More Efficient Serialization and RMI for Java

Michael Philippsen, Bernhard Haumacher, and Christian Nester

Computer Science Department, University of Karlsruhe
Am Fasanengarten 5, 76128 Karlsruhe, Germany
[phlipp|hauma|nester]@ira.uka.de
http://wwwipd.ira.uka.de/JavaParty/

Abstract. In current Java implementations, Remote Method Invocation (RMI) is too slow, especially for high performance computing. RMI is designed for wide-area and high-latency networks, it is based on a slow object serialization, and it does not support high-performance communication networks.

The paper demonstrates that a much faster drop-in RMI and an efficient drop-in serialization can be designed and implemented completely in Java without any native code. Moreover, the re-designed RMI supports non-TCP/IP communication networks, even with heterogeneous transport protocols. We demonstrate that for high performance computing some of the official serialization's generality can and should be traded for speed. As a by-product, a benchmark collection for RMI is presented.

On PCs connected through Ethernet, the better serialization and the improved RMI save a median of 45% (maximum of 71%) of the runtime for some set of arguments. On our Myrinet-based ParaStation network (a cluster of DEC Alphas) we save a median of 85% (maximum of 96%), compared to standard RMI, standard serialization, and Fast Ethernet; a remote method invocation runs as fast as 80 μs round trip time, compared to about 1.5 ms.

1 Introduction

From the activities of the Java Grande Forum [7, 17] and from early comparative studies [6] it is obvious that there is growing interest in using Java for high-performance applications. Among other needs, these applications frequently demand a parallel computing infrastructure. Although Java offers appropriate mechanisms to implement Internet scale client/server applications, Java's remote method invocation (RMI, [20]) is too slow for environments with low latency and high bandwidth networks, e.g., clusters of workstations, IBM SP/2, and SGI Origin.

1.1 Breakdown of RMI cost

We have studied the cost of a remote method invocation on two platforms:

- two PCs: 350 MHz Pentium II, running Windows NT 4.0 Workstation, isolated from the LAN to avoid packet collisions but connected to each other by Ethernet, JDK 1.2 (JIT enabled).

- a cluster of 8 Digital Alphas: 500 MHz, running Digital UNIX, connected by Fast Ethernet, JDK 1.1.6 (regular JIT; the JDK 1.2beta was too buggy).

For three different types of objects we measured the time of a remote invocation of `ping(obj)` returning the same `obj`. A part of this time is spent when communicating over existing socket connections. We have timed this separately (round trip), but including serialization. Finally, we measured the time needed for the JDK-serialization of the argument alone, i.e. without any communication.

Table 1. Ping times (μs) of RMI (=100%), socket communication, and just JDK-serialization. The argument `obj` has either 32 `int` values, 4 `int` values plus 2 `null` pointers, or it is a balanced binary tree of 15 objects each of which holds 4 `ints`.

	μs per object	32 `int`	`4int 2null`	tree(15)
PC	RMI `ping(obj)`	2287	1456	3108
	`socket(obj)`	1900 83%	1053 72%	2528 81%
	`serialize(obj)`	840 37%	368 25%	1252 40%
DEC	RMI `ping(obj)`	7633	4312	14713
	`socket(obj)`	6728 88%	2927 68%	12494 85%
	`serialize(obj)`	4332 57%	1724 40%	9582 65%

Table 1 gives the results; other types of objects behave similarly. For large objects with array data the RMI overhead is about constant, so that serialization and low-level communication dominate the overall cost. As a rule of thumb however, Java's object serialization takes at least 25% of the cost of a remote method invocation. The cost of serialization grows with growing object structures and up to 65% in our measurements. The percentage is bigger for slower JDK implementations. The RMI overhead is in the range of 0.4 to 2.2 milliseconds. Benchmarks conducted by the Manta team [19] show similar results.

1.2 Organization of this Paper

We present work on all three areas (serialization, RMI, and network) to achieve best performance improvements. After a short discussion of related work in section 2, section 3 discusses the central optimization ideas and the design issues of a better serialization. Section 4 presents the key ideas and the design of a much leaner RMI. Both our serialization and our RMI can be used (individually or in combination) as drop-in replacements for standard JDK equivalents. They are written entirely in Java and are portable. Section 5 briefly presents the Myrinet-based ParaStation network that we use instead of Ethernet to demonstrate that our RMI can easily be used over non-TCP/IP networking hardware. Section 6 discusses the benchmark collection and quantitative results.

2 Related Work

Some groups have published ideas on more efficient RMI implementations or on better serializations.

At Illinois University, an improved remote method invocation has been implemented that is based on an alternative object serialization, see [18]. Thiruvathukal et al. experimented with explicit routines to write and read an object's instance variables. In our work, we use explicit routines as well but show that close interaction with the buffer management can improve the performance.

Henri Bal's group at Amsterdam is currently working on the compiler project *Manta* [19]. Manta has an efficient remote method invocation (35 μs for a remote null invocation, i.e., without serialization) but it does not have an efficient RMI package. Manta is based on a transparent extension of Java for distributed environments; the remote invocation is part of that environment and cannot be used as a separate RMI package. Manta compiles a subset of Java to native code on a high-performance network of PCs. The implementation of serialization involves the automatic generation of marshaling routines which avoid dynamic inspection of the object structure and make use of the fact that Manta knows the layout of objects in memory. The paper does not mention performance numbers on serialization of general objects, i.e., graphs. Similar to this work, we use explicit marshaling routines. However, our work sticks to Java, avoids native code, is easily retargetable to other high-performance communication hardware, and is portable. Our serialization and RMI packages can be used (individually or in combination) as a drop-in replacement by anybody. They do not require particular platforms or particular native compilers.

There are other approaches to Java computing on clusters, where object serialization is not an issue. For example, in *Java/DSM* [23] a JVM is implemented on top of Treadmarks [9]. Since no explicit communication is necessary and because all communication is handled by the underlying DSM, no serialization is necessary. However, although this approach has the conceptual advantage of being transparent, there are no performance numbers available to us.

Two groups (Welsch [22], Chang and von Eicken [3]) work on direct access to underlying machine resources for communication purposes. Welsch implemented an RMI which is barely faster than Sun's RMI. Chang and von Eicken implemented an interface to VIA for fast communication but did not provide any support for RMI. To achieve efficiency, both groups use memory regions that are not handled by the JVM.

An orthogonal approach is to avoid object serialization by means of object caching. Objects that are not sent will not cause any serialization overhead. See [11] for the discussion of a prototype implementation of this idea and some performance numbers.

