INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

WebComputing: Design and Performance

by

Kevin M. Ying

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2000

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

4/20/2000
Date

Professor David Arnow
Chair of Examining Committee

4/24/00
Date

Professor Theodore Brown
Executive Officer

Professor Marvin Bishop

Professor Stanley Habib

Professor Paula Whitlock

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

WEBCOMPUTING: DESIGN AND PERFORMANCE

by

Kevin M. Ying

Adviser: Professor David Arnow

The advent of the Java programming language, with its support for Web-deliverable applets, has created a new, promising parallel computing platform that we call *WebComputing*. This platform can break through the conventional boundaries imposed by traditional distributed systems, such as DP, PVM, and MPI. The essential idea is that a master server, or collection thereof, in league with a collection of Web servers coordinates the execution of tasks by applets running in parallel on an ever-changing set of unreliable, heterogeneous client machines. The promise of WebComputing is the potential for achieving an unprecedented degree of parallelism.

WebComputing is in its infancy. It faces many challenging questions that are related to Internet communication delays, unreliable execution environments, and system scalability. Research that can answer the following questions is greatly needed.

1. How does WebComputing communication perform at its current stage?
2. How does communication performance affect the WebComputing architecture design and the programming models?
3. What performance enhancements can be made in terms of scalability and load balancing?

This thesis covers three primary areas of research pertaining to each of these questions. First, it presents a performance study of communication protocols for

WebComputing. The results of this study can be used as guidelines for forthcoming WebComputing system designs and their application implementations. Based on these guidelines, the thesis presents a unique WebComputing system design and implementation, called the SWC framework. Using layered architecture, this framework provides a high-level coherent API and a robust system implementation, which enables the system to be executed as threads on SMP machines, as processes on a collection of networked workstations, or as Java applets in a WebComputing context. Furthermore, this thesis describes the design and implementation of a simple, yet effective scalability enhancement of the SWC framework that incorporates multiple levels of system-wide load-balancing schemes. The measurements of the scalability enhanced system show the effectiveness of this design. In addition, the enhanced framework also provides a deployment system using Java servlet technology.

Currently, several projects (including Monte Carlo computation and classical Operational Research projects) are underway using the SWC framework. In addition, two Computer Science courses were taught using this framework and students took it easily.

Acknowledgements

First, I would like to thank my advisor Prof. David Arnow, my dissertation committee: Prof. Marvin Bishop, Prof. Ted Brown, Prof. Stanley Habib, and Prof. Paula Whitlock. I am especially grateful to my advisor Prof. Arnow for the years of consistent guidance he has provided and for always keeping my best interests in mind. I am also indebted to Prof. Habib for his unequivocal support and guidance over the years.

Pursuing a Ph.D. would have been impossible without the financial supports I have received. Special thanks are due to Prof. Habib, Prof. Arnow, and Prof. Kenneth McAloon for working together to insure the series of fellowships and scholarships I have received. In addition, I would like to acknowledge that this research was supported in part by ONR grant N00014-96-1-1057 and the National Science Foundation's CISE program #CDA-9522537.

I am indebted to my friend Dr. Fabian Zabatta for his consistent input, support, and inspiration. I wish to thank Prof. McAloon for his advice and mentorship, Prof. Carol Tretkoff for her introduction to parallel computing, and Prof. Whitlock for her support and guidance over the years. I also wish to thank Prof. Dayton Clark and Prof. Gerald Weiss for their valuable input. My friends Barry Cohen and C.S. Rani also deserve my thanks for their input and inspiration.

The last thanks is due to my family. I would not be where I am today if not for the selflessness of my mother Susan Ying and my grandfather Dr. S. H. Ying. My brothers, Jeff and John, have given me much support and encouragement. Lastly, I would like to thank my fiancée, Mery Lam, for her unconditional love and selfless support.

Table of Content

CHAPTER 1 ..... **1**

INTRODUCTION **1**

1.1 Introduction *1*

1.2 WebComputing Research *2*

1.3 Contributions *3*

1.3.1 Communication Protocol Evaluation 3

1.3.2 SWC Framework Design 4

1.3.3 Scalability Enhanced SWC Framework 5

1.4 Outline *6*

CHAPTER 2 ..... **7**

DISTRIBUTED COMPUTING **7**

2.1 Parallel and Distributed Computing *7*

2.2 Obstacles in the Distributed Computing Environment *8*

2.2.1 Network Heterogeneity 8

2.2.2 Communication Delay 9

2.2.3 Unreliability 10

2.2.4 Administrative Overhead 11

2.2.5 The Need for Distributed System Architectures 11

2.3 Traditional Distributed Systems and Their Limitations *11*

2.4 Internet Computing *14*

CHAPTER 3 ..... **17**

WEBCOMPUTING **17**

3.1 Introduction *17*

3.2 WebComputing Environment *19*

3.2.1 WebComputing Programming Environments and Their Advantages 19

3.2.2 Challenges to WebComputing 22

3.3 WebComputing Related Works *24*

3.3.1 WebComputing Predecessors 25

3.3.2 WebComputing Projects 27

3.3.3 Experiments 41

3.4 WebComputing Research *44*

CHAPTER 4 ..... **45**

WEBCOMPUTING COMMUNICATION PROTOCOL EVALUATION **45**

4.1 Introduction *45*

4.1.1 WebComputing Communication 45

4.1.2 Protocol Evaluation 46

4.2 Experiments *48*

4.2.1 Experiment Design 48

4.2.2 Data Conversion 50

4.2.3 Experiment Setup 50

4.3 Results *51*

4.3.1 Connection Setup Costs 51

4.3.2 Asymptotic Costs 52

4.3.3 Accumulated Costs 56

4.4 Conclusion *59*

CHAPTER 5 ..... **62**

SWC FRAMEWORK DESIGN **62**

5.1 Introduction to the SWC Framework *62*

5.1.1 SWC Design Objectives 63

5.2 SWC System Design *64*

5.2.1 Master-Worker Programming Model 65

5.2.2 SWC API Design 66

5.2.3 SWC Framework Design 67

5.2.4 Three Implementations in One 70

5.2.5 SWC Internal Design 71

5.3. Performance *74*

CHAPTER 6 ..... **77**

SCALABILITY ENHANCED SWC FRAMEWORK **77**

6.1 Introduction to Enhanced SWC System *77*

6.2 Enhanced SWC Framework Design *78*

6.2.1 Enhancement Overview 78

6.2.2 Design for Supporting Multi-Router Architecture 79

6.2.3 Deployment Enhancement 90

6.3 Outcomes of the Enhancements *94*

CHAPTER 7 ..... **96**

CONCLUSIONS **96**

7.1 Contributions *96*

7.2 Usage *97*

7.3 Future Research *98*

7.4 Final Remark *99*

APPENDIX A **100**

SWC FRAMEWORK USER'S GUIDE **100**

1. Introduction to the SWC Framework 100

2. Basic Programming Specification 100
3. Execution Interfaces 102
4. SWC Application Class Naming Convention 107
5. Example 108

GLOSSARY 111

GLOSSARY OF THE SWC FRAMEWORK: 111
GLOSSARY OF ACRONYMS: 112
GLOSSARY OF TECHNICAL TERMS: 114

BIBLIOGRAPHY 116

List of Figures

Figure 3-1, A Setting for WebComputing 18
Figure 3-2, Sample Charlotte Routine 29
Figure 3-3, Charlotte's Virtual Machine Model and Run-time System 30
Figure 3-4, SuperWeb's WebComputing Blueprint 33
Figure 3-5, Javelin WebComputing Architecture 35
Figure 3-6, Bayanihan Framework 38
Figure 3-7, SAM's Programming Model 39
Figure 3-8, SAM's Target Platform 40
Figure 4-1, Asymptotic Costs of a 255-Byte Object on Solaris 53
Figure 5-1, SWC Framework Programming API 66
Figure 5-2, SWC Framework Design 68
Figure 5-3, SWC Data Flow 70
Figure 5-4, SWC Architecture Implementation 71
Figure 5-5, SWC Framework API Classes 71
Figure 5-6, SWC Framework Runtime Classes 72
Figure 5-7, SWCRouter Design 73
Figure 5-8, SWCRouter Actors 74
Figure 6-1, E-SWC Framework 81
Figure 6-2, E-SWCRouter Actors 84
Figure 6-3, Load Balancing Task Flow Between Two Routers 89
Figure 6-4, The Settings for the E-SWC Framework 92
Figure 6-5, Servlet Architecture for Web Deployment 93
Figure A-1, The SWC Framework API 102
Figure A-2, SWC ControlServlet Starting Page 106
Figure A-3, SWC System Setup Information Configuration Page 107

List of Tables

Table 3-1, Experiments 43
Table 4-1, Setup Costs of Solaris and NT (ms) 51
Table 4-2, Asymptotic Per-datum Costs on Solaris on 100 Mbps Ethernet (ms) 54
Table 4-3, Asymptotic Per-datum Costs on NT on 100 Mbps Ethernet (ms) 55
Table 4-4, 10 Mbps Ethernet and 28.8 Kbps Modem Connections on NT (ms) 56
Table 4-5, Total Time for 100 Iterations Plus Setup Costs on Solaris (ms) 57
Table 4-6, Total Time for Processing 255-Byte Objects on Solaris (ms) 59
Table 5-1, Communication Costs of Different Paths for a 45-Byte Object 76
Table 6-1, A Scalability Performance Test of the E-SWC Framework 95
Table 6-2, Browser Platforms on Which the Current SWC Framework Runs 95

Chapter 1

Introduction

1.1 Introduction

A serial computer has limited computing power due to the fundamental physical limitation imposed by the speed of light. In order to achieve a higher performance, parallel computing uses an ensemble of processors to solve problems. As the demand for computing power increases, parallel computing is finding its way into mainstream computing. Countless industrial problems, such as large-scale molecular modeling for pharmaceutical research, production line scheduling, and weather forecasting, are often highly computationally demanding. Many of the problems that cannot be solved on sequential machines can now be tackled with parallel computing. Not only cutting edge scientific endeavors benefit from parallel technology; even more conventional computing applications take advantage of it. Parallel computing researchers have successfully pushed the boundaries, in quality and applicability, of parallel computing.

Designing *parallel computers* and developing *distributed systems* have been the two alternative approaches to parallelism. This thesis focuses on one of the cutting edge research areas of the latter—distributed *WebComputing*.

1.2 WebComputing Research

The advent of Internet technologies has advanced distributed computing research by increasing the potential scale of parallelism to thousands of computers. It is known as Internet Computing [Pasquale,96]. This trend is further propelled by Java technologies [SunJava]. A number of Java distributed systems for Internet Computing have been developed since 1996. A subcategory of these projects is known as WebComputing. These projects use Java and Java-enabled Web browsers as the platform for developing distributed computing systems.

A number of examples of WebComputing systems are examined closely in this thesis, including the *Charlotte* project developed at New York University (NYU) [BKKW,96], the *Javelin* project developed at University of California at Santa Barbara [CCINSW,97], and the *Bayanihan* project developed at Massachusetts Institute of Technology (MIT) [Sarmenta,98]. However, WebComputing systems are in their infancy. WebComputing can be expected to advance rapidly, as can Internet technologies, to the next generation [NGI].

1.3 Contributions

This thesis contributes new findings and developments in three areas of WebComputing: a communication protocol evaluation (Chapter 4), a prototype design for Small WebComputing frameworks (SWC framework) (Chapter 5), and a scalability enhanced SWC framework (Chapter 6).

Communication Protocol Evaluation (CPE) studies three Java TCP/IP communication protocols and evaluates these protocols based on their performance and attributes in a WebComputing context.

Small WebComputing Framework (SWC Framework) provides a unique, layered architecture that emphasizes both high-level Application Programming Interface (API) design and robust runtime system implementation.

Enhanced SWC Framework (E-SWC Framework) modifies the design of the SWC framework to support multiple server architectures, therefore enhancing the scalability of the SWC framework.

1.3.1 Communication Protocol Evaluation

Performance is the primary issue that must be addressed by WebComputing research. The success of a WebComputing application depends on the performance of its underlying system. The execution environments and the available technologies determine the architecture of a WebComputing system. How do the available communication protocols perform on a given connection setup? What is the best protocol to choose for WebComputing system design? Which communication model should we choose: traditional simple message passing, distributed object communication,

or thread spawning across networks and Java Virtual Machine boundaries? This thesis presents a study of current Java communication protocols and evaluates the issues that concern a WebComputing system developer. The study shows that protocol selection is just one of many factors that affect communication performance. Additional factors, such as the transmitted data type and size, are also important. This study presents a basic performance guideline for both WebComputing systems and their applications.

1.3.2 SWC Framework Design

A WebComputing framework may be perceived as an intermediate layer between applications (top level) and execution environments (bottom level). It must satisfy the requirements of both the top and the bottom. This thesis introduces a WebComputing system—the SWC framework—based on the Communication Protocol Evaluation (CPE). This system is layered to meet the execution environments' demand for a robust architecture and the applications' requirement for a high-level programming interface.

The SWC system uses a dual perspective—bottom up and top down—to achieve an integrated design.

The first perspective is bottom up. It examines the design from a network system developer's perspective. Network communication delays and system unreliability are the primary concerns. The system must provide reliable services and avoid delays and failures. Therefore, the architecture design is geared to improve performance and reduce system overhead.

The second perspective is top down. It concerns the high-level programming interface for WebComputing application development. The framework must support a

simple and straightforward API. Ideally, the development of a WebComputing application would be as easy as multithreaded application design on SMP machines, free from load-balancing and fault-tolerance complications.

These two perspectives have often been presented as mutually exclusive. In this thesis, they are considered equally important and complementary. We strive to present an integrated framework covering both perspectives.

1.3.3 Scalability Enhanced SWC Framework

The SWC framework design supports single-server architecture. Conventionally, a system process can establish only a limited number of network connections: about 510 in UNIX. In practice, this number is usually much smaller because the performance of a single server degrades quickly as the number of clients increases. This creates a performance bottleneck as the WebComputing system scales. A multi-server system seeks to meet this challenge.

Redesign of the SWC framework to support scalability presents several questions. For example, how will the system accommodate load balancing? Will these changes affect the top-level applications? Is there a way to facilitate the deployment of the scaled-up system? This thesis presents the E-SWC framework, an enhanced SWC system design using a four-level load-balancing solution to address these problems. At the same time, the changes E-SWC introduces are transparent to SWC applications. The SWC API remains intact.

1.4 Outline

This thesis begins with a detailed discussion of Parallel and Distributed Computing (Chapter 2), including trends and challenges facing distributed computing. Chapter 3 introduces a new concept of WebComputing and surveys related research. Chapters 4-6 cover the three major new contributions of this work. Because communication protocol selection is a primary concern for WebComputing system developers, experimental results concerning protocol performance are presented in Chapter 4. Based on these performance evaluations, Chapter 5 presents the design of the SWC framework. The scalable E-SWC framework deployed via Java servlet technology [SunServlet] is introduced in Chapter 6. The thesis concludes with remarks on the performance and design of WebComputing systems in Chapter 7.

Chapter 2

Distributed Computing

2.1 Parallel and Distributed Computing

Parallel computers and distributed systems have been two alternative approaches to parallelism. Many researchers have explored different areas of parallel computer design, such as Symmetric Multiprocessor (SMP) systems [Schimmel,94], cache coherence between multiprocessors [Lilja,93], and thread technology [ZY,98], [LB,96]. However, parallel supercomputers often come with a high price tag [Thompson,96] and are beyond the reach of most computer users. In addition, the scalability of a tightly coupled parallel computer is often limited. For example, a Sun Ultra-Enterprise 4,000 supports up to 14 processors and costs about a half million dollars [SunWeb,99]. One of the world's most powerful supercomputers, ASCI Red at Sandia National Laboratories, contains up to 9,152 Intel processors and costs about $55 million [NetLib,98], [SndNL].

The alternative approach is distributed systems. A distributed system is achieved by interconnecting a number of computers through an existing network technology, such

as a Local Area Network (LAN) or Wide Area Network (WAN). It is much less expensive than supercomputers and accessible to a vast majority of networked computer users. Furthermore, distributed systems allow users to gradually upgrade their networked equipment to keep up with changing technologies. This can be called *system evolution capability*, which is absent or close to absent with supercomputers. With the explosive development of networking, distributed computing has evolved into a promising approach to parallelism. As availability of parallel computers increases, distributed systems are able to integrate them into their distributed resources, using technologies such as threads, to achieve an even higher level of parallelism than could ever be realized by a single supercomputer.

2.2 Obstacles in the Distributed Computing Environment

Although distributed computing has an economic advantage, system evolution capability, and the promise of superior computing power over parallel supercomputers, it faces a number of formidable technological challenges. These challenges can be classified into five categories: network heterogeneity, high communication delays and bandwidth limitation, network unreliability, security concerns, and administrative overhead. To understand these challenges, we need to compare various technical aspects of a distributed system to those of a parallel computer.

2.2.1 Network Heterogeneity

Distributed systems, designed to exploit a collection of networked computers, must first meet the challenge of network heterogeneity. In an SMP machine, a common high-speed

interconnection, such as a bus or a crossbar, connects the symmetric processors and the memory modules. The processors share the same system clock and have equal capacities, resources, software, and communication bandwidth. On the other hand, distributed computing is conducted on networks such as LANs and WANs without any shared resources. Even synchronized clocks are not available. The search for effective algorithms to synchronize networked workstations is an active research field in distributed computing.

Distributed computing faces numerous types of heterogeneity, including different hardware architectures, data formats, file systems, operating systems, compilers, machine loads, and network loads. Distributed applications must be ported from one operating system to another. As a system user or developer, one is often trapped by vendor-specific proprietary programming APIs, incompatible protocols, and unpredictable system performance. The programming overhead in porting existing applications across platforms alone is often insurmountable or forbidding, even for sizable companies. The three well-known operating systems—Microsoft Windows, Macintosh, and UNIX—still operate in their own relatively isolated kingdoms. Within UNIX, different vendors provide different flavors, such as SunOS, Solaris, Linux, IRIX, and AIX systems. Each brand of UNIX has more or less incompatible APIs.

2.2.2 Communication Delay

Another obstacle facing distributed computing is communication delays. On an SMP machine, multiple threads may share memories, and communication delays are very small. In contrast, distributed computing suffers from considerable communication

delays. Not only is shared memory absent, but under certain conditions, even direct communications may be unavailable. For example, on a World Wide Web browser, an untrusted Java applet cannot communicate with any hosts other than the HTTP server from which it is downloaded. Web browser security models impose a communication restriction known as "host-of-origin" on Java applets. This limitation is covered in detail in Chapter 3. Using current technology, the communication delay of a distributed system is a few orders of magnitude greater than the delay on an SMP machine. The bandwidth of a LAN is often in the neighborhood of 1-100 megabits per second (Mbps) compared to 2-4 gigabits per second (Gbps) of a system bus on an SMP machine [SunWeb,99]. Communication delays grow as a distributed system scales up. From low-level hardware architecture design to high-level application programming paradigm, distributed computing is profoundly affected by this limitation.

2.2.3 Unreliability

Distributed computing must address system unreliability. Remote system resources are usually shared with other users and applications. Control over these systems is often incomplete or nonexistent. The remote resources are not for the exclusive use of the distributed application and may become unavailable at any time without notification. Not only can the remote system be unreliable; the network itself may be unreliable. Messages between distributed systems may be delayed or lost due to unpredictable network congestion. A distributed system must be flexible enough to adjust to the underlying dynamic execution environment. This capability is called adaptive parallelism.

2.2.4 Administrative Overhead

Finally, security concerns and administrative overhead are additional challenges associated with distributed computing. Permission for remote access opens doors for the flow of both desired and undesired information. It is a loophole that leaves the system open to potential security breaches. A distributed system must adopt automated security mechanisms to minimize potential risks and administrative overhead.

2.2.5 The Need for Distributed System Architectures

All these obstacles and challenges contribute to the complexity of the design and development of distributed applications. Most of the time, the complexity is overwhelming to application developers and must be addressed by introducing a middle-layer architecture standing between the lower-level network protocols and the upper-level distributed applications. This middle-layer architecture is known as a distributed system. A distributed system provides more advanced communication mechanisms and reduces the design complexity of the upper-level applications.

2.3 Traditional Distributed Systems and Their Limitations

Many distributed system research projects have attempted to overcome network heterogeneity, capture network resources, and make them available for computationally demanding applications. The traditional target platform of these systems is a cluster of UNIX workstations, connected by one or several LANs. These LAN-based distributed systems are built on the underlying UNIX's native socket network API. For example, DP [Arnow,95], PVM [GBDJMS,94], and MPI [SOHWD,96] are such distributed systems.

For the purposes of this thesis, they will be called *traditional distributed systems*. These systems often provide a set of ad hoc routines to create processes on remote hosts, to support message passing between parallel processes, and to facilitate communication and coordination across networks. These routines are typically implemented as library packages for C/C++ or Fortran.

These traditional distributed systems have been used with great success in network-of-workstations (NOW) based environments running UNIX. However, there are drawbacks of these systems:

1. Most of the systems are ad hoc library routines and lack support from the primary programming language. The libraries provide idiosyncratic APIs. Message-passing mechanisms are often deeply embedded in the application programs. Consequently, these programs are bound to the particular distributed system and cannot be ported to other systems.
2. They require possession of a user account on every machine where the distributed programs run. The number of machines that are accessible in this way is often limited. This requirement also limits scalability to a predefined domain of networked workstations that are accessible to the distributed application user. This limitation makes the traditional distributed system design not viable for massively distributed computing on the Internet.
3. Most of these systems are not fully automated. Binaries must be compiled for each targeted operating system, probably using different compilers with different options. This is true even for various UNIXes. If a shared file system is not available or different UNIX systems are used, the compiled

binaries must be loaded manually to each of the workstations. In addition to these handicaps, in some cases, daemon processes must be initialized on each workstation prior to the execution of a distributed application. These impediments impose a large amount of administrative overhead in system setup, coordination, and management. Application development for these systems is often long, tedious, and prone to human error.

4. The traditional distributed system libraries often provide low-level communication and coordination APIs. The learning curve is steep. High-level application logic must be mapped to the low-level message-passing mechanisms. Some systems, such as PVM, require multiple explicit instructions for every message transmission. Consequently, low-level message-passing mechanisms often result in prolonged application-development cycles.

