Implementing Protocols in Java: The Price of Portability[*][†]

Bobby Krupczak, Ken Calvert, Mostafa Ammar
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{rdk,ammar,calvert}@cc.gatech.edu

GIT-CC-97-21

August 1, 1997

Abstract

As the number and variety of Web- and network-based applications continues to increase, so does the need for flexible communication protocols and services to support them. Traditionally, a major impediment to deployment of new protocols is the need to upgrade millions of end-systems with compatible implementations. At the same time, Java — a language explicitly designed to support development and distribution of new applications via the Web — is emerging as a (potentially) ubiquitous system platform. It is therefore natural to consider whether Java might speed the introduction of protocols to better support new applications. In this paper, we investigate the tradeoffs involved in using Java for protocol implementation and deployment. Using insights from a Java-based protocol suite and supporting subsystem we have implemented, we describe the benefits of using the Java language and quantify the performance cost of implementing a protocol in Java for various combinations of interpretation and compilation. We find that the performance cost of using Java-based protocols is presently equivalent to four years of hardware performance gains, i.e., interpreted, Java-based protocol performance on current hardware is roughly equivalent to the performance of compiled C code on four-year-old hardware.

[*]This work was supported in part by grants from the National Science Foundation (NCR-9612855, NCR-9628379, and NCR-9305115).

[†]Java, JavaStation, JavaOS, JavaChip, and Solaris are all trademarks of Sun Microsystems, Inc.

1 Introduction

The tremendous growth of the World-Wide Web [2] has brought both new users and new applications to the Internet. Many of these new applications require network services not ideally matched by those provided by the traditional protocols (TCP/IP and UDP/IP). Unfortunately, the difficulty of designing, implementing, and deploying new protocols — which generally requires millions of end-systems to be modified — makes it very difficult to change the status quo, where TCP and UDP are the only transport protocols available. At best, custom protocols are incorporated into the application code itself which inhibits code re-use and requires that the same wheel (e.g. TCP-style congestion control [9]) be re-invented over and over. Yet many new applications — for example, multimedia conferencing — could clearly benefit from such reusable, application-tailored protocols.

Almost simultaneously, a new model for developing and deploying applications based on Java [6, 17] is emerging. Java and the World-Wide Web together are developing into a network-capable application-loader, providing a consistent interface to many network services and, more importantly, a portable and system-independent *application* platform. This naturally leads to the question of whether Java might be useful as a vehicle for rapid development and deployment of new communication protocols.

In this work, we investigate the use of the Java programming language and virtual machine combined with the World-Wide Web as a platform for implementing and deploying protocols. This combination offers several interesting features. *First*, protocols and additional code required to support them can be downloaded and executed on the fly as needed, thus enhancing the flexibility and interoperability of end systems. *Second*, the severity of the protocol code portability problem [12] is greatly reduced because the environment in which protocols execute (the Java virtual machine) has been ported instead of the individual protocols. *Third*, object-oriented features such as inheritance, specialization, and polymorphism can provide additional power for protocol programmers and enhance code re-use [13, 23].

These advantages, however, are not without cost. Presently, the Java virtual machine is invariably emulated in software. Until hardware implementations are available, the use of Java will carry a cost in terms of performance. Moreover, user-space Java platforms (e.g., in Web browsers) present challenges to the faithful implementation of protocol semantics (e.g., with respect to addressing). In this paper, we examine both the benefits and the costs of Java as a vehicle for protocol development and deployment.

The rest of this paper is organized as follows. In the next section we provide background information, review related work, and present a brief a overview of Java. In Section 3 we present the design of a Java-based general-purpose protocol subsystem (which we term *HotLava*). We then analyze the performance of Java-based protocols in Section 4 followed by a discussion in Section 5 of security and addressing issues that arise due to this approach. Section 6 concludes the paper.

2 Background

For the purposes of this work, a *protocol* is a software module implementing a traditional communication protocol specification, e.g. TCP or IP. A *protocol subsystem* (e.g. BSD, the *x*-Kernel, and Streams) provides uniform access to resources such as buffers and timers, and imposes structure (i.e. interfaces) on protocol implementations in order to ease the burden of protocol development. Protocols are traditionally composed in stacks, which provide services to applications. The arrangements in which protocols are composed to provide services are described (and in some cases constrained) by a *protocol graph*.

2.1 Related Work

The idea of configurable and extensible communication subsystems has been the subject of considerable research. Falcone [5], Tschudin [26], and Holzmann [7] proposed extensible subsystems in which protocol code can be downloaded from protocol servers. The Streams [21] and *x*-Kernel [8] subsystems support some level of configurability and extensibility while *Da CaPo* [19] seeks to improve network performance through the elimination of unnecessary protocol functionality. SPIN [20] is an extensible operating system supporting the dynamic addition of protocols at run-time while still providing a level of safety and performance. Although the work described here is similar in various ways to each of these, it differs in its focus on Java, which has sufficient momentum behind it that it actually stands a chance of eventually being a more or less ubiquitous platform.

Traditionally, protocol subsystems have normally resided entirely within the kernel or operating system. This placement has tended to increase the difficulty of developing and testing new protocol implementations. Alternatively, protocols and subsystems can reside in user-space along with applications. Thekkath [25] examined the problems encountered when moving the BSD [14] subsystem into user space. While they too cited flexibility and extensibility as goals, simply moving protocols to user-space does not necessarily improve portability since their implementations are still tied to a particular operating system and CPU architecture.

Others have implemented application protocols in Java; however, we know of no other work that focuses on general-purpose protocol support techniques and rigorously examines performance. Our own previous work [10, 11, 12] has considered protocol portability and proposed two different approaches for overcoming it within the context of traditional subsystems. In both cases, protocol portability was enhanced because protocols could remain in their native subsystem [10, 12] or could be re-used in another through encapsulation [11, 12]. The research described in this paper addresses the protocol portability problem but does so by shifting the problem outward, toward the protocol subsystem and away from the protocol implementations themselves. Because the Java virtual machine has already been ported to the systems on which we wish to operate, a degree of portability comes for free. However, as we shall see, the protocol portability problem is not completely eliminated due to constraints imposed by the way the Java virtual machine is integrated into the underlying system.

2.2 Overview of Java

Java actually comprises two separate entities — a programming language [6] and an underlying machine specification called the Java virtual machine [17]. Together, these can be thought of as roughly analogous to the combination of the C programming language and Sparc CPU architecture. The programming language itself is object-oriented yet is simpler than languages like C++ and Smalltalk. Because Java code is compiled into an intermediate bytecode format for later interpretation, programs written in Java can run on a variety of platforms without recompilation. Because of its intended use with the World-Wide Web, its designers have attempted to make Java safe by removing pointers from the language as well as instrumenting strict array bounds checking and strong typing — all of which can provide impediments to protocol developers.