Whereas the impact of serialization performance on Grande applications is obvious, serialization will become relevant for Corba as well. Future versions of IIOP (Internet inter-ORB protocol) will apply serialization techniques [14].

Horb [5] and Voyager by ObjectSpace [13] are alternative distributed object technologies available for Java. In contrast to our work they are neither drop-in replacements of RMI nor are they designed for high-performance computing over non-TCP/IP networks.

Breg et al. [2] at Indiana University have studied RMI interoperability and performance. In particular they have ported a subset of RMI to run over the Nexus runtime system. In contrast to this work, our design can easily be retargeted to arbitrary communication networks and achieves better performance.

We don't know of any other collection of benchmark programs for RMI, although most of the groups mentioned above have published some quantitative results which are not comparable to each other.

3 Improving Serialization Performance

3.1 Basics of Object Serialization

Object serialization [16] is a significant functionality needed by Java's RMI implementation. When a method is called from a remote JVM, method arguments are either passed by reference or by copy. For objects and primitive type values that are passed by copy, object serialization is needed: The objects are turned into a byte array representation, including all their primitive type instance variables and including the complete graph of objects to which all their non-primitive instance variables refer. This wire format is unpacked at the recipient and turned back into a deep copy of the graph of argument objects. The serialization can copy even cyclic graphs from the caller's JVM to the callee's JVM; the serialization keeps and monitors a hash-table of objects that have already been packed to avoid repetition and infinite loops. For every single remote method invocation, this table has to be reset, since part of the objects' state might have been modified.

In general, programmers do not implement marshaling and unmarshaling. Instead, they rely on Java's reflection mechanism (dynamic type introspection) to automatically derive an appropriate byte array representation. However, programmers can provide routines (`writeObject` or `writeExternal`) that do more specific operations. These routines are invoked by the serialization mechanism instead of introspection. Similar routines must be provided for the recipient.

From the method declaration, the target JVM of a remote invocation knows the declared types of all objects expected as arguments. However, the *concrete* type of individual arguments can be any subtype thereof. Hence, the byte stream representation needs to be augmented with type information. Any implementation of remote method invocation that passes objects by value will have to pass the type as well.

In the remainder of this paper, the term "serialization" refers to the functionality of writing and reading byte array representations in general. The official implementation of serialization that is available in the JDK is called "JDK-serialization". Our implementation is called "UKA-serialization".

For regular users of object serialization, it is a nice feature that the JDK offers general code that can do the marshaling and unmarshaling automatically by dynamic type introspection of the object. For high performance computing however, better performance can be achieved by means of explicit marshaling which can be generated automatically. Manta generates such routines during native compilation. We are working on a tool to generate Java marshaling methods by means of a class-file transformation.

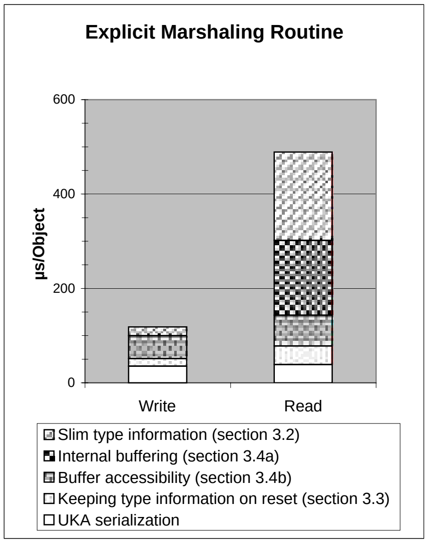

Fig. 1. The full bars show the times needed by the JDK-serialization to write/read an object with 32 `int` values with explicit marshaling routines. We have used JDK 1.2beta3 (JIT enabled) on a 300 MHz Sun Ultra 10/Ultra Sparc IIi, running Solaris 2.6. (We have noticed similar results on our PC and DEC platform with other JDK releases.) The JDK-serialization offers two wire protocols. Although protocol 2 is default, RMI uses protocol 1 because it is slightly faster. Our comparisons use protocol 1 as well. The patterned areas show the individual savings due to the optimizations discussed in sections 3.2 to 3.4. By switching on these optimizations, only the times of the lower white boxes remain. Similar effects can be observed for other types of objects, see Table 2.

The following sections 3.2 to 3.4 will discuss the patterned areas of the bars of Figure 1. The UKA-serialization can avoid all of them. The cost of serialization drops from the total bars to the white boxes when the UKA-serialization is used instead of the JDK-serialization. Suggestions for further improvements and details on the design of the UKA-serialization code are discussed in sections 3.5 to 3.7. The quantitative effects of UKA-serialization are summarized in section 3.8.

3.2 Slim Encoding of Type Information

Persistent objects that have been stored to disk must be readable even if the ByteCode that was originally used to instantiate the object is no longer available. Therefore, the JDK-serialization includes the complete type description in the stream of bytes that represents the state of an object being serialized.

For parallel Java programs on clusters of workstations and DMPs this degree of persistence is not required. The life time of all objects is shorter than the runtime of the job. When objects are being communicated it is safe to assume that all nodes have access to the same ByteCode through a common file system. Hence, there is no need to completely encode the type information in the byte stream and to transmit that information over the network. Instead, the UKA-serialization uses a textual encoding of class names and package prefixes. Shorter representations are possible.

Simplifying type information has improved the performance of serialization significantly, see Figure 1.

3.3 Two types of Reset

To achieve copy semantics, every new method invocation has to start with a fresh hash-table so that objects that have been transmitted earlier will be re-transmitted with their current state.[1] The current RMI implementation achieves that effect by creating a new JDK-serialization object for every method invocation. An alternative implementation could call the serialization's `reset` method instead.

The problem with both approaches is that they not only clear the information on objects that have already been transmitted. But in addition, they clear all the information on types.

The UKA-serialization offers a new `reset` routine that only clears the object hash-table but leaves the information on types unchanged. The dotted areas of the bars of Figure 1 show how much improvement the UKA-serialization can achieve by providing a second `reset` routine.

3.4 Better Buffering

The JDK-serialization has two problems with respect to buffering.

a) External versus Internal Buffering. On the side of the recipient, the JDK-serialization does not implement buffering strategies itself. Instead, it uses buffered stream implementations (on top of TCP/IP sockets). The stream's buffering is general and does not know anything about the byte representation of objects. Hence, its buffering is not driven by the number of bytes that are needed to marshal an object.