These drawbacks limit the traditional distributed systems to a predefined, closely controlled, relatively homogeneous domain (UNIX); impose significant programming and administrative overhead; make standardization difficult; and severely compromise the scalability and robustness of these systems.

As networks grow, the vast pools of resources available on the Internet become increasingly attractive for large-scale distributed computing. However, as large number of computers and networks become accessible as computing resources, the design and maintenance of distributed systems becomes correspondingly complex. The fundamental

design concepts of traditional distributed systems cannot easily be employed in this milieu.

2.4 Internet Computing

By the end of 1998, more than 43 million computers were connected to the Internet [BPMK,99]. This number will reach one billion by 2005 according to Gordon Casey, an Intel director. Most of these computers are idle most of the time and their CPU cycles are "wasted." It is very desirable to develop a distributed system that is able to utilize these wasted resources for large-scale computationally demanding applications. This is known as *Internet Computing* [Pasquale,96].

One organization that has demonstrated the potential power of Internet Computing is `distributed.net`, a worldwide network computing organization. By collecting and applying volunteer computing cycles on the Internet, `distributed.net` successfully cracked RSA Lab's RC5 56-bit secret-key challenge on October 19, 1997. About $2^{56} = 7.2 \times 10^{16}$ key combinations were computed in this experiment. At the peak of this computation, the distributed application was able to process 7 billion keys per second and involved an aggregate computing power equivalent to 26,000 Pentium computers [Sarmenta,98].

SETI@Home [SETI,99] is another Internet Computing project that has attracted much attention from the scientific world. SETI stands for Search for Extraterrestrial Intelligence. SETI@Home is similar to `distributed.net`. A volunteer may download a screen saver from SETI's Web site and install it on his or her computer system. The screen saver receives data from a SETI server and sends results back. Over

a period of three months, this project collected more than one million participants from about 205 countries. Many other successful Internet Computing stories can be found in [Hayes,98] [DistNet] [SETI,99].

However, harnessing this vast resource pool is not a trivial task. The code breaking conducted by `distributed.net` required tremendous managerial effort in organizing and coordinating the event. Onerous requirements are applied to both resource donors and to distributed application developers. A volunteer must visit the organizers' site, download a precompiled binary matching the volunteer's system, install the package, and initiate the program. This provides little or no protection for the volunteer's system. The distributed program can access the entire system and network resources of the volunteer machine, including data files that possibly contain sensitive information. The project organizer, in turn, must obtain unreserved access and complete trust from every volunteer, even if the organizer does not need all the power on the volunteer's system. The security risk keeps many potential volunteers away. This kind of Internet Computing cannot support commercialized computing resource transactions as described in [Pasquale,96]. Rather, almost all of the successful Internet Computing projects are limited to the noncommercial, academic research domain, and the project designs are transparent.

In addition to the security risks imposed on resource volunteers, there are also formidable obstacles facing application developers. A distributed application must be ported to each targeted platform, such as Macintosh, MS Windows, and the various flavors of UNIX. The programs must be compiled and tested for each platform. These requirements impose tremendous programming efforts on application developers. A

project such as distributed.net's takes months of preparation. Development of these applications is often slow and labor-intensive.

A distributed system that supports Internet Computing must meet two primary challenges. First, it must provide security for volunteers' machines in addition to a user-friendly interface. Second, it must run on a variety of platforms and provide application programmers with high-level uniform APIs with reliable run-time support and an abstract programming model to allow distributed application developers to concentrate on application logic instead of details of communication protocols. In other words, a distributed system for Internet Computing must address such issues as programmability, heterogeneity, portability, security, dynamic execution environment, and scalability. Now these issues can be tackled with Java and Java-enabled Web browser technologies.

Chapter 3

WebComputing

3.1 Introduction

In mid-1995 and 1996, Sun Microsystems Inc. and Netscape Communications Corp. released Java and Java-enabled Web browser technologies. Java is based on more than three decades of research on programming languages. It inherited many advantages from object-oriented programming languages like C++ and Smalltalk, and also eliminated many pitfalls of its predecessors. Java is not only a pure object-oriented programming language, but also a programming environment for network computing on the Internet. Unlike programming languages used by the traditional distributed systems, Java has a rich set of network-programming facilities embedded in the language's core class libraries.

All major Web browsers, such as Netscape Communicator, Internet Explorer, and Hot Java, support Java applet technology and enforce mechanisms that ensure secure execution of untrusted codes. Remote volunteer machines can download and execute

Java applets without exposing their systems to security breaches. The combination of Java and Java-enabled Web browsers supports Web-deliverable Java applets and makes a new, promising, distributed computing platform possible. By deploying Java applets across the Internet, a distributed system may harness underutilized networked computing resources and apply them to large-scale distributed applications. This opens the door for inexpensive large-scale distributed computing that uses Java applets and Java-enabled Web browsers. We call this *WebComputing*.

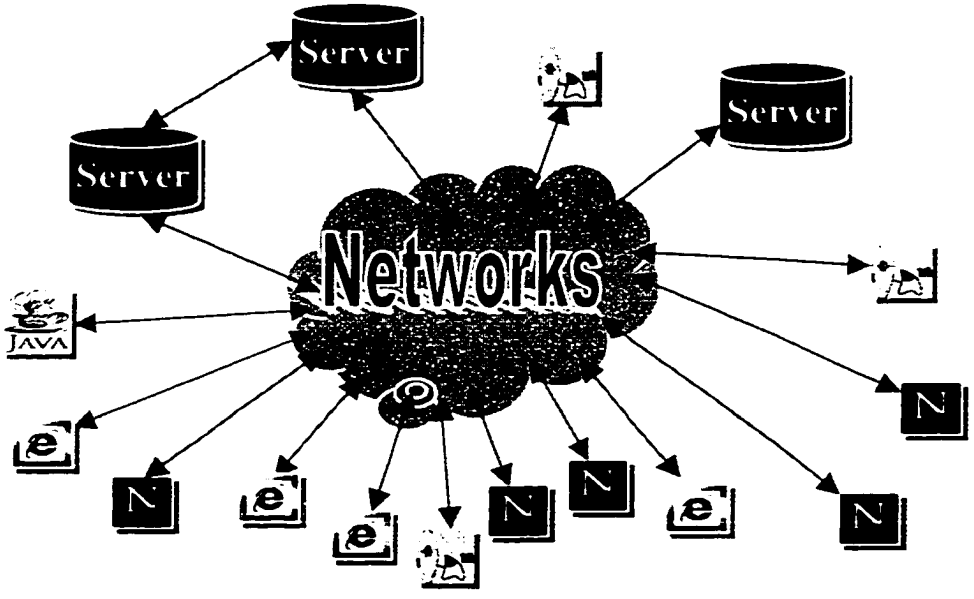


Figure 3-1, A Setting for WebComputing

Figure 3-1 depicts a typical WebComputing setting. The essential idea is that a master server, or collection thereof, in league with a collection of Web servers, coordinates the execution of tasks by applets running in parallel on an ever-changing set

of unreliable, heterogeneous client machines. The promise of WebComputing is the potential for achieving an unprecedented degree of parallelism.

3.2 WebComputing Environment

3.2.1 WebComputing Programming Environments and Their Advantages

WebComputing has many advantages over traditional distributed systems. It overcomes many barriers and problems that cannot be easily solved with a traditional distributed computing system. Its programming environments are primarily based on Java and Java-enabled Web browsers, which differ significantly from what traditional distributed systems offer.

3.2.1.1 Java Virtual Machine (JVM)

A fundamental design concept of Java technology is the Java Virtual Machine (JVM) [SunJVM]. JVM enables cross-platform execution of pre-compiled Java bytecodes. Java applets may be compiled into Java bytecodes and downloaded directly by a Web browser. Thus, JVM presents a uniform interface across operating systems and overcomes the network heterogeneity. For traditional distributed systems, porting an application across various platforms requires formidable effort. In contrast, it is nearly no effort with the JVM.

3.2.1.2 No Account Access Requirement

The traditional distributed systems and systems based on technologies other than Java applets usually require user account access on every participating host to run a distributed

application. Because of the limited number of access accounts a computer user can have, this severely compromises scalability. Systems that use Java applets do not suffer from this limitation. Java applet transmission and remote execution do not require user-account permissions. A Web browser on a volunteer host can download an applet and securely execute it through Sandbox security measures.

3.2.1.3 Built in Security

Java-enabled Web browsers impose security restrictions on any untrusted applet. These restrictions eliminate risks that are inflicted on volunteers' computers by other Internet Computing approaches. Java was designed with security in mind. It provides four layers of security defenses to thwart malicious attacks. These defenses are language attributes, bytecode verification process, "Sandbox Security Model," and digital signatures.

At the first level of defense, a Java program does not have pointers as do C/C++ and many other high-level programming languages. No Java codes have direct access to memory. Overflowing an array, or accessing memory outside the bounds of an array or string, is prohibited. Secondly, the Java run-time environment verifies the bytecodes every time an untrusted class is loaded into the JVM. It ensures that the bytecodes are well-formed and do not contain any illegal instructions. Thirdly, Sandbox Security Models are implemented in all Java-enabled Web browsers. Downloaded applets are treated as untrusted codes and are confined to Sandbox environments provided by the browser. This guarantees secure executions of untrusted codes and limits exposure to risk for the remote host. In addition, Java provides various cryptographic techniques, such as digital signatures, to ensure security as well as functionality. Digitally signed codes are guaranteed to be from the programmer who signs it. Furthermore, Java

provides security for the application developers. Compiled Java bytecodes eliminate the risk of exposing application source codes to the unknown remote hosts.

3.2.1.4 No Requirement for a Shared File System

Traditional distributed systems often require shared file systems to alleviate administrative overhead. On the other hand, data sharing is no longer an arduous task for WebComputing. Application data and programming codes can be transferred at run-time through the HTTP protocol using Universal Resource Locators (URLs). Shared file systems are no longer necessary for running distributed applications.

3.2.1.5 Ubiquitous Programming Platform

The World Wide Web is the unprecedented ubiquitous programming platform that is able to reach almost every corner of the Internet. The potential amount of computing resources exposed to this platform is very large.

3.2.1.6 Friendly User Interface

The Web provides a user-friendly interface. No extraneous technical knowledge is required for computing resource donors. They may easily donate their computing resources by visiting a Java applet embedded Web page designated by a known WebComputing Web site.

3.2.1.7 Less Administrative Overhead

WebComputing requires little administrative effort. Participating hosts may be anywhere on the Internet. Volunteers may simply visit a WebComputing Web site to participate in

an ongoing computation. There is no account requirement for a WebComputing program to access the remote volunteer hosts. As long as the Java-enabled browser and run-time environment are properly set up, little administration is required. Account access, a must for a traditional distributed system, is not necessary in WebComputing.

3.2.1.8 A Large Number of Potential Volunteers

With the elimination of security, account permission, and shared file system problems, any volunteer host on the Internet may participate in the computation and contribute CPU cycles. This breaks through the boundaries imposed on the traditional distributed systems and the potential number of volunteers may be very large.

3.2.2 Challenges to WebComputing

Although WebComputing alleviates problems like heterogeneity and portability, reduces the security risk and administrative overhead, and establishes many advantages over traditional distributed computing systems in terms of scalability and object-oriented programming frameworks, the WebComputing programming environment introduces a number of new challenges and limitations that need to be overcome or circumvented. Slack performance, high communication costs, and unreliable environments are the three most prominent challenges facing WebComputing.

First, the performance of a WebComputing system is crucial to the success of its applications. The performance of this system relies heavily on underlying execution environments and the Java programming language. As Java technology develops, its performance is also under intensive investigation. Now, a number of technologies are available for Java performance optimization. For example, Java Just-In-Time (JIT)

compiler and Java HotSpot performance engine [SunJava] can significantly boost the speed of a Java program. Some literature has reported that the speed of Java is beginning to match the speed of C++ [Mangione,98]. In addition, Java Archive (JAR) technology is available to speed up data and code transmission across networks.

Second, communication in a WebComputing environment is much more expensive than communication in the traditional distributed computing environment. Network latencies get worse as the scale of the distributed system grows. Traditional distributed systems are often confined to one or several interconnected LANs. Network latencies in this kind of environment are usually no more than a few milliseconds (quantified using UNIX ping command). In contrast, WebComputing systems target networks on the scale of multiple LANs, Metropolitan Area Networks (MANs), and WANs. This new environment introduces many delaying factors that are not present in a single or several interconnected LANs. For example, additional routing, intermediate storing and forwarding hosts may impose and exacerbate delays. Network latencies in this environment are ten to several hundred times greater than those in traditional distributed computing environments. Furthermore, according to TCP/IP protocol, intermediate hosts may drop data if the network is congested. When losses are detected, the end hosts must retransmit the data and the delays become unpredictable. Besides the network delays, Java-enabled Web browsers enforce Sandbox security restrictions on untrusted Java applets. One of the restrictions, known as "host of origin," prohibits a Java applet from communicating to any hosts other than the one from which it was downloaded. This severely limits the communication performance of WebComputing systems.

Third, the Internet imposes a much more unreliable programming environment than the LAN environments used by most traditional distributed systems. The scale of WebComputing is beyond the control of the distributed application developer. Volunteer hosts may join an ongoing computation and contribute resources at any point and leave abruptly without warning. The extent of uncertainty is much greater in the Web environment. A WebComputing system must overcome these obstacles and provide reliable service to its upper-level applications. Adaptive parallelism is one of the most important attributes associated with WebComputing systems.

Therefore, a WebComputing system must address two important issues to meet the challenges of this new platform. First, a WebComputing system must provide a flexible programming model, as well as run-time supports, to facilitate communications. This programming model must be able to integrate incoming network resources when volunteer systems become available. At the same time, these communication facilities must provide high-level programming interfaces and recover effectively from communication failures. Second, WebComputing systems must address communication performance. For example, communications between applets must be routed through a server. This may cause server bottlenecks and affect the scalability and robustness of a WebComputing system. The WebComputing system must be designed accordingly and avoid possible performance bottlenecks.

3.3 WebComputing Related Works

Since Java's inception, many distributed computing research projects have been designed to implement distributed systems using Java technologies. These research projects may

be classified into two categories based on whether the system uses the World Wide Web as the distributed platform or not. Projects from the first category are developed solely based on Java applications running in LAN environments. Some of these projects are direct extensions of traditional distributed systems. For example, JavaPVM [Thurman,96] and mpiJava [BCHL,98] are such systems. On the other hand, some of the projects extend the concept of a thread, which was initially developed for SMP machines, to distributed computing by spawning "threads" on remote hosts. Projects from the second category are WebComputing systems. These projects use Java-enabled Web browsers as distributed platforms and deploy Java applets as the primary computing units across networks. Section 3.3.1 reviews projects from the first category. Section 3.3.2 examines several WebComputing projects.

3.3.1 WebComputing Predecessors

JavaPVM library is a direct extension of PVM [GBDJMS,94]. It provides a Java programming interface using Java Native Interface (JNI) [SunJNI] as wrappers around existing standard PVM routines. This approach has two advantages. First, it is built on top of an existing PVM distributed system and is interoperable with legacy PVM programs written in C/C++ or Fortran. Second, it takes advantage of the fact that the current PVM software has better performance than Java [YC,98]. However, this approach to distributed system design originated from traditional distributed systems along with their programming paradigm. It fails to utilize the advantages of Java. Projects taking this design approach are still confined by the limitations imposed on traditional distributed systems. On the other hand, JPVM [Ferrari,97], which is another

extension of PVM, is implemented entirely in Java. JPVM took the advantages of portability and programming mechanisms offered by Java. However, it again originated from traditional distributed systems and inherited a message-passing application programming interface as defined by standard PVM, which offers low-level communication mechanisms using C-like primitive data-type message passing between distributed processes. The low-level programming interfaces are unnatural to object-oriented programming languages like Java. Application programmers using these interfaces are reduced to C-like message-passing APIs, rather than high-level communications between distributed objects. In addition, these projects do not use the World Wide Web, which is ubiquitous on the Internet as the computing platform. Similar to JavaPVM and JPVM, many projects in this category, such as mpiJava, extend standard MPI distributed message-passing interfaces using Java. They also suffer from the drawbacks described above.

Different from JavaPVM, mpiJava, and JPVM, which originated from the message-passing programming paradigm used by traditional distributed systems, ATLAS [BBB,96] is designed to execute parallel multithreaded programs on networked computing resources and is implemented in Java. The architecture of ATLAS consists of three types of entities: clients, managers, and computer servers. The run-time system of ATLAS is directly involved in client-application computing by spawning children and successors (procedures and threads) based on the Cilk's programming model. Cilk [Cilk] is a C-based, parallel-multithreaded programming language together with a runtime system that supports thread scheduling. The run-time library of ATLAS is responsible for hierarchical *work stealing*, thread management, and marshalling objects for

communication across networks, which is also based on a hierarchical tree structure. Local system administrators manage each computer server (computer resource provider) that runs as a daemon. The daemon process runs continuously, either working on some application threads or communicating with its manager or siblings for stealing work. A client running a Java application (not an applet) contacts the local manager to allocate computer servers, then the client connects the server to execute the application. While the server is executing the application, idle servers join the computation by connecting to the manager. Each server maintains a downward stack, which actually is a double-ended queue, to store temporary discrete work. By popping from the bottom for local thread execution and popping from the top for work stealing, ATLAS servers execute locally in depth-first order and steal work in breadth-first order. This effectively exploits the networked resources in a hierarchical fashion, which also corresponds to the Internet hierarchy. However, ATLAS system architecture is structured in a hierarchical manner and is only suitable for running applications that have task graphs that can be mapped to a tree-based structure, such as computation of Fibonacci numbers and ray-tracing applications. In addition, ATLAS does not use Java-enabled Web browsers, which means that its scale is limited to a NOW environment.

3.3.2 WebComputing Projects

Research on WebComputing is in its infancy, and a standard framework is needed. A number of projects, including Charlotte [BKKW,96] at New York University, SuperWeb [AISS,97], Javelin [CCINSW,97] at the University of California, Santa Barbara, and Bayanihan [Sarmenta,98] at the Massachusetts Institute of Technology, have explored the

WebComputing platform and presented their blueprints for architecture design. The following sub-sections present these projects in detail along with SAM [CC,99], a programming model for a common WebComputing setup without a full-blown WebComputing architecture. We will examine these projects from two perspectives: 1) programming model and interface design, and 2) distributed system architecture design. At the same time, we will examine additional design issues, such as fault-tolerance and load-balancing schemes.

3.3.2.1 Charlotte Project

The conceptual programming environment of Charlotte [BKKW,96] consists of two parts: a run-time system and a virtual machine model. The virtual machine model provides an API which supports a reliable shared memory virtual machine for a Charlotte application program and shields this program from underlying hostile execution environments. The virtual machine model is not only a programming interface, but also a high-level logical programming model for distributed applications. Like ATLAS, Charlotte extends the thread-programming model into distributed systems. It establishes a layer of abstraction that helps application developers modeling the actual problems. It facilitates partitioning the problem into finer tasks and routines.

The execution of a Charlotte application consists of alternating sequential and parallel steps. A sequential step is a set of standard sequential Java statements; while a parallel step consists of one or more routines that are bracketed by `parBegin()` and `parEnd()` statements, which are Java function calls. These functions initialize the computing at the beginning and synchronize the computing at the end of each parallel step. A Charlotte routine is similar to defining a user thread class, which must extend

`java.lang.Thread` class and implement a `run()` method. It is a remote thread, which is an instance of a class that extends a Charlotte class called `Droutine` and implements a method called `drun()`. Each `Droutine` object is added to the distributed system by calling `addDroutine()`. Figure 3-2 shows a sample Charlotte routine.

```
public class myRoutine extends Droutine {
     public myRoutine() {...}
     public void drun(int numTasks, int id) {
          // compute...
     }
     public void run() {
          ...
          parBegin();
          addDroutine(this, Size);
          parEnd();
          ...
     }
}
```

Figure 3-2, Sample Charlotte Routine

The execution of a Charlotte `Droutine` object is analogous to the execution of a standard Java thread object. Instead of spawning threads on the local JVM, a `Droutine` is created remotely by the Charlotte system and the `drun()` method of the instance is executed.

The Charlotte run-time system establishes its WebComputing architecture and realizes the virtual machine model on a set of unpredictable, dynamically changing, and faulty machines. The run-time system provides three services: *scheduling*, *memory*, and *computing*. These services are designated as one *manager* and a number of *workers*. The manager provides a management service, which consists of the scheduling, memory, and sequential parts of the computing service. The workers carry out the computing service, which performs the computationally intensive parallel routines. The single manager and the set of workers together implement the virtual machine model. Figure 3-3 shows Charlotte's virtual machine model and its run-time system.

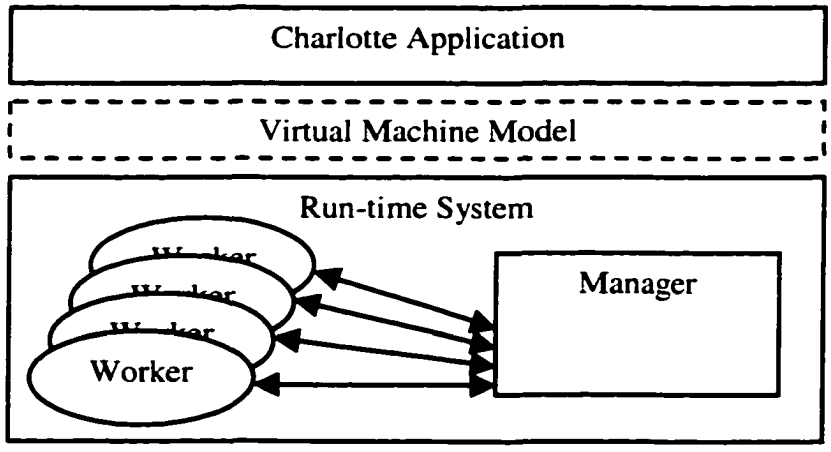


Figure 3-3, Charlotte's Virtual Machine Model and Run-time System

To support the remote-thread programming model, Charlotte provides a distributed-shared memory (DSM) that does not depend on either the underlying operating system, which might impose portability and security problems, or compilers, which might tie the system to a particular language. Charlotte's DSM is implemented at the Java language data-type level. Data, in Charlotte, are logically partitioned into private and shared segments. Private data are local; while shared data are distributed across networks. There is a corresponding Charlotte data type that implements a distributed version for every basic Java data type. Basically, each distributed data object contains a state variable other than the object value. A state of a distributed data object can be `readable`, `not_valid`, or `dirty`. This state information determines the data access mechanism for correct operation. As a result, distributed objects are accessed and modified through explicit function calls like `get()` and `set()`. This leads the Charlotte run-time system to use a concurrent read and exclusive write (CR&EW) mechanism to maintain consistency and coherence of its distributed data. Atomic data updates are performed when routines are finished at each parallel step.