The Java virtual machine implements an interpreter for an abstract CPU architecture[1]. As long as the Java virtual machine has been ported to a system, programs coded within Java will run unmodified without the need for recompilation. It should be noted, however, this approach does not completely eliminate protocol portability issues since the bond between protocol implementations and their surrounding protocol subsystem has not been eliminated. That is, Java protocol implementations cannot necessarily be removed from their accompanying subsystem and run within other Java-based subsystems.

The Java programming language comes with a set of pre-defined class libraries or packages for manipulating graphical interfaces, basic streaming input/output, utilities such as hash tables and vectors, and networking. Networking support is limited to those classes utilizing TCP and UDP protocols (provided by the underlying host system) and related utility functions. Classes for communicating with Web servers, registering Web content handlers, and retrieving and interpreting uniform resource locators (URLs) are built on top of TCP and are also provided. The language is extensible in that new classes (e.g. those making up our protocol subsystem and protocol implementations) can be added dynamically by loading those classes into the virtual machine as needed. Java programs are divided into *applications* which run within stand-alone interpreters and *applets* which are restricted to running within an interpreter contained within a Web browser.

The Java virtual machine provides the ability to dynamically add classes via its *class loader*. The class loader takes requests for new classes (made programmatically from within existing classes and objects) and loads them from file if the interpreter is running as a stand-alone application or loads them from the network if the interpreter is running within a Web browser. Our Java-based protocol approach makes extensive use of the capability to load and execute protocols on the fly.

3 Java-based Protocol Subsystem Design

We next focus on the design of a Java-based protocol subsystem, which we term *HotLava*, using a previously-developed subsystem analysis methodology [12] that identifies the salient properties of protocol subsystems. We then present several different system models available to Java protocol programmers. We design and implement a Java-based protocol subsystem for several important reasons. First, subsystems can improve protocol-code re-usability by imposing structure on protocol implementations and by providing functionality (e.g. memory buffers) common to all protocol implementations. Second, designing and implementing a Java-based protocol subsystem permits us to more accurately compare this approach against traditional protocol subsystems using data taken from our previous work.

[1]Although it is abstract now, Sun and others are actually developing CPUs based on the Java virtual-machine specification; when they becomes available, Java programs will run on a JavaChip without the aid of the interpreter. In effect, the interpreter is being moved into silicon instead of running as a program on another, different CPU. We use *interpreter* and *Java virtual machine* interchangeably when there is no ambiguity.

Protocol structure How a protocol implementation is structured affects how it interacts with the subsystem and with other protocols. The protocol structure defined by *HotLava* is similar to that in traditional subsystems in that it defines input, output, and control methods. However, it also differs in several important ways. First, in *HotLava*, each protocol is implemented as a Java class and is instantiated by users and other protocols desiring its service. Second, protocol classes wishing to operate within *HotLava* inherit the protocol structure from a base protocol class. Using inheritance to impose protocol structure allows *HotLava* to take advantage of language support for software structuring (via inheritance), permits the protocol programmer to enhance functionality through specialization, while still supporting protocol-code reuse. Third, *HotLava*'s protocol structure defines methods for explicit protocol graph manipulation by defining *sethlp* and *setllp* methods for setting the higher and lower-layer protocols respectively. *HotLava*'s protocol structure also defines an explicit *close* method to terminate a communication session since there is no explicit memory or object deallocator in Java. Finally, *HotLava*'s protocol structure dedicates a thread per protocol object; that thread is used to perform periodic, protocol-specific processing like route table pruning, address-resolution (ARP) cache maintenance, and data re-transmission. Performing periodic protocol processing in this manner simplifies protocol code because the use of asynchronous timeouts can be significantly reduced.

Process architecture Subsystems dictate how protocol entities like messages, layers, connections, buffers, and timers are bound to the underlying unit of scheduling. The manner in which they are bound is called a subsystem's process architecture [22]. *HotLava* defines a *vertical* process architecture in which packets are escorted through the protocol graph via non-preemptable threads. When an application wishes to send data, a buffer is created and a thread is attached to it to escort it through the protocol graph in much the same manner as the *x*-Kernel. Thread-based process architectures provide a convenient programming paradigm because each thread possesses its own stack; many threads can be active within a given protocol simultaneously without interfering with one another. If the processing of a particular packet cannot continue (e.g. if address resolution is needed), the thread escorting the packet need only suspend itself until packet processing can continue instead of queueing the packet, setting a timer, and terminating.

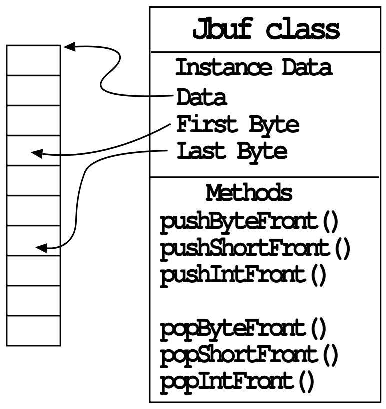


Figure 1: *Jbuf* class

Buffer Support *HotLava* provides buffer support, in the form of *Jbufs* (depicted in Figure 1), which is optimized for the special needs of protocol processing in Java. Rather than implementing a complex data structure as is done in other subsystems, data is stored in a byte array which can then be randomly accessed. Traditional protocol implementations usually overlay a pointer to a header structure on top of a buffer and then access that header's fields through typecasting and pointer de-referencing. In Java, however, header overlaying and pointer de-referencing are not possible because Java does not support pointers[2] so *Jbufs* provide special methods to aid header construction and parsing. Header processing, upon packet reception in *HotLava*-based protocols, entails the consumption of single bytes (e.g. for a single byte header field like protocol version number using *popByte*), two-byte shorts (e.g. for header and data checksums using *popShort*), as well as four-byte integers (e.g. for protocol addresses using *popInt*) one field at a time from the front of a buffer. On transmission, Java-based protocols add headers by prepending them to the front of packets one field at a time in reverse order — the header's last field first and first field last using *Jbuf* primitives such as *pushByteFront*, *pushShortFront*, and *pushIntFront*. In order to reduce byte array re-allocation (which occurs when the *Jbuf* array must be re-sized or when two *Jbuf* arrays are combined), data placement within a *Jbuf* starts at its center so that the buffer may grow through appending and prepending.