The UKA-serialization handles the buffering internally and can therefore exploit knowledge about an object's wire representation. The optimized buffering strategy reads all bytes of an object at once. The patterned areas marked 3.4a in Figure 1 shows the effect of external buffering.

[1] As we have mentioned in the Related Work section, caching techniques could be used to often avoid retransmission.

b) Private versus Public Buffers. Because of the external buffering used by the JDK-serialization, programmers cannot directly write into these buffers. Instead, they are required to use special `write` routines.

UKA-serialization on the other hand implements the necessary buffering itself. Hence, there is no longer a need for this additional layer of method invocations. By making the buffer public, explicit marshaling routines can write their data immediately into the buffer. Here, we trade the modularity of the original design for improved speed. The patterned areas marked 3.4b indicate the additional gain that can be achieved by having the explicit marshaling routines write to and read from the buffer directly.

3.5 Reflection Enhancements

Although we haven't implemented it in the UKA-serialization because of our pure-Java approach, some benchmarks clearly indicate that the JNI (Java native interface) should be extended to provide a routine that can copy all primitive-type instance variables of an object into a buffer at once with a single method call. For example, class `Class` could be extended to return an object of a new class `ClassInfo`:

```
ClassInfo getClassInfo(Field[] fields);
```

The object of type `ClassInfo` then provides two routines that do the copying to/from the communication buffer.

```
int   toByteArray(Object obj,    int objectoffset,
                  byte[] buffer, int bufferoffset);
int fromByteArray(Object obj,    int objectoffset,
                  byte[] buffer, int bufferoffset);
```

The first routine copies the bytes that represent all the instance variables into the communication buffer (ideally on the network interface board), starting at the given buffer offset. The first `objectoffset` bytes are left out. The routine returns the number of bytes that have actually been copied. Hence, if the communication buffer is too small to hold all bytes, the routine must be called again, with modified offsets.

Some experiments indicate that the effect of accessible buffers, see Figure 1(3.4a+b), would increase if such routines were made available in the JNI.

3.6 Handling of Floats and Doubles

In scientific applications, floats and arrays of floats are used frequently (the same holds for doubles). It is essential that these data types are packed and unpacked efficiently.

The conversion of these primitive data types into a machine-independent byte representation is (on most machines) a matter of a type cast. However, in the

JDK-serialization, the type cast is implemented in a native method called via JNI (`Float.floatToIntBits(float)`) and hence requires various time consuming operations for check-pointing and state recovery upon JNI entry and JNI exit. We therefore recommend that JIT-builders inline this method and avoid crossing the JNI barrier.

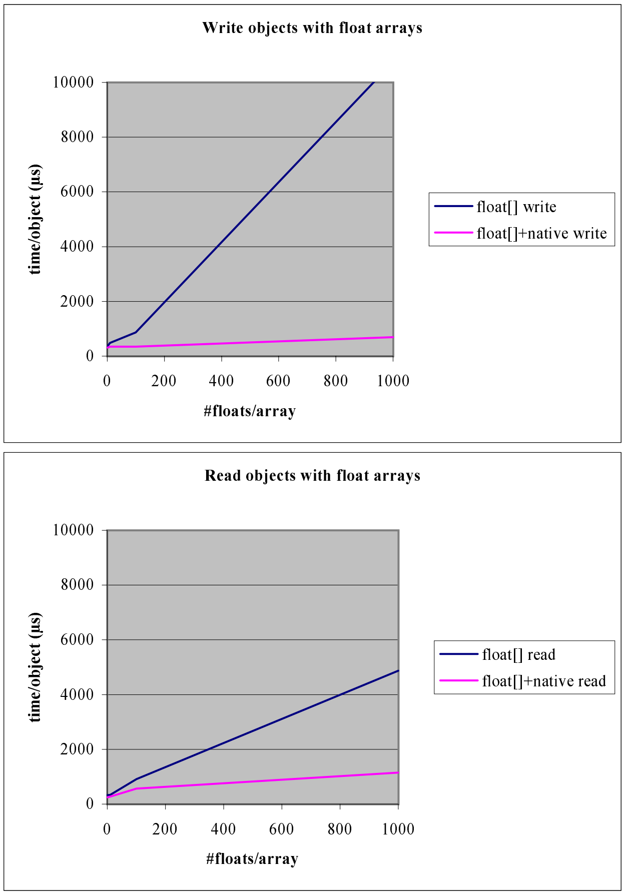

Fig. 2. Serialization of float arrays (same benchmark setup).

Moreover, the JDK-serialization of float arrays (and double arrays) currently invokes the above-mentioned JNI-routine *for every single array element*. We have implemented fast native handling of whole arrays with dramatic improvements, as shown in Figure 2. This, however, cannot be done in pure Java and is left for JVM vendors to fix.

3.7 Design

An important characteristic of the UKA-serialization is that it only improves the performance for objects that are equipped with explicit marshaling and unmarshaling routines discussed above. We call these objects *UKA-aware* objects. For UKA-unaware objects, the UKA-serialization does not help. Instead, standard JDK-serialization is used. Therefore, the JDK-serialization code must – in some way or another – still be present in any design of the UKA-serialization.

In the paragraphs below, we discuss in a increasingly detailed way why straightforward approaches fail and why subclassing the JDK-serialization imposes major problems. Paragraph *(e)* then shows a design that works.

a) CLASSPATH approach fails. The necessary availability of standard JDK-serialization code rules out a design that is based on CLASSPATH modifications.

The straightforward approach to develop a drop-in replacement for the JDK-serialization is to implement all the improvements directly in a copy of the existing serialization classes (`java.io.ObjectOutputStream` and `...InputStream`). The resulting classes must then shadow the JDK classes in the CLASSPATH so that the original classes will no longer be loaded.

The advantage of this approach is that existing code that uses serialization functionality need not be changed in any way. By simply modifying the CLASSPATH, one can switch from the JDK-serialization to a drop-in serialization.

The disadvantage of this approach is that it is not maintainable. Unfortunately, the source code of the JDK-serialization keeps changing significantly from version to version and even from beta release to beta release. Keeping the drop-in implementation current and re-implementing all the improvements in a changing code base is quite a lot of work, especially since existing JDK-serialization needs to survive.

Another straightforward CLASSPATH approach fails: it is impossible to simply rename the JDK-serialization classes and put UKA-classes with the original names into the CLASSPATH. This idea does not work, since the renamed classes can no longer access some native routines because the JNI encodes class names into the names of native methods.

