in the execution of distributed applications only require a conventional Web browser with the Java Runtime Environment enabled.

In this form, the heterogeneity of the participant systems is not a problem anymore, since the Java Runtime Environment is very portable. In fact, Java is supported by innumerable operating systems and hardware architectures. In the same way, the difficulties in administering the dynamic execution environment are reduced. There is no need of installing software or configuring users at each participant host. Finally, security concerns from machine owners disappear. The Java Runtime Environment takes control of all the operations executed by Java applets, since they are run inside the Java sandbox [12].

Another mentioned problem is the high communication delays existing in the Internet. Although, these delays are being reduced by the new hardware and software infrastructures present in modern networks [8], our proposal provides a peculiar mechanism for the communication among application components. This mechanism breaks the limitation imposed by the *host-of-origin* browser security policy that establishes that applets can only communicate with the host where they were loaded from. Using digitally signed applets, users willing to trust the developers' signature can allow direct communication among applets in different machines. However, in order to support hosts belonging to private networks (*i.e.*, behind NAT or PAT) or hosts with users that do not trust specific applets, our proposal also contemplates the possibility of communication through the host-of-origin of the applets.

Besides these basic characteristics, our proposal also includes a new Java class library and a set of debugging and monitoring tools. The Java class library helps developers to write new applications using a simple, high-level view of our infrastructure. In this way, developers can concentrate in defining application's behavior instead of worrying about implementation details. Finally, the set of debugging and monitoring tools allows us to capture and replay every packet exchanged among applets, besides some other actions taken by them. The tools include both text-based and graphical interfaces.

This proposal was evaluated using three different applications, besides some specific microbenchmarks. The results of our evaluations show that the scalability of our infrastructure is very attractive for coarse-grain applications. Fine-grain applications do not scale well, but they could also have some important benefits. The results of our microbenchmarks show that the maximum bandwidth achievable using direct communication between applets is almost the same of using indirect communication through the host-of-origin. However, message latency is reduced by around 70% when direct communication is used instead of indirect one.

The remainder of this paper is organized as follows. In the next section, we discuss some background concepts related to our work. Section 3 describes in detail the proposed infrastructure. The evaluation of our particular implementation is presented in section 4. Then, section 5 analyzes the main works related to our proposal. Finally, section 6 summarizes our findings and concludes the paper.

2. Background

In this section we would like to discuss some concepts that support the main topic of our research. More specifically, we would like to review some ideas related to the World Wide Web, the Java platform, and digital signatures.

2.1. The World Wide Web

The World Wide Web, or simply Web, is the major service deployed on the Internet. The Web is made up of *Web servers* that store and disseminate *Web pages*, which are text documents embedded with HTML (HyperText Markup Language) tags that define how the text will be rendered on screen. Web pages are accessed by the user via a Web browser application such as *Internet Explorer*, *Firefox*, *Opera*, *Netscape*, etc. These Web browsers communicate with Web servers primarily using HTTP (HyperText Transfer Protocol). HTTP allows Web browsers to fetch Web pages from Web servers as well as submit information to them.

Web servers normally store HTML pages. However, they can also be a storehouse for any kind of file delivered to a client application via HTTP. In other words, modern Web pages are "rich" documents that contain text, graphics, animations and videos for anyone with an Internet connection. The browser renders the pages on screen and automatically invokes additional software as needed. For example, animations and special effects are browser plugins, and audio and video files are played by the media player software that either comes with the operating system or from a third party.

2.2. The Java Platform

Java SE (Standard Edition) [18] is a complete environment for application development and deployment. There are two principal products in the Java SE family: the Java Runtime Environment (JRE) and Java Development Kit (JDK). The first one provides the Java Virtual Machine (JVM), the Application Programming Interface, and several other components needed for running applications. The second product permits the development of new applications written in the Java language. Basically, it is a set of compilers and development tools.

In this paper, we are focused in the use of Java applets. Applets are software components written in Java that run in the context of a Web browser or container. Applets can easily be embedded into HTML pages and executed by Java-capable browsers. A Java-capable browser is one with the JRE plugin installed and enabled.