The Charlotte run-time system is geared toward supporting its virtual machine model, namely the remote-thread programming model, which requires DSM support. This requirement complicates system design. Additional memory coherence mechanisms must be incorporated into the run-time system. Each distributed object has three possible states and must be uniquely identifiable system-wide. Based on the current state of a distributed object, each basic memory access operation, such as read and write, could result in data exchange across networks between the worker and the manager. Furthermore, Charlotte's DSM is implemented at the Java application-programming level

to acquire portability and overcome network heterogeneity. This could be very expensive in terms of system performance, as acknowledged by the authors. In addition, the scalability of this system could become an issue. The single manager must support task scheduling and distributed-memory sharing, and provide the sequential part of the computing service. The manager is likely to become overloaded and introduce a performance bottleneck.

It is worth noting that Charlotte introduces an *eager scheduling* algorithm for the run-time system manager assigning tasks to its workers. Under this algorithm, a task is assigned repeatedly to the returning idle workers until it is executed to completion by at least one worker. As long as at least one worker does not fail, this algorithm guarantees completion of the program and provides limited load balancing and fault tolerance. This scheduling algorithm is widely adopted in later WebComputing system designs, such as Javelin, Bayanihan and the SWC framework.

3.3.2.2 SuperWeb

SuperWeb [AISS,97] addresses an array of research issues that must be incorporated into future WebComputing system designs. These include WebComputing economic models, system security for both volunteer hosts and application developers (clients), guarantee of correct execution, and communication latency management. SuperWeb presents a WebComputing blueprint and defines a different approach to WebComputing infrastructure design than the Charlotte project. Instead of approaching the design from an application programmer's point of view in a top-down manner as Charlotte does, SuperWeb is designed from a system-architecture point of view and is built bottom-up.

Figure 3-4 shows the blueprint introduced by SuperWeb. The SuperWeb architecture consists of three types of entities: *hosts*, *clients*, and *brokers*. A broker maps hosts (computing resource donors) to clients (computing resource consumers). A broker consists of three modules: interface, scheduler, and accounting module. Both the clients and the hosts register their WebComputing applications or join an application computation by visiting Web sites that are set up by the brokers' interface module. Then the broker does necessary bookkeeping and schedules available resources for the clients with its accounting module and the scheduler module respectively. The Javelin [CCINSW,97] project, which is introduced in the next section, is an implementation of SuperWeb's blueprint.

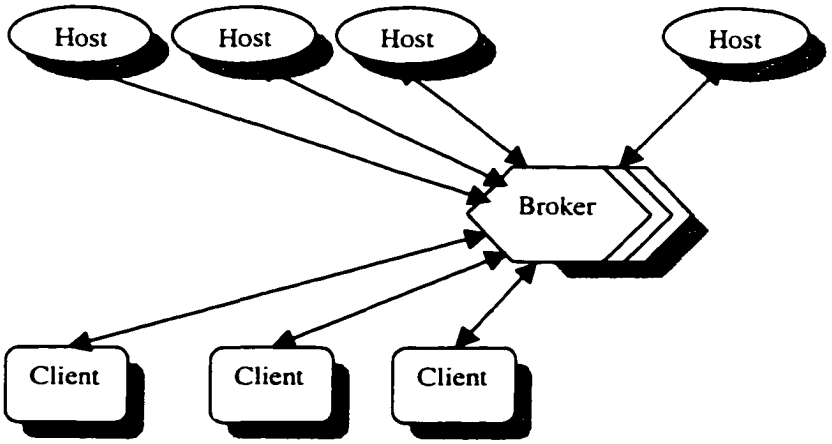


Figure 3-4, SuperWeb's WebComputing Blueprint

3.3.2.3 Javelin Project

Javelin [CCINSW,97] presents a WebComputing prototype based on SuperWeb's blueprint. In addition to the client, host, and broker entities utilized by SuperWeb, Javelin adds a *server* entity. The four entities, described in details, are:

Host. In Javelin, hosts are computing resource volunteer donors. They register their resources by visiting a broker's Web site and activating a broker applet downloaded from this Web site using a Java-enabled Web browser. This downloaded broker applet spawns a small daemon using a thread on the host and awaits tasks to be assigned by the broker. A task in Javelin is represented by an Internet Universal/Uniform Resource Locator (URL), which points to a client Java applet embedded in a Web page on a server that is set up by a client. Upon receipt of a task, the daemon instructs the underlying Web browser to contact the server, upload the client applet from the server, and execute the client applet. This operation is achieved through the `showDocument(URL)` method defined in the `AppletContext` class in the `java.applet` package.

Client. A client is a computing resource consumer. A client application programmer must code distributed tasks as Java applets and embed these applets in HTML files. If a client does not have an HTTP server program running on the client machine, the application developer must upload these HTML files along with all Java applet class files onto an HTTP server (see the following section). The client application developer may then register the distributed tasks by visiting a broker's Web site and submitting the URLs of the client tasks to the broker.

Server. A server is an HTTP server that is accessible to a client. The client uploads its distributed tasks onto the server as Java applet class files and HTML files. Subsequently, this server assigns these tasks to hosts as discussed above. After the

distributed application starts execution, this server serves as a whiteboard and facilitates communications between applets running on different hosts executing the same application. Communications between applets are accomplished in the manner of messages stored and retrieved on this server. Therefore, load balancing between applets could be expensive.

Broker. A broker is built on current Internet infrastructures, such as Common Gateway Interface (CGI) scripts. A broker consists of three modules: interface, scheduler, and accounting. They function as described in SuperWeb [AISS,97]. A broker maintains a pool of hosts and a queue of client tasks. It schedules tasks to be run on hosts by sending tasks to the applet daemon downloaded by a host as discussed above. At the same time, the broker does all necessary bookkeeping. The WebComputing architecture of Javelin is depicted in Figure 3-5.

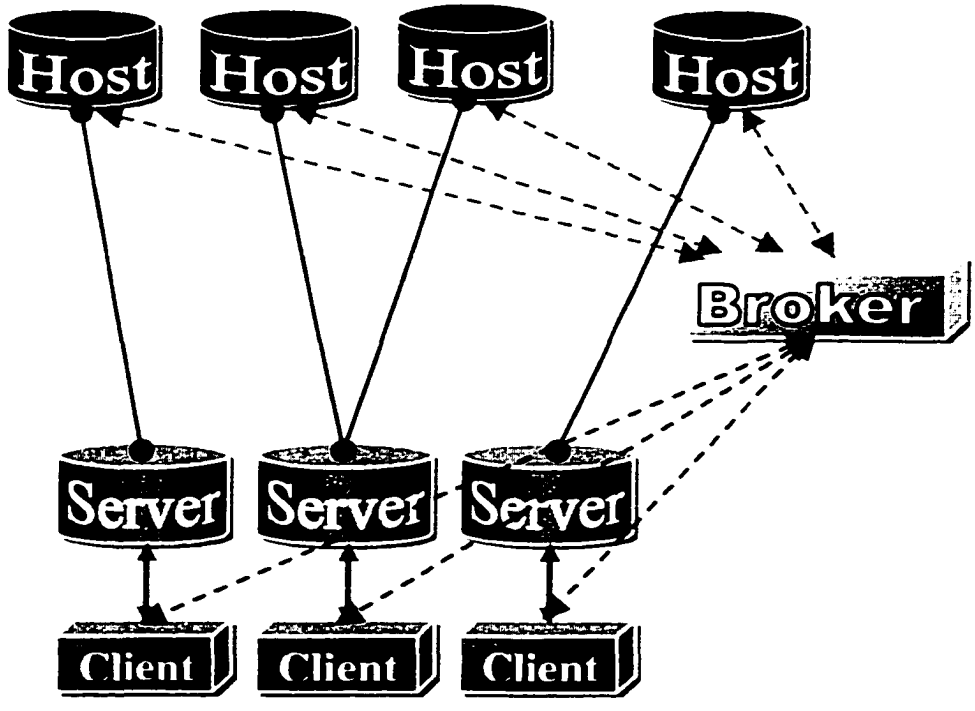


Figure 3-5, Javelin WebComputing Architecture

A broker in the working prototype introduced by Javelin does not treat hosts as a pool of resources, nor does it assume the role of a resource manager and actively participate in a client program execution as a Charlotte manager does. Instead, it plays the role of a "broker" that merely responds to requests and maps each transaction request between clients and hosts. Unlike the Charlotte project, which provides WebComputing application programmers with a high-level programming interface, a client programmer of SuperWeb or Javelin faces all the programming complexity imposed by the underlying dynamic execution environment. For example, client programs must incorporate fault-tolerance schemes to mask out faulty hosts and ensure communication between different programming modules that run on different hosts. To alleviate this formidable task for WebComputing application developers, Javelin provides a run-time library to facilitate communication and coordination between the hosts.

Due to Web browsers' Sandbox security restrictions, the applets running on hosts can not communicate directly with each other. Communication between the applets must be routed through the server from which they were downloaded. The run-time library is implemented in a Java class, which needs to be instantiated on every host. This class provides two levels of APIs. The lower-level API is implemented using Java TCP, UDP, RMI, or CORBA network packages. It provides communication primitives, such as send and receive. In this level, fault-tolerance and load-balancing schemes are not provided. Instead, a client programmer must implement corresponding schemes for their applications. On the other hand, the higher-level API provides functions like I_WANT_WORK and HERE_IS_WORK. Each library instance contains a deque

(double-ended queue). A host pushes tasks that can be done remotely onto its deque while it has work to do; it pops tasks from the deque when it finishes its work and becomes idle. If the deque is empty, the run-time facility collects work from other hosts; load balancing is achieved in this manner.

Although the Javelin architecture provides a set of library functions to facilitate communication coding between different modules of the client programs, it fails to provide a cohesive API or framework for WebComputing applications. A high-level programming model and logic are absent in this research; rather, the services are built on scattered currently available tools, such as HTML coding, CGI scripting, and ad hoc Java library components that need to be instantiated on each applet. Consequently, Javelin does not have a cohesive API. Instead, it depends on whatever tools are available.

3.3.2.4 Bayanihan

Bayanihan [Sarmenta,98] is a WebComputing project developed at MIT. It presents a high level of abstraction using object-oriented approaches for its programming model design. Different from Charlotte, whose architecture is geared towards the virtual machine interface that shields application programmers from the underlying dynamic execution environment, Bayanihan presents a prototype for WebComputing that is highly modularized. Most of the building blocks are accessible to application programmers.

The framework provides a set of generic components that contains engine, manager, and data pool objects. Each generic component may be replaced or redefined by application-specific components by extending the generic base classes. Additional components, such as GUI, are also included in the prototype. The framework supports a master-worker parallel-programming model. However, this programming model is

embedded in the underlying system implementation instead of being exported to the upper-level WebComputing applications. Under this API, application-specific modules are plugged into the framework based on predefined system interfaces. Logic connection and data flow between the user-defined modules are not necessarily required. New parallel tasks are created exclusively by the master program, while worker programs produce results only. The framework-defined API is not completely coherent and imposes limitations on some aspects of application programming. In addition, Bayanihan uses HORB [Hirano,97], a distributed-object package similar to Sun's RMI, as the framework's underlying communication protocol. HORB is a research project developed in Japan and is not necessarily compatible with objects serialized with Sun's JDK [HYI,98]. The Framework design of Bayanihan is shown in Figure 3-6.

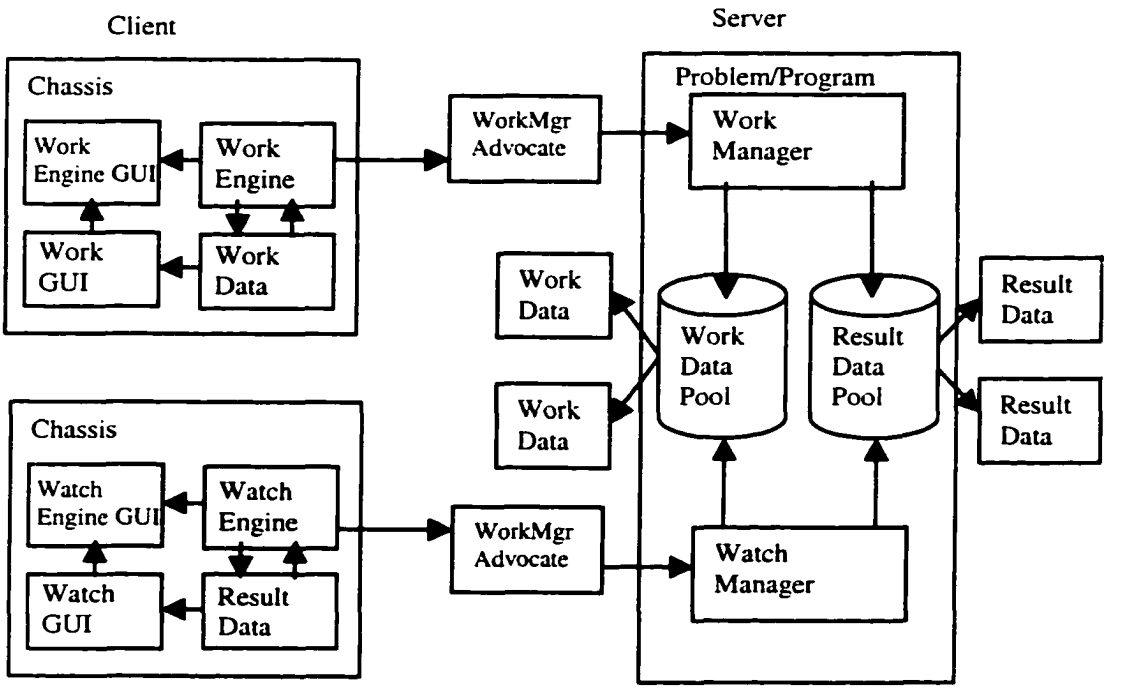


Figure 3-6, Bayanihan Framework

3.3.2.5 SAM

SAM [CC,99] is a programming model proposed for distributing the middle-tier of a three-tier Web-based application. It is not a WebComputing infrastructure like Charlotte or Javelin, which are designed for generic scientific computing purposes. SAM's programming model is designed to split the workload of a heavily loaded Web server into small pieces and distribute these pieces to remote Java applets. Although SAM was created to enhance Web server processing capacity and not for generic parallel computing, its programming model is highly relevant to our WebComputing discussion.

SAM stands for Split-And-Merge. As its name implies, SAM proposes a programming model as listed in Figure 3-7 for distributed computing. This programming model targets a typical Web-based application. These applications often consist of three tiers: the *presentation* tier, which presents results on a client browser; the *computation* tier, which does the core computation; and the *data source* tier, where the computation retrieves and stores data.

```
public interface SAM {
      public Object compute();
      public Object merge(Object[] objs);
      public SAM[] split() throws
                              NotSplittableException;
}
```

Figure 3-7, SAM's Programming Model

Figure 3-8 shows a three-tier Web-based application. The presentation tier often runs as a Web browser on the client machine and issues requests to the computation server. The computation server retrieves data from the data source, which often runs as a DataBase Management System (DBMS). SAM's programming model is designed to partition the computation into tasks and redistribute these tasks to computation workers, which run as applets. The partial results are collected from the workers and the synthesized results are sent to the presentation tier.

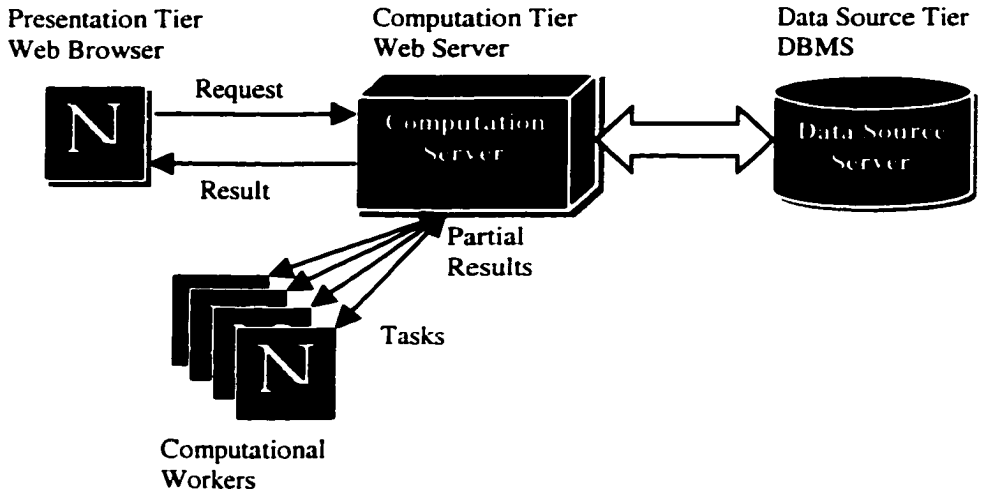


Figure 3-8, SAM's Target Platform

SAM defines a customizable task-partition policy using divide-and-conquer. The concept is a clean, recursive, distributed model. However, it is very restrictive in developing real-life applications. First, SAM's architecture needs support for servlet programming at the computation tier. The programming interface of the servlet is Web-presentation oriented. A servlet execution thread is event-driven and a servlet program

does not have direct control over the thread. Second, this model imposes a self-contained, recursive, distributed task construction. This is very restrictive for most distributed applications. The mapping of application logic to this programming model requires sophisticated programming experience. In addition, the restrictive servlet programming environment renders this approach to WebComputing difficult to follow.

3.3.3 Experiments

The performance experiments presented by the above four WebComputing projects differ in depth. Charlotte [BKKW,96] evaluates its architecture by computing a 3D Ising model, which is a scientific application from statistical physics that involves an exponential number of independent tasks with very little data movement. The experiments were conducted on 10 Sun Sparc-5 UNIX workstations connected by a 10 Mbps Ethernet. The authors [CC,99] conducted three experiments to 1) study performance and overhead; 2) examine the utilization of slow machines in a computation; and 3) analyze how well the system integrates machines into an ongoing computation and how efficiently failures can be masked using the eager scheduling algorithm. The experiment shows that up to 10 worker machines can achieve high efficiency in this application.

Javelin [CCINSW,97] presents three experiments: micro benchmarks, Ray tracing, and a Mersenne Prime application. Micro-benchmark experiments showed that the cost of downloading a Java applet and starting a thread on a JVM is significant compared to message passing after the establishment of an applet. The authors also conclude, as expected, that message passing using Java TCP and UDP sockets APIs is

slow compared to dedicated parallel machines. In addition, the authors report that only computation-intensive parallel applications benefit from the proposed infrastructure because large networks of heterogeneous machines cause unpredictable communication delays. Javelin is suited to coarse-grained applications with a high computation-to-communication ratio. In general, WebComputing is best for non-communication-intensive applications, like prime number factorization, Monte-Carlo, and coarse-grained simulations. The applications used in Javelin's experiments are characteristic. Both Ray tracing and Mersenne Prime problems may be divided into independent tasks and require little communication.

The experiments and conclusions of Charlotte, Javelin, and the preliminary results provided by the Bayanihan project may be found in Table 3-1.

An ideal problem suited for WebComputing should share some common characteristics with the problems listed in Table 3-1.

First, the problem should be easily partitioned into a set of initial tasks. The size of the initial task pool should be easily manageable: not too large, for coarse-grained purposes, nor too small, for load-balancing purposes.

Second, the parallel tasks should be independent. Inter-task communication and coordination must be kept to a minimum because of the high cost of network communication.

Third, the tasks should be computationally intensive. The gain in collected computing power must be large enough to offset the communication cost.

Table 3-1, Experiments

	Facilities	Experiment	Measurements	Comments
Charlotte	10 Sparc-5s, 10 Mbps Ethernet	3D Ising model	1. Performance and Overhead	The program achieves 93% efficiency with 10 machines.
			2. Machines utilization	Charlotte load-balancing technique, which is eager scheduling, is effective.
			3. System integration and failure masking	Charlotte performs well in a dynamic situation and is able to recover from failure effectively.
Javelin	Pentium PCs, 2 Sparc-5s, 5 Ultra-Sparcs, running Solaris 10/100 Mbps Ethernet 64 Sparc-10 nodes of Meiko CS-2	Micro benchmarks	Overhead for initializing applet and basic send cost and receive cost	Only computationally intensive parallel applications benefit from Javelin.
		Ray-tracing	Performance and dynamic behavior of the infrastructure	Close to optimal task distribution using *work stealing* with optimistic propagation and exponential back off.
		Mersenne Prime application		In general, coarse-grained applications with a high computation-to-communication ratio are well suited to Javelin. Increasing application granularity enhances the performance. Java with JIT is about 50% slower than the C version for Mersenne Prime application.
Bayanihan	5 identical 200MHz dual Pentium machines running windows NT 4.0	Factoring Java long integer		The performance of the Java JIT compiler in Netscape is comparable to native C code.

3.4 WebComputing Research

WebComputing is a novel platform for Internet Computing. It provides a means to harness vast computing power from the Internet. A WebComputing system can easily comprise thousands of distributed computing units. WebComputing's potential computational power may allow researchers to tackle problems whose complexity and scale render them impractical or unfeasible using previous technologies. This has created a sense of excitement analogous to that which characterized the early days of parallel computing. However, many research issues regarding WebComputing have only begun to be addressed. Answers to the following questions are needed.

1) How does WebComputing communication currently perform?
2) How does communication performance affect the WebComputing architecture design and the programming models?
3) What performance enhancements can be made?

