JAVA &
Parallelism/Real-time systems

Thesis: 429

by:
D.F. Nooren

Supervisor:
Dr. Th.E. Schouten
Faculty of Mathematics and Computer Science
Section Experimental Computer Science for Technical Applications
University of Nijmegen

April 9, 1998

Preface

This thesis is the result of the graduating project which I fulfilled in the research line Experimental Informatics for Technical Applications (EITA) under the supervision of dr. Th.E. Schouten.

I would like to take this opportunity to thank a number of people who, in one way or another, played a part in the completion of my study. First of all I would like to thank my parents, for supporting me through my years of education, dr. Th.E. Schouten, for his guidance, Lisenka, for her support and understanding, Joachim, for helping me with the layout, friends at the university, for the necessary coffee breaks and card games, and last but not least my friends at the Hertogstraat 125, for their support and helping me to make the right choice in the summer of 1995.

Danny Nooren, April 1998

Contents

Introduction

As a result of the technological developments, faster and more powerful computers are becoming available. Computers with parallel processors are getting cheaper, so they are becoming more available also. Another aspect of the technological developments is the programming language JAVA, which concentrates on the Internet and on the embedded market.

Because the expectations about the Internet are very high, it is interesting to find out in what way JAVA is suitable for parallel (or distributed) machines. With the recent release of a JAVA Chip and other JAVA related products its also interesting to find out in what way Java is suitable for embedded or real-time systems.

The research should give a clear image in what way JAVA is suitable for the use with machines with parallel processors and in what way it is suited for embedded or real-time systems at the moment and what the expectations might be.

Chapter 1
What is Java?

Java is the programming language developed by Sun and released in public alpha and beta versions in 1995. Java is used to create executable content that can be distributed through networks.

In order for users to use Java content, they must have a key piece of Java software; the Java interpreter[1]. To view Java content on the net, a Web browser must be Java-enabled. In the alpha release of Java, available during the summer of 1995, only one browser called HotJava could interpret programs created by the Java language. Other brands of Web browsers have since been upgraded to be able to interpret Java programs, most notably, the Netscape Navigator Web Browser.

Java makes animation, interaction, computation, distributed applications, and new forms of communication possible. Through protocol and content handlers, Java has the potential to make new formats and new protocols available for use on the Web. But Java is not only a programming language. Since the release of Java in 1995, many other Java products have been developed such as a Java Chip and a Java Operating System.

1.1 Java as a Programming Language

According to the information provided by Sun Microsystems [34], Java is a

> ... simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

This characterization identifies the key technical features of Java as shown in de following sections.

1.1.1 Simple

Java is based on the C++ programming language, but many of the C++ language features have been removed by the developers of Java. C++

[1] Also called JVM (Java Virtual Machine), see [30] for more information about the Java Virtual Machine Specification

is a language for object-oriented programming and offers very powerful features. However, as in the case with many languages designed to have power, some features often cause problems. Programmers can create code that contains errors in logic or is incomprehensible to other programmers trying to read it. Because the majority of the cost of software engineering is often code maintenance rather than code creation, this shift to understandable code rather than powerful but poorly understood code can help reduce software costs. JAVA differs from C++ (and C) in these ways:

- JAVA does not support the `struct, union` and `pointer` data types.
- JAVA does not support `typedef` or `#define`.
- JAVA differs in its handling of certain operators and does not permit operator overloading.
- JAVA does not support multiple inheritance.
- JAVA handles command-line arguments differently than C++ or C.
- JAVA has a `String` class as part of the `java.lang` package. This differs from the null-terminated array of characters as used in C++ and C.
- JAVA has an automatic system for allocating and freeing memory (garbage collection), so it is unnecessary to use memory allocation and de-allocation functions as in C++ and C.

1.1.2 Object-Oriented

JAVA supports an object-oriented approach to writing software. JAVA is pure object-oriented: no functions outside classes, and no global variables. Ideally, object-oriented design can permit the creation of software components that can be reused.

Object-oriented programming is based upon modeling the world in terms of software components called *objects*. An object consists of data and operations that can be performed on that data called *methods*. These methods can encapsulate, or protect, an object's data because programmers can create objects in which the methods are the only way to change the state of the data.

Another quality of object-orientation is *inheritance*. Objects can use characteristics of other objects without having to reproduce the functionality in those objects that supports those characteristics. Inheritance thus helps in software re-use, because programmers can create methods that do a specific job exactly once.

Another benefit of inheritance is software organization and understandability. By having object organized according to classes, each object in a class inherits characteristics from parent objects. This makes the job of documenting, understanding, and benefiting from previous generations of software easier, because the functionality of the software has incrementally grown as more objects are created. Objects at the end of a long inheritance chain can be very specialized and powerful.

1.1.3 Distributed

Unlike the languages C++ and C, Java is specifically designed to work within a networked environment. Java has a large library of classes for communicating using the Internet's TCP/IP protocol suite, including protocols such as HTTP and FTP. Java code can manipulate resources via URLs as easily as programmers are used to accessing a local file system using C++ or C.

1.1.4 Interpreted

When the Java compiler translates a Java class source file into bytecodes, this bytecode class file can be run on any machine that runs a Java interpreter or Java-enabled browser. This allows the Java code to be written independently of the users' platforms. Interpretation also eliminates the compile and run cycle for the client because the bytecodes are not specific to a given machine but interpreted.

1.1.5 Robust

Robust software doesn't 'break' easily because of programming bugs or logic errors in it. A programming language that encourages robust software often places more restrictions on the programmer when he or she is writing the source code. These restrictions include those on data types and the use of pointers. The C programming language is lax in its checking of compatible data types during compilation and runtime. C++ was designed to be more strongly typed than C; however, C++ retains some of C's approach toward typing. In Java, typing is more rigorous: a programmer cannot turn an arbitrary integer into a pointer by casting, for example. Also, Java does not support pointer arithmetic but has arrays instead. These simplifications eliminate some of the "tricks" that C programmers could use to access arbitrary areas of the memory. In particular, Java does not allow the programmer to overwrite memory and corrupt other data through pointers. In contrast, a C programmer often can accidentally (or deliberately) overwrite or corrupt data. Java also enables the programmer to catch exceptions.

1.1.6 Secure

Because JAVA works in networked environments, the issue of security should be of concern to developers. In its present form, JAVA puts limits on pointers so that developers cannot forge access to memory where not permitted.

In traditional environments, companies could protect themselves by controlling physical and network access to their computers by establishing policies for what kinds of software can be used on their machines. These steps include building a firewall between the Internet and the company's intranet, obtaining software only from known and trusted sources, and using anti-virus programs to check all new software.

Use of applets potentially adds new security vulnerability. An employee searching an external Web for information might inadvertently load and execute an applet without being aware that the site contains executable code. This automatic distribution of executables make it very likely that software will be obtained from untrusted third parties. Since the applet is imported into the user's browser and run locally, this software could potentially steal or damage information stored in the user's machine on a network file server. Also, since this software is already behind the company's firewall, the applet could attack other unprotected machines on a corporate intranet. These attacks would not be stopped by traditional security measures.

JAVA protects its users from these dangers by placing strict limits on applets. Applets cannot read from or write to the local disk. Stand-alone windows created by applets are clearly labeled as being owned by untrusted software. These limits prevent malicious applets from stealing information, spreading viruses, or acting as Trojan Horses. Applets are also prohibited from making network connections to other computer on the corporate intranet. This prevents malicious applets from exploiting security flaws that might exists behind the firewall or in the underlying operating system. These aspects of JAVA enable a more secure software environments.

In January 1998, Sun released the latest JAVA Cryptography Extension package. This package, a standard extension to the JAVA Development Kit software (JDK), is a comprehensive cryptography toolkit that creates a standard set of application programming interfaces (APIs) for advanced JAVA software-based encryption technologies. The security model core to the next release of JDK software provides permission-based access control of JAVA applets and applications; i.e., different users can be assigned differing levels of access to system resources.

The JCE software package includes cryptography technology standard to all encryption tools, such as symmetric cipher technology (utilizing DES and triple-DES) and key agreement technology (via Diffie-Hellman).

In addition, the JCE software package provides features unique to JAVA application software such as secure JAVA streaming, which allows for the transfer of data across platforms as well across protocols (http and other network I/O, file I/O, etc). According to Sun microsystems [34], the JCE software package also brings all the benefits of being based on an object-oriented language and allows for the encryption of objects in addition to traditional data an information.

For now, the JCE software package is not freely exportable outside of the United States and Canada, although companies and institutions that work within the restrains of the United States federal law (e.g., obtaining export licenses) can export JCE-based applications and products.

1.1.7 Architecture Neutral

The JAVA compiler creates bytecodes that are sent to the requesting browser and interpreted on the browser's host machine, which has the JAVA interpreter or a JAVA-enabled browser installed.

1.1.8 Portable

The quality of being architecture neutral allows for a great deal of portability. However, another aspect of portability is how the hardware interprets arithmetic operations. In C and C++, source code may run slightly different on different hardware platforms because of how these platforms implement arithmetic operations. In JAVA, this has been simplified. An integer type in JAVA, `int`, is a signed, two's complement 32-bit integer. A real number, `float`, is always a 32-bit floating-point number defined by the IEEE 754 standard. These consistencies make it possible to have the assurance that any result on one computer with JAVA can be replicated on another.

1.1.9 High-Performance

Although JAVA bytecodes are interpreted, the performance sometimes isn't as fast as direct compilation and execution on a particular hardware platform. JAVA compilation includes an option to translate the bytecode into machine code for a particular hardware platform. This can give the same efficiency as a traditional compile and load process. According to Sun Microsystems testing, performance of this bytecode to machine code is "almost indistinguishable" from direct compilation from C++ or C programs.

1.1.10 Multi-Threaded

The JAVA language supports so-called multi-threading. Several threads that share a single address space may run concurrently. The most straightforward way to obtain a new thread in JAVA is by calling the method `start()` on an instance of a subclass of `java.lang.Thread` that overrides the `run()`-method of this latter class with the appropriate code. This causes a new thread to execute this `run()`-method while the original thread continues asynchronously with the code that follows the invocation of `start()`. A thread can wait for completion of another thread by calling the method `join()` on this thread.

The JAVA programming language supports a synchronization mechanism that is based on Hoare's [19] monitor and condition paradigm. Mutual exclusion amongst different threads that call methods of the qualifier '`synchronized`' on the same object is guaranteed. Once a lock of a particular object has been acquired, a thread can suspends itself by calling the `wait()`-method on this object. This will release the lock of this object allowing other threads to obtain the lock. If a thread that has acquired the lock of a particular object calls the `notify()`-method, an arbitrary thread that has formerly been suspended by calling the `wait()`-method on this object will awake again, although it must re-acquire the lock of the object before it may continue. Likewise, calling the method `notifyAll()` will awake all suspended threads, each of which must re-acquire the lock before it may continue.

An implementation of the JVM [30] generally uses pre-emptive, priority based scheduling for threads, where the currently running thread always has highest priority among all runnable threads, and as soon as a thread with higher priority becomes runnable, the running thread is pre-emptive in favor of the latter. However, the specification[18] does not guarantee fairness amongst runnable threads with equal priority. Furthermore, the specification requires that any variable that is read or written to *synchronously* is explicitly loaded and stored back at acquiring and releasing locks, so that changes to shared data become visible to other threads that access these variables by means of synchronized methods. *Asynchronously* accessed variables, on the other hand, must be declared as `volatile` to prevent the compiler from optimizations that disable the visibility of such changes, e.g. by keeping a variable that is only read in a register. A variable declared as `volatile` will not be cached by any thread, and any update to a volatile variable by a thread is immediately visible to all other threads[2].

[2] The term `volatile` adopted by JAVA designates a variable as noncacheable, whereas in parallel programming this term has traditionally meant that a variable is cacheable but may be updated by other processors.

1.1.11 Dynamic

Unlike C++ code, which often requires complete re-compilation if a parent class is changed, JAVA uses a method of interfaces to relieve this dependency. The result is that JAVA programs can allow for new methods and instance variables in objects in a library without affecting their dependent client objects.

1.2 JAVA as a Chip

Today the average business person relies on more than 10 microcontrollers. It is expected that by the end of this decade, the average home will contain between 50 and 100 microcontrollers. In fact, there will be millions of cellular phones, set top boxes, personal digital assistants, low-cost network terminals, and other Internet appliances operating in a networked environment and highly optimized for small applications running at top speed. Although Sun Microelectronics [32], a division of Sun Microsystems, claims that JAVA technology is quickly establishing itself as a standard for the rapidly expanding market of low-cost, embedded network computers that will soon populate the enterprice, office and home, many embedded-systems developers are still skeptical about JAVA. While some embedded-systems developers are using JAVA to design embedded-systems, other developers are waiting to see where things are going. The role of JAVA in embedded systems will be discussed in section 2.3. Sun Microelectronics has taken the JAVA phenomenon one step further by casting it into silicon. They have successfully implemented the JAVA code execution in hardware, yielding a boost in performance.

1.2.1 The picoJava Core

The picoJava core directly executes the JAVA instruction set, eliminating the need for existing software layers that impact price or performance. Until now, JAVA code had to run either through an interpreter (see figure 1.1,left), which considerably slows execution, or with a JIT (Just In Time) compiler (see figure 1.1,middle) that can speed execution but expands the code size by a factor of three or more, which is a big problem for embedded applications with their restricted memory requirements.