When a browser is used to view a page that contains an applet, the applet's code is transferred from the Web server to the local machine and then executed by the browser's JVM inside the Java sandbox. The Java sandbox restricts applets from performing unsafe activities. It basically relies on three prongs of defense: the Bytecode Verifier, the Class Loader, and the Security Manager. Together, these three prongs perform load and runtime checks to restrict filesystem and network accesses, as well as browser internals [12].

In the case of applets, the policies adopted by the Security Manager become very restrictive, allowing only a limited set of possible elements (*e.g.*, the host-of-origin constraint). In order to overcome the limitations imposed by the Security Manager, the applet must be digitally signed, and the client must trust (and accept) the applet by confirming the digital signature.

2.3. Digital Signature

Digital signature is a cryptographic mechanism for guaranteeing message authenticity, integrity and non-repudiation. Authenticity allows the recipient of a message to confirm the identity of the sender. Integrity guarantees that the message has not been altered during transmission. And finally, non-repudiation avoids the sender to deny its association with a particular message.

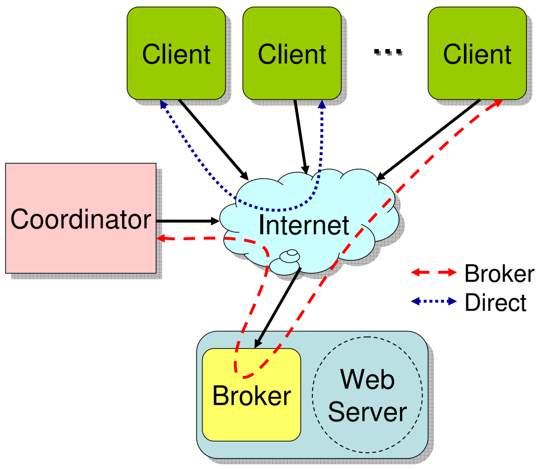

Figure 1. Structure of a typical application.

Public-key digital signature schemes rely on public-key cryptography. In public-key cryptography, each user has a pair of keys: one public and one private. The public key is distributed freely, but the private key is kept secret and confidential.

The distribution of public keys is made through Digital Certificates [6], that for actual security should come from a known and trusted Certificate Authority, which validates that the key belongs to whom the certificates claims to be. There exist several levels of certification, and each one requires more evidence that the information provided is accurate and belongs to the person who ask for that certification.

3. Infrastructure Description

In this section, we describe the main components of the proposed infrastructure and their corresponding interactions.

3.1. Basic Architecture

In our infrastructure, applications are a collection of asynchronous cooperating applets where we can distinguish three main type of components: a communication broker, an application coordinator, and one or more clients. Obviously, because most of the components are accessed through a Web browser, we also need a Web server. Figure 1 shows the typical structure of a distributed application using our generic infrastructure.

The Broker. The communication broker is the central component of our infrastructure. Its main functionality is to pass messages among all the other application components, routing them in a transparent way. However, other functionalities like name service, synchronization management, shared memory support, and error handling can also be integrated.

The broker runs as a standalone application on the same machine that runs the Web server. This is the only component that must be executed next to the Web server, since all the coordinators and clients (*i.e.*, Java applets running on a Web browser) must be able to connect to it without restrictions. This component can be written in any programming language and it is completely independent of the application. In fact, a single broker can be simultaneously used by several applications.

The current implementation of our infrastructure includes a communication broker written in Java that is extremely generic in order to support a huge range of applications. This broker assigns a unique ID to each component connected to it, and it is able to associate symbolic names to those IDs. In this way, the broker also performs name server tasks allowing to know the ID, name, and network address of each component connected to it.

In this specific implementation, messages are routed adding a very simple header to each message. Thus, every incoming message includes the destination ID and its message size. After receiving the message, the broker changes the destination ID by the corresponding source ID and forwards the message to the target component connected to the broker. In this way, the receiver can know where the message is coming from.

Since our implementation only supports asynchronous messages, the broker does not include complex error handling functions. However, our broker is easily extensible and can be adapted to particular work conditions.

It is important to note that several application components communicate through the broker. In other words, this component must be able to support several simultaneous data transfers and name service requests. In order to get an acceptable performance, a multithreaded approach is used by our current implementation.