Since for early versions of the UKA-serialization we have suffered under quick release turn-over we decided that the maintainability problem is more significant than the advantages gained by this approach. Therefore, UKA-serialization is designed as subclasses of JDK-serialization classes.

b) Consequences of Subclassing the JDK-Serialization. Designing the UKA-serialization by subclassing the JDK-serialization causes two general disadvantages.

First, existing code that uses serialization functionality has to be modified in two ways: (a) the UKA-subclass needs to be instantiated wherever a JDK-parent-class has been created before. Additionally (b), every existing user-defined subclass of a JDK-class needs to become a subclass of the corresponding UKA-class, i.e., the UKA-classes need to be properly inserted into the inheritance hierarchy. These modifications are sufficient since the UKA-serialization objects are type compatible with the standard ones due to the subclass relationship.

Even if the source of existing code is not available, the class files can be retrofitted to work with the UKA-serialization. Our retrofitting tool modifies the class file's constant table accordingly. After retrofitting, a precompiled class creates instances of the new serialization instead of the original one.

Using the retrofitting trick we were able to use the UKA-serialization in combination with RMI although most of the RMI source code is not part of the JDK distribution.[2]

The second general disadvantage is that the security manager must be set to allow object serialization by a subclass implementation. There is no way to avoid a check by the security manager because it is done in the constructor of JDK's `ObjectOutputStream`.

For using the UKA-serialization from RMI, this is not a big problem, since the RMI security manager allows serialization by subclasses anyway.

c) Problems when Subclassing the JDK-serialization. Unfortunately, after subclassing the JDK-serialization, the standard implementation can no longer be used. This is due to a very restrictive design that prevents reuse.

Since `writeObject` is final in `ObjectOutputStream` it cannot be overridden in a subclass. The API provides an alternative, namely a hook method called `writeObjectOverride` that is transparently invoked in case a private boolean flag (`enableSubclassImplementation`) is set to true. This flag is true only if the parameter-less standard constructor of the JDK-serialization is used for creation, i.e., only if the serialization is implemented in a subclass. The standard constructor however does (intentionally) not properly initialize `ObjectOutputStream`'s data structures and thus prevents using the original serialization implementation.[3]

There are two approaches to cope with that problem. The first approach uses delegation in addition to subclassing. Although the existing code of the JDK-serialization is not touched, the necessary code gets quite complicated. Moreover, for certain constellations of `instanceof`-usage this approach does not work at all. See paragraph *(d)* for the details.

The implementation of the UKA-serialization does not use the delegation approach. Instead we moderately and maintainably changed the existing JDK-serialization classes to enable reuse. This is more "dirty" but results in a cleaner overall design. See paragraph *(e)*.

d) Subclassing plus Delegation. The only way out without touching the implementation of the JDK-serialization is to allocate an additional `ObjectOutputStream` delegate object within the UKA-serialization. Its

[2] Only three RMI classes needed retrofitting, namely `java.rmi.MarshalledObject`, `sun.rmi.server.MarshalInputStream`, and `...MarshalOutputStream`.

[3] The reason for this design is that it allows the security manager to check permissions. However, the same checks could be done with other designs as well.

`writeObject()` method is invoked whenever a UKA-unaware object is serialized. Since the delegate object can be created lazily, it does not introduce any overhead unless UKA-unaware objects are serialized.

Subclassing plus delegation has two disadvantages. First, it is not as simple as it appears. But more importantly, it does not work correctly under all circumstances.

With respect to simplicity it must be noted, that for the delegate object another subclass of the JDK-serialization is needed for cases where existing code itself is using subclasses of the JDK-serialization. (RMI for example does it.) In addition to the hook method mentioned above, the JDK-serialization has several other dummy routines which can be overridden in subclasses. Therefore, if a standard JDK-serialization stream would be used as delegate, it's dummy routines would be called instead of the user-provided implementations. To solve this problem, the delegate is a subclass of the JDK-serialization and provides implementations for *all* methods that can be overridden in the JDK-implementation. The purpose of the additional methods is to forward the call back to the UKA-serialization and hence to the implementation provided by the user's subclass.

There is no guarantee, that subclassing plus delegation works correct in cases where existing code itself is using subclasses of the JDK-serialization and where UKA-unaware objects provide explicit marshaling routines that use the `instanceof` operator to find out the specific type of a current serialization object. (RMI for example does it.) Since the objects are UKA-unaware they are handled by the delegate. Therefore, the `instanceof` operator does no longer signal type compatibility to the serialization subclasses provided by the code.

Since RMI does exactly this (there are subclasses of the JDK-serialization, and the code uses explicit marshaling routines that check the type of a serialization stream object), subclassing plus delegation does not work correctly with RMI. Especially painful are problems with the distributed garbage collector that are hard to track down due to their indeterministic nature. (However, subclassing plus delegation does work correctly with "well-behaved" users of serialization functionality.)

e) Subclassing plus Source Modification. We now present an approach that works. Being based on subclassing the JDK-serialization, it has the general disadvantages discussed in paragraph *(b)*.

The idea is to moderately modify the source code of existing JDK-serialization classes to enable reusing the existing functionality from subclasses. The modification is kept small enough to not affect maintainability.

Three simple changes are sufficient in every JDK release: First, the code of `ObjectOutputStream`'s regular constructor is copied into an additional initialization method `_init(OutputStream)`. Second, to switch between the subclass and the standard serialization, the access modifier of the above-mentioned flag `enableSubclassImplementation` is relaxed from `private` to `protected`.[4] And

[4] Javasoft recently released a beta version of RMI-IIOP. This software uses an extended serialization that is compatible with Corba's IIOP. This extension faces the same

third, the `final` modifier of `writeObject(Object)` is removed to override the method directly and to save an unnecessary call of the hook method.

Since these modifications are simple they can easily be applied to updated versions of the JDK without too much thought. Because no additional serialization object is introduced, the UKA-serialization works fine, even with RMI.

We hope that Sun will incorporate the ideas of the UKA-serialization in future releases of the JDK so that this “dirty” source code modification will not be necessary for ever.

3.8 Quantitative Improvements

Table 2 shows the effect of the UKA-serialization for several types of objects. For an object with 32 `int` values, instead of 66+354=420 μs on the PCs (JDK 1.2) (2166 μs on DEC, JDK 1.1.6) for serialization and de-serialization the UKA-serialization takes 5+15=20 μs (156 μs), which amounts to an improvement of about 95% (93%). The Alpha is slower because of the older Java version and the poor JIT. The second column of Table 2 shows the measurements (in μs) for an object with four `ints` and two `null` pointers. The last column presents the measurements for a balanced binary tree of 15 objects each of which holds 4 `ints`.