Protocol graph construction In *HotLava*, special *service* classes dynamically construct protocol graphs at run-time as applications need communications services. For example, an application needing AppleTalk services need only create an instance of the AppleTalk *service* class. The code in that service class is responsible for allocating and initializing the various protocol layers and interconnecting them via the *sethlp* and *setllp* methods previously discussed. The service class can be thought of as a UNIX-style *daemon* that is responsible for the construction and maintenance of a particular protocol graph. Java's class-loader is used extensively in the construction of protocol graphs because it provides the ability to incorporate new protocol classes (as well as ordinary Java classes) into the virtual machine. Service classes may refer explicitly to a desired protocol class by referencing its name. If the class is not yet loaded into the virtual machine, the system's class-loader will transparently do so (either by loading the class from a local file or by downloading it from a Web server) without the intervention of the service class itself. Service classes and other *HotLava* protocols may also explicitly load Java classes themselves. For example, they may send each other Java class code (e.g. for a particular *HotLava* protocol class) as data transmitted via a normal transport connection. That data can then be converted by *HotLava* into an executable Java class using the builtin facilities of the class loader.

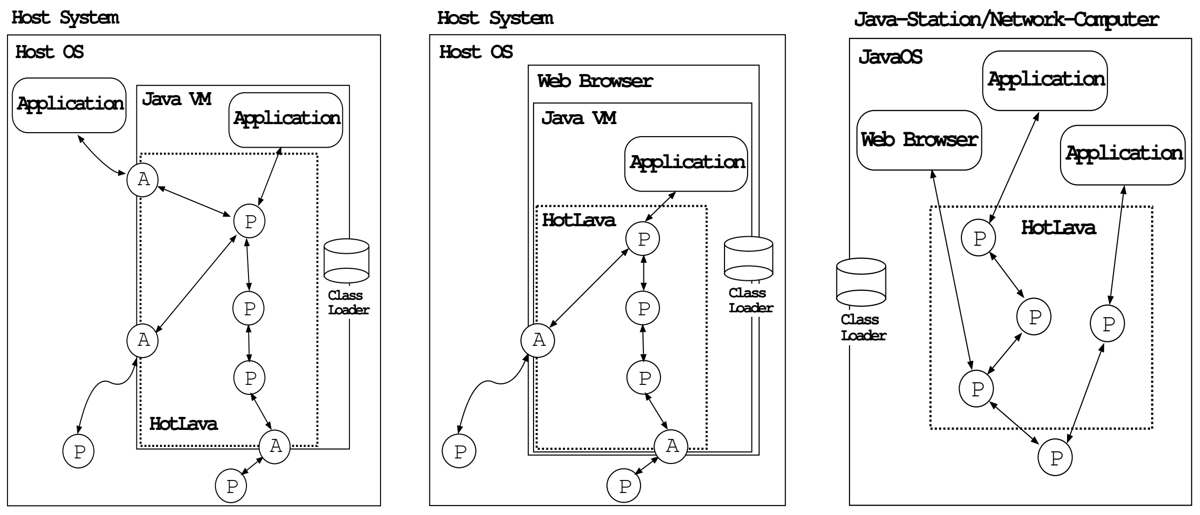
HotLava places the context of the protocol graph within the per-session state information of an object rather than its corresponding class. Our approach differs from that used in the *x*-Kernel which places protocol graph context within the state information of classes[3]. The *x*-Kernel's choice of organization limits the flexibility and configurability of the protocol-graph because all sessions must utilize the equivalent of a

[2]Java actually does possess pointers — they are just not available to programmers. Indeed, all objects (that is, instantiations of Java classes) are accessed by reference so many unnecessary data-touching operations can be avoided. The absence of pointers really only makes protocol header processing a bit more cumbersome. On the other hand, we found that the absence of pointers greatly improves the readability and simplicity of protocol code.

[3]Although not technically an object-oriented system nor implemented using an object-oriented programming language, the *x*-Kernel utilizes an object-based framework where protocols and sessions are roughly equivalent to classes and objects respectively. Protocol classes are placed in the protocol graph with links inter-connecting protocol input (or demux) methods. In object-oriented terms, the *x*-Kernel's *demux* method is roughly equivalent to a class method.

class link between their corresponding protocol and its lower and higher layer protocols. Consequently, all *x*-Kernel protocols must utilize the same protocol graph. Because *HotLava* places protocol graph context in the object, each instantiation of a protocol can potentially be placed at different points in the protocol graph. Multiple, potentially different protocol graphs could exist simultaneously with a particular protocol occupying one position in one protocol graph and a different position in a another protocol graph.

System models Three different models (depicted in Figure 2a-c) for executing Java code exist, each having subtle affects on subsystem design. Consequently, we consider each as a separate system model. In each system model figure, protocols are depicted by circles labeled 'P' while adapter protocols (discussed below) are depicted by circles labeled 'A'. Figures 2a and 2b depict the first two incarnations of *HotLava* within Java virtual machines running on host systems. Those two configurations differ in that the first configuration (Figure 2a) executes as a stand alone process on the host system and the other (Figure 2b) executes within a Web browser running on the host system. When *HotLava* executes within a stand-alone interpreter, Java classes are normally loaded from file first; when run within a Web browser, classes are normally loaded over the network from a Web server. In both cases, access to the underlying system is fairly restricted. When run as a Java application, protocols within *HotLava* can execute native machine code as well as open and close files (and devices). When run within a Web browser, access to the underlying system by *HotLava*-based protocols and other Java applets is more restricted. Because *HotLava* is run within its own process (either stand-alone or within a Web browser), and because several interpreters can operate concurrently, multiple *HotLava*s can operate simultaneously. We discuss the ramifications of this feature in the next section.



(a) Subsystem as Java-Application (b) Subsystem as Java-Applet (c) Subsystem within JavaStation

Figure 2: Java protocol-subsystem models

The incarnation depicted in Figure 2c presents a third execution alternative. In this case, *HotLava* executes alongside other Java applications within a network computer or JavaStation running a small, light-weight operating system (termed the JavaOS [15]) that provides just enough services for the virtual machine to operate directly on hardware platforms[4] without the aid of a host operating system. In this configuration, a single virtual machine executes both *HotLava* and applications. In this configuration, only a single *HotLava* subsystem can execute at any one point it time.

Accessing Network Devices Sending and receiving packets on real physical media requires access to devices and associated device drivers regardless of the particular environment or programming language within which they may be implemented. How protocols in *HotLava* access devices depends on the execution model and on the use of adapters (depicted as circles labeled with 'A') in Figure 2) tailored to that particular configuration. Adapter protocols [10, 12] are special purpose modules that provide conversion services for packets wishing to exit one subsystem and enter another. Adapter protocols are situated at the boundary between subsystems for which they provide adaptation services and operate by making each protocol subsystem believe that protocols outside the subsystem are actually operating within it.