Chapters 4-6 cover the three primary areas of WebComputing research pertaining to each of these questions.

Chapter 4

WebComputing Communication Protocol Evaluation

4.1 Introduction

4.1.1 WebComputing Communication

Communication performance is the primary challenge that must be addressed in the WebComputing environment. Given the latency and bandwidth limitations of the Internet, applications that can be parallelized with a minimum of inter-task communication are ideal for WebComputing. Nevertheless, some communications are necessary and inevitable. Logically, we may classify the communication of a WebComputing system into three categories:

- Application-to-Application
- Applet-to-Server Application
- Applet-to-Applet

In a WebComputing context, a server or a collection of servers is used to coordinate and distribute a computation. A server process often runs as a Java

application and coexists with an HTTP daemon on the same host, while an applet runs in a Java-enabled Web browser. In practice, because of Sandbox security restrictions, a Java applet cannot communicate with any host other than the one from which it was downloaded. Only the first two communication categories (Application-to-Application and Applet-to-Server Application) are certain to be realizable with direct communication. The third category (Applet-to-Applet) requires indirect routing through the server. As a result, Applet-to-Applet communication can only be realized with a combination of multiple Applet-to-Server Application communications. This thesis concentrates on the performance of the first two categories of communications.

4.1.2 Protocol Evaluation

The efficiency of a WebComputing application depends on many elements beyond the control of the implementer of a WebComputing system or framework. Among these are the appropriateness of the application to the platform, the implementation of the Java Virtual Machines (JVM) that support the applets, and the network conditions that affect latency and bandwidth. This thesis focuses on an issue that is in the control of such an implementer—the choices of network protocols in realizing Application-to-Application and Applet-to-Server Application communication. The Java core libraries developed by Sun Microsystems, Inc. provide several protocol choices for Internet TCP/IP networks:

- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)
- Java Remote Method Invocation (RMI) [SunRMI]

Because of the extremely dynamic execution environment, WebComputing communication protocols should be chosen based on the protocols' attributes and their

ability to adapt to this environment. Although the choices may be influenced by concerns other than, or in addition to, communication efficiency (for example, the questionable scalability of a connection-oriented approach), it is efficiency that we consider here.

The specific Java Development Kit (JDK) implementation that we used is the Production Releases 1.1 for both Sun Solaris and Microsoft Windows NT operating systems developed at Sun Microsystems, Inc. These releases contain Just-In-Time (JIT) compilers that are able to deliver high performance and are widely available for free. When Java-enabled Web browsers were required for testing Applet-to-Server Application communication, the default Java run-time environments were used. The rationale for using default setups is that a WebComputing system should not impose additional requirements on volunteers other than visiting a designated Web page. Additional requirements may diminish the willingness of a volunteer to participate. The primary testing platform for these experiments was a number of Solaris workstations and PCs connected by a 100 Mbps Ethernet. In addition, this thesis also explores a number of network resources that we had access to and collected experimental results. These include communications on a 10 Mbps Ethernet connection and a 28.8 Kbps modem connection.

To determine the communication performance of a WebComputing system, we needed to examine not only the low-level network connections, but also the upper-level application data transmitted. Different communication protocols may perform differently when handling various data sizes and types. For example, in the case of passing message objects through networks, these objects must be serialized before they can be sent out; object serialization could be a significant performance factor and needed to be quantified.

Consequently, the design choice of communication protocols and data representations in WebComputing applications may directly affect the performance of these systems.

An array of experiments was designed to evaluate the three protocols. This chapter reports the empirical study of their performance based on currently available Java network packages. The results may provide a guideline for future system design and enhancement, and WebComputing research. Even the slightest enhancement of communication performance at the system level can improve the performance of the applications on a grand scale. This study may also provide intelligible information for upper-level WebComputing application developments.

4.2 Experiments

4.2.1 Experiment Design

Our goal was to evaluate the performance of Java network communication facilities, as they would be experienced in a WebComputing context. This performance depends on Java's end-to-end communication costs, which include the time spent in preparing data for transmittal and reassembling transmitted data on the remote host. This in turn is affected by the particular data that is transmitted. An appropriate measurement in this regard is a ping-pong-like test: data of a certain type is transmitted from one process to another, processed negligibly, and returned to the original process.

The ping-pong tests were run, with 10, 100, 1,000, and 10,000 iterations. This enabled us to determine the asymptotic cost of per datum transmission.

To make these measurements, a number of experiments were conducted, comparing the performance of UDP, TCP, and RMI (as provided by Sun's JDK1.1) in communicating various kinds of data values and object types:

1. Two Java primitive data types: `int` and `double`
2. Byte arrays of five sizes: 16 bytes, 256 bytes, 1K bytes, 4K bytes, and 64K bytes
3. One Java library object: a 13-character String object
4. Three user-defined objects: 42 bytes of internal state, 255 bytes of internal state, and 1K bytes of internal state

Performance differences were determined in three dimensions: protocol selection, data type, and data size.

The experiments were also conducted on different platforms:

1. Two Ultra-Sparc-10 machines running Solaris 7 connected by a 100 Mbps Ethernet.
2. Two Gateway 500MHz PCs running Windows NT 4 connected by a 100 Mbps Ethernet.
3. Two Gateway 500MHz PCs running Windows NT 4 connected by a 10 Mbps Ethernet.
4. Two Gateway 500MHz PCs running Windows NT 4 connected by a 28.8 Kbps Modem.

These experiments showed performance differences in two additional dimensions: operating system and network-connection speed.

4.2.2 Data Conversion

Before the data were transmitted using the UDP protocol, they were converted to byte arrays using Java's `DataOutputStream` class for primitive data types, `String.getBytes()` method for `Strings`, and `ObjectOutputStream` class for objects. The output streams were composed with a `ByteArrayOutputStream` class to extract byte array data representations. The TCP protocol provides stream connections. Data were transmitted using corresponding stream classes from the `java.io` package. Byte arrays were transmitted using the `read()/write()` methods from the `InputStream/OutputStream` classes that were obtained from the TCP `Socket` class. RMI provided automatic data conversions.

Along with the above experiments, we measured the setup costs of the three protocols. Because of the ephemeral nature of the task-processing applets in WebComputing clients, these setup costs are relevant.

To measure the setup costs, a program was created for each protocol. For UDP, the `DatagramSocket` setup costs were measured. For TCP, the costs for `ServerSocket` setup and the client-side `Socket` setup were measured. For RMI, the server-side setup cost included the time for local registry establishment and the time for binding the remote object. For the RMI client-side setup cost, the time for remote object lookup was measured.

4.2.3 Experiment Setup

The measurements were carried out using JDK's Production Release version 1.1.7 and its default Just-In-Time (JIT) compiler on both Solaris and Windows NT. For Solaris, two

300 MHz Sun Ultra-Sparc-10 workstations with 128 MB of RAM each, running Solaris 7 operating systems, were used. For NT, two 500MHz Gateway PCs with 128 MB of RAM each, running Windows NT server 4.0, were used. One Solaris and one NT machine were employed as servers corresponding to their clients of the same type. These machines were connected with a 100 Mbps Ethernet or a 10 Mbps Ethernet as detailed in each experiment. The latencies on both networks were below 1 millisecond (ms) (tested with "ping" facility). Netscape Communicator 4 [Netscape] with a 1.1.5 Java run-time environment was employed when testing applets.

4.3 Results

Application-to-Application network communication performances were examined first. These results reflect the core Java networking performance.

4.3.1 Connection Setup Costs

The setup costs on Solaris and NT are shown in Table 4-1. The UDP setup was the most efficient among the three protocols. Based on server-side setup costs, it shows that TCP was about 10-30% more expensive than UDP, and RMI costs were about 15-22 times higher than the setup costs of UDP.

Table 4-1, Setup Costs of Solaris and NT (ms)

	SOLARIS			NT		
	UDP	TCP	RMI	UDP	TCP	RMI
Server Setup	33	43	727	29	32	572
Client Setup	32	50	626	30	45	463

It is worth noting that improper DNS server configuration could result in a dramatic increase in connection setup costs. NT setup costs are very sensitive to DNS system configuration compared to UNIX systems, which often provide multiple redundancies for network information management and are often maintained by professional system administrators. In addition to DNS, a standard UNIX system provides (1) network host files and (2) NIS/NIS+ services. These redundancies ensure reliable network setups. On the other hand, an NT system is often used as a personal workstation and lacks centralized network administration. It is more prone to system mis-configuration, especially when centralized Dynamic Host Configuration Protocol (DHCP) and Windows Internet Name Service (WINS) are not available on the LAN.

The initial DNS query often takes much longer to process. Once the information is cached on the local system, subsequent queries are faster. Initial RMI client setup costs were as high as 5,007 milliseconds (ms) in the NT experiments. As the information was cached by the local system, subsequent costs dropped to an average of 463 milliseconds, as shown in Table 4-1.

4.3.2 Asymptotic Costs

The initial setup costs are relatively high for connection-oriented protocols, especially for RMI. As the number of communication iterations goes up, the connection setup overhead becomes less significant. Figure 4-1 shows the asymptotic costs of the ping-pong experiment of the 255-byte object on Solaris.

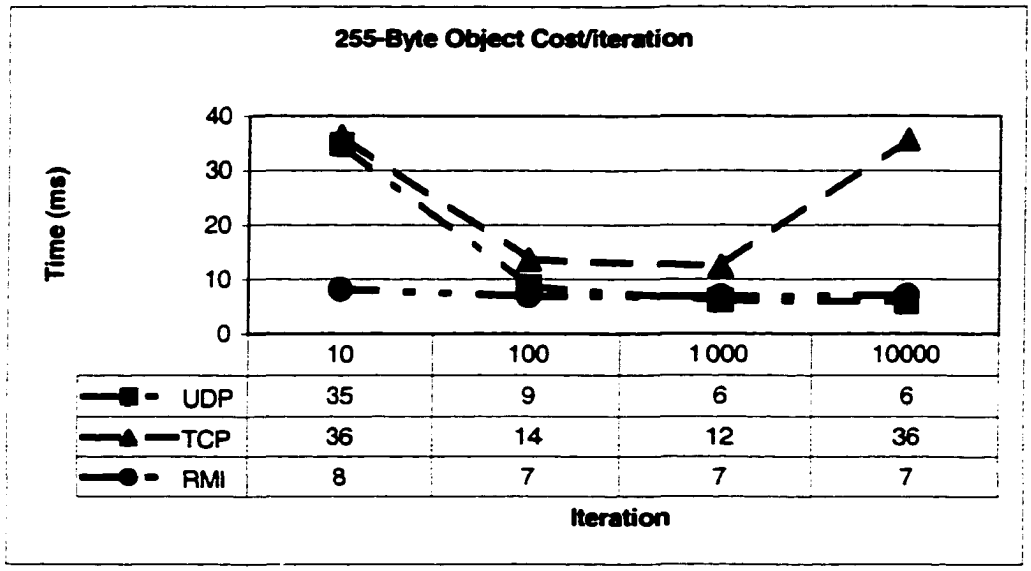


Figure 4-1, Asymptotic Costs of a 255-Byte Object on Solaris

As shown in Figure 4-1, the costs of per datum transmission decrease as the number of iterations goes up—except for TCP, which exhibits abnormal behaviors, as we will see in later experiments. Asymptotically, RMI has outstanding performance given the services it provides.

Table 4-2 shows the asymptotic per datum costs for transmitting various data types on Solaris. The results show that many factors affect communication performance. These include:

- Communication protocol selection
- The data type used for representing a message
- The size of a data unit

Table 4-2, Asymptotic Per-datum Costs on Solaris on 100 Mbps Ethernet (ms)

DATA	UDP	TCP	RMI
int	1	89	2
double	1	90	2
16 bytes	<1	<1	3
256 bytes	<1	<1	3
1K bytes	1	1	4
4K bytes	1	*	4
64K bytes	14	*	20
13-char String	2	1	3
42-byte object	4	3	2
255-byte object	9	14	7
1K-byte object	27	75	27

* Program hung and the testing process was suspended.

It is worth noting that some measurements could not be made using TCP or UDP. In the cases of the latter, repeated iterations of the large byte array failed (there was data loss). It is not surprising, given the nature of UDP. More surprising was the much more frequent failure of TCP. These failures may reflect problems with the buffering and/or the default value of Maximum Transmission Unit (MTU) used by TCP in the particular implementation of Java we examined, and could be related to the few anomalies seen in the TCP columns in Table 4-2 and Figure 4-1. Interestingly, RMI did not suffer these problems, even though it uses the TCP protocol. One might speculate that in this implementation, RMI does not use the Java TCP classes, but rather an alternate way of getting the TCP services.

Table 4-3 shows the corresponding NT performance on 100 Mbps Ethernet. TCP anomalies persisted in the NT experiments and got worse in some cases (TCP's `int` and `double`). On average, primitive data types, such as `int`, `double`, and bytes, did not

perform better in the NT experiments than in the Solaris. At the same time, objects, especially the large objects, experienced better performance in the NT experiments than in the Solaris. The ratios of speed enhancement in some cases are even better than the ratio difference of the system clock cycles of the machines we used. Many factors could contribute to the differences in performance of the two operating systems, besides the difference in clock cycles. Still, one might speculate that JDK could be highly optimized for object performance, especially that on large objects, on an NT platform.

Table 4-3, Asymptotic Per-datum Costs on NT on 100 Mbps Ethernet (ms)

DATA	UDP	TCP	RMI
int	<1	300	4
double	1	301	4
16 bytes	1	<1	4
256 bytes	<1	<1	4
1K bytes	1	<1	4
4K bytes	1	1	5
64K bytes	14	16	22
13-char String	2	1	4
42-byte object	5	6	3
255-byte object	7	9	5
1K-byte object	14	29	12

Network connections also affect performance. Table 4-4 compares the results of the same experiments on an NT platform using a 10 Mbps Ethernet connection and a 28.8 Kbps modem connection. Performance degrades in all categories as the network speed goes down, especially the performance of large data types, such as 64K bytes. This shows that large-size data transmission is less reliable on slow network connections, especially for TCP.

Table 4-4, 10 Mbps Ethernet and 28.8 Kbps Modem Connections on NT (ms)

DATA	10 MBPS ETHERNET			28.8 KBPS MODEM		
	UDP	TCP	RMI	UDP	TCP	RMI
int	1	301	6	123	701	367
double	1	302	6	126	704	383
16 bytes	1	1	7	130	139	386
256 bytes	1	*	8	327	344	528
1K bytes	3	3	9	755	821	1199
4K bytes	9	8	15	2973	*	3242
64K bytes	127	*	804	*	*	*
13-char String	3	2	6	143	139	392
42-byte object	7	8	4	150	157	212
255-byte object	10	13	7	345	304	385
1K-byte object	20	51	15	939	974	979

*Program hung or failed to return before 30-minute timeout for 100 Ping-Pong iterations.

On extremely slow networks, such as the 28.8 Kbps modem connection, the cost of actual data transmission becomes more significant relative to the cost of data preparation on the end hosts. Java primitive data types and small byte array data types, which required less data preparation and performed well on high-speed network connections, show the largest increase in performance cost on slow network connections. The performance of data types that had relatively high preparation costs on high-speed networks, such as Java objects, show better resistance to slow connections. The results show that the overall performance of TCP closes up the gap between itself and the other protocols.

4.3.3 Accumulated Costs

Because of the unreliable, ephemeral nature of the clients in a WebComputing context, it is useful to consider the data in Table 4-5, which presents the total time to complete 100

iterations—*plus* the setup cost on Solaris on a 100 Mbps Ethernet connection. During the experiments, the JVMs on both server and client used approximately 9-10 MB of RAM. The JVMs also employed multiple threads (5-16 LWPs [ZY,98] were observed), even though the application programs were single-threaded.

Table 4-5, Total Time for 100 Iterations Plus Setup Costs on Solaris (ms)

DATA	UDP	TCP	RMI
int	104	8977	956
double	112	9074	974
16 bytes	69	73	1042
256 bytes	75	80	1049
1K bytes	86	97	1084
4K bytes	157	*	1177
64K bytes	1470	*	2761
13-char String	226	118	992
42-byte object	462	355	946
255-byte object	905	1407	1426
1K-byte object	2773	7505	3378

* Program hung and the testing process was suspended.

Figure 4-2 presents the same data shown in Table 4-5, but provides clearer protocol comparison. The y-axis is in logarithmic scale.

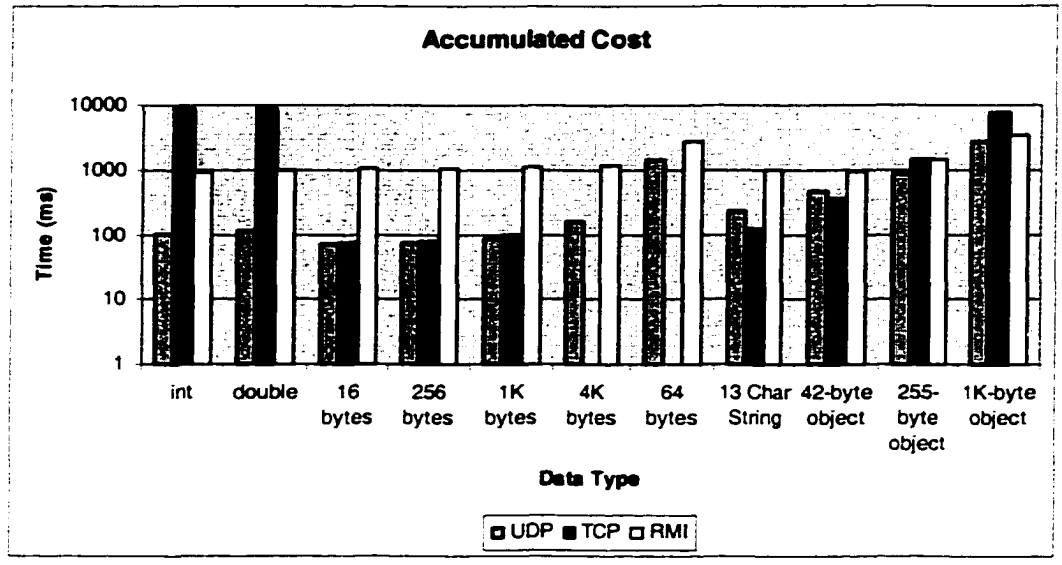


Figure 4-2, Accumulated Costs for UDP, TCP, and RMI Protocols

For Applet-to-Server Application communication, the experiments show that the Netscape browser took more than 10 seconds to start up and load Java run-time classes. For experiments executing a small number of iterations, we observed that the initial run of the testing applet was almost always significantly slower than the subsequent runs; the difference became negligible in experiments executing a large number of iterations. This initial slowness of applets might be attributed to code verifications enforced by the Sandbox security model and to caching. The performance of TCP was comparable with Application-to-Application performance. On the other hand, UDP performance varied. In addition, the browser's Sandbox implementation prohibited symmetric RMI communications.

Along with the above experiments, performance results using JDK 1.2.1 with its default JIT compiler on the same hardware setups were collected and compared with the above experimental results. The comparison shows that JDK 1.2.1 yielded significant improvements in connection setup costs in all three protocols, and transmission costs for

large objects were also greatly reduced for all three protocols. However, the experiments also show that there was no improvement or even slightly poorer performance for primitive data types (such as integer and byte arrays) and small data objects.

Finally, we created two programs to simulate data conversion for network transmission: one for UDP and one for TCP. The efficiency of these programs was used to estimate the data processing costs for these communication protocols. Two threads were employed for each of these programs. Instead of sending message objects to the network, they were bounced between the two threads. For TCP, pipe stream classes were used for connecting the two threads. For UDP, byte arrays were passed between the threads. The costs for data processing on the end host were significant. To give an idea of the cost, the results for processing a 255-byte object is shown in Table 4-6.

Table 4-6, Total Time for Processing 255-Byte Objects on Solaris (ms)

255-BYTE OBJECT	UDP	TCP
Networked ping-pong	872	1364
Threaded ping-pong	571	506

4.4 Conclusion

This chapter presents a number of experiments conducted on different platforms using different protocols. The results show that the performance of Java communication varies in five dimensions:

1. Java communication protocol selection
2. Data type used for message representation

3. Data size
4. JVM implementation for different operating systems
5. Network connection speed

This evaluation can aid system developers in selecting the communication protocols that best suit the design of a WebComputing system. Application developers can also use this evaluation to inform their decisions with regard to application implementation.

After careful examination of the performance results, we concluded that UDP was almost always the winner in performance, in accordance with its lightweight nature, but did not perform well in small-object transmission (the 13-Character String object). Although data loss is possible with UDP, it happened less often than one might have supposed, and overall, UDP had the best performance average of the three protocols. TCP as currently implemented in JDK1.1 apparently suffers from numerous anomalies. Asymptotically, RMI had outstanding performance, providing reliable high-level communication services. As the size of the object increased, its performance benefit became visible. It performed respectably with Java primitive data types and outperformed both UDP and TCP on Java object communications. But its high startup costs made it much more expensive than UDP and TCP, even when amortized over 100 iterations of communications. For all three protocols, although per-byte costs dropped dramatically with larger arrays and objects, the low absolute costs of small communications make them preferable to large ones in a WebComputing system. This is especially true for UDP.

Our measurements suggest that for communication between server and applet in the context of Internet-based WebComputing, UDP's performance, coupled with other relevant considerations, makes it the protocol of choice, especially if the size of the data being transferred is small. On the other hand, in Application-to-Application communication, where reliability of the underlying processes is not at question and where larger data transfers may be more natural, the relatively small performance benefit of UDP is unlikely to be enough to justify its use. In that case, RMI is the method of choice.

It is also clear that the performance in Java—even on a platform presumably so well-developed as Solaris on an Ultra-Sparc system—is spotty as well as version-and-platform dependent. JVM implementation on Windows NT is comparable to the JVM implementation on Solaris.

Chapter 5

SWC Framework Design

5.1 Introduction to the SWC Framework

The current trend in WebComputing research is the exploration of design approaches. The Charlotte [BKKW,96] project pioneered the research in this field and introduced a distributed-thread-programming model for WebComputing. Javelin [CCINSW,97] presented a working prototype for WebComputing, which assessed the feasibility of this new distributed platform. SuperWeb [AISS,97] discussed numerous future research issues associated with WebComputing. Bayanihan [Sarmenta,98] [SH,99] presented a structured framework for WebComputing which emphasized modularized framework design.