Table 2. Improvements for several types of objects.

μs per object		32 `int`		`4int 2null`		tree(15)	
		w	r	w	r	w	r
PC	JDK serialization	66	354	31	153	178	448
	UKA-serialization	5	15	3	11	41	107
improvement %		**92**	**96**	**90**	**93**	**77**	**76**
DEC	JDK serialization	700	1466	271	591	1643	3148
	UKA-serialization	54	102	32	71	216	397
improvement %		**92**	**93**	**88**	**88**	**87**	**87**

While for flat objects about 90% or more of the serialization overhead can be avoided, for the tree of objects the improvement is in the range of 80%. This is due to the fact that the work needed to deal with potential cycles in the graph of objects cannot be reduced significantly.

4 Improving RMI’s Performance and Flexibility

With KaRMI, we have re-designed and re-implemented the RMI of JDK 1.2. The central idea is to provide a lean and fast framework to plug in special

problems as ours. But instead of modifying the access modifier, the authors provide a native library routine to toggle the flag. We consider this to be even more “dirty” than our approach since it is no longer platform independent.

purpose or optimized modules. The framework can be used to plug in optimized implementations of RMI functionality, implementations that trade some of RMI's functionality for speed, or specialized modules, e.g. for special purpose communication hardware or garbage collectors. Currently available are optimized implementations, low-level communication modules for both Ethernet and ParaStation hardware, and several distributed garbage collectors.

The following subsections give an overview of KaRMI, reason about the performance improvements, discuss KaRMI's way to support non-TCP/IP communication hardware, and cover KaRMI's approach towards alternative distributed garbage collectors.

4.1 Clean Interfaces between Design Layers

Similar to the official RMI design, we have three layers (stub/skeleton, reference, and transport).[5] In contrast to the official version however, our design features clear and clearly documented interfaces between the layers. This has two essential advantages. First a performance advantage: In KaRMI a remote method invocation requires just two additional method invocations at the interfaces between the layers and does not create temporary objects. The second main advantage is that alternative reference and transport implementations can easily be added (see section 4.3).

The crucial design flaw of the official RMI is that it exposes its socket implementation to the application level. For example, the application can export objects at specific port numbers. From their level of abstraction, sockets belong to the transport layer. Making them a part of the API and thus visible at the application level, causes that the official RMI *must* use sockets at the transport layer, even if the underlying networking hardware does not support sockets well. Hence, socket semantics need to be implemented in an RMI package that uses a non-TCP/IP network. This is not only unnecessary for Grande applications, it even slows down performance.

The only way out, and hence the only way to exploit the performance of high-speed networks for Grande applications is to separate Java's sockets from the design of RMI. Thus, in KaRMI the application can no longer use the socket factory for example to switch to secured sockets; but we consider this lack irrelevant for Grande applications on a closely connected cluster. (It is of course possible to implement a special purpose transport that uses secured sockets. But this cannot be influenced by the application on a per-object basis.)

[5] A short introduction for those readers not too familiar with the architecture of RMI: The *stub/skeleton layer* maps a specific method invocation onto the generic invocation interface provided by the reference layer. On the side of the callee, the up-call to the application is performed. The *reference layer* is responsible for carrying out the semantics of the invocation. For example, it determines whether the server is a single object or a replicated object requiring communication with multiple locations. The *transport layer* is responsible for low-level communication without any knowledge about the semantics.

Both the clean interfaces between layers and the separation of RMI's design from transport level sockets allow for KaRMI to work fast and to work with high performance communication hardware.

4.2 Performance Improvements

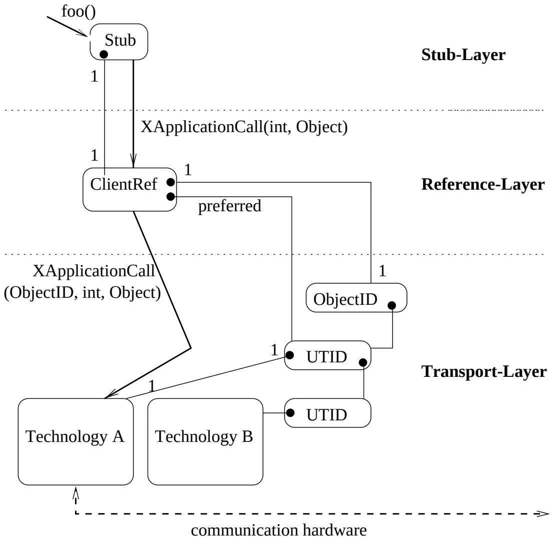

Fig. 3. The three layers of the KaRMI design are shown. A method call is handed from the stub to its reference object (`ClientRef`). The reference object has an `ObjectID` which in turn points to a chain of UTIDs. Each UTID has a pointer to the corresponding technology object that talks to the underlying communication network.

Figure 3 shows the caller-side of the implementation of a remote method invocation in KaRMI. A method call that arrives at the stub is forwarded to its reference object (`ClientRef`) by calling `XApplicationCall(int, Object)`. In a straightforward implementation of remote stubs, the `int` argument of `XApplicationCall` specifies the number of the method to be called; the `Object` argument is the list of parameters for the remote call. Other semantics are possible. The interface provides several copies of `XApplicationCall(...)`, one for each return type (encoded in the method's name: replace `X` for the name of a primitive type, `Object`, or `Void`). Whereas the original RMI always uses return values of type `Object` and hence has to wrap primitive types, KaRMI avoids costly wrapping. Analogously, for the up-call on the side of the callee, the methods are called `XdoApplicationCall(..)`.

The reference object is responsible for addressing the remote object. For that purpose, it stores an `ObjectID` (or several object IDs in case of replication). As will be discussed below, the reference object calls `XApplicationCall(ObjectID,` `int, Object)` of a suitable technology object which performs the low-level communication. The details of addressing and technology objects will be discussed in section 4.3.

Whereas for each remote method invocation, KaRMI creates a single object, RMI needs about 25 objects plus one for each argument and non-void return type. If the connection to the remote side needs to be created first or cannot be re-used, standard RMI needs even more objects. The clean layering leads to better performance since object creation is costly in Java.