Figures 2a and 2b depict the use of adapters in virtual machines running *HotLava* as a Java application or Java applet. Adapters are implemented partly in Java and partly in native system code tailored to the operating system hosting the virtual machine and to the protocol subsystem in which the other protocol operates. For example, protocols in *HotLava* can access Streams Ethernet devices by utilizing an adapter protocol class named *Ether*. That class implements the methods defined by *HotLava*'s protocol structure in Java but declares other methods as *native* methods which are implemented in the system's native language (e.g. C for Solaris). At run time, when the *Ether* class is loaded, the virtual machine is linked with a Solaris shared library implementing those particular native methods. When outgoing packets enter the Java *Ether* protocol, they traverse out[5] of the virtual machine and into native-C methods which convert them into a format dictated by the Transport Layer Interface (TLI) [1] and Streams and then send them using the underlying Solaris Ethernet device driver. The *Ether* adapter protocol also periodically checks if packets have been received from the underlying Solaris Streams Ethernet device; if so, native-C methods are again used to read and convert the packet from TLI and Streams to *Jbufs* used within *HotLava*.

Accessing network interface devices on JavaStations is simpler for two reasons. First, because JavaStations execute the virtual machine on JavaOS instead of a host operating system, one level of indirection can be eliminated. Second, network interface device drivers are implemented in Java and execute alongside protocol code within the same virtual machine (Figure 2c). For the case of accessing an underlying network interface to send and receive Ethernet packets, the *HotLava Ether* adapter protocol need only refer to the Ethernet driver directly without the need to access native code.

[4]In this configuration, the JavaOS could be running on a JavaChip or on a non-Java microprocessor. The JavaStations we experimented with had microSparc-II processors in them because JavaChips were not yet available.

[5]Native code is dynamically linked into the address-space of the interpreter at runtime so packets do not actually exit the virtual machine when they are passed to native methods because they continue to remain in the same address-space. Logically, however, one can think of it as exiting the virtual machine.

4 Java-Based Protocol Performance

Because the Java virtual machine is generally implemented in software, and the language itself is object-oriented, the performance of protocols implemented in Java and compiled into Java bytecodes is inevitably suspect. Although Java proponents argue that its performance is good enough for its intended applications, we desire a more precise quantification of the price paid in performance for the portability and extensibility offered by this approach. In this section, we analyze the performance of the AppleTalk implementation (depicted in Figure 3) we have running in Java and *HotLava*, and compare it against one we have previously developed in C and Streams. Both implementations are fully functional and have been tested against each other, against native Macintosh implementations, and against AppleTalk routers.

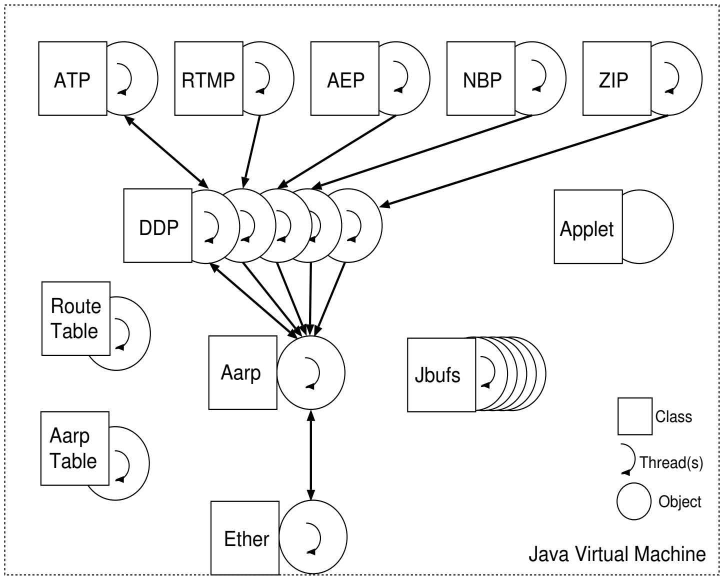
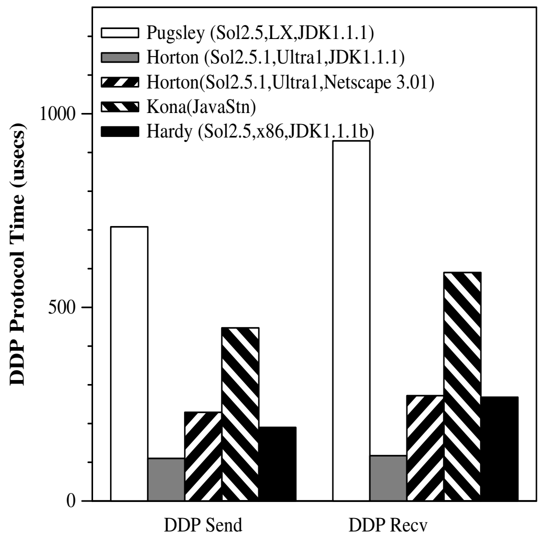


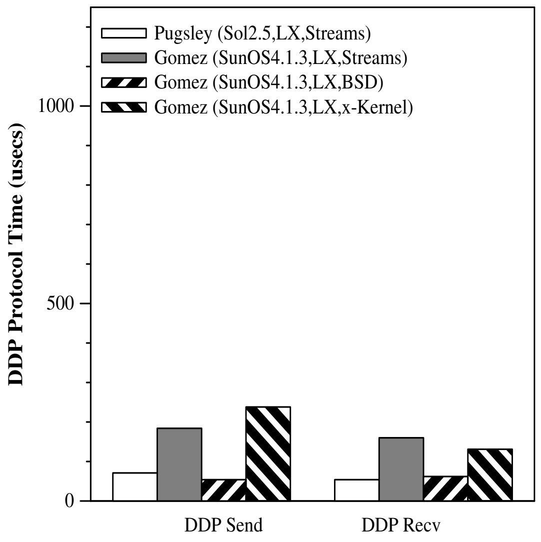
Figure 3: JavaTalk — AppleTalk within Java and *HotLava*

We chose the AppleTalk protocol suite [24] for several important reasons. First, AppleTalk provides a broad set of services equivalent to the Internet protocol suite, is in widespread use, and provides a reasonable platform upon which to base our evaluation. Second, because we implemented Java-based and Streams-based code from a common source base, we can ensure that each implementation has undergone the same level of tuning and (where possible) utilized the same implementation algorithms. Third, by having kept the operating system and hardware constant (Solaris 2.5 on SparcStation-LXs), we eliminated any performance differences attributable to differences in the underlying system. Therefore, we can assert that *the differences in performance between our implementations in C/Streams and Java/HotLava measure only the impact introduced by Java and the virtual machine interpreter*. We conducted our performance measurements using a simple client and server exchanging 40 000 four-byte packets in loopback mode. All values shown are averages over multiple samples; the 95% confidence interval is at most 10% of measured values.