This chapter presents a Simple WebComputing framework (SWC framework) designed and implemented at Brooklyn College of the City University of New York. SWC is a small, simple framework, originally developed as a tool for students, that simplifies developing and deploying WebComputing applications. It exceeded its

predecessors in many ways. To understand this, let us review the SWC framework design objectives.

5.1.1 SWC Design Objectives

Based on the CPE performance study presented in the previous chapter, we understand that the design of a distributed system for WebComputing must take performance constraints into consideration. At the same time, it must also satisfy the requirements of high-level application development.

The objective of the SWC framework design is to meet the challenges of both the network system, which emphasizes high performance, and the application, which requires a high-level programming API. This thesis strives to tackle the design from these two apparently opposite perspectives and provide an integrated framework design that excels at both aspects. The SWC framework presented here shields an application developer from the underlying, unreliable, and ever-changing Internet execution environment, while providing system developers with freedom in designing architecture.

To gain a better understanding of the objective, we may examine the design of a WebComputing system from the opposite perspectives presented above: the design of the programming interface for application development and the design of the system architecture for optimal performance. These two perspectives are equally important for WebComputing system design.

The former is a top-down view of the system that concerns the programming interface for WebComputing application development. The programming API must be simple, straightforward, and intuitive. It should be free from the details imposed by the

communication mechanisms and the setup requirements of the execution environment. Fault-tolerance and load-balancing issues must be addressed by the system instead of application developers. Ideally, the development of a WebComputing application would be as simple as, or even simpler than, multithreaded design on an SMP machine.

The latter perspective is a bottom-up examination of the design with regard to network system development. From this perspective, maximizing network system reliability and minimizing communication delays are primary concerns. Besides providing reliable services, the framework architecture must be geared towards improving performance and reducing system overhead.

The two design perspectives often have been presented as mutually exclusive. To solve this problem, we provide a layered WebComputing system design that separates the programming interface from the underlying WebComputing run-time system. As a result, the SWC framework contains two layers: the API layer and the system run-time layer. This design approach shields application developers from the underlying dynamic and unreliable execution environment and makes the application programs portable, yet provides freedom in architecture design.

5.2 SWC System Design

We will start our exposition by introducing the system's programming model (section 5.2.1) and its programming API (section 5.2.2). The following sections discuss the layered-framework design (section 5.2.3) and its lower-level run-time system implementation (section 5.2.4). After this overview of the system, section 5.2.5 covers the internal class hierarchy, along with the architecture of the SWCRouter module.

5.2.1 Master-Worker Programming Model

Communication overhead, as we observed in the previous chapter, requires that system architecture design thoroughly consider communication performance. A distributed shared-memory programming model, such as distributed threads, might not be the optimal choice in a WebComputing environment. To minimize communication overhead, a programming model should naturally reflect the cost and steer the application developers in such a manner as to minimize expensive communication operations without sacrificing high-level programming interface.

As a result, the SWC framework employs a Master-Worker programming model for application development. This programming model has a number of attributes that make it the choice for WebComputing:

1. It has been widely used and proven effective in many distributed computing systems, such as in DP, PVM, and MPI distributed programming environments.
2. It fits well with object-oriented design methodology. Here in the SWC system, it is supported by the Java programming language.
3. It adapts well to an environment in which communication is relatively expensive. Network latencies and bandwidth limitations on the Internet are extremely high.
4. It is a versatile and general parallel programming model and can be transformed to many other programming models, including the Linda tuple space model [CG,90].
5. A large set of problems can be parallelized using this model.

5.2.2 SWC API Design

The SWC framework provides a Master-Worker MIMD parallel programming model. The essential SWC API contains two abstract classes and three interfaces (Figure 5-1).

```
public interface SWCUnit extends Serializable {}
public interface SWCWorkUnit extends SWCUnit {}
public interface SWCResultUnit extends SWCUnit {}

abstract class SWCMaster {
  final void sendWork(SWCWorkUnit swu) {…}
  final SWCResultUnit receiveResult() {…}
  abstract void beTheBoss();
}

abstract class SWCWorker {
  final synchronized void send(SWCUnit su) {…}
  final synchronized SWCWorkUnit receiveWork() {…}
  abstract void doTheWork();
}
```

Figure 5-1, SWC Framework Programming API

To use the SWC framework, an application programmer must extend the framework-defined abstract classes. The `SWCMaster` class defines the master's

behavior, while the `SWCWorker` class defines the computation of a task and is instantiated on volunteer hosts. The `SWCWorkUnit` and the `SWCResultUnit` interfaces label the data that define particular tasks or the result transmitted between the framework modules. They are also used as mechanisms for recognizing task equivalence and task completion. Using these classes, the framework itself oversees the entire computation and handles all communications.

At the programming interface level, the master process (SWCMaster) defines an initial set of tasks and integrates the results of those tasks, possibly defining new tasks along the way. The tasks comprising the computation are defined by lightweight data objects, received and carried out by workers (SWCWorker) that run as Java applets, and generated dynamically by both the master and the workers themselves. The master and any worker can also broadcast control information through the server (SWCRouter) to all workers. The workers do not communicate directly with each other — they simply return the results of the tasks and request new ones to carry out.

5.2.3 SWC Framework Design

The SWC system provides a simple yet elegant WebComputing programming interface that defines a high-level Master-Worker parallel programming paradigm. Based on this programming model, a WebComputing user may build his application without descending to low-level communication protocols and is free from concerns about the underlying dynamic execution environment. This abstraction separates WebComputing programming interface from underlying specific WebComputing architecture design.

This may greatly enhance the portability of a WebComputing program and ensure robustness of the architecture design. Figure 5-2, presents the basic SWC framework.

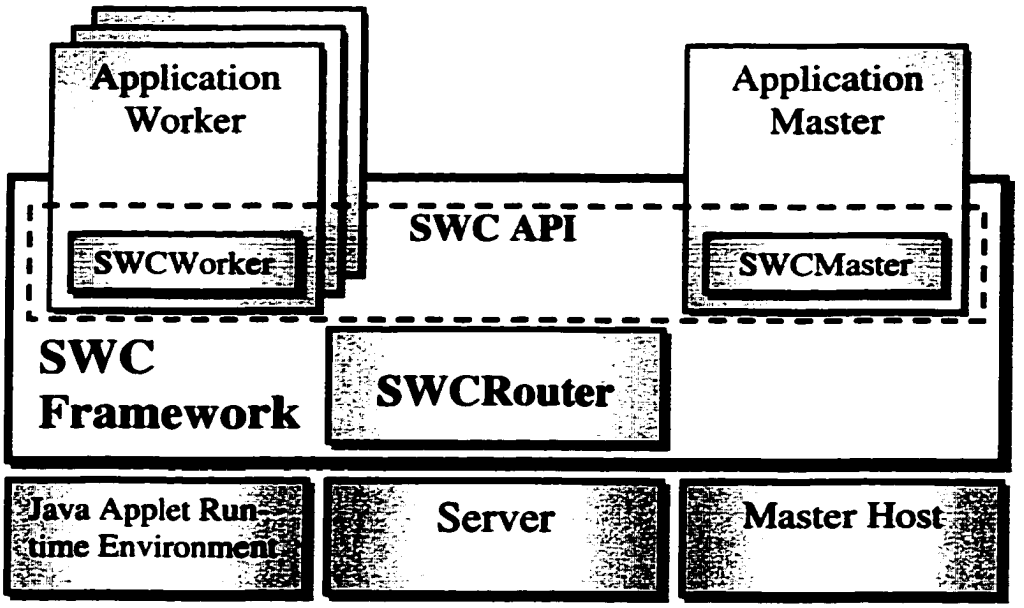


Figure 5-2, SWC Framework Design

Figure 5-2 illustrates the SWC architecture. The large middle box represents the SWC system proper and contains two layers. The upper layer, shown in the dashed box, includes the SWC programming API. The lower layer provides the support for this API.

An SWC computation can be initiated using a form in a Web page provided by the server. A servlet [SunServlet] responds by creating the necessary objects within the HTTP server, along with an external master process. The master process and the server communicate using the connection-oriented TCP protocol and need not reside on the same machine. The master process, using programmer-provided classes, creates an initial set of task definitions, which are sent off to the server. The server (SWCRouter)

maintains a collection of task definitions, and uses eager scheduling, as is commonly done in WebComputing [AISS,97] [BKKW,96] [CCINSW,97], to assign them to applets that have been downloaded to volunteering Web clients. The most obvious candidates for WebComputing will not require large quantities of data to define tasks. Because of the unreliable nature of the applets, the desire to eliminate system-imposed limits on the number of connections on the server as a potential bottleneck, and the performance consideration [YAC,99], server-applet communication uses the connectionless UDP protocol. Because the server does not consider a task complete until the results are actually received, lost packets do not compromise the integrity of the application. To avoid failure to utilize an applet as a result of communication failure, applets use a timeout mechanism to repeatedly send the server their most recent response until the server provides additional work or instructs them to terminate.

As the server (SWCRouter) receives responses from the applets (SWCWorker), it determines whether these are additional task definitions, in which case they are added to its collection of task definitions (WorkHeap), or results from completed tasks, in which case they are passed to the master (SWCMaster). The latter may, in response, generate new task definitions or control information for the server to distribute to the applets. Control information is immediately broadcast to applets and is made available to all future ones. Figure 5-3 illustrates SWC communication between different system modules.

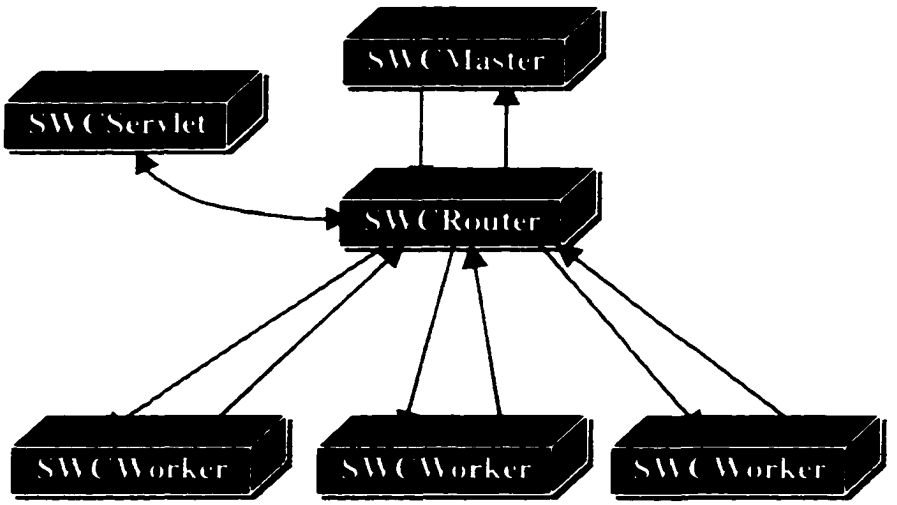


Figure 5-3, SWC Data Flow

5.2.4 Three Implementations in One

Because of its independent architecture design, SWC is able to provide three implementations of its architecture (Figure 5-4). One is thread-based and runs on an SMP parallel machine, which could serve as a development environment when a Web-based one is not available or is inconvenient. Another implementation is based on independent system processes. The third implementation serves our main purpose—WebComputing. It consists of a multithreaded, servlet-enhanced HTTP server that provides an application control page, creates the master process, downloads the applets and handles all communication. It also provides the classes that define, for both the master process and the applets, their computational structure and their communication tools.

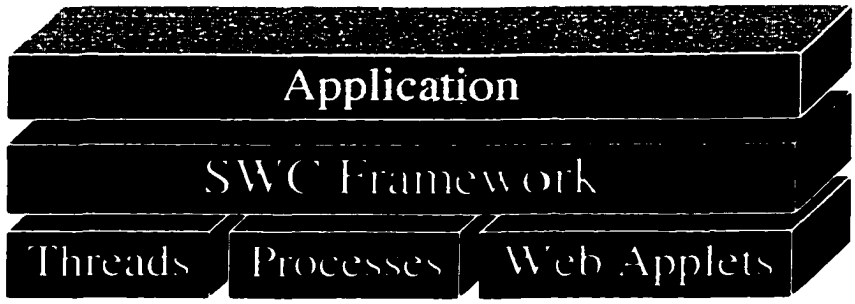


Figure 5-4, SWC Architecture Implementation

5.2.5 SWC Internal Design

5.2.5.1 Class Hierarchy

The internal SWC framework design reflects and enforces the separation of the programming API and the run-time system. There are two sets of classes. The first set supports the exported programming API. The `SWCMaster` and the `SWCWorker` classes belong to this set. Both these classes inherit from the `SWCActor` class (Figure 5-5). They are exported to top-level SWC applications as public classes and support the execution of application master and workers respectively.

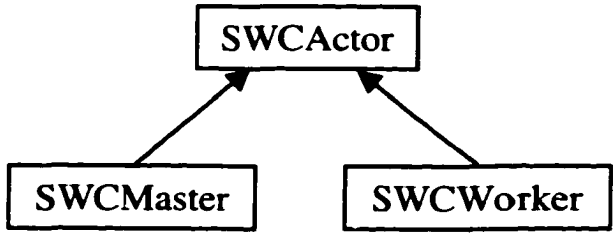


Figure 5-5, SWC Framework API Classes

Parallel to this structure, the second set of classes creates a platform on WebComputing environments and instantiates the `SWCMaster` and `SWCWorker` classes accordingly (Figure 5-6).

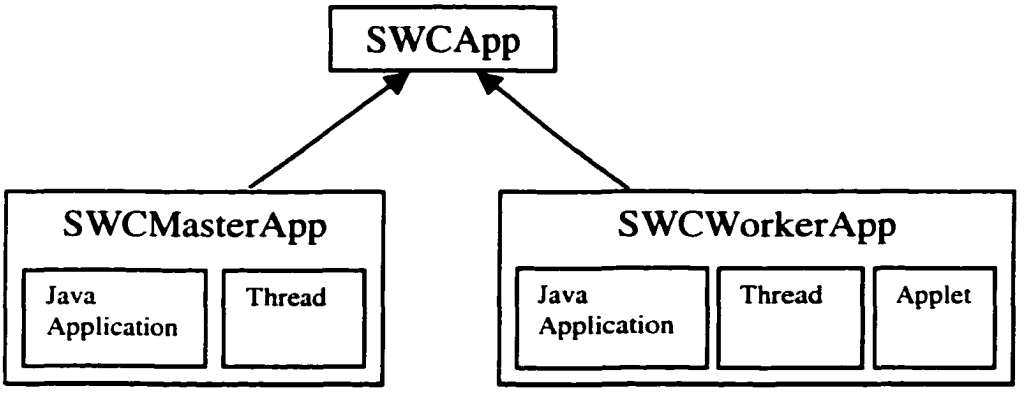


Figure 5-6, SWC Framework Runtime Classes

5.2.5.2 SWCRouter Design

Both the SWCMaster and the SWCWorker modules communicate directly with the SWCRouter module. The SWCRouter module contains another set of classes that is essential to the SWC system. The module is responsible for managing all the underlying data flow between the master and the workers. SWCRouter is designed as a composite of multiple objects (See Figure 5-7), including the WorkHeap object, DestinationHeap object, GlobalManager, and a number of Router's Actors. The Router's Actor box, shown in Figure 5-7, is a collection of closely related objects that function independently while cooperating with the other objects through buffers or direct messages.

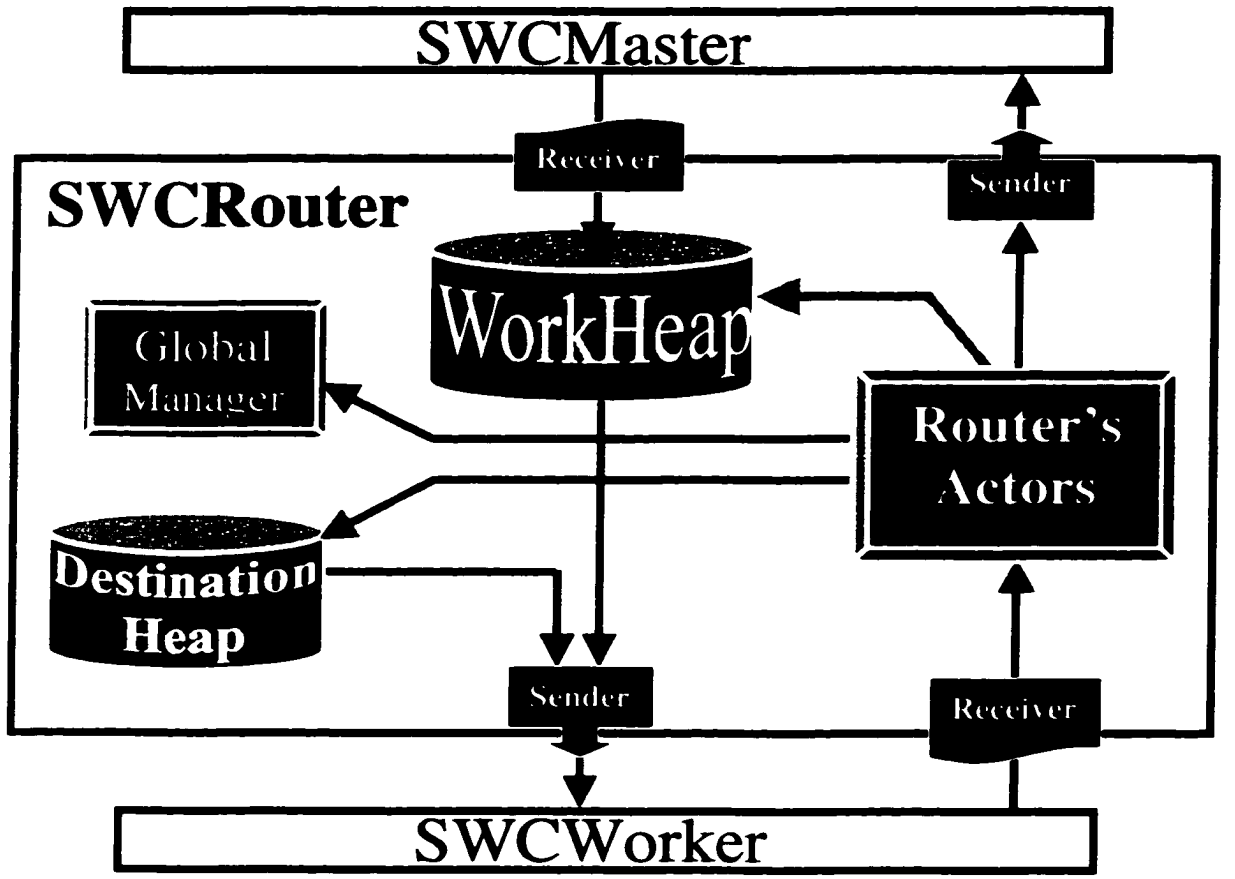


Figure 5-7, SWCRouter Design

The Router's Actors include three primary objects: *BeenThere*, *Distinguisher*, and *Reaper*. In a nutshell, BeenThere is responsible for eliminating duplicated packages returned by workers. Upon receiving packages, Reaper checks and removes completed tasks from the WorkHeap. At the same time, Distinguisher assigns new tasks to the WorkHeap, submits computational results to the SWCMaster, or forwards global data objects to the GlobalManager (Figure 5-8). The actors run concurrently as independent threads.

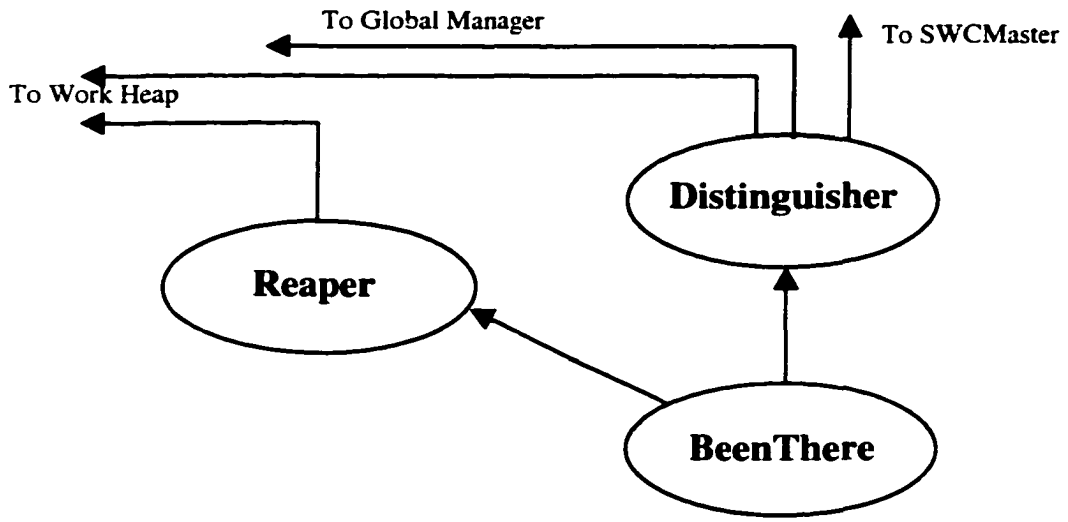


Figure 5-8, SWCRouter Actors

5.3. Performance

The efficiency of a WebComputing application depends on many elements beyond the control of the implementer of a WebComputing system or framework. Among these are: first, the appropriateness of the application to the platform because of the extreme network latency and bandwidth limitations of the Internet; second, the implementation of the Java Virtual Machines (JVM) that supports the applets; and third, the network conditions that affect latency and bandwidth.

We benchmarked the SWC system performance. Through this study, we hoped to recognize potential performance bottlenecks and identify possible performance enhancements.

To provide a sensible estimation of the system's performance, we needed to quantify the communication overhead, which is primarily generated by data transfers between WebComputing system modules. Data flow of the SWC system can be classified into three categories:

- **MW:** The SWCMaster sends newly defined tasks to SWCWorkers.
- **WM:** The SWCWorker sends SWCResultUnits to the SWCMaster.
- **WW:** The SWCWorker sends SWCWorkUnits to the SWCRouter for rescheduling.