The Coordinator. As mentioned, our infrastructure is oriented to run collections of asynchronous cooperating applets. In this way, we propose to implement task-level parallelism using a centralized task queue in our applications. This computational model is attractive for three reasons: (i) some parallel/distributed applications are best expressed as a collection of coarse-grain tasks, (ii) the task queue model has implicit load balancing, and (iii) fault tolerance mechanisms are easier to implement in such a model.

Based on this, the coordinator is the component in charge of taking control of one application. It implements the task queue interface besides some load balancing and fault tolerance mechanisms. Advanced load balancing mechanisms are useful to exploit clients with different processing power characteristics, while fault tolerance mechanisms are very important to deal with unreliable clients.

The coordinator basically initiates building the task queue according to the initial conditions of the application, connecting to the communication broker, and waiting for clients. As clients arrive, the coordinator assigns them a new task and waits for their corresponding responses. At the end, the coordinator can finish writing the solution computed by all clients to permanent storage devices.

An important characteristic of the coordinator is the fact that it does not need to be run at the same machine where the Web server is executing. In fact, the coordinator can be executed as another client, using the Java-applet technology. However, reliability is a key characteristic needed by this component.

The Clients. The clients are the entities that actually perform the data processing in our applications. Each client is an applet that executes inside a Web browser. A host willing to participate in the execution of a distributed application only needs to access the Web page containing the corresponding applet.

The code of the clients can be general enough to compute anything we want. However, clients are restricted to communicate with the host-of-origin, exclusively. In this way, clients start connecting to the broker running on the Web server where they are from, and requesting the ID of their coordinator. Then, clients ask for their corresponding execution parameters (or tasks) to the coordinator of the application. Finally, after computing each assigned task, clients send back to the coordinator the results of their computation.

Direct Communication. The main advantage of the communication model described until now is the possibility of reaching any computer in the Web, regarding of the client being behind a NAT or proxy. In fact, we can have several clients running on the same machine without any problem. In addition, this approach fulfills all the security restrictions for applets executed inside Web browsers.

However, the big disadvantage of this communication model is the extra overhead of passing through the broker (see line labeled 'Broker' in figure 1). Latency and throughput problems can easily appear due to the extra hop. In fact, the broker could easily become a bottleneck for some applications.

In order to solve this problem, we propose to use direct communication among application components. In this communication model, the clients have the ability of communicating among themselves without using the broker (see line labeled 'Direct' in figure 1). However, in this case we are violating the restrictions imposed by the security model adopted by Web browsers. Thus, we need to attach a digital signature to the applets in order to prove the identity and good intentions of the signer. The user executing the applet can accept or not this digital signature. If the user executing the applet trusts and accepts the digital signature, the applet can overcome the host-of-origin restriction. Otherwise, the applet does not have other alternative that communicating through the broker.

The main disadvantages of this new communication model are the need of real IP addresses for every application component and its conditioning to digital signature approval by the remote user/host. On the other hand, its main advantages are the better performance and significant scalability of our applications.

3.2. Libraries and Tools

An extra goal of this work is to facilitate the development of new applications using our proposed infrastructure. In order to achieve this goal, we have created a new Java class library and some debugging and monitoring tools. These extra elements of our infrastructure are described in the following paragraphs.

The Java Class Library. Our library provides generic classes for the development of clients and coordinators used by new applications. Note that the broker developed in this work does not have to be modified unless the user needs new functionalities.

The library has two main classes: The first one facilitates to connect any applet to the communication broker, and the second one allows to connect two applets directly. Both classes create input and output streams on which messages can be received and sent, respectively. In addition, any applet can use the connection to the broker for setting its symbolic name, asking for another ID or IP address, synchronize among components, etc.

As we can see, our library is mainly oriented to support an efficient and easy communication among applets. However, we are planning to extend this library to support complex error handling methods, advanced fault tolerance and load balancing techniques, shared memory abstractions based on Linda-style tuples [20], etc.

The Application Monitor. We have developed a standalone application that can connect to the broker and display the instantaneous state of any distributed application attached to that broker. The level of details showed by the monitor can be dynamically configured at the broker.