According to Sun Microelectronics [32, 33], The JavaChip architectures are microprocessors which are optimized for JAVA and a better overall performance and small memory footprint. To boost throughput, picoJava directly executes the JAVA intruction set, eliminating the need for an interpreter or JIT compiler (see figure 1.1,right). The processor is

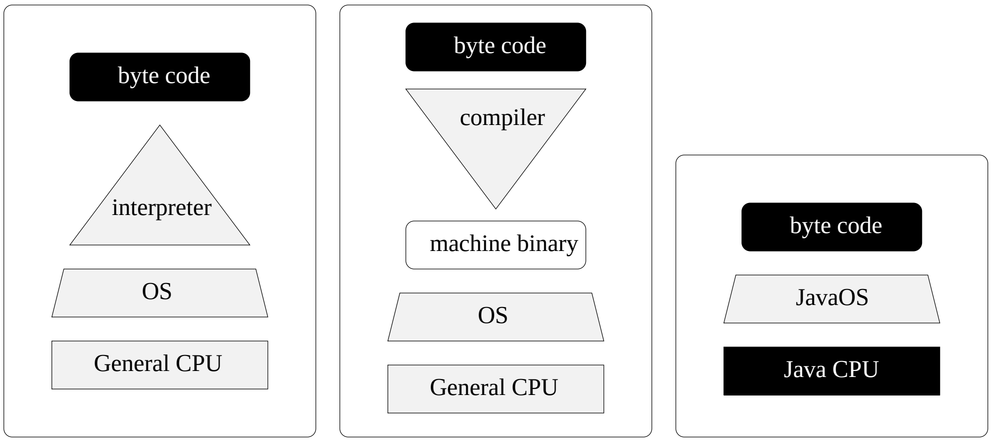

Figure 1.1: Byte code interpreted, compiled, and direct execution

tailored to JAVA requirements, offering 100% conformity with the JAVA Virtual Machine specification [30].

1.2.2 picoJava Architecture

Streamlined for maximum run-time performance, picoJava is a simple architecture as seen in figure 1.2. According to Sun Microelectronics [32], great care was taken to make the core easy to customize for embedded applications, with variable sized caches and with or without a floating point unit along side a standard integer unit.

The picoJava core includes a RISC-style pipeline and a straightforward instruction set. Only those instructions that directly improve JAVA execution are implemented in hardware. JAVA bytecodes are stored in an instruction cache that can range from zero to 16 kbytes in size. A 12-byte instruction buffer decouples the instruction cache from the main execution unit. All instructions consists of an 8-bit opcode and zero or more operand bytes as defined in the JAVA Virtual Machine Specification [30]. Up to 5 bytes at the head of the instruction buffer can be decoded and sent to the next stage of the 4 stage RISC-like pipeline for execution in a single cycle. There is an on-chip stack cache used by the execution unit to access operands needed by the functional units, which is the equivalent of a register file used in most RISC designs. The stack contains both integer and floating point data and there is no overhead for passing operands to and from the stack for either unit. This facilitates moving data between the floating point and integer unit. The data cache, like the instruction cache, can range in size from zero to 16 kbytes. To improve the hit rate it is designed as a two-way, set-associative mem-

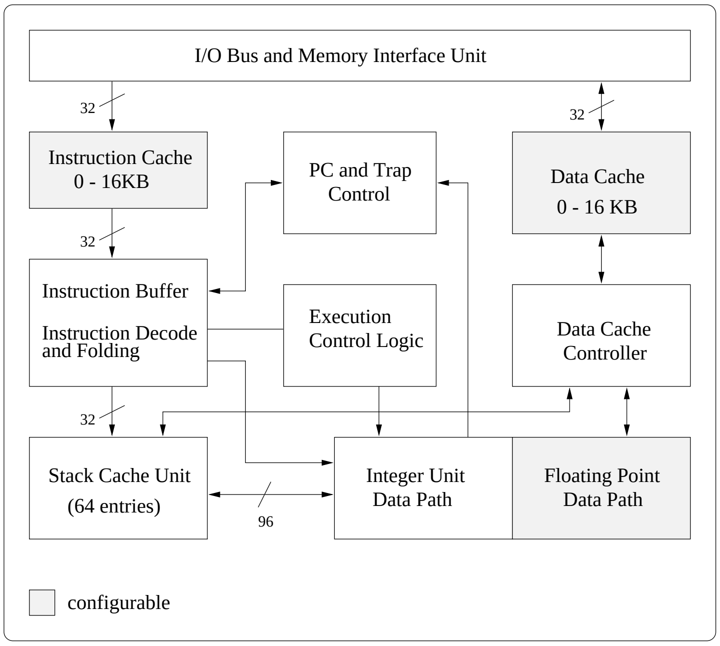

Figure 1.2: picoJava block diagram

ory. The datapath between the picoJava data cache and the pipeline is 32 bits. An I/O bus and memory interface unit serves as the link between the picoJava core and any other interfaces.

According to Sun Microelectronics [32], with the configurable instruction and data caches, and the option to include or exclude the floating point unit, embedded designers can customize the core processor to meet their specific area, performance and power requirements. Furthermore, the unique implementation of a hardware stack and a creative approach to decoding helps picoJava deliver exceptional performance and execution throughput.

1.2.3 picoJava Performance

The picoJava core architecture gives designers maximum flexibility when building a JAVA-based application. Depending on cost and performance goals, the designer can choose the appropriate I/O and functional blocks for the target environment. As mentioned in section 1.2.2, picoJava allows variable-sized instruction and data caches and the option of including or excluding a floating point unit. This means that developers of cost-intensive applications can take advantage of these configurable features, which can be critical in embedded systems with restricted footprints.

The picoJava architecture was specifically designed to run complex, multi-threaded and object-oriented JAVA code. Because previously proposed JAVA benchmark suites did not offer these characteristics, two JAVA programs were chosen as benchmarks for the analysis of the performance. They provide the advanced programming activity likely to be found in a typical JAVA application. The two benchmarks are the JAVA compiler `javac` with over 25,000 lines of JAVA technology in 170 different classes and a raytracer with substantial floating point activity (3,500 lines in 32 different classes). To prove that picoJava does indeed deliver the expected performance gains, Sun Microelectronics simulated a representative suite of performance benchmarks with the picoJava core. Specifically they looked at JAVA code running native on the picoJava core and compared it with the same code executed on Intel's 486 and Pentium using both a JIT compiler and an interpreter. The picoJava's performance was emulated using a simulator for the core. All the measurements were scaled to the same clock rate of 100 MHz.

		JIT compiler	JIT compiler	Interpreter	Interpreter
	picoJava /100	Pentium /100	486 /100	Pentium /100	486 /100
`javac`	1.8s	9.3s	10.7s	20.4s	27.3s
raytracer	13.0s	64.5s	109.5s	174.3s	254.8s

Table 1.1: Execution times after scaling to 100 MHz

Table 1.1 lists the results from the benchmarks after scaling the execution times to 100 MHz. Since the picoJava simulator does not accurately simulate I/O, a 0.8 second penalty, based on experiments with a Sun Sparc system running the JAVA interpreter, was added (and 0.4s for the raytracer). The table shows, for example, that picoJava is 15 to 20 times faster than a 486 with an interpreter at equal clock rates, and still 5 times faster than a Pentium with a JIT compiler at equal clock rates.

Sun Microelectronics state, that this performance advantage gives an embedded design team a remarkable degree of flexibility when designing an embedded application around the picoJava core. For example, if minimal power consumption is critical, the designers could run the core ten times slower to reduce power requirements. And, since picoJava is 5 to 20 times faster on JAVA applications, power dissipation can be 5 to 20 times less than on a conventional CPU doing the same function. Although the performance gains will vary depending on the size of the caches and whether or not the floating point unit is included, picoJava would still deliver substantial performance improvement over other leading embedded processors when using Java applications.

Further informations about the systems used for the benchmarks can be found in [33]. Information about coming microprocessor chips microJava and ultraJava, which are based on the picoJava core and are specially designed for embedded Java systems, can be found at [36].

1.3 Java as an Operating System

The JavaOS operating system is a small, efficient and dynamically extensible operating system that brings the design advantages of the Java program language platform to an operating system, which executes Java applications directly on hardware platforms. According to Sun Microsystems [34], JavaOS has been designed specifically to support Java and Java applets without having to provide backward compatibility with other legacy applications. This approach improves performance by eliminating needless layers of software and enabling the operating system to run in minimal memory configurations. JavaOS enables users to run the same applications across many different hardware platforms, including the x86, SPARC and ARM microprocessor architectures. And because JavaOS device drivers are written in Java, they can easily be ported or dynamically extended. JavaOS supports the Java language's multi-threaded capabilities and tailored memory management, enabling it to run multiple threads at the same time so that users can download networked information while also running Java applications. Java products using JavaOS can be booted from ROM, over a network, or from hard disks. Network computers can boot JavaOS and download it into memory from a server, while handheld products can embed JavaOS into ROM for instant-on capabilities.

JavaOS employs a layered architecture which is divided into platform-specific and platform-independent code, see figure 1.3. Each layer of the architecture can be independently updated. The platform-specific code, which is compiled to the native code, consists of the kernel and the Java Virtual Machine. The platform-independent part contains the device drivers, the JavaOS Graphics and JavaOS Windowing systems, networking classes and support for the full Java application programmer's interface (API).

The JavaOS Microkernel supports booting, interrupt handling, multiple threads, traps and DMA handling, enabling users to run multiple applets at the same time or download information while running a Java application. The JavaOS Virtual Machine supports the Java bytecode interpreter loop, execution handling, memory management, threads, class loading, and a bytecode verifier. JavaOS extends the memory model to

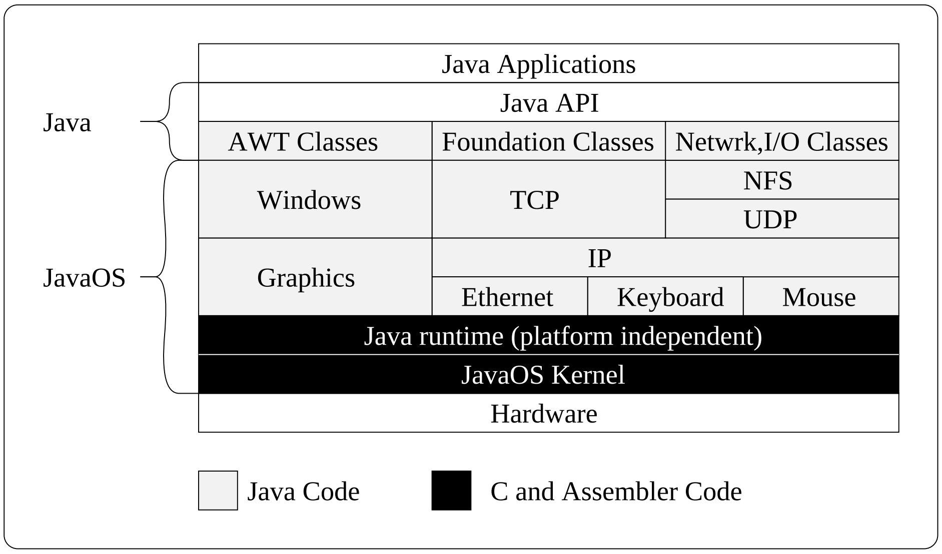

Figure 1.3: Java Platform running on JavaOS

optimize memory usage and accommodate low-memory conditions.

JavaOS Device Drivers are written in Java and are portable and extensible. Network Classes in JavaOS, also written in Java, include standard networking protocols such as TCP/IP, UDP and ICMP for basic transport and routing. And JavaOS enabled clients can access files on a NFS server and be managed using SNMP. The JavaOS Window System manages all drawing to the screen, and The JavaOS Graphics System supports all common graphic calls. Both the graphics and window subsystems are crafted to support JAVA's Abstract Windowing Toolkit (AWT) in a memory-efficient way. JAVA Application Environment (JAE) Support enables the same applets and applications to run on other servers or platforms running the JAVA Virtual Machine.

CHAPTER 2
Embedded and Real-Time JAVA

According to the designers at Sun Microsystems, is JAVA also suitable for Embedded or Real-Time systems. Before proceeding further it is worth trying to define the phrase 'real-time system' more precisely.

2.1 Definition of a real-time system

There are many interpretations of the exact nature of a real-time system; however, they all have in common the notion of response time, the time taken for the system to generate output from some associated input. The Oxford Dictionary of Computing [48] gives the following definition of a real-time system. A real-time system is...

> Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Here, the word timeliness is taken in the context of the total system. For example, in a missile guidance system output is required within a few milliseconds, whereas in a computer-controlled car assembly line the response may be required only within a second.

The definition of a real-time system according to Young [49] is: A real-time system is...

> any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period.

In their most general sense both these definitions cover a very wide range of computer activities. For example, an operating system like UNIX may be considered real time in that when a user enters a command he will expect a response within a few seconds. Fortunately, it is usually not a disaster if the response is not forthcoming. These types of systems can be distinguished from those where failure to respond can be considered just as bad as a wrong response. Indeed, for some, it is this aspect that distinguishes a real-time system from others where response time is important but not crucial. According to Burns and Welling [9] depends the correctness of a real-time system not only on the logical result of the computation, but also on the time at which the results are produced.

Real-time systems can be distinguished between hard and soft real-time systems. Hard real-time systems are those where is it absolutely imperative that responses occur within the specified deadline. Soft real-time systems are those where response times are important but the system will still function correctly if deadlines are occasionally missed.

In a hard or soft real-time system the computer is usually interfaced directly to some physical equipment and is dedicated to monitoring or controlling the operation of that equipment. A key feature of all these applications is the role of the computer as an information processing component within a larger engineering system. Applications like these are also known as embedded computer systems [9, 41, 49].

2.2 Characteristics of a real-time system

According to Burns and Welling [9], a real-time system possess many special characteristics which are identified in the following sections.

Clearly, not all real-time systems will exhibit all these characteristics, however, any general-purpose language which is to be used for the effective programming of real-time systems must have facilities which support these characteristics.

2.2.1 Size and complexity

Most of the problems associated with developing software are those related to size and complexity. Writing small programs presents no significant problem as they can be designed, coded, maintained and understood by a single person. If that person leaves the company or institution, then someone else can learn the program in a relatively short period of time. Burns and Welling [9] even say that there is an art or craft to their construction and *'small is beautiful'*. Unfortunately, not all software exhibits this most desirable characteristic of smallness.

Real-time systems react on events in the real world, but the real world is continuously changing and evolving. Therefore real-time systems have to be maintained constantly. Although real-time software is often complex, features provided by real-time languages and environments enable these complex systems to be broken down into smaller components, which can be managed effectively. The JAVA programming language as mentioned in sections 1.1.1 and 1.1.2 being a simple and object-oriented language provides this 'required' maintainability of 'simple' smaller components.