Other improvements result from the following ideas:

– The standard RMI uses costly calls of native code and the expensive reflection mechanism to find out about primitive types.[6] There are two native calls per argument or non-void return value plus five native calls per remote method invocation. These can be avoided by a clever serialization. KaRMI uses native calls *only* for the interaction with device drivers.
– In contrast to the official RMI, KaRMI's reference layer detects remote objects that happen to be local and short-cuts object access. Of course, arguments will still be copied to retain RMI's remote object semantics. But no thread switches and no communication are needed in KaRMI.
– The RMI designers prefer hash-tables over other data structures. Hash-tables are used where arrays would be faster or where KaRMI can avoid them completely. Although clearing hash-tables is slow, the RMI code frequently and unnecessarily clears hash-tables before handing them to the garbage collector.
– A little slowdown is caused on some platforms by the fact that the RMI code contains a lot of debugging code that is guarded by boolean flags. At execution time, these flags are actively evaluated. KaRMI does not have any debugging code (instead we remove the debugging code by means of a pre-processor.)

4.3 Technology Objects hide Network Hardware

In contrast to RMI, KaRMI can deal with non-TCP/IP networks. It is even possible to use several types of networking hardware at the same time on the same node.

For that purpose, KaRMI introduces the notion of a communication technology. For each networking hardware available to a node `A`, a corresponding technology object `T` is created upon initialization. `T` contains all the addressing information needed by other nodes to reach `A` via this technology. Moreover, `T` implements everything `A` needs to access the low-level hardware for reaching other nodes via the network of technology `T`. E.g., on a node that has both an Ethernet and a Myrinet available, there will be two technology objects. Both technology

[6] Leaving Java through the JNI and re-entering is much too slow in the current JDK. For Grande Applications that are likely to capitalize on existing fine-tuned native code, a significant penalty cannot be tolerated.

objects implement the transport layer's `XApplicationCall(...)`. Depending on the object to be reached, the reference layer uses the best technology object.

When a reference to an object `R` is made available to a remote node, unique technology identifiers (UTIDs), one for each available technology, are passed to the remote node (by means of serialization). At the remote side, the transport layer examines the incoming UTIDs to check which technologies can be used. This check can be done, even if the recipient has never heard of the technology represented by an incoming UTID. As long as both nodes have at least one common technology, the reference layer can directly use the preferred technology to address the remote object `R`. Otherwise, a bridge object is automatically created on the boundary between two technology domains.

We have implemented an Ethernet technology class that is designed for TCP/IP-sockets and uses Java's sockets. The implementation is quite similar to RMI, i.e., we re-use socket connections, a watch-dog closes connections which are no longer used, etc. Several of the performance improvements mentioned in the previous section are implemented in this class.

In addition to the Ethernet technology class, we have implemented an optimized technology on top of the low-level communication layer of the ParaStation hardware (see section 5). For the optimization, we exploit that packets are guaranteed to be delivered in order. Furthermore, there is no need to protect against network errors, e.g., temporary unavailability of some nodes, connection failure, etc.

Nodes that are located in our cluster instantiate two technology objects, one for Ethernet and one for Myrinet, and use the latter when talking to each other. Since other nodes outside of the cluster are not equipped with Myrinet they instantiate just the technology object for Ethernet. The cluster nodes use the slower protocol when talking to them.

4.4 Pluggable Garbage Collection

Distributed garbage collection is complicated due to common failures of distributed systems such as lost, duplicated, or late messages, as well as crashes of individual nodes. No distributed garbage collection algorithm has been presented yet that is efficient, scalable and fault-tolerant at the same time [15].

RMI's distributed garbage collection is well-designed for wide-area networks where messaging problems, node crashes, and network problems are likely. However, on a closely connected cluster of workstations, a distributed garbage collection algorithm faces different environmental conditions: there is no need for extra messaging to make the algorithm fail-safe, hence more efficiency can be reached.

Since there are more efficient distributed garbage collection algorithms for stable networks, KaRMI provides the opportunity to plug them in, i.e., there is a clean interface for the distributed garbage collector which is independent of the technology layer as well.

For each communication technology, a different distributed garbage collector can be used, e.g., a fast cluster collector within the ParaStation network, an

RMI-like fault-tolerant collector over the LAN. The bridge objects mentioned above which are automatically generated at the boundary between communication technologies make different collectors work together correctly.

4.5 Restrictions

KaRMI is a drop-in replacement for standard RMI. After some changes to the BOOTCLASSPATH environment variable and new generation of stubs and skeletons, existing code can make use of KaRMI's improved performance, even without re-compilation.

However, KaRMI does have some restrictions. As mentioned above, the most significant one is that KaRMI cannot deal with code that explicitly uses the socket factory or port numbers. As we have reasoned above, we consider sockets to be an implementation feature that belongs into the transport layer and therefore do not support them in the API. An unavoidable restriction is that KaRMI cannot be used if existing code uses undocumented RMI classes (IBM's San Francisco project does.)

Minor incompatibilities are that stubs and reference objects are no longer a subclass of `RemoteObject` and that only one registry can be started per JVM. But these changes are unlikely to be noticed.

Since we consider the following three features low-priority for Grande applications, we postponed their implementation. First, KaRMI's current technology objects do not fall back into HTTP if they cannot access the remote host. Second, the current technology objects can only load class files from a common file system, they do not ask a web server to provide class files. Finally, we did not yet implement activation of remote objects on demand.

5 ParaStation Network

ParaStation [21] is a communication technology for connecting off-the-shelf workstations to form a supercomputer. The current ParaStation hardware is based on Myrinet [1], fits into a PCI slot, employs technology used in massively parallel machines, and scales up to 4096 nodes.

ParaStation's user-level message passing software preserves the low latency of the communication hardware by taking the operating system out of the communication path, while still providing full protection in a multiprogramming environment. We have used ParaStation's highly efficient direct access to the low-level communication ports, although we have experimented with ParaStation's emulation of UNIX sockets as well.

For the present study we have used a ParaStation implementation on 8 Digital Alpha machines with 500 MHz. In this environment, ParaStation achieves end-to-end (process-to-process) latencies as low as 48 μs (round trip) and a sustained bandwidth of more than 50 Mbyte/s per channel.

6 Benchmarks and Results

Quite often contributors to the RMI mailing list ask for benchmark results or for a collection of benchmark programs. A lot of speculative discussion is devoted to the problem whether RMI over IIOP will be faster or slower than standard RMI. Unfortunately, there does not seem to be a collection of benchmark programs available (except for various simple ping-pong programs that often measure platform characteristics such as cache-sizes or the garbage collector's efficiency instead of the RMI performance.) The Java Grande Forum's benchmarking activity lacks an RMI benchmark as well.