Applications that use broker-assisted communication do not generate any extra traffic, since the broker is the one in charge of sending event notifications to the monitor. Applications that use direct communication among applets receive a notice from the broker that force them to send event notifications to the broker. In this case, event notifications are created by the own Java class library guaranteeing a very short representation of every event in order to generate a small amount of traffic. It is also important to mention that each event notification carries a millisecond-level timestamp that can be very useful in debugging processes.

This application monitor is able of creating a log file using all the event notifications sent by the broker. The log can be used later for recreating the execution of applications through the application replaying option. This option allows us to debug an application using a dynamically adjusted replaying speed, and both text-based and graphical interfaces.

3.3. Application Examples

In order to probe the feasibility of developing distributed applications using our infrastructure, we have written three parallel applications: numerical integration, matrix multiplication, and successive over relaxation. These applications incorporate basic load balancing techniques and simple fault tolerance algorithms. However, as we can see in the section 4, the performance of these applications is very impressive.

Numerical Integration. In this particular implementation, the integration coordinator starts splitting the interval of integration in N parts and waits for clients. When a client asks to the coordinator for its computation parameters, the coordinator sends to the client the lower and upper bounds of the assigned interval and the corresponding δ value. The client receives the parameters and integrates the coded function using the Newton-Coates method. After finishing, the client sends back to the coordinator its partial result and asks for another interval.

The integration coordinator continues sending the corresponding parameters and receiving the partial results until all the integration parts have been computed. At the end, the overall integration result is printed.

As we can see, this application is easily parallelizable, since there is almost no communication among application components. The computation in the clients is completely independent from each other. Thus, this application is extremely scalable and we do not need direct communication among clients.

Matrix Multiplication. In this application, the coordinator starts establishing the matrices A and B to be multiplied. Then, the coordinator splits the computation in N tasks, where each task has a determined number of rows from A and columns from B. After that, the coordinator waits for clients. When a client asks for a new task, the coordinator sends back the size of the matrices, and the elements assigned to that multiplication. When the computation is done, the client sends to the coordinator the results with their corresponding row and column identifications.

On the other side, the coordinator keeps track of all assigned tasks and stores the results sent by clients. After all the elements of the solution have been computed, the coordinator finishes writing a file with the solution.

As in the numerical integration case, this application is easily parallelizable. However, there is much more communication between the coordinator and the clients, since the coordinator sends parts of the matrices A and B to each client. Although there is no communication among clients, the communication with the coordinator is extremely demanding on network bandwidth.

Successive Over Relaxation. In our SOR implementation, the coordinator starts defining the matrix to process and splits it in N equal parts. Each part also includes the lower and upper adjacent rows. After that, the coordinator waits for clients to arrive. When a client asks for its task, the coordinator sends back just one part of the matrix, including the relative position of that part. Then, each client starts computing $a_{i,j} = (a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1})/4$. After all the elements have been computed, clients must exchange shared rows with their neighbors in order to proceed with the next interaction. These steps are repeated by a predefined number of interactions. At the end, clients send their corresponding parts back to the coordinator which writes the solution to disk.

As we can see, this application has a lot of communication among clients. If fact, this is a typical regular application that is not easily parallelizable. Every computing pass requires exchanging shared rows with the lower and upper neighbors. Based on this, we have decided to implement direct communication between adjacent clients to reduce communication overhead. In this alternative SOR implementation, clients send shared rows directly to their neighbors without passing through the broker.

4. Performance Evaluation

In this section we present some performance results from two microbenchmarks and three parallel applications. But before that, a brief description of our methodology is given.

4.1. Methodology

All the experiments were executed in a network of four Sun Blade 1500 and two Sun Fire V240. The Sun Blade machines are being used as clients, while the Sun Fire ones are our servers. All the machines are running Solaris 9/04 and are interconnected by a Fast-Ethernet network with a 3Com 3300 switch. The version of our JDK and JRE is 1.5.0_08.

With relation to our applications, numerical integration integrates $y = (x^2 + 0.25)^{-1}$ in the interval $0 \leq x \leq 100$ using $\delta = 10^{-8}$. The matrix multiplication multiplies matrix $A[10, 8000]$ by matrix $B[8000, 20]$ using double-precision floating point representation. Finally, our SOR application uses a 16000×500 matrix of doubles and a thousand interactions.