2.2.2 Manipulation of real numbers

In order to calculate what changes must be made to the input variables, so that a desirable effect on the output takes place, it is necessary to have a mathematical model of the plant which is being controlled. Often, a plant is modeled as a set of first order differential equations. These link the output of the system with the internal state of the plant and its input variables. Changing the output of the plant involves solving these equations to give the required input variables.

Most physical systems do not respond immediately when the input has changed. If a certain change in the output is required within a fixed time-period, it will add to the complexity of the mathematical model and the physical system. The fact that, in reality, linear first-order equations are only an approximation to the actual characteristics of the system also presents complications.

Because of these difficulties, the complexity of the model and the number of distinct inputs and outputs, most controllers are implemented as computers. However, the introduction of a digital component changes the nature of the control cycle. Input and output of the controller/ computer are now discrete values.

The computer can solve the differential equations by numerical techniques, taking into account that the plant outputs are now being sampled. The control algorithms can be mathematically complex and require a high degree of precision. A fundamental requirement of a real-time programming language, therefore, is the ability to manipulate real or floating-point numbers. As mentioned in section 1.1.1, the JAVA programming language is based on the C++ programming language. Although some powerful features have been removed, the JAVA programming language still supports floating point arithmetic operations, see section 1.1.8.

2.2.3 Reliability and safety

The more that society let the control of vital functions be handled by computers, the more important it becomes that those computers do not fail. Reliability is an important issue in real-time systems. I.e., when an error occurs, the systems should not shut down completely and endanger human lives. The system should only be allowed to fail in a controlled way, i.e. if possible, with interaction with the system operator. According to Sun Microsystems, as mentioned in sections 1.1.5 and 1.1.6, The JAVA programming language with its restrictions on datatypes and its capability to catch exceptions, is a robust and a secure language.

2.2.4 Concurrent control of separate system components

An embedded system will tend to consist of computers and several co-existing external elements with the computer must simultaneously interact. It is the nature of these external real-world elements that they exist in parallel. Fortunately, the speed of modern computers is such that usually the actions of real-world events may be carried out in sequence but give the illusion of being simultaneous. In some embedded systems, this may not be the case, i.e. where the data is to be collected and processed at various geographically distributed sites, or where the response time of the individual components cannot be met by a single computer. In these cases it is necessary to consider distributed and multiprocessor embedded systems.

A problem with the production of software for systems where concurrency is required is how to express that concurrency in the structure of the program. When a programming language is used which does not support concurrency, it will bring extra complications. For the programmer this means that the already difficult task becomes more complicated, because he/she must consider activities which are irrelevant for the controlling task. It makes it also more difficult to prove the correctness of program and will not make the program code easier, resulting in less maintainability.

Programming languages that support concurrency are Ada, Mesa and Occam, and of course the relatively new JAVA. Lately there are more programming languages in development that support concurrency such as Tempo, lyric, A,UM-90, PICT, Oz, Joyce/Linda and many others. It is not clear when each programming language is out of development. The JAVA programming language supports multi-threading for the control of separate system components (see section 1.1.10), avoiding that the programmers task becomes more complicated.

2.2.5 Real-time facilities

Response time is crucial in any embedded system. It is difficult to design and implement systems which will guarantee that appropriate output will be generated at the appropriate times under all possible conditions. To do this and make full use of all computing resources at all times is often impossible. Therefor real-time systems are usually constructed using processors with considerable spare capacity, thereby ensuring that 'worst-case behavior' does not produce any unwelcome delays during critical periods of the system's operation.

When a programmer can specify times at which actions are to be performed and when actions are to be completed, and can respond to situations where the timing requirements cannot be met, then we can speak of real-time facilities. They enable the program to synchronize with time itself. For example, it may be necessary to samples readings from sensors at certain periods of the day or at regular intervals, for example, every 5 seconds.

In order to meet response times it is necessary for a system's behavior to be deterministic and, moreover, predictable. Although the JAVA programming language provides the class `java.util.Date`, which represents a specific instant in time with millisecond precision, the determinacy is a problem when using JAVA in real-time systems. This problem is discussed in section 2.3.2.

2.2.6 Interaction with hardware interfaces

The nature of embedded systems requires the computer components to interact with the external world. They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers. Devices may also generate interrupts in order to signal to the processor that certain operations have been performed or that an error has occurred. In the past, the interfacing was done by the operating system or the programmer had to control and manipulate registers and interrupt at assembly level. Because of the variety of devices and the time-critical nature of there associated interactions, their control must be direct, and not through a layer of operating system functions. Furthermore, reliability requirements argue against the use of low-level programming techniques. with the `java.io` package, the JAVA programming language supports a wide variety of I/O classes.

2.2.7 Efficient implementation

Since real-time systems are time-critical, efficiency of implementation will be more important than in other systems. It is interesting that one of the main benefits of using a high-level language is that it enables the programmer to abstract away from implementation details, and concentrate on solving the problem at hand. A programmer of an embedded system cannot afford this luxury, and must be constantly concerned with the cost of using particular language features. For example, if a response to some input is required within a microsecond there is no point in using a language feature whose execution takes a millisecond!

2.3 JAVA in Embedded systems

Embedded systems provide much of the intelligence in today's 'smart' devices. Microwave ovens, telephones, even toaster ovens increasingly use them. And factories, transportation systems, and most of the rest of the infrastructure of modern life use them in massive numbers. Embedded systems rely on microprocessors and microcontrollers[1]. Although there still is a significant market for 4-bit microcontrollers, increasingly the business is moving to 8-, 16-, and 32-bit controllers. In higher-end jobs, such as controlling a laser printer or running the sensors and controllers on a machine tool, embedded systems often rely on full-blown RISC processors. Intel's i960 series has been a big hit in the embedded systems market, and Sun has versions of its SPARC processors that are widely used as well.

According to Ripps [41, 42], Currently most embedded programs are coded in C and assembly. Some major problems of C for embedded systems are the permissiveness of C operations, which can lead to undisciplined coding practices and ultimately to unstable execution. And the absence of universal standards for multithreading, shared data protection, and intra-thread communication and coordination, which can make the program hard to transfer to alternate platforms. JAVA solves these problems.

While some companies expect or hope that JAVA will supersede C as the primary programming language for embedded systems in the near future, there is still an enormous number of lines of C code in operation. Companies will have to work with that code for many years as the transition to JAVA runs its course. Many systems will have to be a mixture of legacy C code and JAVA enhancements. It is not trivial to integrate an overall application with some components written in JAVA and others in C or assembler. Part of the problem arises from security issues. JAVA can not guarantee the security of the system if execution disappears into 'unknown' regions of code. For example, the non-JAVA code could make OS support service calls, that alter the application's threading and data-protection aspects.

The JAVA interpreter expects to be able to load any needed class libraries and code components from a disk. Many embedded systems are housed within a highly-constrained box, and have to operate in the harsh factory floor environment. Lack of space, power or cooling, or excessive vibration may preclude a standard hard or floppy disk. The embedded application should be structured in such a manner that the JAVA programs can start from ROM disk. The JavaOS, as mentioned in section

[1] a computer on a chip that combines a microprocessor with some ROM, RAM, I/O channels, and perhaps some other specialized devices.

1.3, can be tailored in such a manner, so that it can boot from ROM, over a network, or from hard disks. The problem concerning size and speed is discussed in section 2.3.1. Another problem concerning Real-Time is discussed in section 2.3.2.

Aiming at the Embedded Market, Sun Microsystems released the Java Card, PersonalJava and EmbeddedJava API's, designed for embedded applications.

2.3.1 Size and Speed

By the standards of embedded systems where designers are used to working with 8-bit controllers with a few thousand bytes of ROM and RAM, Java seems big and power-hungry. The resource issue is inherent in Java. With Java, space for the Java Virtual Machine and decoding all the classes on the fly has to be allowed. And if a JIT compiler is added in order to save time, memory consumption will be increased. PersonalJava (see section 2.3.4), for example, might take up a megabyte of RAM for the JVM and classes.

Java is also slow, which becomes a critical limit if the timing of the system is tight (see section 2.2.5). Better Java interpreters, JIT compilers, compiled Java, and Java processors will speed up execution considerably.

In an embedded system where an Instruction Set Architecture (ISA) other than one of the new Java-chips is used, it seems there is little benefit for Java's universal executable format. Because there is a significant time penalty for execution by interpretation, the Java source is likely to be compiled into the ISA-specific instructions of the target hardware, as is currently done with C. Although during program development it may be convenient to debug individual Java components by executing them on any available Java-capable platform, such as a PC or SPARC workstation, it can be expected that in most cases the embedded program that finally goes out in the field will have been compiled into native code, where as a result, no Java interpreter and libraries of class code is needed. Nevertheless, some embedded programs may benefit from retaining the capacity of dynamically load new Java source code. Those are applications that must be altered partially or totally while in the field, because, as mentioned in section 2.2.1, the real world has changed and evolved and so, i.e., the process control needs to be changed also.

Another way to shrink the size and boost speed of Java is to use a microprocessor that realizes the Java Virtual Machine. In August 1997, Rockwell Collins produced the JEM1, the first Java processor, which was designed not for sale, but to power its own avionics systems. Patriot Scientific Corp. has a Java processor that is a port of the JavaOS to the company's own 32-bit 100MHz PSC1000 processor, which sells for less than

$10 in high volumes. As mentioned in section 1.2, Sun Microsystems is planning to release JAVA processors microJava and ultraJava. No release date has been set yet.

2.3.2 The Real-Time Problem

Although JAVA was originally designed for embedded systems, there is a fundamental irony in the fact that it is not well adapted to hard real-time applications. The main reason is that JAVA's garbage collection feature interferes with the determinacy real-time systems demand (see section 2.2.5). There are, however, a lot of embedded and control applications which are not hard real-time. JAVA's portability, object orientation and ease of software development are extremely attractive for embedded systems and control applications that don't demand real-time operation. Although speed is important in real-time applications, the key issue is determinacy, or predictability. A real-time system must be able to perform an operation within a known time window and be able to do it every time that operation is called. Determinacy is where JAVA's garbage collection feature works against it. Garbage collection is one of the things that simplifies Java programming, so the programmer doesn't have to worry about de-allocating memory when an object has finished executing. But the programmer[2] can't fully control when garbage collection will execute and has no control over how long garbage collection will take to complete. This introduces a fundamental indeterminacy into JAVA that programmers must live with and vendors can only try to work around.

The usual way around these problems is to graft the JAVA virtual machine onto a real-time operating system and to write the real-time pieces of code in languages such as C or assembly. This is what most of the RTOS companies with JAVA offerings are doing. NewMonics Inc.[22] introduced a different approach: pERC, which replaces the JVM with a virtual machine that includes a couple of new constructs.

pERC provides specific enhancements to the JAVA environment, making JAVA more suitable for use in embedded applications, while retaining the many benefits of JAVA. pERC includes a JAVA-compatible Virtual Machine with an improved garbage collector, and selected class library support. pERC's Real-Time extensions will allow the deterministic execution of tasks. The pERC technology is JAVA-like rather than JAVA and it puts a load on the application, which has to decide how to handle possible indeterminacies as they occur.

The current version of JavaOS (1.0) as mentioned in section 1.3, still

[2] programmers have limited control, but if the JVM senses it is running out of memory it will initiate garbage collection

has this garbage collection problem.

2.3.3 JAVA Card

JAVA Card is a specification that enables the 'Write Once, Run Anywhere'[3] capabilities of JAVA on smart cards[4] and other devices with limited memory. Because of JAVA Card API applications written for one smart card platform can, for the first time run on any other smart card platform. JAVA Card API 1.0 was developed in conjunction with leading members of the smart card industry. Since the release of the JAVA Card API 2.0, it has been adopted by over 95% of the manufacturers in the smart card industry. According to Sun Microsystems, Its almost universal adoption by the smart card industry is due to several unique benefits that Java Card provides:

- Applications written for one card can run on all Smart Cards.
- Multiple Applications can run on a single card. JAVA's inherent design makes it easy to securely run multiple applications on a single card.
- Faster and simpler application development. Previously, applications for smart cards were written in assembly-code type languages. JAVA Card allows development in JAVA using off-the-shelf JAVA development tools.

The specification contains detailed information for building JAVA virtual machines in very small memory environments which have at least 16K of read-only memory, 8K of EEPROM, and 256 bytes of random access memory.

2.3.4 PersonalJava and EmbeddedJava

Also aiming at the Embedded Market, Sun Microsystems last year introduced PersonalJava and EmbeddedJava. PersonalJava is a new JAVA Application Environment for network-connectable applications on personal consumer devices for home, office, and mobile use. Like JAVA, PersonalJava consists of core and standard extension (optional) APIs. It has been designed specifically for resource limited environments, with the addition of specific features required by consumer applications. With the

[3] 'Write Once, Run Anywhere' is a Trade Mark of Sun Microsystems

[4] A smart card is a card embedded with either a microprocessor and a memory chip or only a memory chip with nonprogrammable logic. The microprocessor card can add, delete, and otherwise manipulate information on the card, while a memory-chip card, i.e. a phone card, can only undertake a predefined operation.

arrival of the World Wide Web, the need for consumer devices be web-connectable has grown, meaning they must support the download and execution of applets and applications. Many consumer devices have long life cycles, requiring support over a long period of time. JAVA is ideally suited for these devices, offering dynamic linking and loading which enables remote updates and patching of software. Also, JAVA saves code storage space since JAVA bytecodes are generally more compact than native machine code. PersonalJava runs on top of the JAVA Virtual Machine, ensuring that code runs across a variety of operating systems and microprocessors. According to Sun Microsystems, writing PersonalJava applications leverages the ease of development, code reuse, and inherent security built into the JAVA language. Applications in development for PersonalJava include hand-held computers, set-top boxes, game consoles, mobile hand-held devices, and smart phones to name a few.