The systems on which performance measurements were taken and on which testing was performed indirectly demonstrate the level of portability achieved by utilizing Java and the virtual machine for protocol and



(a) DDP in Java/*HotLava*



(b) DDP in Native C/BSD,Streams,*x*-Kernel

Figure 4: Java-based DDP protocol processing Times

subsystem development. For example, the JavaTalk (AppleTalk within Java) code in its compiled, bytecode format ran un-modified and without re-compilation on the following platforms: Solaris on SparcStations, Solaris on Intel, JavaStations, Windows[6] NT 3.51 on Intel, and within Netscape Navigator and Microsoft Internet Explorer Java interpreters. This level of portability is extremely appealing notwithstanding the small amount of native code necessary to access network-interface devices and the potential performance overhead introduced by the Java virtual machine. We discuss that overhead next.

4.1 Protocol Processing Time

Figure 4 compares performance of the AppleTalk Datagram Delivery Protocol (DDP) coded within Java and *HotLava* (Figure 4a) against that coded in C and Streams(Figure 4b). In order to provide more context, we also chart DDP's performance in C and the BSD and *x*-Kernel subsystems (taken from our earlier work [10]). These measurements depict send and receive protocol processing times per DDP packet. The results for the system Pugsley indicate that the use of Java increases protocol processing latency by a factor of approximately 10 for sending and 17 for receiving when running the subsystem as a Java application. Overall, in that configuration, DDP send and receive processing took approximately 700 and 900 μ-seconds respectively. However, when we ran JavaTalk on an Ultra-1 SparcStation (on the system named Horton), DDP protocol processing time was reduced (resulting in a performance improvement) to only 117 and 110 μ-seconds for send and receive processing respectively. DDP send/receive processing times for Solaris 2.5

[6]Unfortunately, a paper covering this kind of topic is replete with terms that are trademarked. "Windows" and "Windows NT" are all trademarks of Microsoft. "Navigator" is a trademark of Netscape Communications.

on Intel hardware (the system Hardy) and on a JavaStation (the system Kona) demonstrate better performance than Pugsley with processing times well under 500 μ-seconds.

The DDP processing times merit a couple of observations. First, DDP's performance on today's workstations (Sun Ultra 1) is roughly equivalent to the performance of native DDP implementations (e.g. C/Streams) on hardware that is approximately four years old (Sun Sparcstation LX) running the same operating system. Second, Java compiler and interpreter technology is in its relative infancy with performance tuning only just beginning. As compiler and virtual machine performance improve, so too will the performance of protocols coded within Java. Indeed, we have anecdotal evidence supporting this "rising tide lifts all ships" argument — when we compiled and executed our protocol implementations in the Java Development Kit (JDK) version 1.1.1, their performance doubled over that measured using an earlier version of the JDK.

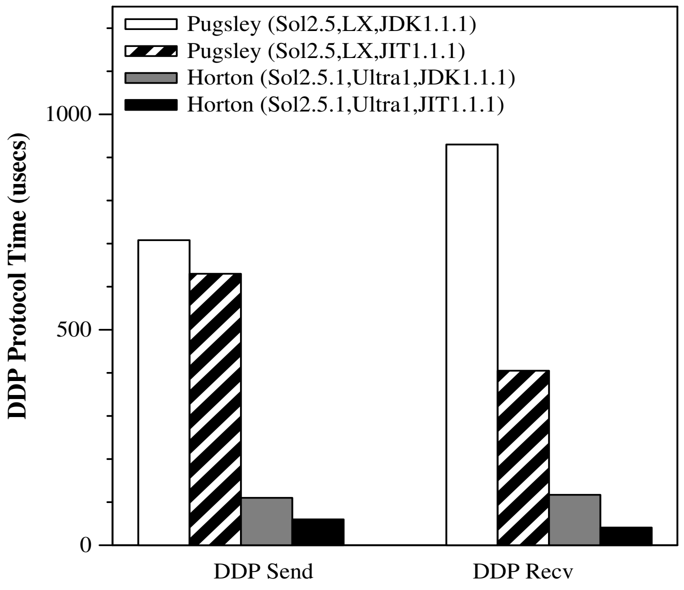


Figure 5: Performance improvement via just-in-time compilation

4.2 Just-in-time Compilation

The performance of interpreted Java-based protocols and subsystems will probably always trail their natively developed and compiled counterparts. However, approaches such as just-in-time compilation [3, 16], which performs runtime compilation of Java bytecode into that directly usable by the underlying host system, may reduce this gap. Figure 5 depicts the performance improvements attributable to the just-in-time (JIT) compilation of our JavaTalk implementation. For these measurements, we ran the same tests under the same conditions as before but used an interpreter which performed just-in-time compilation. The results indicate that on slower machines (our SparcStation-LX system named Pugsley), JIT-DDP performed approximately 12% faster on send and roughly 200% faster on receive. On a faster machine (our Ultra-1 system named Horton), JIT-DDP performed roughly 50% and 250% faster on send and receive processing respectively. Receive-time processing improved by a greater factor than did send-time processing when just-in-time compilation was applied. We attribute some of this improvement to the fact that receive-time processing generally took longer and so there was more room for improvement. However, it is unlikely that this is the entire explanation.

There is a non-zero overhead imposed by just-in-time compilation which converts Java bytecode to native machine code (presumably) when a Java class is first referenced. We were unable to assess this overhead due to a lack of control over when and how just-in-time compilation occurs and due to a lack of overall documentation on the just-in-time compiler. However, one can assume that there is a break-even point before which the overhead of just-in-time compilation is greater than the performance overhead of Java interpretation and after which the performance improvement of executing native code outweighs the cost of just-in-time compilation. Obviously, the longer the particular communication session, the greater the savings.