Communication	Latency (ms)	Bandwidth (Mbps)
Broker-assisted	1.037 ± 0.004	93.04 ± 0.05
Direct	0.315 ± 0.012	94.83 ± 0.06
Improvement	69.6%	1.9%

Table 1. Microbenchmark results.

Microbenchmarks. We have built two basic microbenchmarks in order to evaluate latency and bandwidth communication. The architecture used by the microbenchmarks is the same described in section 3: We have a master (or coordinator) and a slave connected to the broker. These components can use a broker-assisted or direct communication between them.

The latency test consists in sending a 0-byte message from the master to the slave and waiting for the corresponding response. This process is repeated 10 thousand times and the average time is measured. On the other hand, for the bandwidth test, the master sends 16000-byte messages without waiting for any response or acknowledgment. After 10 thousand messages, the average bandwidth is computed.

4.2. Results

Microbenchmarks. Table 1 summarizes the latency and bandwidth results from our microbenchmarks. As we can see, the latency in the direct communication approach is approximately 70% lower than in the broker-assisted one. With relation to the bandwidth measurements, the differences are less significant. Both communication approaches practically saturate the 100-Mbps Ethernet links.

We can conclude that direct communication is useful in terms of latency, mainly. However, even the bandwidth can be improved when the broker becomes a bottleneck. For instance, consider the case where several copies of the bandwidth microbenchmark are running simultaneously using broker-assisted communication. In this situation, each microbenchmark shares the bandwidth of a single broker.

In order to determine the actual network limits, we also wrote a few benchmarks in C. The latency of a ping-pong communication is 0.22 ms and the maximum achievable bandwidth is 94.9 Mbps. Thus, we can also conclude that direct communication is very close in performance to the values obtained using low-level programming.

Real Applications. Figure 2 shows the speedup of numerical integration using up to 4 computing machines. We can see a perfect linear speedup, since there is almost no communication among application components.

In similar way, figure 3 shows the speedup of matrix multiplication. In this case, the application is serialized by the coordinator during data distribution and result merging, being data distribution the most demanding activity. However, our original application presents a super-linear speedup, as seen in figure 3 for the line labeled ‘Blocking’. This behavior is due to the blocking of the application coordinator when waiting for client requests. Once the coordinator is blocked, the JVM takes a long time for unblocking it. However, while more clients has the application, it is less probable that the coordinator becomes effectively blocked.

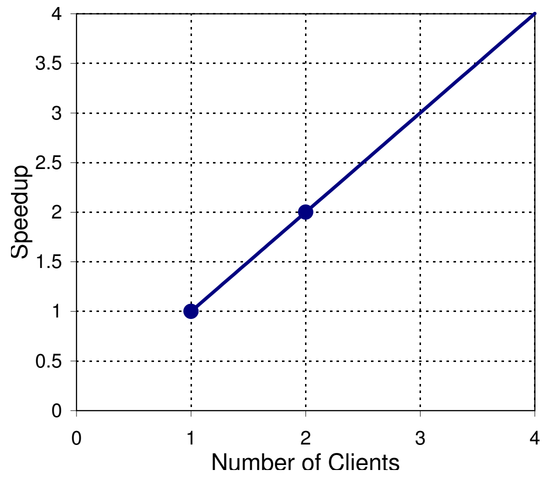

Figure 2. Speedup of numerical integration.

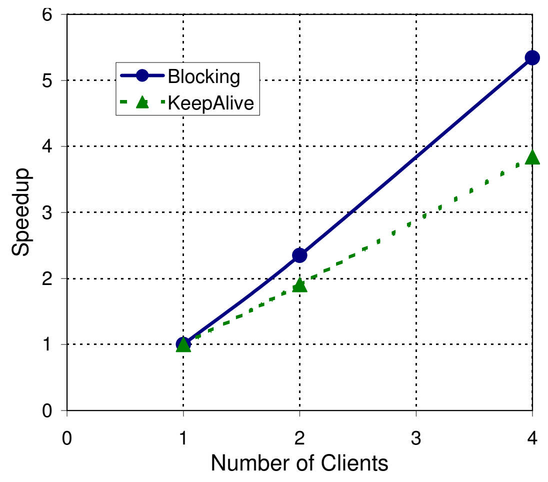

Figure 3. Speedup of matrix multiplication.