EmbeddedJava is a new JAVA Application Environment for high volume embedded devices. Similar to Java, EmbeddedJava consists of core and standard extension APIs, and is designed specifically for severely resource constrained environments. PersonalJava and EmbeddedJava include a feature level subset of JAVA, therefore PersonalJava and EmbeddedJava applications are upward compatible to JAVA. EmbeddedJava applications run on real-time operating systems, and are optimized for the constraints of small memory footprints and diverse visual displays. EmbeddedJava enables manufacturers of devices to take advantage of the portability and flexibility of JAVA in their product software. Since embedded software must run in resource constrained environments, application development requires the ability to select only the functionality required for a specific product. For example, if a device (such as a pager) requires no file system, the file system support is left out. Following this principle, EmbeddedJava is designed to be highly modular, scalable and configurable, while requiring minimal memory. A design goal of real-time embedded applications such as network routers and switches, is to preserve the real-time support offered by the hardware in the final product. In other words, fine granularity performance is required. According to Sun Microsystems, for this reason, EmbeddedJava is designed to be used on top of a real-time operating system. Performance and porting considerations are built into its design.

2.4 JAVA in the Real World

With most of the embedded code written C and assembly, the general agreement is that JAVA's portion of the embedded systems market is miniscule and that it will become significant over the next several years.

The problems concerning size and speed can be solved in the near future as Korean manufacturer LG Semicon Co. expects to ship its version of the picoJava processor early next year. NEC Corp., which also has licensed the picoJava design from Sun, hasn't yet announced its plans. Fujitsu Microelectronics, Mitsubishi Electronics, Thomson Sun Interactive, and Toshiba Systems Group also reportedly are working on picoJava-based chips.

Sun announced that Apple Computer, Hewlett-Packard, Hitachi, IBM, Microsoft, Novell, Silicon Graphics, SunSoft, The Santa Cruz Operation (SCO), and Tandem will embed the Java platform into their operating systems.

Most of the major makers of software for embedded systems development are making JAVA part of their future. Along with many of the smaller firms in the industry, they have announced JAVA products or products that will work with JAVA. Wind River Systems offers a version of its Tornado development environment for JAVA. It is also integrating JAVA into VxWorks, its real-time operating system. The Wind River products include a port of Sun's JAVA Application Environment (JAE) to VxWorks. This includes the JVM and all the JAVA classes, so any JAVA application can execute on a device running VxWorks. Microtec, a division of Mentor Graphics, has also announced a suite of development tools for JAVA, including a JAVA compiler and an extension of its X-Ray development environment to include JAVA. The tools are scheduled to reach the market in early 1998. Cygnus Solutions has announced an entire suite of embedded systems tools for JAVA that should be available early in 1998. The Cygnus tools actually are based on the company's popular GNU tools for C and C++. The company has added a JAVA compiler, jcl, which is a front end for the gcc compiler and shares most of its code with the company's other front ends. The main advantage of this approach, aside from time-to-market for Cygnus, is that the tools already are available for most popular processors and controllers, giving JAVA developers a broad selection of target environments. Integrated Systems Inc. has teamed with NewMonics to use pERC as the basis of its JAVA development products. Besides pERC itself, NewMonics should have a tool suite available early in 1998. The tools include a JIT compiler, an execution time analyzer that is used to tune applications on a specific system, and a pERC-to-JAVA translator called Percolator.

It is not clear how these tools will deal with the Real-Time problem as discussed in section 2.3.2. Most likely these solutions are JAVA-like instead of 100% pure JAVA.

Visa and Sun have reached an agreement in which Visa will contribute to the development and promotion of a JAVA-based smart card microprocessor core based on the JAVA Card standard.

Accelerated Technology Inc. (ATI) and NSI Com are building a Real-Time, embeddedJava-based embedded JAVA Virtual Machine. In late 1998, Insignia Solutions also plans to deliver an Embedded Virtual Machine (EVM), which is, according to Insignia Solutions, designed for the rapidly growing JAVA embedded systems market, an d will offer a migration path compatible with legacy systems.

NEC is developing WebPanel, a portable touchscreen, JAVA-based Internet device designed for home and on-the-road use. It should be able to send/receive e-mail, surf the Web using Sun's HotJava browser, and execute Java applets. Mobile communication will take place at 32 kilobytes per second. The first prototype of the 133MHz MIPS-powered unit is expected by the end of March 1998. Future versions may be powered by upcoming JAVA chips from Sun.

Motorola has licensed the entire suite of JAVA platform technologies, and will use the JAVA platform as a cornerstone of new product development across its diverse product portfolio. Motorola ships many tens of millions of embedded silicon solutions and radio products a year worldwide as the market leader in various industries. This largest JAVA technology licensing agreement ever is an important step for JAVA in the embedded market.

Despite the Real-Time problem, as mentioned above, many embedded systems with less Real-Time contrains have been implemented using JAVA.

2.5 Conclusion

Although many of the existing embedded systems are coded in C and assembly, JAVA is increasingly being used for the development of embedded and real-time systems. The JAVA programming language does support many of the characteristics of a real-time systems, but has to overcome the shortcomings concerning size, speed and predictability to become a language for hard real-time systems. With the development of JAVA-based chips and JAVA-based RTOS, these problems might be solved in the near future.

Despite its resource demands and lack of real-time capability, JAVA is currently being used for the development of many embedded systems. As more communicating microprocessors will be used in a single appliance, JAVA, with its user interface, secure communication mechanism, and network capabilities has the potential to become the market leader in embedded systems in the near future.

CHAPTER 3
Parallel JAVA

As mentioned in subsection 2.2.4 it may be necessary, under certain circumstances, to consider distributed or multiprocessor systems. Some large parallel problems such as analysis of molecular sequence [46] are very hard to solve without parallel computing. The increasing performance and connectivity of ordinary workstations[1] and PCs make it possible that parallel computing will be done on local area networks, or even on the Internet in the near future. Distributed systems require that computations running in different address spaces, or even on different hosts, be able to communicate.

The JAVA language supports sockets as a basic communication mechanism, which are flexible and sufficient for general communication, but require the client and server to engage in applications-level protocols to encode en decode messages for exchange. The design of such protocols can be error-prone and usually result in extra work. With the interim release of JDK 1.0.2, Sun Microsystems, first introduced Remote Method Invocation (RMI) [35]. The release of JDK 1.1 includes full support of RMI.

3.1 Remote Method Invocation

According to Sun Microsystems [34], RMI enables the programmer to create distributed JAVA-to-JAVA applications, in which the methods of remote JAVA objects can be invoked from other JAVA virtual machines, possibly on different hosts. A JAVA program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses Object Serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism. An overview of the RMI system architecture is done in section 3.4.

According to Philippsen and Zenger [39], JAVA offers good support

[1] Also called COW (Cluster Of Workstations) or NOW (Network Of Workstations).

for medium grain shared memory parallelism and for medium to coarse grain distributed parallel applications with restricted communication needs, but find that JAVA does neither offer any significant support for cluster computing or workstation-based parallel computing nor does it address more involved or irregular communication needs. As cluster interconnection technology is improving and cluster reliability and availability is growing, JAVA's current mechanisms become insufficient. According to [39], explicit socket communication is too low-level for comfortable parallel programming and RMI is too restrictive and its overhead for dealing with network problems is too verbose for a cluster setting. Both sockets and RMI result in increased program size and thus reduce programmer productivity and code maintainability. Yu and Cox [50] also have experienced that with RMI the programmer must spend significant effort on such problems as data replication and the optimization of the remote interface to improve communication efficiency. They [39, 50] have been working on different projects to solve these shortcomings. About the same time Sun Microsystems first introduced RMI, Nagaratnam and Srinivasan [38] had also been working on Remote Objects. Hummel, Ngo and Srinivasan [21] explored the suitability of using JAVA as a high-performance parallel-computing language en propose a standard library of SPMD routines. Kalé, Bhandarkar and Wilmarth [25] also aren't satisfied with the JAVA Remote Method Invocation and give another design and implementation of Parallel JAVA. Others like Bilk and Gannon [5, 6] and Carpenter, Chang, Fox, Leskiw and Li [11] explored true parallel computing and JAVA as a language for High Performance Computing.

All these different projects and others can be roughly classified into 4 groups: Local Parallelism, Distributed Memory, Distributed Computing and Global Computing.

3.2 Local Parallelism

Although the interpretation of bytecode is substantially faster than the interpretation of most high level languages, still a performance penalty must be paid for portability. For many applications, this is not a major drawback. In other situations, as mentioned in section 2.2.5, performance may be more essential. 'Just In Time' compilation can, in these cases, be useful where *at run-time* the bytecode is compiled into native machine code. Performance close to the performance of compiled languages can be obtained with this approach. Because it is very likely that the demand for more computing power is to remain, other means to speedup JAVA programs have to be found. Expressing parallelism in JAVA itself clearly has the advantage that the transformed program into bytecode remains portable. Speedup can be obtained on any platform on which the implementation of the JAVA Virtual Machine supports the true parallel execution of threads.

Bik and Gannon [5, 6] developed a JAVA restructuring compiler which automatically exploits implicit parallelism in loops and multi-way recursive methods. They state that letting a compiler deal with the transformations that make implicit parallelism explicit simplifies the task of the programmer and makes the parallelization less error-prone.

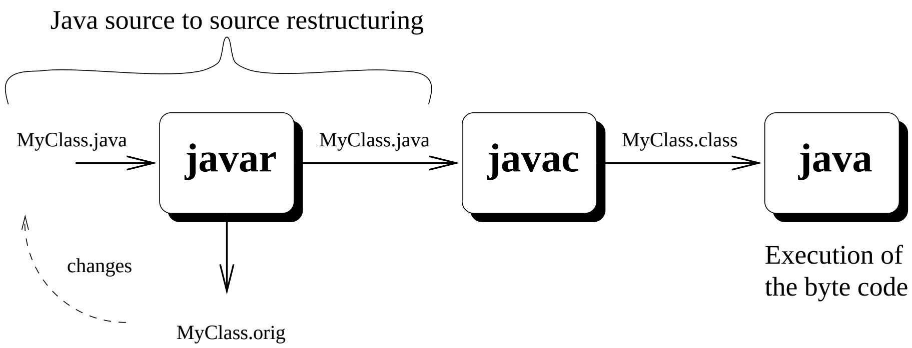

Figure 3.1: Restructuring, compiling, and interpreting

Figure 3.1 illustrates their approach to automatically exploiting implicit parallelism in JAVA programs. A JAVA program `MyClass.java` is used as input of their source-to-source JAVA restructuring compiler `javar`. First, the compiler identifies the loops and the multi-way recursive methods that can exploit implicit parallelism. Parallel loops are either detected automatically by means of data dependence analysis, or identified explicitly by the programmer by means of annotations. Because, according to Bik and Gannon [6], automatically detecting implicit parallelism in multi-way recursive methods can be very hard, in their paper they simply assume that such parallelism is always identified explicitly by means of annotations. Thereafter, the compiler transforms the input program into a form that uses the multi-threading mechanism of JAVA to make all implicit parallelism explicit. Because parallelism is expressed in JAVA itself, the transformed program can still be compiled into bytecode by any JAVA compiler, and subsequently interpreted by any bytecode interpreter.

In their paper, Bik and Gannon [6] give some examples and show how the restructuring compiler can exploit implicit loop parallelism in loops and how implicit parallelism in multi-way recursive methods can be made explicit, and present the results of a series of experiments that show the potential of the transformations presented in the paper. A prototype of the `javar` restructuring compiler and a prototype of a bytecode parallelization tool `javab` are made available on the Internet, see [6].

Roudier and Ichisugi [43] state that the JAVA language lacks facilities for language extension, which are adopted by other object-oriented languages. For example, C++ provides a macro preprocessor, operator overloading and template facilities. Smalltalk and CLOS have closures and metaclass facilities. These features extend language constructs and operators and supplement the inheritance mechanism which extends class, that is module definitions (as well as data types if class and type notions are unified) in a very hard way. Because they find that without such extension possibilities it is difficult to make parallel libraries which can be employed easily or seamlessly by end-users, they developed an extensible JAVA preprocessor kit, EPP. EPP can be used to introduce new language features, possibly associated with new syntax. Basically, EPP itself is, like `javar`, a source-to-source preprocessor of JAVA. According to Roudier and Ichisugi, EPP enables a preprocessor programmer to write an extension as a separate module, so-called EPP plugins. In their paper, Roudier and Ichisugi have defined Tiny Data-Parallel Java as an example application of their extensible Java preprocessor EPP. In the Tiny Data-Parallel Java language, some methods are identified as data-parallel methods and translated by EPP in JAVA multi-threaded codes, executed on a large number of virtual processors.

Demaine [13] describes the design of a JAVA package that implements the main features of Reppy's [40] Higher-Order Concurrency extension to the basic CSP model by Hoare [20]. In this extension, communication algorithms (or *events*) are first-class objects and can be created and manipulated dynamically. In addition, threads are automatically garbage collected and *channels* are first class, that is, they can be passed over to other channels. A channel is a uni-directional connection between two processes, which effectively correspond to the port abstraction. The main goal of Demaine's paper is to bring the capabilities of higher-order concurrency to a larger community and feels that garbage collection of processes and first-class events and channels offer a powerful and easy to use abstraction for concurrent programming that does not significantly harm performance. Although only a multi-threaded, single-process version of higher-order concurrency has been implemented, Demain states that the package is designed using a model that is very close to a distributed system, and should be easy to modify to use multiple computers in parallel.

3.3 Distributed Memory

A typical programming idiom for highly parallel machines where the data provides the parallel dimension is called data-parallel or single-program multiple-data (SPMD) [29]. Parallelism can conceptually be specified as a loop whose iterations operate on elements of a, perhaps multidimensional, array. Data dependences between parallel-loop iterations lead to a *producer-consumer* type of sharing, wherein one iteration writes variables that are later read by another, or collective communication, wherein all iterations participate. The communication pattern between iterations is often very regular, for example a bidirectional flow of variables between consecutive iterations.

As mentioned in section 1.1.10, JAVA uses a memory model wherein an update to a shared variable is only guaranteed to be visible to other threads when they execute synchronized code. For instance, a value assigned by a thread to a variable outside a synchronized statement may not be immediately visible to other threads because the implementation may elect to cache the variable until the next synchronization point.