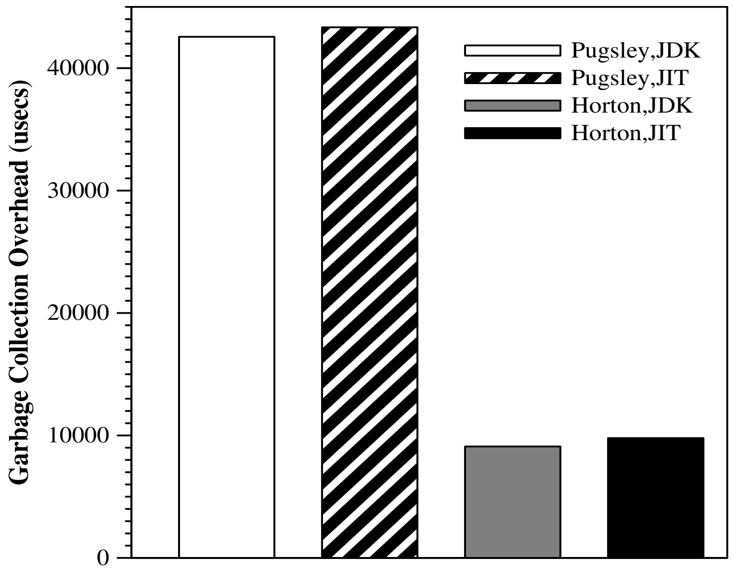


Figure 6: Java Garbage collector Overheads

4.3 Garbage Collection Overhead

Memory management in Java programs is much simpler than those in traditional systems programming languages (e.g. C) because memory need not be explicitly deallocated due to the fact that the virtual machine's garbage collector periodically reclaims unused memory. Garbage collection can greatly simplify code[7], but that simplification does have its price because the virtual machine must periodically suspend execution of programs and root through memory to reclaim that no longer in use. Consequently, the overhead of garbage collection should also be taken into account when considering the cost/benefit tradeoffs

[7]For example, our subsystem does not define a buffer-deallocation convention since memory need not be explicitly freed at all — the garbage collector will find it when it is no longer in use.

of developing and executing protocols in Java. One can consider the costs of garbage collection to consist of two parts — a fixed part relating to the cost of suspending current programs and starting the garbage collector itself and a variable part relating to the particular garbage collection algorithm in use and how much memory it must root through. Because the garbage collector is normally only invoked when the virtual machine is running low on memory and because that condition can occur non-deterministically, protocols sensitive to delay and protocols providing real-time services may not be able to function correctly.

In order to gain an understanding of garbage collector overhead, we wrote a very simple program that took advantage of the fact that the garbage collector can be explicitly invoked. By repeatedly invoking the garbage collector and by sampling the start and end time and dividing by the number of iterations, we were able to measure the time each pass took. Although not large in absolute time, the results depicted in Figure 6 indicate that garbage collection introduces a substantial overhead when compared against the time it takes to process a DDP packet. For example, on our Sparc-LX system (Pugsley), garbage collection takes almost 50 times longer than the time it takes to process a single DDP packet (43 000 vs. 930 μ-seconds). On our faster Ultra-1 system (Horton), that gap widens to almost 80 : 1 (9 100 vs. 117 μ-seconds). Just-in-time compilation had no effect on garbage collection (the plots in Figure 6 labeled "JIT") since the virtual machine already implements it in native code. Although garbage collection will automatically occur at non-deterministic points in time when memory is low, protocols can themselves preemptively invoke it when they are not processing packets. However, even if a protocol knows that it will be idle for a period of time sufficient to perform garbage collection, it may not know if all other protocols will likewise be idle.

4.4 Entering and Exiting the Virtual Machine

In order to send and receive packets via network-interfaces located in the underlying host system, packets must travel from inside the Java virtual machine to outside via native methods (e.g. C functions). Consequently, the latency (if any) of this boundary crossing must be considered when analyzing the viability of this approach. To gain an understanding of how long it takes to invoke a native-C method from within Java (the latency to exit and re-enter the Java virtual machine), we constructed a null Java class whose sole purpose was to invoke a native-C and a Java method. By invoking each method 1 000 000 times and measuring the time before and after, we were able to assess the time per invocation for each type of method. The difference between them represents how much longer a native-C invocation takes. Figure 7 contains measurements indicating the amount of time (in μ-seconds) that native-C and Java method invocations cost when passed a 1K-byte buffer. As with our other measurements, all values were averaged over multiple samples with a 95% confidence interval of at most 10% of measured values.

The measurements for both SparcStation-LX (Pugsley) and Ultra-1 (Horton) systems indicate that the difference between native-C and Java method invocations is quite small (less than 2 μ-seconds), and in most cases, only 20-25% more. Because native method invocations are only used to enter or exit the virtual machine and because only one such invocation will probably be used during the entire protocol processing of a packet, one can consider the extra 1 or 2 μ-seconds to be negligible. It should also be noted that this overhead is not present when executing in JavaStations (the system model depicted in Figure 2c) because no native code is necessary to access the underlying network-interface devices. Lastly, just-in-time compilation reduced the difference between native-C and Java method invocations (the plots labeled "JIT" in Figure 7)

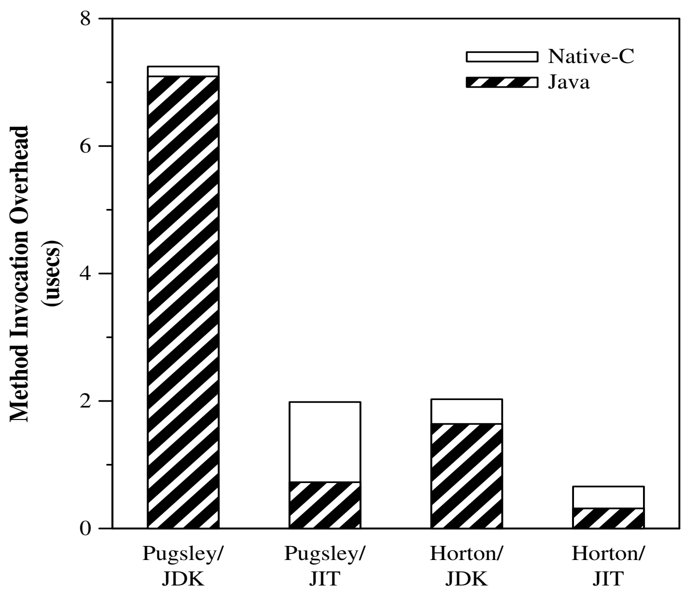


Figure 7: Native-C and Java Method Invocation Times (μ-seconds)

by a factor of almost 4 : 1 on SparcStation-LXs and roughly 3 : 1 for Ultra-1 systems.

5 Discussion

In this section we discuss some of the issues resulting from our subsystem design and implementation choices. We first focus on security and addressing and then examine architectural limitations imposed by the three different system models we previously identified.