Evaluating the communication costs for these three categories provided insight into the system's behavior. It also provided information that, along with the communication patterns of specific applications, a WebComputing application developer may use to estimate their communication overhead.

To study these costs, performance measurements were carried out using JDK's Solaris Production Release version 1.1.7 and its default just-in-time (JIT) compiler on two Sun Ultra-Sparc-10 workstations with 128 MB of RAM each, running Solaris 7 operating systems. One machine was employed as a server, the other as a client. These machines were connected with a 100 Mbps Ethernet. The network latency was below 1 millisecond (ms) (tested with UNIX ping facility). We also used Netscape Communicator 4.5 with a 1.1.5 Java run-time environment when applet implementation of the SWC system was tested.

Table 5-1 shows the communication costs for different categories of application data flow using an object with 45-byte internal state. For both thread and process measurements, SWCMaster, SWCWorker, and SWCRouter ran on a single workstation

as three threads and three processes respectively. For the Web Applet measurement, two workstations were employed. Both the SWCMaster and the SWCRouter ran on the server as different processes, while a SWCWorker ran as a Java Applet in a Netscape browser on the client. In the first experiment (MW and WM), a 45-byte object was bounced back and forth between the SWCMaster and the SWCWorker. The cost of a round trip was recorded. The second experiment (WW) shows the data flow cost of the third communication category.

Table 5-1, Communication Costs of Different Paths for a 45-Byte Object (Averaged Over 100 Iterations)

Flow Path	Thread	Process	Web Applet
MW and WM	50 ms	50 ms	60 ms
WW	7 ms	8 ms	19 ms

Chapter 6

Scalability Enhanced SWC Framework

6.1 Introduction to Enhanced SWC System

The performance of a SWC system server diminished quickly as the number of client connections increased. This implied that the single-router architecture of the SWC framework may impose limitations on the scalability of the system and become a performance bottleneck, and led us to hypothesize that a multi-router architecture was needed to enhance scalability and support massively distributed computation. This chapter presents the E-SWC framework, a scalability enhanced SWC framework design that extends the initial architecture to support multiple distributed SWCRouters. However, this extension raises a number of design issues and invalidates many initial implementation approaches.

To adhere to the initial design principle, three constraints were set for the enhancement redesign. First, the changes needed to be transparent to the initial SWC applications. The changes were thus limited to the run-time system of the framework and

did not affect upper-level applications. The exported framework API remained unchanged. Second, the redesign had to be simple, yet effective. Because of the programming and execution environments, a WebComputing system can suffer from significant communication overhead. Complex run-time systems can introduce additional overhead from which we wished the system to refrain. Third, the redesign needed to maximally reuse the existing building blocks and keep the design adherent to the initial architecture—a basic software engineering design principle.

6.2 Enhanced SWC Framework Design

6.2.1 Enhancement Overview

The changes introduced by the E-SWC redesign can be classified in two categories: the redesign for supporting multi-router architecture and the enhancements for large-scale system deployment.

The multi-router architecture redesign includes two areas of enhancement. First, the TCP Senders and Receivers, the components in charge of connection-oriented communication between different system modules, were enabled with multicasting capabilities. Second, the SWCRouter architecture was redesigned to support a system-wide load-balancing scheme. The changes were made with the intention of adding minimum complexity and maximum functionality. Some design issues are covered in later sections.

The deployment enhancement also consists of two parts. First, the system setup information was consolidated to facilitate distributed multi-router management. Second, a server deployment system was introduced using Java servlet technology.

6.2.2 Design for Supporting Multi-Router Architecture

6.2.2.1 Multicasting Enhanced TCP Sender and Receiver

In the single-router SWC framework, the Sender and Receiver objects provide an end-to-end communication channel between system modules. The enhanced multi-router architecture needs additional channels for different types of communication. For example, before the enhancements, the SWCMaster communicates only with the SWCRouter. Under the multi-router architecture, the master must be able to use various types of communication to communicate with a group of SWCRouters. Instantiating Sender and Receiver objects for every pair and different type of end-to-end communication channel is not a viable solution. For the E-SWC framework, both the TCP Sender and the TCP Receiver are enhanced to support multicasting. The enhanced Sender supports a set of built-in collective communication schemes, including end-to-end delivery, broadcasting, random-polling, and round-robin schemes. Each of these schemes will be discussed in detail later in this chapter. The enhanced Receiver supports multiple connections by spawning a thread for each one. The enhanced Sender and Receiver support inter-module many-to-many communication.

6.2.2.2 Load Balancing at Multiple Levels

It is often impossible to predict the size of a parallel subtask that is assigned to a processor. Therefore, static partition of a parallel problem is inadequate for balancing workload among the participating processors. Load-balancing schemes were used to dynamically redistribute tasks among the processors, achieve maximum execution and avoid processor starvation. The use of load-balancing schemes is essential for optimizing

the execution of a parallel application. This topic is beyond the scope of this thesis and is covered in detail in [KGGK,94]. The E-SWC framework provides load-balancing support.

The framework was designed in layers. The enhancements focus on the low-level run-time system and leave the SWC API intact. The E-SWC architecture is shown in Figure 6-1. Bilateral inter-module communications occurs at four different levels, including (see Figure 6-1):

1. Connection-oriented communication between the SWCMaster and a group of SWCRouters. The communication between these modules can be either one-to-one-based or broadcasting-based determined by the SWC module in question.
2. Connection-oriented communication between different SWCRouters. Each pair of routers may communicate directly.
3. Connectionless communication between an SWCRouter and its SWCWorkers.
4. SWCRouter Control through SWCServlet.

For the E-SWC system, load balancing must be addressed at four different levels corresponding to each of the four inter-module communications:

1. The SWCMaster assigns tasks to a group of SWCRouters.
2. The SWCRouters redistribute their tasks.
3. The SWCRouters assign tasks to their SWCWorkers.
4. The SWCServlet assigns volunteer clients to the SWCRouters.

At the first level, the SWCMaster must assign tasks equally to each SWCRouter. At the second level, a finer dynamic load-balancing scheme is needed for rescheduling

tasks among the SWCRouters. A router without work can request work from other routers. At the third level, the SWCServlet must schedule the incoming volunteers evenly among the SWCRouters. Each router should have a roughly equal number of volunteers. Finally, the routers must assign tasks more or less equally among their volunteer applets.

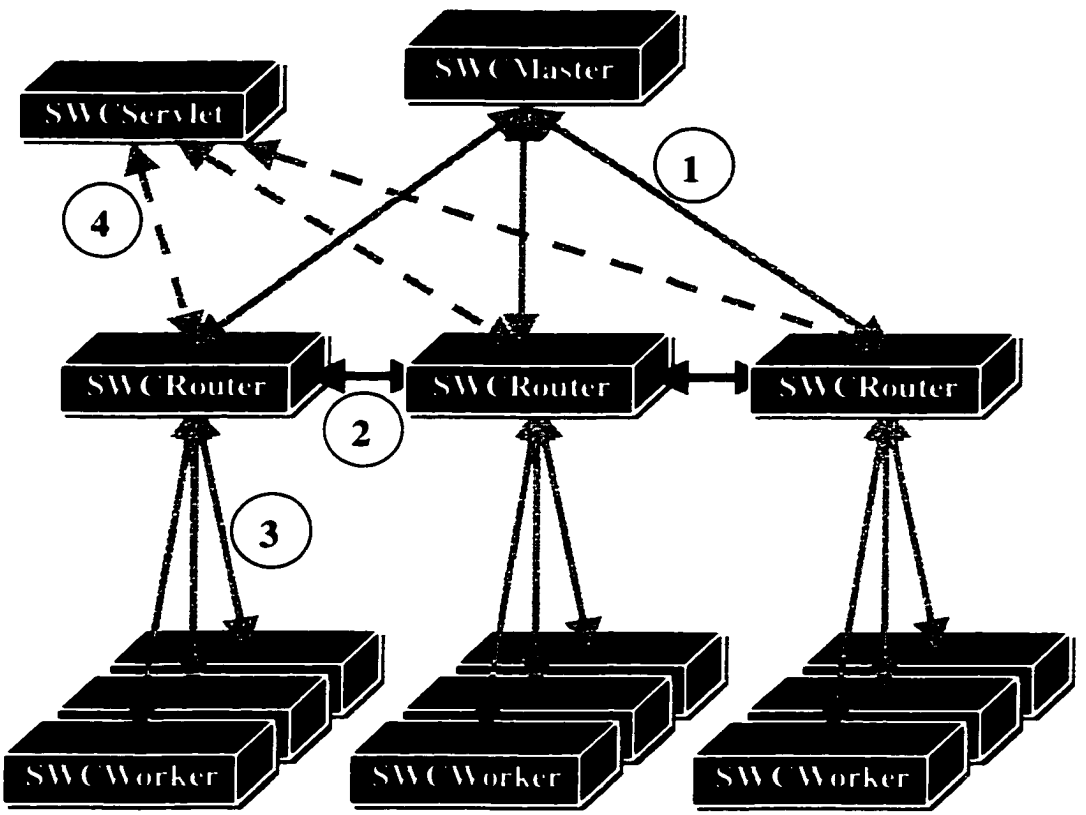


Figure 6-1, E-SWC Framework

To ensure that the design objectives are met, the E-SWC framework uses different load-balancing schemes at each level of communication. First, the E-SWC system uses a

round robin scheduling algorithm for the SWCMaster to assign tasks to the SWCRouters. In later releases, this scheme may be replaced with priority-based schemes using dynamic workload information on each router. Second, the E-SWC system uses a *random polling*-scheduling algorithm for dynamically balancing load among the SWCRouters. When a router's WorkHeap becomes empty, it randomly selects another router and requests work. The selected router shares half of its workload. These tasks are not removed from the selected router; rather, they are pushed to the back of the scheduling queue. When a shared task is done, the router notifies the original router of the completion of the task. This provides a fine-grained load-balancing scheme. Third, we used the *eager scheduling algorithm* for SWCRouters assigning tasks to volunteer applets (SWCWorkers). This approach maximally reuses existing components from the initial SWC architecture design. Finally, the E-SWC system uses a *round robin* algorithm for the SWCServlet assigning volunteers to the SWCRouters. The first and the last load-balancing schemes, as described above, provide a top-level coarse-grained approach and simplify the design of both the SWCServlet and the SWCMaster.

6.2.2.3 A Distributed Extension of the Eager Scheduling Algorithm

The inter-router load-balancing scheme can be better understood by examining the redesign of the E-SWC run-time system.

The initial SWC system design provides an eager scheduling scheme for task tracking and scheduling. This is accomplished by the three Router's Actors: the Distinguisher, the Reaper, and the BeenThere actor (Figure 5-8). Under the eager scheduling algorithm, a task is assigned to idle workers repeatedly until it is completed. A tracking scheme is required for task management, for which a task ID is used. The

Router's Actors keep track of finished tasks which can be removed from the system. However, because the task-labeling and -tracking scheme depends on a router's internal state, a simple eager scheduling algorithm is not scalable for the multi-router architecture.

To support inter-router load balancing, the E-SWC system introduced a new actor—the *ControlManager*, see Figure 6-2. The ControlManager is responsible for communication and load balancing among the SWCRouters. This design extends the existing eager scheduling algorithm across a server boundary and is achieved in a distributed manner. It provides a scalable, system-wide load-balancing solution for the multi-router system.

The functionality of a ControlManager is described as follows: it can generate requests to another randomly selected SWCRouter for sharing work and respond to such a request. When the local WorkHeap becomes empty, the router's ControlManager randomly selects a remote router, establishes a connection if necessary, and requests work. Upon receiving such a request, the remote router's ControlManager prepares half of its workload from its WorkHeap, if the WorkHeap is not empty, and sends the work back to the requesting router. The shared tasks are not removed from the WorkHeap of the router in question; rather, they are pushed to the back of the scheduling queue. Upon receiving the shared tasks, the requesting router labels them as remote, puts them onto its WorkHeap, and distributes them to its workers. A SWCWorker may return `SWCResultUnits` and/or `SWCWorkUnits` to its router. Upon receiving the returned packages, the Distinguisher determines if the task is local or remote. If it is local, the router proceeds as it would in the single-router framework. If it is remote, the ControlManager forwards the returned packages to the remote router. The remote

ControlManager receives the returned packages and passes them to the BeenThere actor as if the packages were returned by that router's own workers.

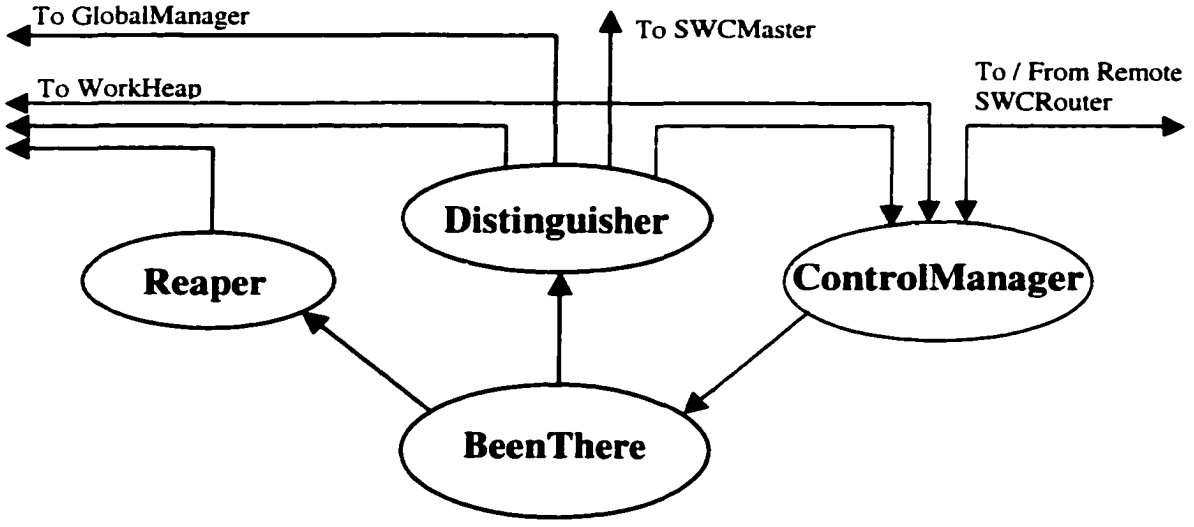


Figure 6-2, E-SWCRouter Actors

The ControlManager is a self-contained module and interacts with other actors using the existing interfaces. This design maximally reused the existing building blocks of the initial SWC system and kept the E-SWC system design adherent to the original architecture. This approach kept the impact of the multi-router load-balancing design to a minimum.

The design of the ControlManager module is simple and effective. It is based on the existing eager scheduling algorithm, which operates independently at each router's subsystem (a SWCRouter and its workers). A ControlManager, which is instantiated at each router, cooperates with other ControlManagers. Together, they provide a distributed scheme for inter-router data flow. This distributed scheme extends the independent eager scheduling scheme at each router to a distributed multi-router architecture and achieves

load balancing among the routers. It provides a distributed extension of the eager scheduling algorithm.

6.2.2.4 Problems Encountered

During the design process, we encountered two problems: duplicated results and a race condition. We examine these problems and their solutions in the following sections. Through this, we explain additional details of the E-SWC framework.

Duplicate Results

The initial SWC framework uses a centralized WorkHeap counter scheme to label and track tasks and ensure correct completion of a computation. New tasks generated by both the SWCMaster and the SWCWorkers were stored on the single WorkHeap. The singleton object can easily enforce a system-wide labeling and tracking scheme. A counter is employed to assign a unique ID to each new task obtained from either the master or a worker. This ensures task tracking and eliminates duplicated results.

The E-SWC framework introduces multiple routers, and each router contains its own WorkHeap. Due to load balancing, a task may be reassigned to a different router. The original WorkHeap counter scheme can no longer guarantee the system-wide uniqueness of a task's ID. A naive suggestion is to build an ID scheme on the master, like the one employed by the single-router system. A centralized approach may suggest an easy guarantee of system-wide task uniqueness; however, this is not a viable solution because both the SWCMaster and a SWCWorker can generate new tasks. Tasks generated by a worker would not be able to acquire unique IDs using this approach. A labeling scheme on either end (the SWCMaster or the SWCWorker) cannot cover the

entire system. Thus, the tracking scheme must be incorporated into the SWCRouter subsystem in a distributed fashion.

Solution

To make the modification minimal and efficient, the E-SWC system utilizes the existing WorkHeap counter scheme as part of its distributed labeling and tracking scheme. A new task is labeled independently on each router. This is the task's original ID. When a shared task is obtained from another router, it is labeled as remote and assigned a new ID, local to the new router. The original router ID and the original task ID are kept. When a task's results (defined as `SWCResultUnit`) or newly defined tasks (defined as `SWCWorkUnit`) are returned from a worker, the Distinguisher checks the task on the WorkHeap. If the task is local, it behaves as before—forwarding results to the master and putting new tasks on the WorkHeap. If the task is remote, the Distinguisher forwards the returned packages to the ControlManager; subsequently, the ControlManager forwards them to their original router. Upon receiving these packages, the original router treats the packages as if they were returned by the router's own workers. The `SWCResultUnits` are forwarded to the master and the `SWCWorkUnits` are pushed onto the router's WorkHeap. Therefore, this completes the inter-router load-balancing scheme.

First, let's examine the eager scheduling algorithm in a single-router architecture. Under the SWC framework, the master assigns a task to the only router. For example, this task is labeled with ID 10, depending on the router's internal state. Worker W1 becomes free and the router assigns task 10 to W1. Instead of removing the task, as can

be done in a reliable system, the router keeps the task and pushes it to the back of the scheduling queue. W1 starts working on task 10. At the same time, worker W2 becomes free, contacts the router, and also receives task 10 for processing. At this time, both W1 and W2 are working on the same task. Because a task is processed in a deterministic fashion, W1 and W2 produce the same set of results in the same order. These results are packaged using sub IDs associated with the task's ID. For example, "10.1" means "task 10, result package 1" or "10.15" means "task 10, result package 15." A worker indicates the completion of a task in the last package. Using the task-tracking scheme, the router knows which returned package is new and which package is a duplicate. The router keeps track of newly returned packages and ignores duplicates. Task 10 is kept on the WorkHeap till all its packages are returned.

This works well with the single-server architecture. The E-SWC framework, which supports multi-server architecture, requires addition mechanisms to ensure the system-wide consistency. Let's examine a scenario of the inter-router task flow as a scalable extension of the eager scheduling algorithm. The master assigns tasks to the routers using a round-robin algorithm or a priority-based scheme. Each task is uniquely assigned to a router (Because servers are presumed to be reliable, no task-tracking is used). Assigned tasks are labeled at each router with router-dependent IDs. They are the original IDs of these tasks. Let's say that a task is assigned ID 10 on Router A (see Figure 6-3). Router B might also have a task 10. However, these two tasks are different. If no inter-router load balancing takes place, both tasks are processed locally at each router. On the other hand, if Router A finishes first, this triggers the ControlManager of Router A (CM1) to randomly select another router from which to request work. CM1

contacts CM2 on Router B. Upon receiving the request, CM2 splits its work in half and forwards half to CM1. Let's say that task 10 of Router B is forwarded to Router A. This task is not removed from Router B's WorkHeap; instead, it is pushed to the back of B's scheduling queue, as in the single-router SWC system. CM1 receives task 10 from CM2 and labels it as Router A's task 20. In addition, it keeps "B-10" as the task's original ID. This indicates that task 20 on Router A is a remote task. Then, Router A proceeds as in the single-router architecture—scheduling task 20 to its workers. For example, the workers return package "20.1" and "20.2" as the results of task 20. The BeenThere actor of Router A (B1) eliminates duplicated results and returns the packages to the Distinguisher of Router A (D1). D1 examines the packages and knows that task 20 is a remote task because it is labeled with "B-10." D1 forwards "20.1" and "20.2" to CM1. CM1 communicates with Router B and returns "10.1" and "10.2" to CM2 by switching the task's ID to its original ID. CM2 forwards these packages to B2. B2 processes "10.1" and "10.2" as if they were returned by Router B's own workers and proceeds accordingly. Duplicated results of task 10 are discarded. A task is computed regardless of whether it is done locally or remotely. If a task, like task 10 on router B, is redistributed among the routers, it is completed and removed independently on each router. This completes the inter-router task flow. This scheme is an extension of the existing eager scheduling in a distributed fashion and enables scalable architecture.

The extended eager scheduling algorithm described above implicitly assigns a unique system-wide ID to every newly defined task. This implicit ID is the combination of both the router ID and the task's ID associated to the router. As long as the scheme

ensures a unique entry point for every newly defined task, this extended scheme ensures correct completion of the tasks, no matter where they are processed.

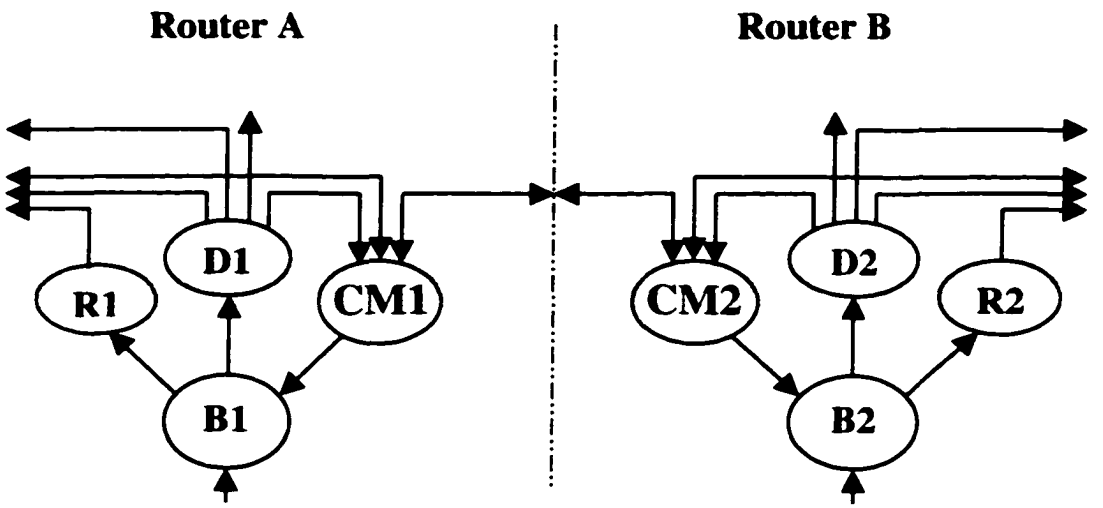


Figure 6-3, Load Balancing Task Flow Between Two Routers

In future SWC releases, we may introduce a *task migration* mechanism to reduce additional communications. A task on a router's WorkHeap may or may not have its result packages returned by a worker. If a task has returned packages, it would not be eligible for migration because these packages will have changed the internal states of the Router's Actors. Otherwise, a task with no returned package would be eligible for migration. This task would then be removed from the WorkHeap and transferred to the requesting router. The requesting router could then label the migrating task as local instead of remote, as described above. Doing so, the migrating task would replace its original task ID and router ID with a new set of IDs. Additional coordination between the Router's Actors and the two routers might be needed for task migration.