Hummel, Ngo and Srinivasan [21] state that exploiting data locality is arguably one of the most important optimizations on parallel machines, and that the ability to perform data locality optimizations, such as caching and prefetching, depends on the underlying language memory model. Communication optimizations require an analysis of memory access patterns in an SPMD program, and therefore, the ability to perform communication optimizations also depends on the language memory model.

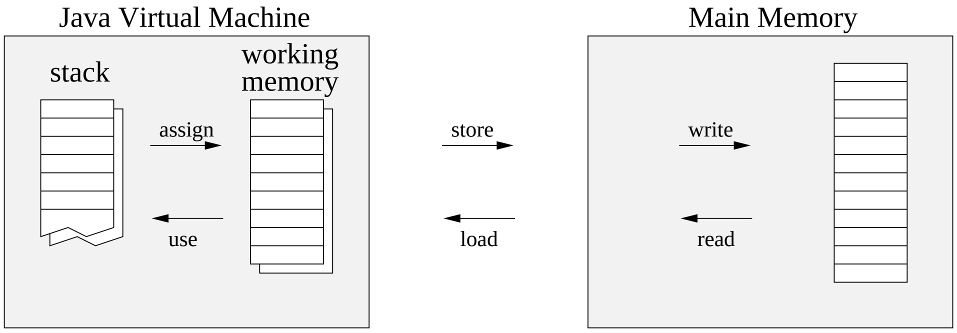

Figure 3.2: JAVA memory abstraction

In the JAVA memory model (see figure 3.2), every thread has, in addition to a local stack, a working memory in which it can temporarily keep copies of the variables that it accesses. The main memory contains the master copy of all variables. The interactions between the stack and the working memory are *assign, use, lock*, and *unlock*; the interactions between the working memory and the main memory are *store* and *load*, and those within the main memory are *write* and *read*. As mentioned in section 1.1.10, any update to a volatile variable by a thread is immediately visible to all other threads. This disables some code optimizations. Java defines a complex set of rules which define the required ordering and coupling of the interactions between the memories. For instance, the *read/write* actions to be performed by the main memory for a variable must be executed in the order of arrivals.

According to Hummel, Ngo and Srinivasan [21] the JAVA shared memory data-model is weakly consistent, but is it not weak enough to allow the oblivious caching and out of order access of variables between synchronization points. For example, JAVA requires that the updates by one thread be totally ordered and the updates to any given variable/lock be totally ordered. A JAVA implementation does not have enough freedom to fully exploit optimization opportunities provided by caches and local memories. Hummel, Ngo and Srinivasan state that a stronger memory consistency model at the language level forces an implementation to adopt at least the same, if not stronger, memory consistency model. A strong consistency memory model at the implementation level increases the complexity of code optimizations. The compiler/run-time-system have to consider interactions between threads at *all* points where shared variables are updated, not just at synchronization points. To perform many standard compiler optimizations such whole program information is necessary. With the separate compilation of packages, as in the case with JAVA where whole program information is not always available, it is difficult to distinguish between shared and private variables. In such situations, therefore, several common code optimizations are effectively disabled.

Hummel, Ngo and Srinivasan [21] state that the following features are idiomatic in data-parallel programming:

- Creating and starting multiple threads simultaneously.
- Producer-consumer operation for synchronization and data communication between threads.
- Collective communication/synchronization between a set of threads.

Although JAVA provides constructs for parallelism, there is no mechanism for creating and starting threads in a ThreadGroup simultaneously. A `forAll` loop can be implemented as a serial loop in JAVA:

```
SPMDThread spmdthread[] = new SPMDThread[NumT];
for (i=0; i<NumT; i++) {
  spmdthread[i] = new SPMDThread();
  spmdthread[i].start();
}
```

Hummel, Ngo and Srinivasan find that a simple high level construct for creating multiple threads is more desirable, and if thread startup is not well synchronized across the parallel machine, the serialization will result in idle time that degrades the speedup of a program.

Implementing the producer-consumer data sharing using the synchronized statements, `wait`, `notify` and `notifyAll` is problematic, because `notify`s are not queued, and so enforcing a specific ordering is complicated. In their paper, Hummel, Ngo and Srinivasan give a simple example of producer-consumer code where it is not guaranteed that the produce method will be invoked after the consumer code, and therefore an early `notify` will be lost. A programmer must therefore add mutual exclusion and auxiliary variables to enforce the correct ordering, resulting in a low-level style of programming. They also find that it is not clear that a compiler can always generate efficient code for these producer-consumer methods. The JAVA language specification [18] claims that `wait` and `notify` are especially appropriate in situations where threads have a producer-consumer relationship. According to Hummel, Ngo and Srinivasan, this applies mainly to bounded buffer applications where it is not necessary to associate a producer with a specific consumer. Because the implementation of `notify` selects a random thread on the associated wait queue of an object, and Hummel, Ngo and Srinivasan find that producer-consumer semantics require that the `notify` select a specific thread from the wait queue, they use a separate object for each producer-consumer pair to ensure that only one thread can appear on the wait queue. Collective communication in JAVA must also employ mutual exclusion and auxiliary variables.

Hummel, Ngo and Srinivasan propose a SPMD-java library in which each thread has a local partition of each shared array. They find that localizing shared data-structures is a natural extension of the per-thread working-memory model supported by JAVA. This design choice makes it necessary that threads communicate explicitly. The library provides methods for creating thread arrays and for establishing communication links between them according to various topologies. The library also includes methods for producer-consumer data sharing and for collective communication. They also provide methods to specify several common communication channel topologies, such as one, two, and three dimensional grids or a binary tree. There are methods for broadcasting and multicasting variables to threads in a `ThreadGroup`. A barrier method is provided to synchronize phases of a parallel algorithm. The library includes methods for common reduction and scan operations, which they find, are the cornerstones of many parallel algorithms. A reduction applies a binary operation to reduce the elements of an array into a single element, while a scan applies a binary operation to reduce subsequences of the element of an array.

Ivannikov, Gaissaryan, Domrachev, Etch and Shtaltovnaya [23] also propose a SPMD library called DPJ. In their paper they mainly describe the DPJ library and give an example of the *QuickSort* algorithm using the DPJ library.

Launay and Pazat [28] have been working on the Do! project which includes the automatic generation of distributed memory programs from shared memory multithreaded JAVA programs. They state that, nowadays, it is necessary to develop new tools and programming environments for NOWs to help *non-experts* programmers to cope with parallelism, distribution, heterogeneity and reconfigurability in the context of many different applications ranging from scientific computing, and data mining to interactive and virtual reality applications. The JAVA language is used both for the source and the generated code with no modification to the language. the programming model is explicitly parallel: parallelism is expressed by the definition of parallel control and data structures. These definitions are written using a specific library within a framework designed for the Do! project. The code distribution, mapping of task on processors, is guided by user annotations in the source code, describing tasks and data distribution. The execution model is based on the use of remote objects and Remote Method Invocations (see sections 3.1 and 3.4). The process of code generation consists of two main steps. First, from a shared memory program with partial annotations for tasks and data distribution, the missing distribution is computed to obtain a program where the distribution of all objects is specified. Secondly, from this shared memory annotated parallel program, a distributed memory program is generated.

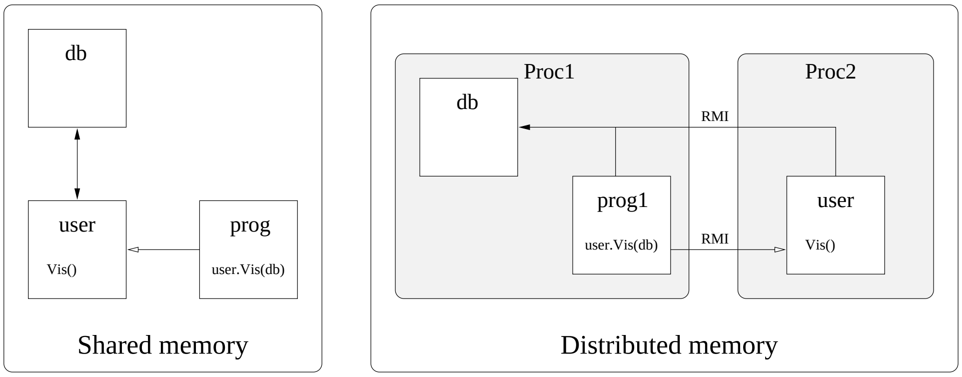

Figure 3.3: Transformation from shared memory to distributed memory

Figure 3.3 shows an example of a basic transformation of a shared memory program , in this case a sequential program, into a distributed program. A database represented by the object `db` is accessed by a user interface `user`. The program invokes the visualize method `Vis` of object `user` and passes a reference to `db` to the visualize method.

If the program is going to executed on a distributed architecture and the database is going to be located on one machine (`proc1`) and the user interface on another (`proc2`), annotations `ON Proc` need to be added to specify were objects should be created. Their Tool automatically generates a distributed program were `prog1` runs on machine `proc1` and `prog2` runs on machine `proc2`. Naming problems are solved, and remote accesses are automatically generated, allowing transparent access to remote objects.

The generated program is a distributed program, running in a distributed memory environment. According to Launay and Pazat [28] the generated program is, due to the tasks distribution and the nesting of parallelism, not SPMD as it is the case when sequential programs are distributed using the *Owner Computes rule*. Thus, the communication mechanisms they use take into account the fact that the processor which owns an object does not know when this object is accessed and which processor generates the access. On each processor, a thread (distinct form the local computing thread) manages remote creation and remote method invocation on local fragments of objects. Both operations are implemented with the standard JAVA RMI package (see sections 3.1 and 3.4).

3.4 Distributed Computing

As mentioned before, the JAVA language supports sockets for general communication, which require the client and server to engage in applications-level protocols to encode and decode messages for exchange. An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. The programmer is not working directly with sockets, but has the illusion of calling a local procedure. The arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR. RPC however, according to Sun Microsystems [35], does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require *remote method invocation* or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

In the JAVA distributed object model, a *remote object* is one whose methods can be invoked from another JAVA Virtual Machine, potentially on a different host. An object of this type is described by one or more *remote interfaces*, which are JAVA interfaces that declare the methods of the remote object. *Remote method invocation* (RMI) is the action of invoking a method of a remote interface on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object. According to Sun Microsystems [35], the JAVA distributed object model is similar to the JAVA object model in the following ways:

- A reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote).
- A remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in JAVA syntax for casting.
- The built-in JAVA `instanceof` operator can be used to test the remote interfaces supported by a remote object.

And the JAVA distributed object model differs from the JAVA object model in these ways:

- Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
- Nonremote arguments to, and result from, a remote method invocation are passed by copy rather than by reference. This is because references to objects are only useful within a single virtual machine.
- A remote object is passed by reference, not by copying the actual remote implementation.
- The semantics of some of the methods defined by class `Object` are specialized for remote objects.
- Clients must deal with additional exceptions that can occur during a remote method invocation, because the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects.

The RMI system consist of three layers (see figure 3.4): the *stub/skeleton layer*, the *remote reference layer*, and the *transport layer*. A remote method invocation from a client to a remote server object travels down through the layers of the RMI systems to the client-side transport, then up through the server-side transport to the server.

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of

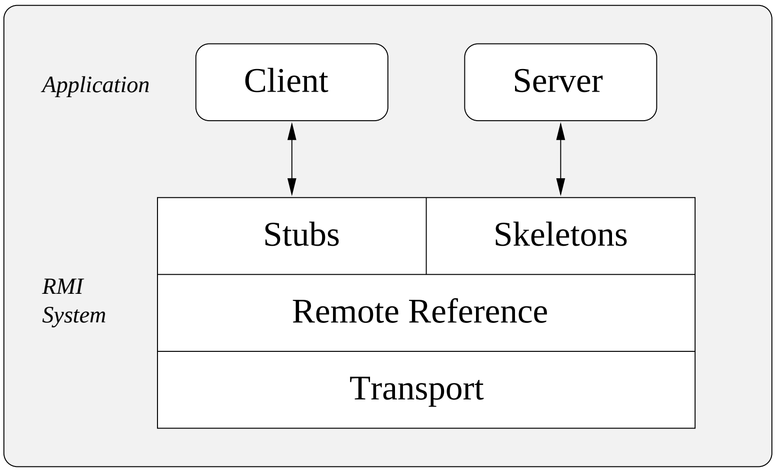

Figure 3.4: The RMI system layers

any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. Marshal streams employ a mechanism called *object serialization* which enables JAVA objects to be transmitted between address spaces. Objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference. A *stub* for a remote object is the client-side proxy for the remote object. Such a stub implements all the interfaces that are supported by the remote object implementation. A *skeleton* for a remote object is a server-side entity that contains a method which dispatches calls to the actual remote object implementation. Stubs and skeletons are generated using the rmic compiler. The appropriate stub and skeleton classes are determined at run time and are dynamically loaded as needed, see [35] for details about Dynamic Class Loading.

The remote reference layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics-whether the server is a single or a replicated object requiring communications with its replicas. Also handled by the remote reference layers are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring to objects that are implemented in (a) servers that are always running on some machine, and (b) servers that will run only when some method invocation is made on them (activation). At the layers above the remote reference layer, these differences are not seen. The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented *connection*.

The transport layer is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the target of remote calls) residing in the transport's address space. In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to occur before handing off the request to the server-side skeleton. The skeleton for a remote object makes an call up to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side.

For more details about the stub/skeleton, remote reference, and transport layer see the Remote Method Invocation Specification [35].

As mentioned in section 3.1, Philippsen and Zenger [39] find that JAVA does neither offer any significant support for cluster computing or workstation-based parallel computing nor does it address more involved or irregular communication needs. They also state that explicit socket communication is too low-level for comfortable parallel programming and that RMI is too restrictive and its overhead for dealing with network problems is too verbose for a cluster setting. To overcome these shortcomings they introduce *JavaParty* [39].