5.1 Security and Addressing Considerations

In addition to the problem of performance, other, equally important ramifications of using this Java-based approach should also be considered. Probably the single most important consideration centers around network security and the potential for its abuse by this apporach. There are many aspects to network security: address spoofing (e.g. using the addresses of other systems), malicious behavior (e.g. intentionally transmitting incorrect routing information), and packet capturing (e.g. capturing of passwords and other sensitive data) are just a few. We first address why these security concerns are exacerbated by Java-based protocols and then, where possible, identify potential solutions.

The manner in which applications, protocols, and protocol subsystems coexist in the same address space inside the Java virtual machine is one reason why Java-based protocols can exacerbate security problems. In traditional systems, protocols normally reside within the operating system kernel and are separated from applications by a user/kernel boundary that, coupled with different privilege levels, prevents applications from improperly configuring protocols or loading and executing malicious protocol code. In Java, no such user/kernel boundary nor differing levels of privilege exist. Consequently, the execution of arbitrary and potentially malicious protocol code cannot be prevented because the Java virtual machine has no knowledge of the semantic meaning of packets that may emanate from within it.

Traditional protocol specifications have assumed that certain semantics are enforced by their underlying implementations. Those semantics include the binding of addresses to unique physical resources (e.g. network interface cards and system CPUs) and the assumption that only a single instance of a particular protocol can operate simultaneously (e.g. only a single IP, TCP, and UDP can operate inside a given system). When all protocol implementations execute within the operating system kernel and privilege levels protect the user/kernel boundary, network security may be provided. However, because Java-based protocols execute within user space and because multiple instances of the same protocol can be executing simultaneously in different subsystems, those semantics cannot be enforced.

Java-based protocol implementations can violate those semantics and the assumption that they are enforceable by their implementations in at least three ways. First, because multiple Java virtual machines can execute simultaneously in host-based configurations (see Figures 2a and 2b), multiple, independent copies of the same protocol could therefore execute simultaneously. Second, multiple copies of a given protocol could potentially assign the same address to different physical resources thus creating confusion with peer protocols. Third, current address assignment algorithms are typically unable to map between different instantiations of a given protocol. For example, if two different UDP implementations are operating within a given system and a UDP packet is received by IP, which UDP implementation should receive it? Handing the packet to both UDP implementations may cause unintended privacy violations because a maliciously behaving UDP could capture data not intended for it. Unfortunately, current in-kernel IP implementations pass the packet to each UDP because they have no way to distinguish which implementation should legitimately receive it.

Several different approaches can be applied to solve the addressing and security issues highlighted by *HotLava*. They are:

- In-kernel end-point servers can be used to arbitrate addressing among user-level subsystems (both Java and non-Java based) [25].
- Dynamic address assignment protocols like DHCP [4] and AppleTalk's Name Binding Protocol (NBP) can be extended to allow the assignment of multiple, different logical addresses per physical entity. For example, each different JavaTalk instantiation would each receive a different network-layer address even though they may be operating on the same system.
- The Java virtual machine's notion of security can be enhanced to encompass network addressing so that protocol address spoofing can be prevented. For example, the virtual machine could be extended to prevent multiple network-layer protocols from using the same address.
- A new addressing architecture obviating the need to couple network addresses to physical resources can be developed and deployed.

We discuss each of these in turn.

End-point servers The first solution involves the development and deployment of kernel-based end-point servers which provide address assignment and arbitration among multiple subsystems. This solution, though, is not without its own costs. Address assignment is now tied to an in-kernel module which cannot be changed nor modified as easily as one in user-space. Further, the end-point server method only works for certain protocol families and must be modified each time a new protocol family is added. This solution also does not prevent user-level protocol implementations from circumventing the end-point server by utilizing the services of protocols lower in the protocol graph. For example, if we used an end-point server on top of IP to arbitrate the provision of transport-layer addresses, it cannot prevent a protocol implementation from contacting IP directly to send and receive packets. This solution is also non-portable in that the end-point server is operating-system specific and must be ported to each system wishing to execute Java-based protocols. Lastly, this solution can introduce performance impediments as layered demultiplexing is re-introduced rather than eliminated.

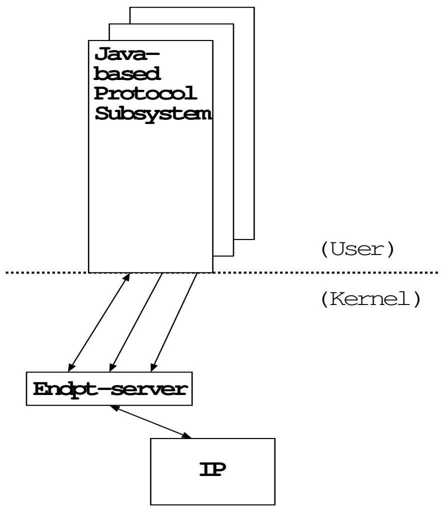


Figure 8: Example end-point server

Dynamic address assignment protocols The second solution involves the extension of current dynamic address assignment protocols in order to support multiple addresses per physical system. For example, DHCP could be modified to assign multiple IP addresses to the same MAC-layer address. In effect, each different subsystem currently in use would obtain its own IP address. However, this solution is not without its costs. The domain-name address resolution protocol [18] may also require modification which can be extremely expensive for a fielded implementation. Lastly, this solution does not necessarily prevent spoofing attacks because it still requires cooperating user-level subsystems — there is no mechanism to prevent a protocol implementation from using any address it chooses.

Extending Java Security The third solution involves extending the notion of security within the Java virtual machine to encompasses network addressing. Currently, the Java virtual machine provides some level of security through the restriction of access to underlying files and to whom UDP and TCP packets can be sent (via Java's TCP and UDP socket classes). These security mechanisms could be augmented to mediate access to IP and Ethernet and to disallow certains address assignments or block improper address usage. However, this solution presents considerable costs in that existing virtual machines would require modification. Further, this solution is piecemeal at best because the virtual machine could only provide security for protocol addressing that it specifically knows about. Deploying new protocol families would require upgrading all existing virtual machines.

New Address Architecture The last solution would obviate current security problems by decoupling the assignment of protocol addresses to physical devices. Solutions would probably still be based on dynamic address assignment but would not link security and authentication to network addressing as is done in current addressing architectures. Instead, public-key encryption and authentication techniques could be used. However, this approach is probably the most costly since current protocols and operating systems would require modification and because current encryption and authentication algorithms are generally computationally expensive.