A Race Condition

Each router contains a number of actors that have access to shared data objects and are able to modify them through provided interfaces. Java provides some synchronization mechanisms to prevent data from being corrupted. However, a race condition can still occur. Under the E-SWC framework, the Reaper removes completed tasks from the WorkHeap. At about the same time, the Distinguisher needs to reference the task to determine its origin and act accordingly. This action may occur either before or after the removal of the task. Therefore, a race condition occurs.

Solution

Tasks removed from the WorkHeap are not discarded right away; instead, they are moved to a temporary storage, where they await eventual removal from memory after being referenced by the Distinguisher. This temporary storage provides a window for a possible delayed query by the Distinguisher. The window can be relatively small, because the interval between the actions taken by the Reaper and the Distinguisher is usually small. Tasks in this storage area do not affect the normal behavior of the WorkHeap. They are treated as finished tasks and cannot be reassigned. Thus this storage area does not affect the size of the WorkHeap.

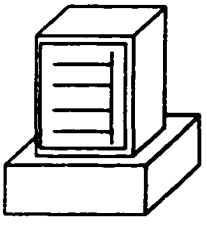
6.2.3 Deployment Enhancement

6.2.3.1 System Setup Information

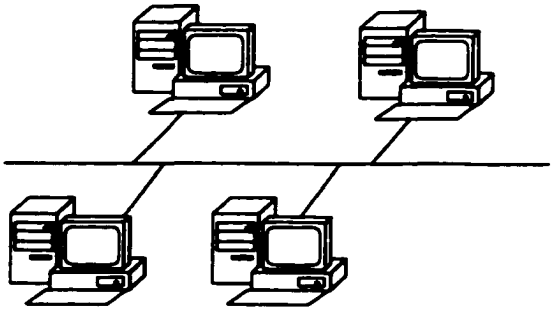
In the SWC framework, both global name-value pairs and the system setup information are put indiscriminately on GlobalManager, a global name space. Information, such as the master's host address, the routers' host addresses, and the port numbers associated

with each server, is dumped to GlobalManager. The E-SWC system introduces additional objects—`MasterData` and `RouterData` to manage system setup information. This change is necessary for efficient communications between multiple interconnected servers to ensure consistent and coherent system configuration. It cleans up the protocol between the servers and system modules in a logical fashion. It also reduces the security risks in the initial SWC release (Information stored on GlobalManager propagates to volunteer applets; hence, the system setup information is accessible to volunteer applets). In the E-SWC system, extraction of setup information is done in a specific module that separates the system core from its configuration. This is necessary for configuring servlet supports and benefits the framework implementation by untangling the specific requirements of various execution environments.

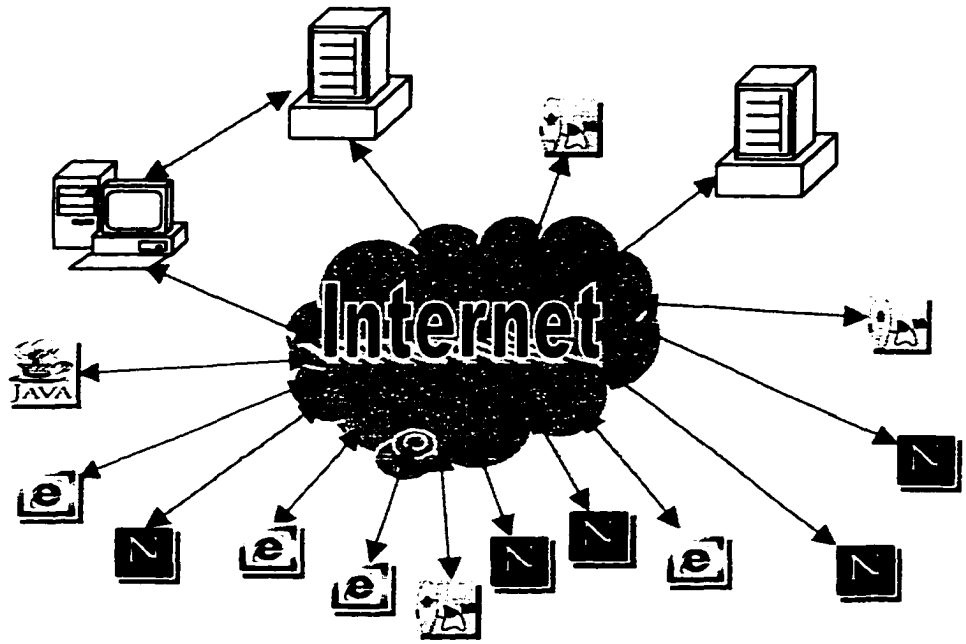
Due to the limitations on process implementation, the initial framework supported multi-process application only on a single machine. The cleaning up of system setup information eliminated this limitation. The E-SWC framework can run as threads on an SMP machine, as distributed processes on a network of workstations (NOW), and as applets in a WebComputing environment, see Figure 6-4.



a. SWC can run on an SMP machine as threads.



b. SWC can also run on a collection of networked workstations as distributed processes.



c. SWC runs on a WebComputing setup.

Figure 6-4, The Settings for the E-SWC Framework

6.2.3.2 Servlet Enhancement for System Deployment

The changes in system setup interfaces allow a servlet enhancement to be incorporated into the framework. This enhancement consists of three servlet modules: *ControlServlet*, *AppletServlet*, and *RouterServlet*, see Figure 6-5.

The ControlServlet provides users with a Web interface for setting up the system execution environment (see Appendix A for User's Guide). A user may specify the host names on which the master and the routers would reside, along with their port specifications. The user may also use the servlet to automatically generate a system configuration, load an existing configuration, save a newly specified configuration, and start or stop an application.

RouterServlet is responsible for starting and stopping the SWCMaster module and the SWCRouter modules based on the instructions issued by the ControlServlet.

AppletServlet is responsible for assigning upcoming volunteer clients to the group of SWCRouters. This servlet provides a coarse-grained load-balancing scheme as described in section 6.2.2.2.

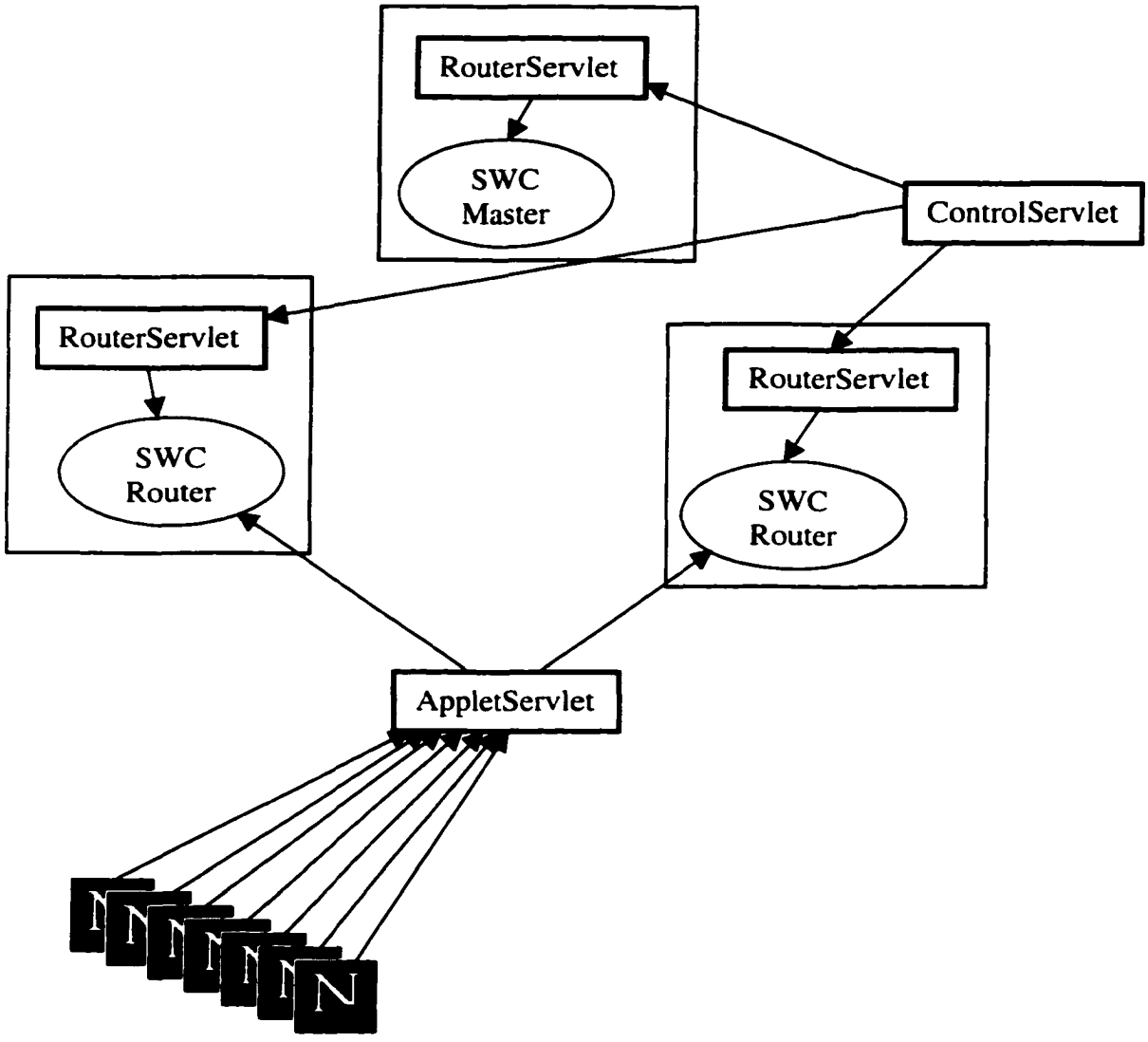


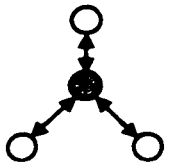
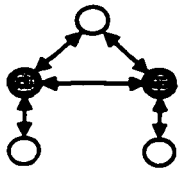
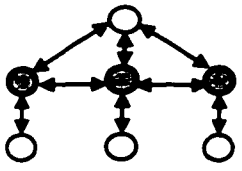
Figure 6-5, Servlet Architecture for Web Deployment

6.3 Outcomes of the Enhancements

Scalability is essential for building WebComputing systems. The performance of a single server diminishes quickly as the number of volunteers grows. This chapter presents a simple approach to support scalability. Based on the layered framework design, these enhancements focus on the lower-level architecture and leave the top-level framework API intact. The modifications can be classified into two categories: multi-router framework redesign and system-wide deployment using servlet technology. The design is aimed at simplicity by providing system-wide changes without massive replacement of the existing building blocks. In fact, all the changes were provided in self-contained modules and the design impacts were kept to a minimum. Using a simple and effective algorithm at each level, the scalability-enhanced E-SWC framework provides four levels of load-balancing schemes in a distributed fashion.

To determine the performance benefit of the enhancements, we tested the E-SWC framework with distributed processes on a NOW environment (a collection of Solaris Ultra-Sparc-5s connected with a 10 Mbps Ethernet—a normal departmental faculty network). It is a one-to-one binding of the processes (SWCMaster, SWCRouter, and SWCWorker processes) and the workstations. We used a communication intensive application for this test to quantify the server performance bottleneck. The distributed tasks were processed negligibly on the workers and returned back to the routers. The results are shown in Table 6-1. The same application was run on four different setups, as shown in the left-most column of the table. By introducing additional servers (shown as shaded circles), we were able to boost performance significantly. The standard deviations (σ) of the performance are also shown in the right most column.

Table 6-1, A Scalability Performance Test of the E-SWC Framework

Setup	Performance (ms)	σ
	448,532	25,006
	189,511	15,826
	144,805	10,409

The current SWC (or E-SWC) framework implementation has been tested and run successfully with volunteers using the browsers and operating systems shown in Table 6-2. Because Netscape's commitments to open standards, we expect the framework to run on Netscape 6.0, which will be available sometime this year (2000).

Table 6-2, Browser Platforms on Which the Current SWC Framework Runs

	UNIX (Solaris)	Windows NT	Macintosh
Netscape 4.5	X	X	
Internet Explorer 4			X
Internet Explorer 5			X

Chapter 7

Conclusions

7.1 Contributions

Because of the latency and bandwidth limitations of the Internet, communication efficiency is a primary challenge facing WebComputing. A system designer must make informed decisions in selecting communication protocols. This thesis presents a Communication Protocol Evaluation (CPE) of UDP, TCP, and RMI protocols, based on available Java packages. The detailed evaluation can be use as a guideline for forthcoming system design and application development.

A WebComputing system must not only provide efficient communication and adapt to the dynamic execution environment, but also support high-level application programming interfaces. This thesis presents the SWC framework with both performance and simple programming API in mind. It presents a two-layered system design – the run-time system and the abstract programming interface. The layered design makes its applications portable and provides freedom for robust architecture

development. At the upper layer, the SWC framework provides a simple coherent programming interface. Application programmers simply extend four predefined classes of the framework using a Master-Worker programming paradigm. Because of the robust lower layer implementation, the same application program can run as threads, as processes, and as Java applets for WebComputing. This provides a versatile distributed programming environment. First, an SWC application may be developed and tested on thread- and process-based platforms. Then the same program may be deployed and executed on large-scale process- or Java applet-based platforms.

Based on this design, we provided a scalability enhanced E-SWC system. This system incorporates load-balancing schemes at four levels using simple, yet effective, algorithms. The changes in architectural design emphasized simplicity and effectiveness. The E-SWC framework enhanced the scalability of the initial SWC system, and the changes are transparent to an SWC application.

7.2 Usage

Although SWC applications must be written in Java and can easily be developed in an object-oriented fashion, the framework is equally "friendly" to a procedural or imperative style. In particular, it is not terribly difficult to port C or even Fortran applications to run as SWC programs. Several projects (including Monte Carlo computations and classical Operational Research projects) are underway using SWC. Two Computer Science courses in the spring of 1999 were taught using SWC and it is worth noting that students —graduate and undergraduate—took to it easily.

7.3 Future Research

As stated earlier, WebComputing system design is in its infancy. Much research will be developed in the next few years as network technologies develop. Because of its close tie to the Internet, WebComputing research will greatly benefit from technology developed for the Internet. Researchers must pay close attention to the technologies that could further advance WebComputing system development, such as Internet security research and high-speed network development. Communication latencies and bandwidth limitations may diminish in the future. Next Generation Internet will provide a much broader and faster platform for WebComputing systems than are currently available.

There are several promising areas for the future development of the SWC framework. The system can be extended to support multiple SWC application programs. One way to achieve this is to share the SWCRouter module between multiple applications. However, to do this, the current application task-tracking scheme must be revised. At the same time, volunteer applets may be automated to support multiple applications in a fair fashion. This would require new system-wide scheduling mechanisms.

In a WebComputing setup, mutual trust between resource consumers and resource donors is virtually nonexistent. Therefore, corresponding security measures should be incorporated into the SWC system. The system may also enforce certain security mechanisms on behalf of the application developers. For example, the system could use random testing and/or majority vote mechanisms to ensure that a volunteer is trustful and only the correct results returned by volunteers are accepted.

Another area in which research could be fruitful is the examination of currently available parallel programming models, such as Linda tuple space programming model [CG,90] or even remote thread programming models for parallel programs, to assess the feasibility for supporting such programming interfaces without compromising performance while fully exploiting the layered architecture design.

The SWC system uses a Java servlet scheme to facilitate deployment. This imposes an additional requirement for making an HTTPD daemon available on every server. To eliminate this requirement, a skeleton server module could be introduced. If the essential functionality of an HTTPD server is available and can be packed with the SWC modules, the server-side services may also be deployed to the volunteer side of the computing resources.

Another interesting performance factor that needs to be investigated is the server congestion break point. Many factors could quickly diminish the performance of a server. Factors, such as the cost for Java object serialization, could be significant and should be made known to system developers. Publishing such results could help system developers to be more aware of system limitations and plan accordingly.

7.4 Final Remark

In conclusion, design and performance are the two interplaying complementary aspects of a WebComputing system. The performance prescribes the WebComputing system design; the design has significant impacts on the system's performance. The proposed research above is intended to provide a guideline for future WebComputing research and development.

Appendix A

SWC Framework User's Guide

1. Introduction to the SWC Framework

SWC is a small framework for developing and deploying WebComputing applications. This framework supports a form of MIMD programming model using a "Master-Worker" programming paradigm. The parallelism is achieved through distributing multiple copies of a worker program as Java applets across networks and executing them remotely in parallel. The SWC framework is an ongoing project and its programming interface is subject to change in subsequent releases.

2. Basic Programming Specification

First, an SWC application needs to import the SWC package, which provides the high-level programming interface of the SWC framework. Basically, a user needs to provide two application-specific modules: a *Master* module and a *Worker* module using the Master-Worker programming paradigm. Each module may contain a collection of Java classes. However, the application master must extend the `SWCMaster` class and implement `beTheBoss()` method. The application worker must extend the `SWCWorker` class and implement `doWork()` method.

On the master side, the application master may initialize a set of tasks in `beTheBoss()` method, wrap them in work units and send them to the workers using `sendWork()` method inherited from the `SWCMaster` class. The work units are application-specific, defined by a class that implements `SWCWorkUnit` interface. Computational results from the workers are received with a blocking call to `receiveResult()` method, also inherited from the `SWCMaster` class. This method returns an application-specific result unit, which is a class that implements the `SWCResultUnit` interface. Upon receiving the result units, the master integrates the results and/or defines new tasks.

On the worker side, a worker may receive an assigned task from the master with a blocking call to the `receiveWork()` method and does the work in the `doWork()` method. Analogous to the master, the worker inherits the `receiverWork()` and `send()` methods from the `SWCWorker` class. Two types of packages may be sent back to the master: a result package and/or an new task package. The result package should be sent back as an application-specific result unit, while the incomplete task should be sent back as an application-specific work unit. Incomplete tasks are rescheduled by the SWCRouter automatically, without going through the master. The `SWCWorker` class provides two send methods in the current SWC release. The `send()` method should be used for any results or work units that are not the last unit for the current task; while, the `sendLast()` method should be used to send the last unit for the current assigned task. This method indicates completion of the current task. Figure A-1 shows the SWC API.

```
public interface SWCUnit extends Serializable {}

public interface SWCWorkUnit extends SWCUnit {}

public interface SWCResultUnit extends SWCUnit {}


abstract class SWCMaster {
   final void sendWork(SWCWorkUnit swu) {…}
   final SWCResultUnit receiveResult() {…}
   abstract void beTheBoss();
}


abstract class SWCWorker {
   final synchronized void send(SWCUnit su) {…}
   final synchronized SWCWorkUnit receiveWork() {…}
   abstract void doTheWork();
}
```

Figure A-1, The SWC Framework API

3. Execution Interfaces

The SWC system consists of three primary modules: the SWCMaster, the single or multiple SWCRouters, and the SWCWorkers. The SWCRouter, which is responsible for distributing parallel tasks to SWCWorkers, is provided by the system. The SWCMaster

and the SWCWorker modules are application-specific and are provided by the SWC system users.

An SWC application may be run in three different execution environments, as threads on a Symmetric Multiprocessor (SMP) machine, as processes on a collection of interconnected workstations (NOW), or as applets in a WebComputing setup. An SWC application, named Sum, is used as an example in the following sections. The complete Sum program is provided in the following Example section.

Before developing an SWC application, a user should properly install the package on the system he/she intends to use. In addition, the package location should also be included in Java CLASSPATH environment variable.

3a. As Threads

To execute the application as threads, a user may issue command at a shell prompt.

```
$ java SWC.SWC Sum -w 3 MastersAddress=localhost
MastersPortGroup=15000 R0_RoutersAddress=localhost
R0_RoutersPortGroup=15200 n=100
```

This command starts the SWC system running application `Sum` with three worker threads. In addition, this command also specified the hosts and the port group starting numbers for both the master thread and the router thread. Additional name value pairs may also be appended at the end of the command line, like "n=100". These name-value pairs are stored as global variables.

3b. As Processes

The three primary modules of the SWC system: the SWCMaster, the SWCRouter, and the SWCWorker, must be invoked separately on each of the hosts. For example, a user wants to run the SWCMaster on host *hamilton*, SWCRouter 1 on host *turing*, and SWCRouter 2 on host *russell*. The user should log on to *hamilton* and issue the command:

```
[hamilton]$ java -DYouAre=Master SWC.SWC Sum
MastersPortGroup=15000 MastersAddress=hamilton
NumRouter=2 R0_RoutersAddress=turing
R0_RoutersPortGroup=15200 R1_RoutersAddress=russell
R1_RoutersPortGroup=15300
```

Also, the user should issue the commands on host *turing* and *russell* respectively.

```
[turing ]$ java -DYouAre=R0 SWC.SWC Sum
MastersPortGroup=15000 MastersAddress=hamilton
NumRouter=2 R0_RoutersAddress=turing
R0_RoutersPortGroup=15200 R1_RoutersAddress=russell
R1_RoutersPortGroup=15300
```

```
[russell]$ java -DYouAre=R1 SWC.SWC Sum
MastersPortGroup=15000 MastersAddress=hamilton
NumRouter=2 R0_RoutersAddress=turing
R0_RoutersPortGroup=15200 R1_RoutersAddress=russell
R1_RoutersPortGroup=15300
```

At this point, the SWC system is deployed on three server machines and ready to accept inbound SWCWorkers. To start a worker, the user may log on to an available host and issue the command:

```
[newton]$ java -DYouAre=Worker0 SWC.SWC Sum
RoutersAddress=turing RoutersPortGroup=15200
```

Each SWCWorker needs only to know the SWCRouter's information that it communicates to. The SWCMaster and other SWCRouter's setup information may be omitted.