According to Philippsen and Zenger, JavaParty adds transparently remote objects to JAVA purely by declaration while avoiding disadvantages of explicit socket communication, the programming overhead of RMI, and many disadvantages of the message-passing approach in general. JavaParty is specifically targeted towards and implemented on clusters of workstations. A multi-threaded JAVA program can easily be turned into a distributed JavaParty program by identifying those classes and threads that should be spread across the distributed environment. In order to do so, Philippsen and Zenger introduce a new class modifier `remote`, which is the only extension of JAVA. JavaParty provides a shared address space, although objects of remote classes may reside on different machines their methods and variables, both static and non-static, can be accessed in the same way as in pure JAVA. No explicit communication protocol needs to be designed and implemented by the programmer, because JavaParty hides addressing and communication mechanisms from the user and handles network exceptions internally. JavaParty is location transparent, the programmer does not need to map remote objects and remote threads to specific nodes of the network; compiler and runtime system deal with locality and communication optimization. Objects that are stored locally are used locally at the cost of a pointer indirection instead of expensive operating system overhead.

Philippsen and Zenger have implemented the JavaParty Compiler JPC, see figure 3.5

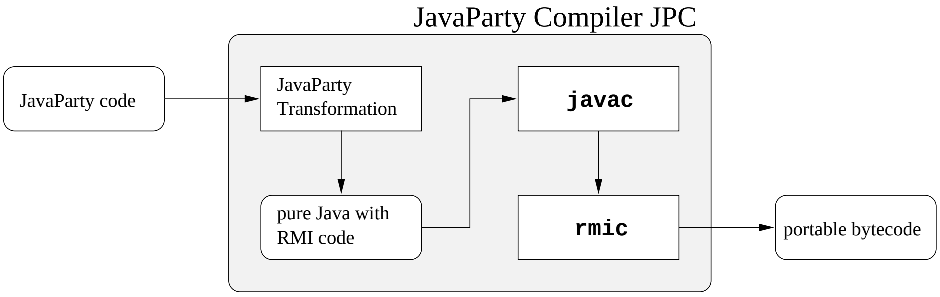

Figure 3.5: The JavaParty Compiler

The JavaParty code (pure JAVA code with the `remote` extension) is transformed into pure JAVA with RMI code. The transformed code is then compiled by the JAVA compiler `javac`. The resulting RMI portions are then fed into Sun's RMI compiler `rmic` (stud and skeleton generator). Because no changes to the standard JDK are made, portable bytecode is the result. JavaParty currently does not offer any means for data parallelism. However, work is nearing completion that fills that gap: they are currently adding a `forall` statement that is pre-processed into threaded execution with remote threads by means of another source-to-source transformation.

Before Sun Microsystems first introduced JAVA RMI, Nagaratnam and Srinivasan [38] also have been working on remote objects in JAVA. Instead of using the existing language constructs of JAVA to facilitate remote object creation and remote method invocation, Nagaratnam and Srinivasan choose to modify the language and provide new language structures, because they felt it was more challenging. They also state that adding support for remote objects into the language itself would be necessary for JAVA to be used as a distributed-programming language. With this approach changes to the compiler and interpreters for all platforms are required in order to maintain full portability. No new publications about there approach to remote objects in JAVA are made since the summer of 1995, which could be an indication that they are no longer working on that particular project.

Yu and Cox [50] implemented JAVA/DSM, a parallel JAVA Virtual Machine implemented on the Threadmarks DSM system [26]. A DSM system provides a shared memory abstraction over a group of physically distributed machines. It automatically handles data communication between machines and eliminates the need for the programmer to write message-passing code. Yu and Cox state that with the RMI package, the programmer is exposed to the distributed nature of the system and a lot of extra work may be necessary, namely maintaining the coherency of replicated data, optimization of the remote call interface, and reference marshaling. With the shared memory platform, all of these issues are implicitly handled by the DSM system and, according to Yu and Cox, the program is much easier to write. Their implementation includes a distributed garbage collector and supports the JAVA API with very few changes. These changes are:

- The heap is allocated using ThreadMark's shared memory allocation routine.
- Classes loaded by the JVM are put in a shared memory. The loading is synchronized.
- During the garbage collection, the garbage collector must recognize objects created by remote machines, mark those referenced locally, and send the unused remote references to their owners after the collection.
- To support data conversion, Yu and Cox require a mapping from an arbitrary address to the object's handle. Their solution is to require that all objects on a page are the same size, and put a pointer to the handle at the beginning of every object.

Yu and Cox made most of the changes to the memory management module, and say that the rest of the JAVA Virtual Machine is virtually unchanged. The modified garbage collector allows each machine to perform independent garbage collections.

Because JAVA RMI is synchronous, the calling thread waits for the result of the remote method execution. Implementing asynchronous RMI on top of synchronous RMI implies providing a callback method on the client object. Therefore, one has to implement the client object also as a remote object and pay the unnecessary overhead of the return message. Kalé, Bhandarkar and Wilmarth [25] designed and implemented a parallel extension to JAVA, supporting the dynamic creation of remote objects and efficient asynchronous method invocation. It also supports object groups, which, according to Kalé, Bhandarkar and Wilmarth, are essential for efficient and modular programming for parallel applications. The parallel JAVA extension is implemented using Converse [24] interoperability framework, which makes it possible to integrate parallel libraries written in JAVA with modules in other parallel languages in a single application. The implementation does not require any modifications to the JAVA compiler or to the JAVA Virtual Machine. Instead an interface compiler is used to generate wrapper code for each parallel class. The user only has to specify the names of parallel classes, and the signatures of their remotely invokable methods to the interface translator. Kalé, Bhandarkar and Wilmarth's parallel Java will only run on machines where there exists both Converse and JAVA installations.

Before Sun first released the RMI package, Satoshi [45] introduced HORB. HORB is an ORB (Object Request Broker) for JAVA that extends JAVA as a distributed object oriented language. It is very similar to JAVA's RMI. HORB supports dynamic remote object creation and connecting of already existing remote objects on a server. Although Satoshi [45] claims that the compiler and the runtime are coded in JDK 1.0 and therefore operate on any JAVA system, HORB will not run on the JAVA Virual Machine in the JDK 1.0.2 package. No updates or announcements have been made since May 1996 to solve this problem.

3.5 Global Computing

As the World Wide Web continues to grow, it's providing an important infrastructure for scientific and engineering computation. According to Fox and Furmanski [16], the distributed computing hardware of the Web has a remarkable potential compute performance. Of course, the Web does not support the low latency and high bandwidth required by most parallel simulations. However, many [16, 3, 2, 8, 15, 12, 44, 1, 4] believe that an attractive scientific computing environment can be built on top of Web software. Although there are some differences in the proposed desirable properties for a global computing infrastructure [3, 8, 2, 44, 1, 4], they all have the following desirable properties in common:

- **Scalability**, the ability to scale up to millions of nodes.

- **Heterogeneity**. The global computing infrastructure must support a variety of platforms and operating systems, all of which must interoperate.

- **Fault tolerance**. According to Baldeschwieler, Blumofe and Brewer [3], an application should tolerate most failures and should have a probability of completion consistent with traditional applications. Faults in distributed systems can be generally be classified into *stopping faults* and *Byzantine faults* [31]. Stopping faults cover cases of processors crashing or leaving, and are automatically handled by adaptively parallel systems. Byzantine faults cover all other kinds of faults such as data loss or corruption due to faulty network links, or faulty processors, as well intentional malicious attacks. Byzantine faults can be handled using replication techniques such as *majority voting* or the more sophisticated and reliable algorithms in [31]. According to Sarmenta [44], another way to achieve fault tolerance is by simply choosing more fault tolerant problems. Such problems include those that do not require 100% accuracy in the first place, such as sound or graphics processing where a little static or a few scattered erroneous pixels would be unnoticeable or can be averaged out to make them unnoticeable.

- **Adaptive parallelism**. By its nature, the Web is heterogeneous and dynamic. Nodes can have different kinds of CPUs, and can join and leave a computation at any time. Even nodes with the same type of CPU cannot be assumed to have equal or constant computing capacities, since each can be loaded differently by external tasks, especially in systems which try to exploit users' idle times. For these reasons, models for global computing systems must be adaptively parallel. That is, they must not assume the existence of a fixed number of nodes, but must dynamically exploit a varying collection of resources, so that long-running applications can grow or shrink.
- **Security**. The system must ensure that foreign applications obey local limits on data access and resource utilization, so that users will be willing to add their machines to the global pool.
- **Ease of use**. The system should be easy to use.

Baldeschwieler, Blumofe and Brewer [3] propose ATLAS, a system architecture for global computing. The goal of the ATLAS system is to exploit the networked resources of the world as a giant distributed computer, and to develop an infrastructure that exploits idle resources both within and among institutions.

ATLAS combines the existing mechanisms from JAVA and Cilk [7] into a global computing infrastructure. Cilk is an algorithmic, C-based parallel multi-threaded programming language together with a runtime system that employs a thread-scheduling algorithm based on the technique of *work stealing* in which idle processors steal threads from randomly chosen 'victims'.

Based on multi-threading, the ATLAS programming model is largely borrowed from Cilk and adapted to JAVA. An essential feature of this programming model is that it abstracts all physical resources so that the runtime system is free to manage all physical resources in the highly dynamic manner required for global-scale execution. ATLAS applications are written mostly in JAVA with some use of C or C++ when extra performance is needed. Pure JAVA applications have the advantage that they are heterogeneous and do not require special libraries. When a native library is used, the application is limited to run on machine that has or can dynamically link that library.

An ATLAS program contains one or more ATLAS procedures which consists of a sequence of non-blocking, continuation-passing threads, see figure 3.6.

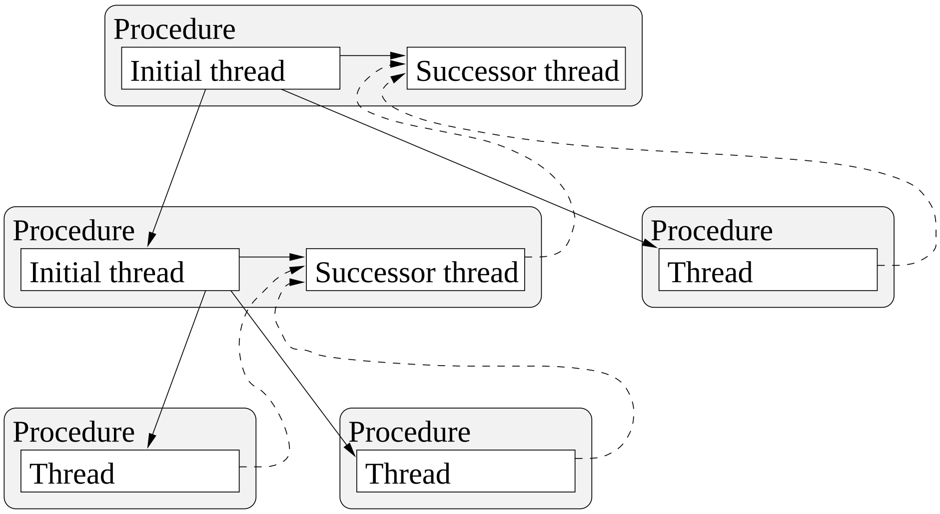

Figure 3.6: The ATLAS programming model

A procedure spawns a child when a thread of the parent spawns a child thread that becomes the initial thread of the child procedure. A spawn is the parallel analogue of a procedure call. The difference is that a spawned child may execute concurrently with its parent (and its siblings). After spawning one or more children, a procedure may synchronize with its children by blocking until one or more of its children complete. Since a thread cannot block, a procedure must employ a sequence of threads in order to obtain blocking behavior. In addition to spawning children, a thread may spawn a successor that is part of the same procedure. The successor thread cannot begin execution until it receives all of its arguments which are returned from the children. Thus, the successor thread implements the blocking behavior of procedures by waiting for arguments from the children.

An ATLAS programmer employs threads to express the parallelism and locality in the program, while the runtime system takes care of the details of protocols, scheduling, and resource management. The ATLAS system architecture consists of clients, managers, and compute servers, see figure 3.7.

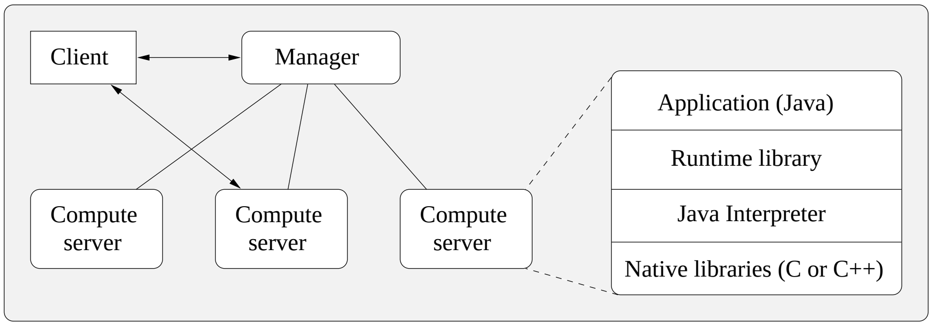

Figure 3.7: The ATLAS system architecture

A client with an application to run contacts the local manager to find an up and idle compute server. It then connects to this server to run the application and provides feedback to the user while the application is running. During execution, idle servers steal work from those that are busy, so that the application eventually spreads to all available resources. The Runtime library of a compute server is responsible for work stealing, thread management, and marshalling and unmarshalling objects for communication to other (heterogeneous) servers. The JAVA interpreter runs the application and JAVA half of the runtime system. The native libraries (if any) extend the server with new capabilities that applications may exploit or even require.

Each compute server runs a daemon. It runs all the time, either working on some application thread or pinging its manager and siblings for threads to steal. The server process runs with low priority and relinquishes the node whenever there is non-ATLAS work to do. Porting the runtime system to a new platform requires a JAVA interpreter for that platform and conversion of the native part of the runtime system.