In order to verify our previous explanation, we developed a new matrix multiplication where the coordinator never blocks. It implements a *busy-wait* for client requests. In this case the speedup of the line labeled ‘KeepAlive’ is not as good as the numerical integration, but we reach an almost linear speedup.

Finally, the speedup of the SOR application is presented in figure 4. Our first version uses broker-assisted communication (line labeled ‘Broker’) and runs on a system with

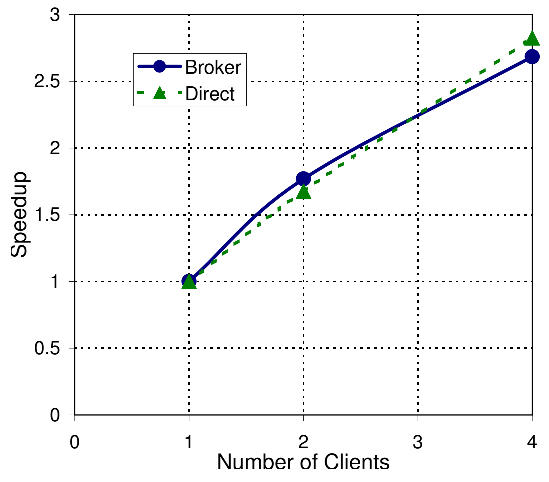

Figure 4. Speedup of SOR.

up to 4 computing machines. We can see that there is a significant speedup, but it drops quickly when the number of nodes is increased. This behavior can be explained by the strong synchronization required by the application, and because the speedup is computed against the application running in one node where there is no communication at all.

Our second version of this application uses direct communication between clients. Although the performance of the second version is very similar to the obtained by our first version, we can see a more linear speedup. The reason that explains why a broker-assisted communication is better that a direct communication when 2 clients are used is the TCP slow-start mechanism. However, when 4 machines are used, the extra scalability of direct communication outperforms the overhead of the TCP slow-start algorithm.

5. Related Work

There are several works related to integrate networked computers into a global computing resource. Most of these systems require the maintenance of binaries for all the architectures used in the computation, and the program has to reside at each site (or at a shared filesystem). They also require the user or the system to have an account on each machine participating in the computation. All these factors severely limit their use as a metacomputing facility on the Web. In addition, current systems have almost no support for dynamic load balancing or fault tolerance.

Works focused on Java-based distributed computing include JPVM [10], JMPI [9], ATLAS [2], ParaWeb [7], JavaParty [17], SMPD Programming in Java [13], WebFlow [5], Ninflet [19], Charlotte [4], Javelin [15], and KnittingFactory [3].

JPVM and JMPI use Java to overcome heterogeneity, but are not intended to execute on anonymous machines. What they provide is a message passing interface for Java standalone applications (not applets). ATLAS furnishes a global computing model based on Java, ensuring scalability through a hierarchy of managers. It relies on native code which eliminates the guarantees of a secure program execution. ParaWeb creates a global computing infrastructure using extensions to the Java programming environment (through a parallel class library) and the Java runtime system. Users need to install those extensions in order to allow a transparent remote execution of Java threads. JavaParty supplies mechanisms (built on top of Java RMI) for the transparent distribution of remote objects. JavaParty requires to run a Java process, called *LocalJP*, in all the machines used by the computation. Similarly, the usage of the SMPD programming model on a distributed shared memory abstraction [13] requires to run a local Java process that acts as a runtime environment. WebFlow is one of the earlier workflow systems supporting centralized application composition in Grid environments. It includes a complete programming paradigm and coordination model for Java that exclude typical Web browser users. Finally, Ninflet is an infrastructure for migratable objects which is being targeted for idle cycle based parallel computing. Resource providers need to run the *Ninflet Server* daemon on their hosts.

As we can see, most of the previous works do not allow that every single host in the Internet can participate in distributed computations using common Web browsers. They all need a local account or at least the possibility of running local Java processes on each machine.