3c. As WebComputing Applets

To execute the application as applets, the users are expected to do additional setup. Three Java servlets should be installed for the deployment system. They are ControlServlet, AppletServlet, and RouterServlet. ControlServlet should be installed on the server that the SWC application developers can access. AppletServlet should be installed on the server that volunteer donors know and visit. RouterServlet should be installed on every server that is part of the SWC computation, including the host that runs the SWCMaster module.

Figure A-2 shows the initial page for a user to start or create a setup configuration.

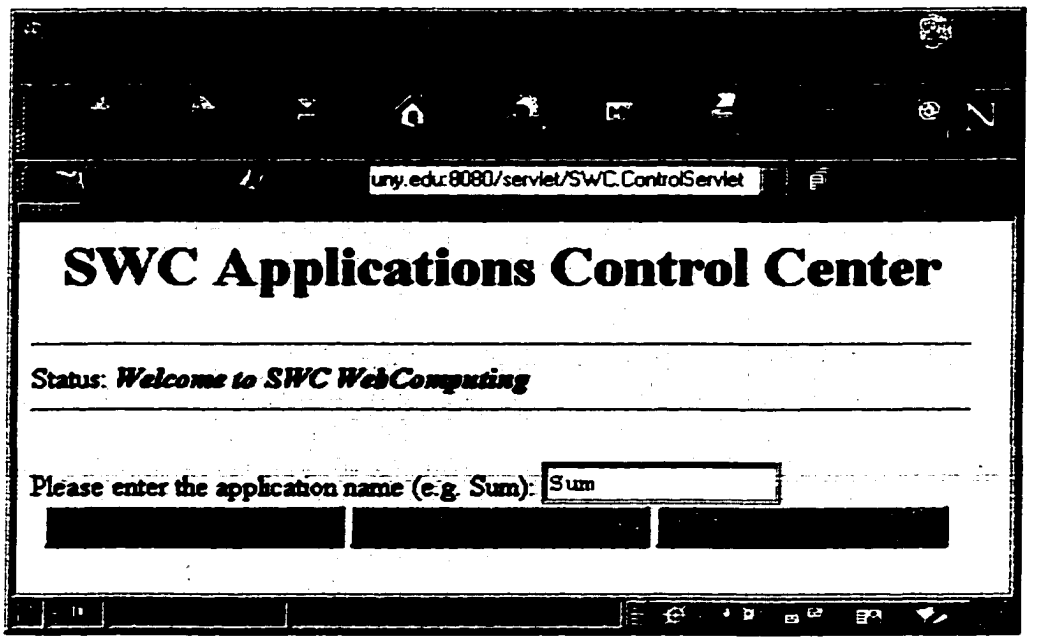


Figure A-2, SWC ControlServlet Starting Page

The user should supply the name of the application. The user may choose to load an existing setup configuration or to create a new setup for the application. ControlServlet brings the user to a new page, shown in Figure A-3. This page contains all the setup information needed to start an SWC application. In the case that the "New Configuration" button is clicked in Figure A-2, the system will provide a default setup. The user may choose to use this setup or change the information that appears in textboxes. After configuring the setup information, the user may start an execution by clicking on the "Start the Application" button.

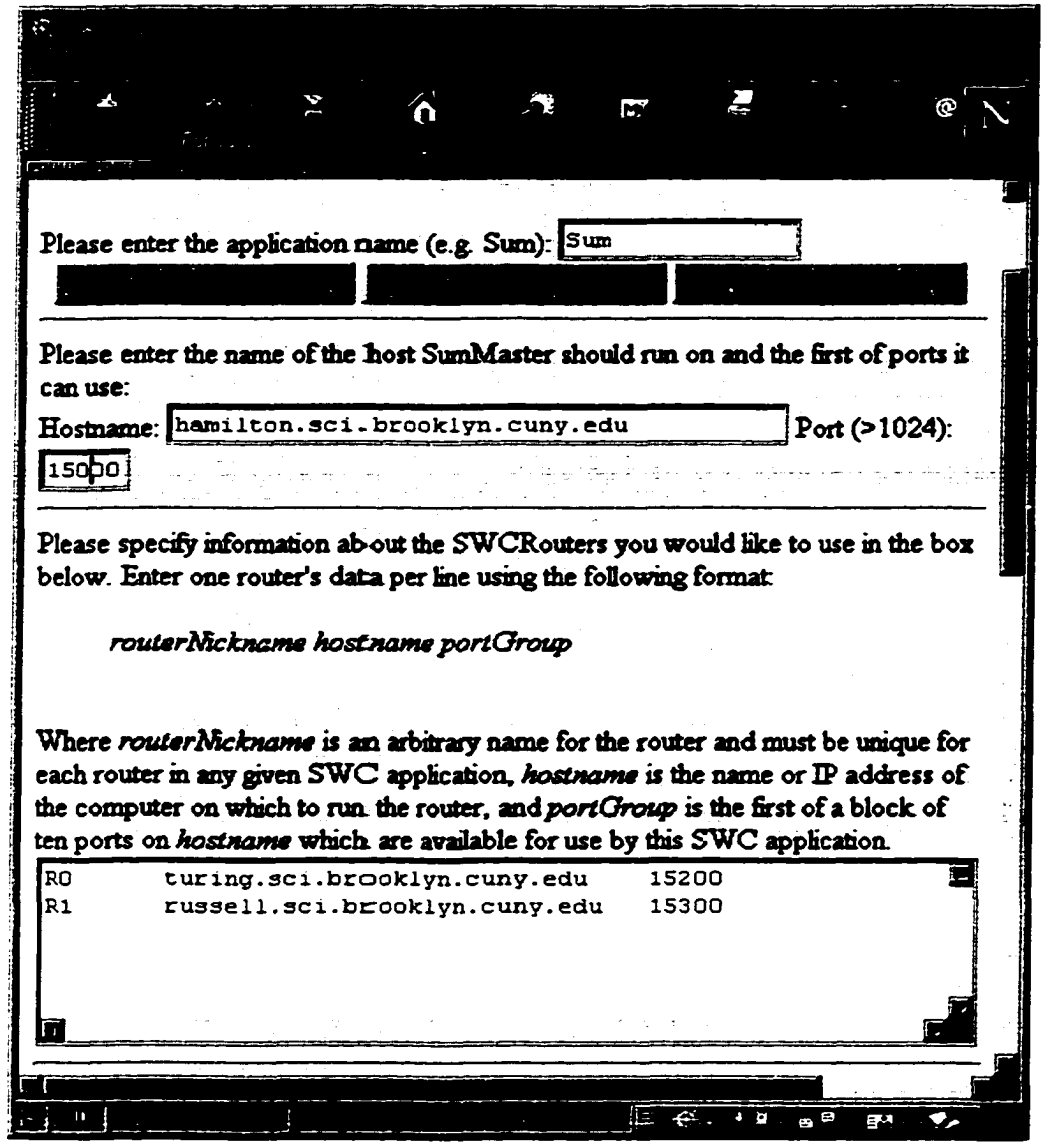


Figure A-3, SWC System Setup Information Configuration Page

4. SWC Application Class Naming Convention

An SWC system requires the users to follow a simple naming convention for developing SWC applications. In the current release, the SWC system requires the application developers to choose a name for the application and use it as a prefix for the class

naming. The application master and worker classes must be suffixed with "Master" and "Worker" respectively. For example, if the application name is XYZ, then the application classes must be defined as following:

```
XYZWorker
XYZMaster
XYZWorkUnit
XYZResultUnit
```

(Note: SWC users are required to follow the naming convention for both master and worker classes. Naming for WorkUnit and ResultUnit is optional).

5. Example

Following is a complete SWC sample application. It computes the summation of n integer numbers in a distributed fashion. Four essential SWC classes are implemented.

```
//: SumMaster.java
import SWC.*;
import java.io.*;

public class SumMaster extends SWCMaster {
  public void beTheBoss() {
    int n = 1000;
    for(int i = 1 ; i < n; i += 2) {
      sendWork(new SumWorkUnit(i, i+1, 1, 1));
    }
```

```
    SumResultUnit sru = (SumResultUnit)receiveResult();
    while(sru.nR < n) {
      SumResultUnit sru2 = (SumResultUnit)receiveResult();
      sendWork(new SumWorkUnit(sru.r, sru2.r, sru.nR,
                               sru2.nR));
      sru = (SumResultUnit)receiveResult();
    }
    System.out.println("The Answer is "+sru.r);
  }
}
```

```
//: SumWorker.java
import SWC.*;
public class SumWorker extends SWCWorker {
  public void doWork() {
    while(true) {
      SumWorkUnit swu = (SumWorkUnit)receiveWork();
      SumResultUnit sru =
              new SumResultUnit(swu.x+swu.y,
                                swu.nX+swu.nY);
      sendLast(sru);
    }
  }
}
```

```
//: SumWorkUnit.java
import SWC.*;
class SumWorkUnit implements SWCWorkUnit {
  SumWorkUnit(int x, int y, int nX, int nY) {
    this.x = x;
    this.y = y;
    this.nX = nX;
    this.nY = nY;
  }
  int x, y;               // #'s to add
  int nX, nY;             // # of values used to form x & y
}
```

```
//: SumResultUnit.java
import SWC.*;
class SumResultUnit implements SWCResultUnit {
  SumResultUnit(int r, int nR) {
    this.r = r;
    this.nR = nR;
  }
  int r;                  // result of addition
  int nR;                 // # of values used to form result
}
```

Glossary

Glossary of the SWC Framework:

Actor. An actor is an active thread that interacts with other Router's Actors. As a whole, they manage the task flow of an SWCRouter.

BeenThere. One of the Router's Actors. It eliminates duplicated returning packages from the router's workers and forwards them to the Reaper and the Distinguisher.

Communication Protocol Evaluation (CPE). See Chapter 4 for details.

ControlManager. A Router's Actors introduced in the E-SWC framework. It is a self-contained module that is responsible for load balancing among the routers.

CPE. See Communication Protocol Evaluation.

Distinguisher. One of the Router's Actors. Upon receiving packages from the BeenThere, it forwards them to their corresponding destinations depending on the package types.

E-SWC—Enhanced SWC framework. See Chapter 6 for details.

GlobalManager. An object resides in an SWCRouter. It contains global variables as name-value pairs.

Sender. A framework module that sends data onto the network. There are two types of senders: TCP Sender and UDP Sender.

Reaper. One of the Router's Actors. It removes finished tasks from the WorkHeap.

Receiver. A framework module that receives data from the network. There are two types of Receivers: TCP Receiver and UPD Receiver.

SWC—Small WebComputing framework. See Chapter 5 for details.

SWCMaster. Also called a master. It is the master module of the framework.

SWCRouter. Also called a router or a server. This module provides the conceptual framework and oversees the communication between the SWCMaster and the SWCWorker. It provides task-scheduling, task-tracking, and load-balancing supports to upper-level applications.

SWCWorker. Also called a worker. It is the worker module of the framework. It is distributed across networks and constitutes the parallel computation.

WebComputing. An approach to distributed computing that uses Java applets to automatically distribute a computation across the Internet.

WorkHeap. An object residing in an SWCRouter. It stores unfinished tasks for distribution.

Glossary of Acronyms:

API—Application Programming Interface. A set of protocols and tools for building software applications.

CGI—Common Gateway Interface. A specification for transferring information between an HTTP server and CGI programs, which can process this information dynamically and provide interaction between a user and a Web server.

CORBA—Common Object Request Broker Architecture. An architecture that enables communications between networked objects, regardless of what programming language they were written in or what operating systems they are running on.

CR&EW—Concurrent Read and Exclusive Write. A method used in parallel computing to ensure correct concurrent access to shared data or memory.

DBMS—DataBase Management System. A set of programs that enables database access and modification.

DHCP—Microsoft Dynamic Host Configuration Protocol. A protocol for assigning dynamic IP addresses to devices on a network.

DNS—Domain Name System. An Internet service that translates domain names into IP addresses.

DP. A distributed system developed by Dr. David Arnow.

DSM—Distributed Shared Memory. A service for distributed memory management provided in some distributed systems.

GUI—Graphical User Interface. A graphic interface for user-computer interaction in place of command-based interaction.

HTML—HyperText Markup Language. The set of rules that govern the construction of a Web page.

HTTP—HyperText Transport Protocol. The protocol used by Web servers and their clients.

HTTPD. HTTP daemon process.

JDK—Java Development Kit. Sun's software development kit for developing Java programs.

JIT—Java Just-In-Time compiler. A code generator that converts Java bytecode into machine language instructions.

JNI—Java Native Interface. A Java API allows a Java program to access the language of a host system and determines the way Java integrates with the native code.

JVM—Java Virtual Machine. A self-contained operating environment for Java programs. It behaves as if it is a separate operating system.

LAN—Local Area Network. A general-purpose local network that connects a variety of devices.

MAN—Metropolitan Area Networks. A data network that is larger than a LAN and smaller than a WAN in terms of geographic breadth.

MIMD—Multiple Instruction stream, and Multiple Data stream. In a parallel machine or a distributed system, each processor is capable of executing a program independent of the other processors.

MPI—Message Passing Interface. A message-passing programming interface defined for distributed computing. A number of systems were implemented based on this interface.

MTU—Maximum Transmission Unit. The largest physical packet that can be transferred on a network.

NIS/NIS+ —Network Information System. An information system that manages network related data, such as host names and IP addresses.

NOW—Network Of Workstations. A targeted platform for distributed systems.

PVM —Parallel Virtual Machine. A traditional distributed system first developed at Oak Ridge National Laboratory during summer 1989.

RAM—Random Access Memory. A type of computer main memory.

RMI —Java Remote Method Invocation. A set of protocols for communication between distributed Java objects.

SMP—Symmetric Multi-Processor. SMP architecture is a tightly coupled multiprocessor system in which processors share a single copy of the operating system and resources that often include a common bus, memory and an I/O system.

TCP—Transmission Control Protocol. One of the main protocols in TCP/IP networks. It is a connection-oriented protocol and provides reliable services.

TCP/IP. A suite of communication protocols used to connect hosts on the Internet.

UDP—User Datagram Protocol. Unlike TCP, UDP is a connectionless protocol that runs on top of IP protocol. It does not guarantee reliable services.

URL—Universal/Uniform Resource Locator. A unique identification of a network resource.

WAN —Wide Area Networks. A network that spans a relative large geographic area.

WINS—Microsoft Windows Internet Name Service. A system that provides name resolution, especially for Windows systems.

Glossary of Technical Terms:

Distributed System. A software system that supports parallel computing on a collection of networked computers.

Java Applet. A program designed to be executed from within another application, such as a Java-enabled Web browser.

Java Application. A Java program designed to be executed directly on a JVM. It starts with the "main" thread of control.

Load Balancing. Parallel computing needs to balance workload among the processors in order to achieve the maximum performance. See [KGGK,94] for more information.

LWP-Light Weight Process. A Solaris' kernel-level object of execution. See [ZY,98] for detail.

Parallel Computer. A computer with multiple CPUs.

Sandbox Security Model. A set of security measures that are imposed on Java applets by a Java-enabled Web browser.

Scalability. The ability to achieve performance proportional to the number of processors used in a parallel system.

Serial Computer. A computer with a single CPU.

Servlet. Generic extensions to Java-enabled Web servers. A servlet is like a CGI program, but it is in Java and is cheaper than a CGI program.

Thread. An encapsulation of the flow of control in a program. A classical process may contain multiple executing threads. Each thread has its own context and may share memory address and other computing resources with other threads. It is much cheaper than a classical process and used extensively in operating systems that support SMP architecture.

Bibliography

[AISS,97] A.D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. *SuperWeb: Research Issues in Java-Based Global Computing*, Concurrency: Practice and Experience, June 1997.

[Arnow,95] D. Arnow, *DP - A Library for Building Reliable, Portable Distributed Programming Systems*, Proceedings of the USENIX Winter '95 Technical Conference, New Orleans, January 1995.

[AWYC,99] David Arnow, Gerald Weiss, Kevin Ying, Dayton Clark. *SWC: A Small Framework for WebComputing*. Proceedings of the International Conference on Parallel Computing (ParCo99). Delft, Netherlands, August 1999.

[BBB,96] J. Baldeschweiler, R. Blumofe, and E. Brewer, *ATLAS: An Infrastructure for Global Computing*, Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, Connemara, Ireland, September 1996.

[BCHL,98] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. *mpiJava: A Java interface to MPI*. Presented at First UK Workshop on Java for High Performance Network Computing, Europar 1998.
http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html

[BKKW,96] Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. *Charlotte: Metacomputing on the Web*, Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996.

[BKKK,97] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem. *KnittingFactory: An Infrastructure for Distributed Web Applications*. TR1997-748, Computer Science Department, New York University.

[BKKK2,98] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem. *An Infrastructure for Network Computing with Java Applets*,

Proceedings of ACM Workshop on Java for high-performance Network Computing, February 1998.

[BPMK,99] Ben Petrazzini and Mugo Kibati, *The Internet in Developing Countries*, Communications of the ACM, Vol. 42, No. 6, June, 1999.

[CC,99] T. Chuang and D. Chen. *Distributed Middle-tier: A Programming Model for Web-based Scalable Java Computing*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Vol. 2, August, 1999.

[CCINSW,97] P. Cappello, B. Christiansen, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu, *Javelin: Internet-based parallel computing using Java*, the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1997.

[CG,90] Nicholas Carriero and David Gelernter, *How to Write Parallel Programs : A First Course*, MIT Press, 1990.

[Cilk] Cilk project: http://supertech.lcs.mit.edu/cilk/

[DistNet] Distributed Net: http://www.distributed.net/

[Ferrari,97] Adam J. Ferrari, *JPVM: Network Parallel Computing in Java*, Technical Report CS-97-29, University of Virginia, Charlottesville, Virginia, December 1997. JPVM project URL: http://www.cs.virginia.edu/~ajf2j/jpvm.html

[GBDJMS,94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[Hayes,98] Brian Hayes, *Collective Wisdom*, American Scientist, Vol.86, No.2, March-April, 1998, Page 118-122.

[Hirano,97] S. Hirano, *HORB: Distributed Execution of Java Programs*, Proceeding WWCA'97, Lecture Notes in Computer Science, Vol. 1274 (Springer, Berlin, 1997) 29-42. http://www.horb.org/

[HYI,98] Satoshi Hirano, Yoshiji Yasu, and Hirotaka Igarashi, *Performance Evaluation of Popular Distributed Object Technologies for Java*, ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[KGGK,94] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Commings, 1994.

[LB,96] B. Lewis and D. Berg, *A Guide to Multithreaded Programming - Threads Primer*, SunSoft Press 1996.

[Lilja,93] Lilja, David J. *Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons*, ACM Computing Surveys. 25.3 (Sept. 1993): 303-335.

[Mangione,98] C. Mangione, *Performance tests show Java as fast as C++*, Java World, Feb. 1998

[Netlib,98] The world's largest supercomputer—ASCI Red at Sandia National Laboratories: http://www.netlib.org/benchmark/top500.html

[NGI] Next Generation Internet (NGI) Initiative: http://www.ngi.gov/

[NinJa,98] Ninja RMI implementation: http://ninja.cs.berkeley.edu/

[Pasquale,96] J. Pasquale, *Towards Internet Computing*, ACM Computing Surveys, 1996.

[Sarmenta,98] Luis F. G. Sarmenta. *Bayanihan: Web-Based Volunteer Computing Using Java.* 2nd International Conference on World-Wide Computing and its Applications (WWCA'98), Tsukuba, Japan, March 3-4, 1998.

[Schimmel,94] Curt Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison Wesley, 1994.

[SETI,99] Search for Exraterrestrial Intelligence (SETI). SETI@HOME http://setiathome.ssl.berkeley.edu/, 1999

[SH,99] Luis F. G. Sarmenta, Satoshi Hirano. *Bayanihan: Building and Studying Volunteer Computing Systems Using Java.* To appear in Future Generation Computer Systems Special Issue on Metacomputing, Vol. 15, No. 5/6. Elsevier Publication, 1999.

[SndNL] Sandia National Laboratories: http://www.sandia.gov/

[SOHWD,96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1996.

[SunJava] Sun Java Web site: http://java.sun.com/

[SunJNI] Sun Java Native Interface: http://java.sun.com/products/jdk//1.2/docs/guide/jni/index.html

[SunJVM] Sun's Java Virtual Machine Specification: http://java.sun.com/docs/books/vmspec/index.html

[SunRMI] The Java Remote Method Invocation (RMI) Home Page: http://java.sun.com/products/jdk/rmi/index.html

[SunServlet] Java Servlet Development Kit (JSDK): http://java.sun.com/products/servlet.

[SunWeb,99] Sun Web site: http://www.sun.com/servers/enterprise/, 1999.

[Thompson,96] Thompson, T., *The World's Fastest Computers*, Byte, January 1996.

[Thurman,96] D. Thurman, *JavaPMV: The Java to PVM Interface*, December 1996. JavaPVM is renamed jPVM, URL: http://www.isye.gatech.edu/chmsr/jPVM/

[YAC,99] Kevin Ying, David Arnow, and Dayton Clark. *Evaluating Communication Protocols for WebComputing*. Proceedings of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), CSREA Press, Las Vegas, July, 1999.

[YC,98] N. Yalamanchilli and W. Cohen, *Communication Performance of Java-based Parallel Virtual Machines*, ACM Workshop on Java for High-Performance Network Computing, 1998.

[ZY,98] Fabian Zabatta and Kevin Ying, *A Thread Performance Comparison: Windows NT and Solaris on A Symmetric Multiprocessor*, Proceedings of the 2nd USENIX Windows NT Symposium, August 1998.