Alexandrov, Ibel, Schauser and Scheiman [2] also proposed a global computing architecture: SuperWEB. In their model there are three kinds of participants: brokers, clients and hosts. Brokers coordinate the supply and demand for computing resources. Hosts offer their resources (CPU time, memory, permanent storage) as a commodity by registering with a broker. Clients are users or computers that need extra computational resources. For example, a client that wants to run a task requiring 500 billion floating point operations may register a request for CPU resources with a broker. It's the brokers responsibility to find hosts capable of executing the client's task. The role of a host or a client is not fixed. A Machine may register as a host when being idle, and act as a SuperWeb client when its owner needs additional computing power.

According to Alexandrov, Ibel, Schauser and Scheiman [2], The SuperWeb architecture makes computational resources a freely tradeable commodity that can be captured by a microeconomic model of supply and demand. Like other commodity exchanges, fluctuations in supply and demand result in a changed cost for the services provided. For example, if many clients are demanding resources, the cost of requesting a hundred CPU hours will increase. If there are no longer such demands for resources, the cost of such resources will drop.

Alexandrov, Ibel, Schauser and Scheiman implemented the first prototype using Netscape browsers and JAVA applets. In the prototype, hosts register their CPU by visiting a broker web page. This causes the host to download a JAVA applet, which stays in contact with the broker and downloads client's JAVA objects for execution until the user leaves that broker web page. Clients submit tasks by visiting a different web page and submitting a form to request resources. On this form, they specify the JAVA program (and URL) they wish to have executed. As clients request resources, the broker maps these to the next available host, using a simple round-robin algorithm. Once a host is selected, the broker retrieves the client applet (URL) and submits it to the host. The host starts executing it and delivers the result directly to the client. Finally, after finishing a task, the host contacts the broker again and signals its readiness for new tasks.

A key design feature of this architecture is that the host software runs completely at the user-level, and only requires that registered machines have a WWW browser, such as Netscape, which can run trusted code, such as JAVA applets, see section 1.1.6. It does not require clients to actually have access, e.g., login privileges, to the host machines running their codes.

According to Brecht, Sandhu, Shan and Talbot [8], parallel computing on the web will likely take two predominant forms. The first is where people will wish to run third-party JAVA applications faster. This may mean simply sending the application to a faster compute engine for execution, or it may mean sending a single application that is capable of running in parallel to a variety of compute engines, either local or remote. Since no assumption can be made about the mechanisms for parallelism used by these third-party applications, Brecht, Sandhu, Shan and Talbot state that the most suitable approach is one that extends the JAVA runtime system but makes no assumptions about the program. This would allow existing JAVA applications that use only standard JAVA concurrency facilities to be executed in parallel. The second is where people will wish to write applications in JAVA that exploit parallelism, and distribute these applications to other users. In this case, according to Brecht, Sandhu, Shan and Talbot, the most suitable approach is one which extends the programming environment but not the runtime system. This would allow JAVA programs written explicitly for parallelism using the parallel programming environment to run in parallel on existing unmodified JAVA interpreters. For these two cases Brecht, Sandhu, Shan and Talbot introduce the JAVA Parallel Class Library (JPCL) and the JAVA Parallel Runtime System (JPRS) which are part of their ParaWeb project. According to Brecht, Sandhu, Shan and Talbot [8], the goal of ParaWeb is to allow users to execute serial programs on faster compute servers or parallel programs on a variety of possible heterogeneous hosts.

The parallel class library in ParaWeb provides mechanisms for the remote creation and execution of threads, and facilitate communicate between these threads. A programmer can create a thread that executes on a remote machine by simply extending the JPCL `RemoteThread` class and then instantiating the new class. Communication is currently provided in the form of standard message passing *send* and *receive* primitives.

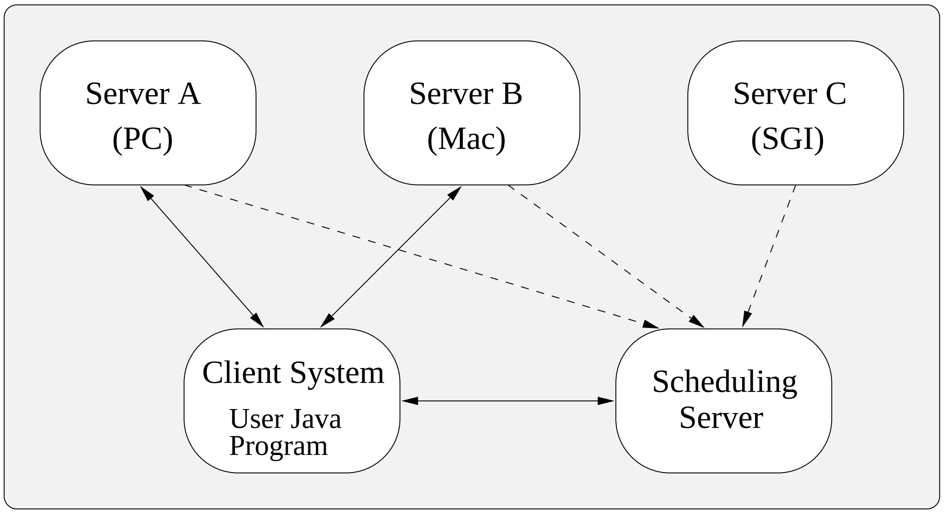

Figure 3.8: Running a parallel program using the Java parallel class library

A parallel program is executed in the following way. First server sites (Server A, Server B, Server C) start the JAVA interpreter and initially run a deamon which has been compiled into JAVA byte-code, see figure 3.8. Each server daemon then registers with a local scheduling server which is used to keep track of which server daemons are available. At this point these server daemons are ready to accept and execute JAVA byte-code on behalf of the permitted clients. A JAVA program creates a thread on a remote machine by simply instantiating a new object that has extended the `RemoteThread` class. When this object is instantiated on the client machine, the library code implementing the `RemoteThread` class contacts the scheduling server and requests the address of a server daemon that will execute code on behalf of the client. The scheduling server replies with an available server daemon. The instantiated object on the client sends the compiled byte-code to the remote server daemon. The server daemon then calls the run method of the uploaded `RemoteThread` class which starts the execution of the thread. In the example in figure 3.8, the scheduling server has determined that Server A and Server B should be used for the execution of the program. The server daemon executes the client's code and sends results back to the client. When the client application completes, it contacts the scheduling server to inform it that it has finished using the server daemons. Because the server daemons execute standard JAVA byte-code within an unmodified interpreter, the server daemon is completely portable and capable of being executed on any computer which executes JAVA byte-code.

In the implementation of the ParaWeb JPRS, Brecht, Sandhu, Shan and Talbot modified the JAVA interpreter to allow threads to be instantiated on any remote machine that is running the modified interpreter. The JAVA interpreters on each machine coordinate with each other to maintain the illusion of a global shared address space for the application they are running, in much the same way as distributed shared memory systems, such as the Threadmarks system [50], maintain the illusion of shared memory. In this approach, the definition of the JAVA Virtual Machine [30] remains unchanged, as does the byte-code of existing JAVA programs. Thus, any compiled JAVA application utilizing threads may be executed in parallel, by distributing those threads to different machines. In their paper, Brecht, Sandhu, Shan and Talbot [8], show an example of a parallel multiplication using JAVA threads. In the JPRS, upon the invocation of a thread, the modified interpreter on the local machine (the client) contacts one of the other machines also running the modified interpreter (the server). The client must then upload the class that is to be executed, along with any classes that may be referenced by this class, to the server. To do this, upon instantiation of a thread on a remote machine, replicas of all classes are made on the remote machine. Then consistency mechanisms ensure that the replicas on the different machines remain consistent during the course of the parallel computation.

Sarmenta [44] also presents a system for global computing: the Bayanihan project. The prototype of Bayanihan is based on HORB [45], providing remote object capabilities in JAVA similar to those of Sun's RMI. Figure 3.9 shows the architecture of the Bayanihan system.

On the client side, an engine runs its own thread, and takes care of receiving and processing data objects from its corresponding manager on the server. A work engine, for example, would take care of getting

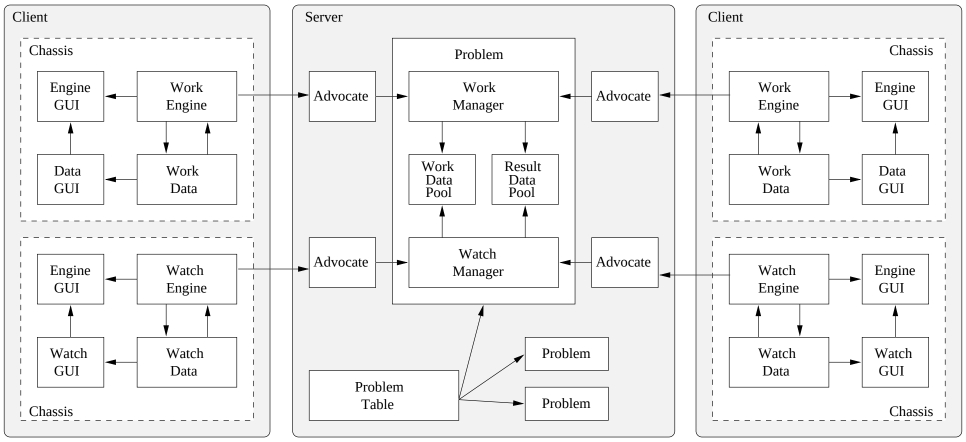

Figure 3.9: Bayanihan system architecture

work data from the work manager, executing the work, and requesting more work when it's done. Data objects are generally polymorphic and know how to process themselves. To execute work, a work engine need only to pass the work data a pointer to itself, and call the work data's `run()` method. The data will compute itself, using the work engine to communicate with the work manager, if necessary. Both the engine and data have associated GUI types for user interface. The engine and the GUIs are all contained in a chassis object, which can be an applet or application.

On the server side, an advocate object serves as the engine's representative and forwards the engine's calls to the manager. The manager has access to several data pools. These pools may be shared by other managers serving other purposes. In figure 3.9, for example, the work and result pools are shared by the work and watch managers, allowing the watch manager to watch the progress of the computation and inform watch engines of such things as new results and statistics. The whole set of associated managers and data pools compose a problem. There may be many independent problems existing in a server at the same time. A problem table keeps track of these, and can be used by volunteers to choose the problem they want to participate in.

Acharya, Ranganathan and Saltz [1] decribe the design and implementation of Sumatra, an extension of JAVA that supports resource-aware global computing. In the design of Sumatra, they have not altered the JAVA language. All added functionality was provided be extending the JAVA class library and by modifying the JAVA interpreter, without affecting the virtual machine interface.

Sumatra adds four programming abstractions to JAVA: *object-groups, execution-engines, resource-monitoring* and *asynchronous events*. An object-group is a dynamically created group of objects. Objects can be dynamically added or removed from object-groups. All objects within an object-group are treaded as a unit for mobility[2]-related operations. This allows the programmer to customize the granularity of movement and to amortize the cost of moving and tracking individual objects. Objects-groups also allow the programmer to control the life-time of objects. Objects that are included in an object-group continue to live on a host even after the thread that created them completes execution or migrates to some other host. Objects that do not belong to an object-group are subject to garbage-collection as usual.

An execution-engine corresponds to the notion of a 'location' in a distributed environment. In concrete terms, it corresponds to an interpreter executing on a host. Multiple engines can exist on a single host. Sumatra allows object-groups to be moved between execution-engines. An execution-engine may also host active threads of control. Currently, multiple threads on the same engine are scheduled in a *run-to-completion* manner. Acharya, Ranganathan and Saltz plan to implement other scheduling strategies in the future. Threads can move between engines.

The resource-monitoring allows programs to either query the level of resource availability or to control the extent to which various resources are used by the program itself as well as library code it is linked to. Both on-demand as well continuous monitoring are supported. For continuous monitoring, the resource-monitoring interface allows programs to register jitter thresholds which determine which resource changes should be reported.

In Sumatra, asynchronous events are used to notify executing programs about urgent changes in their execution environment. The notifications can come either from the interpreter or from the external environment (the operating system or some other administrative process). Acharya, Ranganathan and Saltz expect that the interpreter would use asynchronous events to notify the program about violations of resource-restrictions requested by the program itself; they expect that the external environment would use asynchronous events to inform the program about changes in the environment of the interpreter including resource revocation. Sumatra allows programs to register handlers for asynchronous events. The handlers for asynchronous events are able to inspect the current state of execution and can take appropriate action including mov-

[2] According to Acharya, Ranganathan and Saltz, mobile programs can move an active thread of control from one site to another during execution. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-side instead of fetching all the data to its current location.

ing away from the current execution site or changing the resourcerestrictions in force.

In Sumatra, computation begins at a single site and spreads to other sides in three ways:

- Remote method instantiation
- Remote thread creation
- Thread migration

Remote Thread instantiation corresponds to the familiar notion of RPC (Remote Procedure Call) whereby the calling thread is suspended while an operation is performed, on its request, at a remote site. Remote thread creation differs from remote method instantiation in that the new thread is independent of the creating thread; the creating thread continues once the creation is complete. Finally, thread migration involves stopping the execution of the calling thread at the current site, transferring its state to another site and resuming execution at that site.

Baratloo, Karaul, Kedem and Wyckoff [4] describe the design and implementation of a system called Charlotte. Charlotte is build on top of JAVA without relying on any native code. Their programming model can be conceptually divided into a virtual machine model and a run-time system. The virtual machine model provides a reliable shared memory machine to the programmer and isolates the program from the execution environment. The run-time system realizes this model on the Web[3]. An execution of a Charlotte program consists of alternating sequential and parallel steps. A sequential step consists of standard JAVA code. Computationally intensive parts of a program are done in parallel steps by one or more routines, a routine is analogous to a standard thread in JAVA, except for its capability to execute remotely. A parallel step starts with the keyword (which is in reality a function call) `parBegin` and ends with `parEnd`. There can be any number of routines defined within one parallel step. The closing `parEnd` server as a synchronization barrier.