6.1 Benchmark Collection

We therefore have put together an initial collection of RMI benchmark programs: several kernels and some small applications. The kernels test the specific behavior of RMI under certain controlled stress situations. The small applications use a lot more remote method invocation than is adequate for solving the problems, i.e., often even a sequential implementation could be faster. But on the other hand, they test frequent communication patterns or test RMI's performance when interfered by a lot of thread scheduling activities or synchronization. We do not claim that the benchmark collection is representative for applications that use RMI, but it is a good start to evaluate the performance of RMI's basic capabilities. The collection is available to the public from [8]; please contribute additional RMI benchmark programs.

Each program in the collection is executed often for every parameter setting to cope with finite clock resolution, execution time fluctuations due to cache-effects, JIT-warmup, and outliers (e.g., due to operating system interrupts or other jobs).

For the programs in the collection, the parameter `obj` can be created by a factory. Currently, the benchmark collection comprises factories for the following types of objects; it is easy to add additional factories.

`null`	null pointer

`byte[`n`]` `int[`n`]`
`float[`n`]`
	arrays with n elements (the current benchmark collection uses n=50, 200, 500, 2000, 5000, 20000)

4 `int` 32 `int`	an object with 4 or 32 `int` values
`tree(`n`)`	a balanced binary tree of objects with 4 `int` values and a total of n nodes (the current benchmark collection uses n=15)

a) Kernels that test RMI between two nodes.

- `void ping()`
- `void ping(int, int)`
- `void ping(int, int, float, float)`
- `void ping(obj)` and `obj ping(obj)`
- `void pingpong(obj)` and `obj pingpong(obj)`
  In contrast to a simple `ping` that returns immediately, the client calls back the server (`pong`) before both remote invocations return.

b) Kernel that tests server overload by calls from several clients.

- `obj star(obj)`
  All clients wait at a barrier before they concurrently call a single method at the server.

c) Small application kernels.

- Hamming's problem (from [4]). Given an array of primes a, b, c, $\ldots$ (every second prime in our implementation) output, in increasing order and without duplicates, all integers of the form $a^i \cdot b^j \cdot c^k \ldots \leq n$.
  The remote method invocations have only primitive type arguments. Several threads execute per client node (but have little work to do).
- Paraffin Generation (from [4]). Given an integer n, output the chemical structure of all paraffin molecules for $i \leq n$, without repetition and in order of increasing size. Include all isomers, but no duplicates. The chemical formula for paraffin molecules is $C_i H_{2i+2}$. Any representation for the molecules could be chosen, as long as it clearly distinguishes among isomers. The implemented solution is based on [10].
  This benchmark is designed in the classic master-worker way. Graphs of objects are passed as arguments and return values.
- SOR successive overrelaxation. SOR is an iterative algorithm for solving Laplace equations on a 2d-grid. In every iteration the value of every grid point is updated based on the values of the four neighboring grid points. The parallel implementation is based on the red/black algorithm. During the red phase only the red points of the grid are updated. Red points have only black neighbors and no black points are changed during the red phase. During the black phase the black points are updated in a similar way. Rows are distributed across the available processors. Between phases the values of border points need to be exchanged. The RMI implementation is provided by Jason Maassen [12].

6.2 Results

We have studied four software configurations on three hardware platforms. *Software:* We timed the behavior of the benchmark collection four times, namely on pure RMI, on RMI with the UKA-serialization, on KaRMI with the JDK-serialization, and on KaRMI with the UKA-serialization. Each run timed 64 individual constellations (each kernel for each type of parameter object + small applications). *Hardware:* We have used the two hardware platforms mentioned in section 1.1. In addition, the Alphas have been connected through the ParaStation network.

The lower three box plots in Figure 4 show 2·64 results for each software configuration. The lowest box plot shows the improvement over standard JDK that can be achieved by switching over to the UKA-serialization (plus regular RMI). The second row shows the effect of KaRMI (plus JDK-serialization). The third row shows the combined effect (UKA-serialization plus KaRMI). These first three rows consider Ethernet on PC and Fast Ethernet on Alpha only. The top box plot shows what happens if the ParaStation network and the corresponding KaRMI technology object is used on the DEC cluster (64 results).

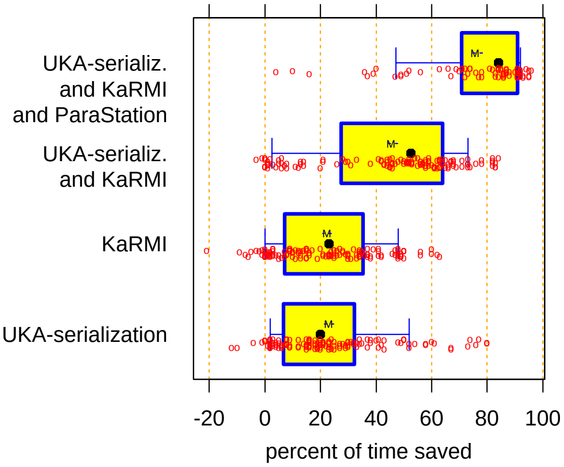

Fig. 4. Percentage of time saved by UKA-serialization, by KaRMI, by a combination of both, and by additional use of the ParaStation network.

For each individual time taken, a small circle represents the percentage of time saved in comparison to the standard RMI implementation. The M represents the mean, the fat dot is the median. The boxes contain the middle half of the circles. For example on ParaStation half the measurements save about 70-90% of the runtime. For a quarter of the measurements, more than 90% of the runtime can be saved. The H-lines indicate the 0.1 and 0.9 quantiles, that is only the smallest tenth of the data are left of the H. A small excerpt of the raw data is given in Table 3.

The box plots show that UKA-serialization and KaRMI almost always improve performance, both when used individually and especially when used in combination. Without special purpose communication hardware a median of 45% of the runtime can be saved; sometimes up to 71%. On ParaStation, a median of 85% can be saved, sometimes up to 96%. Note that it is impossible to use ParaStation with regular RMI, only with KaRMI the ParaStation communication hardware can be employed. Only with both the UKA-serialization and KaRMI, a remote method invocation can run as fast as 117 μs. With a newer JIT from DEC we reached 80μs. However, this JIT was still too buggy to use it throughout this paper.

In each of the box-plots of Figure 4 there are some circles close to zero improvement. These represent kernel measurements with array arguments. When large amounts of data are sent, the low-level communication time is the dominating cost factor that hides any performance improvements in the serialization and in the RMI implementation. Table 3 and Figure 5 give details. On the PCs

Table 3. Excerpt of the raw data. Both individually and in combination, UKA-serialization and KaRMI save time, in particular by means of the ParaStation network. Some DEC timings are left out (...) since their trend is similar to the PC timings.