5.2 Architectural Limitations

Several architectural limitations are imposed by the different system models we have previously identified and by current Java compilers and interpreters. When executing protocols and their accompanying subsystems within Web browsers (see Figure 2b), access to underlying network devices via native code is often prevented by coarse-grained security policies implemented by that Web browser. For example, some browsers (e.g. Netscape Navigator) may not allow Java code to utilize native methods nor access files on the underlying system — capabilities needed by adapter protocols for sending and receiving packets on underlying physical media. Other browsers (e.g. HotJava) do allow access to files and network-interface devices via native code but require special configuration. Further, Web browsers may not be able to provide or curtail access to these resources on a per-class basis. Even if they could, access to the underlying system devices is often restricted to the root/administrator user.

Although the amount of native code required by *HotLava* is small, the fact that it is required in host-based incarnations (Figures 2a and 2b) can itself be limiting. For example, our Ethernet adapter protocol, which allows Java-based protocols access to the underlying Ethernet device driver on Solaris 2.x systems, required only 500 lines of C-code. Its counterpart, the Ethernet Java class, required only 250 lines of Java code. An adapter protocol permitting access to IP on Solaris 2.x required even fewer lines of native C-code (approximately 180). When considered in context of our entire JavaTalk implementation, which required approximately 4 000 lines of Java code, the percentage of native code is fairly small (6%). As the amount of Java-based protocol code grows (e.g. as additional protocols are implemented), this percentage becomes even smaller. Still, the development and deployment of corresponding native code for each desired platform may be prohibitive and can detract from the desirability of Java-based protocols. However, it is envisioned that distributing ready-made adapter protocols for popular platforms and media types will help alleviate this burden. It should be noted that native code is not required for network-computer based configurations (Figure 2c) because their operating system and device drivers are all coded within Java.

Finally, despite the fact that the Java programming language has built-in features for multi-threading (e.g. synchronization primitives) and despite the fact that they are supported by the virtual machine, true parallelism is not yet available. For example, if the underlying system has multiple CPUs, applications (or protocols) cannot specify which CPU to run a thread on nor can threads migrate from CPU to CPU. Support for parallelism can be an important approach for improving the performance of protocols as well as applications and should almost certainly be added in future versions of the JDK.

6 Conclusions

The Java programming language and virtual machine offer enticing opportunities for developing portable protocol and subsystem implementations. Further, the ability to load and execute new protocol code as needed allows application programmers to download custom-tailored protocols on the fly. We have investigated the use of Java and its virtual machine as a protocol development and execution environment. We have developed a portable, Java-based protocol subsystem taking advantage of many of the enhanced features provided by Java and have measured some aspects of the performance overhead introduced by its interpretation.

Our results indicate that the performance overhead is roughly $10:1$ over natively compiled and executed code and that techniques such as just-in-time compilation can reduce it. At present, the gap between interpreted Java code and natively compiled and executed protocol code is roughly equivalent to the gap in performance of native code between four year old hardware and that available today. While improvements in Java compiler and interpreter technology are underway, the gap between Java and native code cannot continue to decline indefinitely. At some point, performance tuning of compilers and interpreters should level off and then roughly match operating system tuning efforts. When that occurs, performance improvements attributable to increased processor speeds will most likely benefit Java-based and native protocol code equally. Consequently, this approach may never be optimal for time-sensitive applications or those needing the utmost in performance.

Although this approach permits the development of portable protocol implementations that are virtually system-independent, a small amount of native, system-dependent code may still be necessary to access network-interface devices. We examined how much native code is required and found it to be minimal, accounting for less than 6% of total Java-based protocol code.

References

[1] AT&T. *UNIX System V Network Programmer's Guide*. Prentice-Hall Inc., 1987.

[2] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[3] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *11th Annual Symposium on Principles of Programming Languages (POPL-11)*, pages 297–302, January 1984.

[4] R. Droms. Dynamic host configuration protocol. Request for Comments (Experimental) RFC 1541, Internet Engineering Task Force, oct 1993. (Obsoletes RFC1531).

[5] Joseph R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[6] James Gosling, Bill Joy, and Guy Steele. The Java^{TM} language specification. Technical report, Sun Microsystems, October 1996. Version 1.0.

[7] Gerard J. Holzmann. Standardized protocol interfaces. *Software - Practice and Experience*, 23(7):711–93, July 1993.

[8] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[9] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM-1988 Symposium*, pages 314–329, August 1988.

[10] Bobby Krupczak, Mostafa Ammar, and Ken Calvert. Multi-Subsystem Protocol Architectures: Motivation and Experience with an Adapter-Based Approach. In *Proceedings of IEEE INFOCOM*, March 1996.

[11] Bobby Krupczak, Ken Calvert, and Mostafa Ammar. Protocol Portability through Module Encapsulation. In *Proceedings of the International Conference on Network Protocols (ICNP-96)*, pages 56–63, October 1996.

[12] Bobby Krupczak, Kenneth L. Calvert, and Mostafa H. Ammar. Increasing the portability and re-usability of protocol code. *IEEE/ACM Transactions on Networking*, 1997. (to appear).

[13] R. Greg Lavender, Dennis G. Kafura, and Chris J. Tomlinson. Implementing communication protocols using object-oriented techniques. Technical Report TR-93-11, Department of Computer Science, Virginia Tech, 1992.

[14] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1st edition, 1989.

[15] Peter W. Madany. JavaOS: A standalone java environment. Technical report, Sun Microsystems, May 1996. White Paper.

[16] Sun Microsystems. Java JIT compiler overview. http://www.sun.com/workshop/java/jit/.

[17] Sun Microsystems. The Java virtual machine specification. Technical report, Sun Microsystems, August 1995. Version 1.0 Beta Draft.

[18] P. Mockapetris. DNS encoding of network names and other types. Request for Comments RFC 1101, Internet Engineering Task Force, April 1989.

[19] Thomas Plagemann, Bernhard Plattner, Martin Vogt, and Thomas Walter. Modules as building blocks for protocol configuration. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*, September 1993.

[20] Pardyak Przemyslaw and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 201–212, October 1996.

[21] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[22] Douglas C. Schmidt and Tatsuya Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, May 1993.

[23] Douglas C. Schmidt and Tatsuya Suda. An object-oriented framework for dynamically configuring extensible distributed communication systems. *IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, December 1994.

[24] Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppenheimer. *Inside AppleTalk*. Addison-Wesley, 1st edition, 1989.

[25] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[26] Christian Tschudin. Flexible protocol stacks. In *ACM SIGCOMM-1991 Symposium*, pages 197–205, September 1991.