From previous cited projects, all but Charlotte, Javelin, and KnittingFactory fail to take advantage of Web browsers in bringing distributed computing to every-day users. These systems were specifically designed for parallel programming over the Web. By leveraging the ability of browsers to download and execute remote applets, they provide the means for any user, anywhere on the Internet, using any Java-capable browser to participate in a parallel computation. Charlotte, Javelin and KnittingFactory have a common feature in their design: they require a host running a Web server in addition to a standalone Java application. The role of the standalone application is to distribute work among browsers and to act as a message forwarding agent for communication among applets.

Although KnittingFactory can work in the same way as the other two proposals, it also has some extra characteristics. First, the possibility of violating the Java sandbox restrictions using a customized RMI reference forwarding[1]. And second, the capability of running coordination processes, called *Initiators*, on any machine other than the original Web server.

Our proposal is similar to Charlotte and Javelin, but we

[1]The proposed approach for direct communication between applets only works in Java 1.1.

also allow direct communication among application components using digitally signed applets. The digital signature mechanism also permits to overcome some other limitations imposed by the Java sandbox, besides the host-of-origin constrain.

Another important difference of our proposal is the fact that the application coordinator can be run outside the Web server. Thus, we have most of the advantages attributed to KnittingFactory, but with a much simpler approach. Finally, our study includes a detailed performance evaluation of real parallel applications.

6. Conclusions

This work describes a simple and generic infrastructure for running parallel and distributed applications on the Web. In order to execute existing applications, users only need a Java-capable Web browser. The proposed infrastructure also provides a new Java class library and some monitoring and debugging tools for supporting the development of new applications. Basically, the developer needs to write an application coordinator and its corresponding clients, since our communication broker can be reused without modification. All applets must be made public through the Web server running on the machine where the broker is executing.

Our results show that it is important to use direct communication among application's components when possible, since latency and bandwidth performance are significantly influenced by the communication model. Other important factors to take in consideration are fault tolerance and load balancing, since the Web is not either a reliable nor a homogeneous platform.

A near-future extension to our work is to use more than one broker, allowing to have better fault tolerance and scalability in our applications. However, this requires the development of more complex name services and coordination algorithms. In the same way, we are planning to implement shared memory support using Linda-style tuples [20].

References

[1] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Software – Practice & Experience*, 32(15):1437–1466, December 2002.

[2] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An infrastructure for global computing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[3] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with Java applets. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

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

[5] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran. Webflow – a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency – Practice and Experience*, 9(6):555–577, 1997.

[6] S. A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, 2000.

[7] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[8] P. Dickens, B. Gropp, and P. Woodward. High performance wide area data transfers over high performance networks. In *Proceedings of International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, pages 254–262, 2002.

[9] K. Dincer. Ubiquitous message passing interface implementation in Java: JMPI. In *Proceedings of the 13th International Parallel Processing Symposium*, 1998.

[10] A. Ferrari. JPVM: Network Parallel Computing in Java. *Concurrency – Practice and Experience*, 10(11-13):985–992, 1998.

[11] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report RFC 2616, Internet Engineering Task Force, June 1999.

[12] A. Herzog and N. Shahmehri. Performance of the Java Security Manager. *Computers & Security*, 24(3):192–207, 2005.

[13] S. F. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, 1997.

[14] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57R1, HP Labs, Palo Alto, CA, July 2002.

[15] M. O. Neary, B. O. Christiansen, P. Cappello, and K. E. Schauser. Javelin: Parallel computing on the Internet. *Future Generation Computer Systems*, 15(5–6):659–674, 1999.

[16] M. P. Papazoglou and W.-J. van den Heuvel. Web services management: A survey. *IEEE Internet Computing*, 9(6):58–64, November–December 2005.

[17] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[18] Sun Microsystems Inc. Java Technology. `http://java.sun.com`, June 2007.

[19] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: A migratable parallel objects framework using Java. *Concurrency: Practice and Experience*, 10(11–13):1063–1078, 1998.

[20] G. C. Wells. New and improved: Linda in Java. *Science of Computer Programming*, 59(1–2):82–96, 2006.