The Charlotte package provides three major services: a scheduling service, a memory service, and a computing service. In the current prototype, the scheduling service, the memory service, and parts of the computing service executing the sequential steps are fused into a single process: the manager. The computing service performs the computationally intensive parallel routines. Baratloo, Karaul, Kedem and Wyckoff refer to the processes, that perform the computing service, as workers. A single manager and one or more workers together implement the virtual

[3]According to Baratloo, Karaul, Kedem and Wyckoff [4], the Web can be seen as a set of unpredictable, dynamically changing, and faulty machines.

machine model that the program was written for. The manager executes the program until it reaches a parallel step. Upon reaching this point, it calculates the number of jobs for the workers to execute. The manager then waits (if necessary) for workers to contact it. In general, this will happen after a worker finishes a previous assignment and is willing to work. However, a worker could just appear any time and be integrated into the computation. In general, the manager keeps giving jobs to workers until they are done.

Because the programming model is in standard JAVA, enhanced with a few classes for expressing parallelism, heterogeneity and security are provided to the extent supported in JAVA. The run-time environment realizes automatic load balancing and fault tolerance, both critical to the effective utilization of the Web. Although Charlotte is currently handicapped by the slow performance of interpreted JAVA, Baratloo, Karaul, Kedem and Wyckoff [4] expect this will be transparently resolved once JAVA compilers are used.

3.6 Conclusion

The JAVA programming language and its multi-threading mechanism enables several different execution threads to run at the same time inside the same program, in parallel, without interfering with each other. If the program runs on a JAVA Virtual Machine which supports true parallel execution of threads on a multi-processor computer, execution times, compared with a single processor computer, will be reduced, resulting in a better performance. In order to speedup JAVA programs, some preprocessors have been proposed such as Bik and Gannon's `javar` which automatically exploits implicit parallelism, and Roudier and Ichisugi's EPP which also generates multi-threaded code.

When the performance requirements can not be met by a single computer, or when data is distributed over several computers, JAVA's RMI can be used. Because many aren't satisfied with JAVA's RMI, several different proposals have been made. Together with the previous mentioned preprocessors, they can be roughly classified into 4 groups: Local parallelism, Distributed memory, Distributed computing and Global computing.

Several SPMD libraries have been proposed, all providing a distributed memory environment with different extensions such as Hummel, Ngo and Srinivasan's SPMD-library in which each thread has a local partition of each shared array and Launay and Pazat's Do! project which generates a by user annontations guided distributed memory program.

Close to the group Distributed memory lies the group Distributed computing. Philippsen and Zenger's JavaParty adds transparently remote objects to JAVApurely by declaration while avoiding disadvantages of explicit socket communication, the programming overhead of RMI, and many disadvantages of the message-passing approach in general. No changes to the standard JDK are made, resulting in portable bytecode. Nagaratnam and Srinivasan's approach to remote objects in JAVA included the modification to the JAVA language to provide new language constructs. The result of this approach is that the bytecode is no longer portable to standard JAVA Virtual Machines, and can only run on modified virtual machines. No new publications are made since the summer of 1995, indicating that the project is no longer worked on. Yu and Cox's approach to distributed computing consists of Java/DSM, a parallel Java Virtual Machine implemented on the Threadmarks DSM system. Their implementation includes a distributed garbage collector and supports the JAVA API with very few changes. Portability relies on the availablilty of their modified virtual machine. Kalé, Bhandarkar and Wilmarth's approach consist of a parallel class which supports the dynamic creation of remote objects and efficient asynchronous method invocation. The implementation does not require any modifications to the JAVA compiler or to the JAVA Virtual Machine, but will run only on machines where there exists both Converse and JAVA installations. Another approach to remote objects is Satoshi's HORB which also supports dynamic remote object creation, but will not run on JAVA Virtual Machines other than the one from the JDK 1.0 package.

Of all the proposed systems for distributed computing other than JAVA's RMI, JavaParty is the only system which results in portable bytecode and only needs the standard JDK package.

In the last group, Global computing, several proposals for a global computing architecture have been made. All having the following desirable properties in common: scalability, heterogeneity, fault tolerance, adaptive parallelism, security and ease of use. Baldeschwieler, Blumofe and Brewer propose ATLAS, a system to exploit the networked resources of the world as a giant distributed computer. ATLAS applications are mostly written in JAVA with some use of C or C++ when extra performance is needed. Pure JAVA applications have the advantage that they are heterogeneous and do not require special libraries. When a native library is used, the application is limited to run on machines that have or can dynamically link that library. Similar to ATLAS is Alexandrov, Ibel, Schauser and Scheiman's SuperWeb. The SuperWeb architecture makes computational resources a freely tradeable commodity that can be captured by a microeconomic model of supply and demand. The prototype is implemented using Netscape browsers and JAVA applets. This makes the SuperWeb, unlike the ATLAS system, a fully portable and secure system. Brecht, Sandhu, Shan and Talbot introduce a parallel class library and a parallel run-time system for two different approaches. With the

Java Parallel Class Library JPCL, programmers can exploit parallelism in programs, and execute the server daemons standard JAVA code within an unmodified interpreter. With the Java Parallel Run-time System JPRS, the definition of the JAVA Virtual Machine remains unchanged, as does the bytecode of existing JAVA programs. Portability relies on the availability of the modified run-time system. Sarmenta presents the Bayanihan project. The first prototype is based on HORB, and is similar to the previous systems. Sarmenta is currently also working on other approaches. For the design of the Sumatra, Acharya, Ranganathan and Saltz did not alter the JAVA language, but extended the JAVA class library and modified the JAVA interpreter, without affecting the virtual machine interface. Finally, Baratloo, Karaul, Kedem and Wyckoff introduded Charlotte. Charlotte is build on top of JAVA without relying on any native code. Because the programming model is on standard JAVA, enhanced with a few classes for expressing parallelism, heterogeneity and security are provided to the extent supported by JAVA. However, portability also relies on the availability of their modified virtual machine.

Of all the proposed architects for global computing, SuperWeb is the only systems which uses pure JAVA code without modifications to the JAVA Virtual Machine.

CHAPTER 4

Conclusions & Further Research

In this document JAVA has been identified as an object-oriented, multi-threaded programming language, as a configurable chip, and as a dynamically extensible operating system. To find out in what way JAVA is suited for embedded or Real-Time systems, the definition and characteristics of a Real-Time system have been discussed. To find out in what way JAVA is suited for parallel computing, an inventory of parallel related research work has been classified into 4 groups.

As a result of the rapid development of JAVA products like a JavaChip, a JavaOS, and several API's (PersonalJava, EmbeddedJava, Card), JAVA has the potential to become an important player in the embedded software market. The JAVA programming language supports many of the characteristics which a general-purpose language must have when it is to be used for effective programming of a Real-Time system. Currently most embedded programs are coded in C and assembly, but it is expected that JAVA will supersede C as the primary programming language for embedded systems in the near future. With the arrival of new JIT compilers and other new tools, the size and speed problems will eventually disappear. The problem concerning Real-Time can hopefully be dealt with when RTOS companies introduce their modified virtual machines or other Real-Time tools.

At the moment of this writing (end of March), Sun Microsystems announced that the EmbeddedJava specification is available for public review and comment. Because the EmbeddedJava platform is a JAVA application environment that runs on top of a real-time operating system, and consists besides the Application and EmbeddedJava Classes of hardware-specific classes which control the hardware in an embedded device and are either supplied by the device manufacturer or the RTOS reseller, it is very interesting to study this API and it's development in a follow-up or near future thesis research. Because the Virtual Machine is either supplied by JavaSoft or through an RTOS reseller, the previously mentioned problem concerning Real-Time could be (partially) solved and is therefore also another interesting topic for further study.

JAVA supports concurrency through the multi-threading mechanism and monitor synchronization, which makes it a suitable language for parallel computing on a single computer. With the classification into 4 groups, its clear that parallel computing can be done in different ways. A certain way can be more suited for a specific problem than another way. The question how suited the several tools really are, is something that needs to be studied in another follow-up research. Because of the rapid development of JAVA and JAVA related products, its better to study those tools which do not alter the JAVA programming language or the JAVA Virtual Machine. For local computing, Bik and Gannon's preprocessor `javar` is an interesting tool for further research, because it generates standard multi-threaded JAVA code. For distributed computing, Philippsen and Zenger's JavaParty is a candidate for further research, because it also generates standard JAVA code, and they are still working on it to improve it and to develop new tools. For global computing, SuperWeb is the only system that uses pure JAVA code without modifications to the JAVA Virtual Machine. But the use of pure JAVA code or a unmodified JAVA Virtual Machine is not a guarantee for successful global computing. Because the success also depends on the ease of use of the system, and other users be willing to volunteer their idle resources, all the proposed systems are topic for further research.

Bibliography

[1] Anurag Acharya, M. Ranganathan, and Joel Saltz. *Resource-aware Meta-computing*, March 1997.

[2] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. *SuperWeb: Research Issues in Java-Based Global Computing*, 1997.

[3] Eric J. Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. *ATLAS: An Infrastructure for Global Computing*, 1996.

[4] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. *Charlotte: Metacomputing on the Web*, April 23, 1997.

[5] Aart J.C. Bik and Dennis B. Gannon. *Automatically Exploiting Implicit Parallelism in Java*, 1997.

[6] Aart J.C. Bik and Dennis B. Gannon. *javar - a prototype Java restructuring compiler*, 1997. `http://www.extreme.edu/hpjava`.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, July 1995. `http://theory.lcs.mit.edu/~cilk/`.

[8] Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot. *ParaWeb: Towards World-Wide Supercomputing*, September 19, 1996.

[9] Alan Burns and Andy Wellings. *Real-time systems and their programming languages*. Addison-Wesley publishing company, 1989.

[10] Dennis Caromel and Julien Vayssiere. *Java// Distributed, Concurrent and Parallel Computing in Java*, 1997. `http://zenon.inria.fr/sloop/javall/`.

[11] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. *Experiments with "HPJava"*, February 1997.

[12] Mani K. Chandy, Adam Rifkin, Paolo A.G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman. *A World-Wide Distributed System Using Java and the Internet*, 1997.

[13] Erik D. Demaine. *Higher-Order Concurrency in Java*, 1997.

[14] A. Ferrari. *The Java Parallel Virtual Machine*, 1996. `http://www.virginia.edu/~ajf2j/jpvm.html`.

[15] Geoffrey C. Fox and Wojtek Furmanski. *Towards Web/Java based High Performance Distributed Computing - an Evolving Virtual Machine*, 1996.

[16] Geoffrey C. Fox and Wojtek Furmanski. *Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modelling*, 1997.

[17] Al Geist. *JavaPVM The Java to PVM Interface*, 1997. `http://www.epm.ornl.gov/pvm/`.

[18] J. Gosling, B. Jay, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] C.A.R. Hoare. *Monitors: An operating system structuring concept*, 1974.

[20] C.A.R. Hoare. *Communicating Sequential Processes*, 1978.

[21] Sussan Flynn Hummel, Ton Ngo, and Harini Srinivasan. *SPMD Programming in Java*, June 1997.

[22] NewMonics Inc. *Embedded Real-Time Development in the Java Language*. `http://www.newmonics.com/WebRoot/technologies/PERC/escw.html`.

[23] V. Ivannikov, S. Gaissaryan, M. Domrachev, V. Etch, and N. Shtaltovnaya. *DPJ: Java class library for development of data-parallel programs*, 1997.

[24] L.V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. *Converse: An Interoperable Framework for Parallel Programming*, April 1996.

[25] L.V. Kalé, Milind Bhandarkar, and Terry Wilmarth. *Design and Implementation of Parallel Java with Global Object Space*, 1997.

[26] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. *Treadmarks: Distributed shared memory on standard workstations and operating systems*, January 1994.

[27] Pascale Launay and Jean-Louis Pazat. *A Framework for Parallel Programming in Java*, December 1997.

[28] Pascale Launay and Jean-Louis Pazat. *Generation of distributed object-oriented programs*, September 1997.

[29] Ted G. Lewis. *Foundations of Parallel Programming.* Computer Society Press, 1994.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[31] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kauffman Publishers Inc., 1996.

[32] Sun Microelectronics. *picoJava I Microprocessor Core Architecture.* `http://www.sun.com:80/microelectronics/whitepapers/`.

[33] Sun Microelectronics. *Sun Microelectronics' picoJava I Posts Outstanding Performance.* `http://www.sun.com:80/microelectronics/whitepapers/`.

[34] Sun Microsystems. *Java Home Page.* `http://java.sun.com/`.

[35] Sun Microsystems. *Java Remote Method Invocation (RMI).* `http://java.sun.com/products/jdk/rmi/index.html`.

[36] Sun Microsystems. *Java Systems - Embedded Products.* `http://www.sun.com:80/javastation/embed/embedprod.html`.

[37] M. Morrison and J. December a.o. *Java unleashed.* Sams net, 1996.

[38] Nataraj Nagaratnam and Arvind Srinivasan. *Remote Objects in Java*, 1997.

[39] Michael Philippsen and Matthias Zenger. *JavaParty - Transparent Remote Objects in Java*, June 1997.

[40] John H. Reppy. *Higher-order concurrency*, June 1992.

[41] David L. Ripps. *An Implementation Guide to Real-Time Programming.* Prentice Hall, 1989.

[42] David L. Ripps. *Java for embedded systems*, September 1996.

[43] Yves Roudier and Yuuji Ichisugi. *Integrating data-parallel and reactive constructs into Java*, October 1997.

[44] Luis F. G. Sarmenta. *Bayanihan: Web-Based Volunteer Computing Using Java*, November 16, 1997.

[45] Hirato Satoshi. *Preliminary Performance Evaluation of a Distributed Java: HORB*, 1997.

[46] Volker Strumpen. *Coupling Hundreds of Workstations for Parallel Molecular Sequence Analysis*, March 1995.

[47] David A. Thurman. *JavaPVM The Java to PVM Interface*, 1997. `http://www.isye.gatech.edu/chmsr/JavaPVM/`.

[48] var. *The Oxford Dictionary of Computing*. Oxford University press, 1989.

[49] S.J. Young. *Real Time Languages: Design and Development*. Halsted Press, 1982.

[50] Weimin Yu and Alan Cox. *Java/DSM: A Platform for Heterogeneous Computing*, July 1997.