		RMI	UKA	KaRMI	UKA +	UKA +
timings in μs					KaRMI	KaRMI +
						ParaStation
PC	void ping()	745	689 (8%)	385 (48%)	360 (52%)	NA
	void ping(2 int)	731	673 (8%)	619 (15%)	398 (46%)	
	obj ping(obj)					
	• 32 int	2287	1106 (52%)	1935 (15%)	674 (71%)	
	• 4 int, 2 null	1456	905 (38%)	1104 (24%)	464 (68%)	
	• tree(15)	3108	1772 (43%)	2708 (13%)	1311 (58%)	
	• float[50]	1462	1192 (18%)	1095 (25%)	859 (41%)	
	• float[5000]	37113	36432 (2%)	37123 (0%)	37203 (0%)	
	paraffins	19013	12563 (34%)	18350 (3%)	7121 (53%)	
DEC	void ping()	1451	...		511 (65%)	117 (92%)
	void ping(2 int)	1473			793 (46%)	194 (87%)
	obj ping(obj)					
	• 32 int	7633			1232 (84%)	328 (96%)
	• 4 int, 2 null	4312			1123 (74%)	279 (94%)
	• tree(15)	14713			2485 (83%)	1338 (91%)
	• float[50]	2649			1264 (52%)	483 (82%)
	• float[5000]	16954			12590 (26%)	8664 (49%)
	paraffins (2 PE)	56870			15580 (73%)	19600 (66%)
	paraffins (8 PE)	42290			9450 (78%)	13860 (67%)

(left diagram) the performance improvements approaches zero for big array data. Because of the higher bandwidth, on ParaStation we still see an improvement of above 40% with big arrays, but the general trend is the same. Without the array arguments, all the box plots would shift further towards higher improvements.

Interestingly, the Fast Ethernet implementation saves more runtime than the ParaStation implementation for the Paraffin application. This is due to a thread scheduling problem which causes the current ParaStation library to busily wait at the communication card, when Java threads cannot continue.

For a few circles one can see a slowdown if either the UKA-serialization or KaRMI are used alone. For the UKA-serialization, this is caused by the fact that the standard implementation uses a native routine to create uninitialized array objects. Our implementation currently calls the standard constructor within Java instead.[7] For KaRMI the slowdown can be explained by the fact that the 1.1.6 stubs used on the Alpha handle primitive type arguments more efficiently. The worst circle (-20%) will go away as soon as JDK 1.2 is available on the Alpha.

7 Conclusion

KaRMI is a more efficient RMI for Java. The UKA-serialization is a faster serialization. Both can be used (individually or in combination) as drop-in replace-

[7] We will soon fix that by calling the same native method.

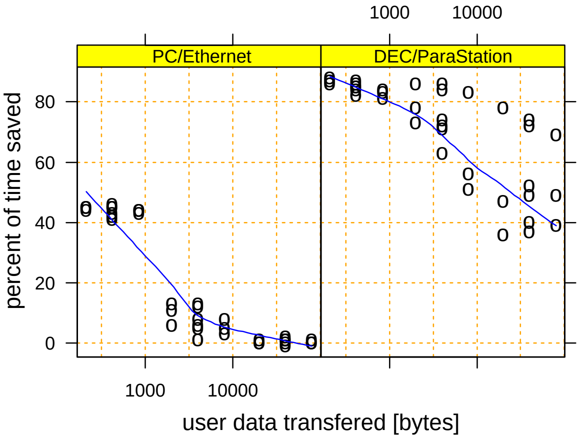

Fig. 5. The percentage of time saved by a combination of the UKA-serialization and KaRMI vanishes with growing array argument sizes.

ments for standard JDK implementations. On PCs connected through Ethernet, a median of 45% of the runtime can be saved, which means a more than sixfold performance improvement. In addition, our re-design and re-implementation allows to use high-performance communication hardware as well. On a cluster of DEC Alphas a median of 85% can be saved; currently a simple remote ping runs as fast as 80 μs. As a by-product, we have developed a benchmark collection for RMI.

Acknowledgments

We would like to thank Lutz Prechelt for help with the statistical evaluation of the raw benchmark data. Matthias Gimbel suffered through the beta-testing. The Java Grande Forum and Siamak Hassanzadeh from Sun Microsystems provided the opportunity and some financial support to find and discuss shortcomings of both the JDK-serialization and the official RMI.

References

1. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jarov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
2. Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, September–November 1998.
3. Chi-Chao Chang and Thorsten von Eicken. Interfacing java to the virtual interface architecture. In *ACM 1999 Java Grande Conference*, San Francisco, 1999. 51–57.

4. John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Elsevier Science Publishers, Holland, 1992.
5. Satoshi Hirano, Yoshiji Yasu, and Hirotaka Igarashi. Performance evaluation of popular distributed object technologies for Java. *Concurrency: Practice and Experience*, 10(11–13):927–940, September–November 1998.
6. Matthias Jacob, Michael Philippsen, and Martin Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice and Experience*, 10(11–13):1143–1154, September–November 1998.
7. Java Grande Forum. http://www.javagrande.org.
8. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
9. P. Keleher, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter USENIX Conf.*, pages 115–131, January 1994.
10. Donald E. Knuth. *The Art of Computer Programming*, volume I. Addison-Wesley, Reading, Mass., 1973.
11. Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java Remote Method Invocation (RMI). In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, 1998.
12. Jason Maassen and Rob van Nieuwpoort. Fast parallel Java. Master's thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, August 1998.
13. ObjectSpace. Voyager. http://www.objectspace.com.
14. OMG. *Objects by Value Specification*, January 1998. ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf.
15. David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.
16. Sun Microsystems Inc., Mountain View, CA. *Java Object Serialization Specification*, November 1998. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf.
17. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, November 7–13, 1998. panel handout.
18. George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–926, September–November 1998.
19. Ronald Veldema, Rob van Nieuwport, Jason Maassen, Henri E. Bal, and Aske Plaat. Efficient remote method invocation. Technical Report IR-450, Vrije Universiteit, Amsterdam, The Netherlands, September 1998.
20. J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.
21. Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. ParaStation: Efficient parallel computing by clustering workstations: Design and evaluation. *Journal of Systems Architecture*, 44(3-4):241–260, December 1997. Elsevier Science Inc., New York.
22. Matt Welsch. NinjaRMI. http://www.cs.berkeley.edu/~mdw/ninja/.
23. Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.