and Distributed Computing, April 12-16, 1999, San Juan, Puerto Rico.

More Efficient Object Serialization

Michael Philippsen and Bernhard Haumacher

University of Karlsruhe, Germany
phlipp@ira.uka.de and hauma@ira.uka.de
http://wwwipd.ira.uka.de/JavaParty/

Abstract. In current Java implementations, Remote Method Invocation is too slow for high performance computing. Since Java's object serialization often takes 25%–50% of the time needed for a remote invocation, an essential step towards a fast RMI is to reduce the cost of serialization.
The paper presents a more efficient object serialization in detail and discusses several show-stoppers we have identified in Sun's official serialization design and implementation. We demonstrate that for high performance computing some of the official serialization's generality can and should be traded for speed. Our serialization is written in Java, can be used as a drop-in replacement, and reduces the serialization overhead by 81% to 97%.

1 Introduction

From the activities of the Java Grande Forum [2, 9] and from early comparative studies [1] 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) is too slow for environments with low latency and high bandwidth networks, e.g., clusters of workstations, IBM SP/2, and SGI Origin. Explicit socket communication – the only reasonable alternative to RMI – is not object-oriented, too low level, and too error-prone to be seriously considered by a broad user community.

Since Java's current object serialization often takes at least 25% and up to 50% of the time needed for a remote invocation, an essential step towards a fast RMI is to reduce the cost of serialization as far as possible.

1.1 Basics of Object Serialization

Object serialization [8] 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 mechanisms (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 implement the above functionality.

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".

1.2 Cost of JDK-Serialization

We have done measurements demonstrating that Java's current object serialization takes 25%–50% of the cost of a remote method invocation.

μs per object	32 `int`		4 `int`, 2 `null`		tree(15)	
RMI `ping(obj)`	4620	100%	3220	100%	6170	100%
Socket `send(obj)`	3290	71%	1820	56%	4720	76%
`Serialize(obj)`	1756	38%	765	24%	3081	50%

Table 1. Ping times (μs) of RMI (=100%), socket communication, and just JDK-serialization. The argument object `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 `int`s.

On two Suns (300 MHz Sun Ultra 10 (Sparcs IIi), running Solaris 2.6) we have used JDK 1.2beta3 (JIT enabled) on some RMI benchmarks.[1] Table 1 gives

[1] 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 results. For three different types of objects we measured the time of a remote invocation of `void ping(obj)`. In addition, we have taken the time needed for the communication over existing Java socket connections. This includes serialization. Finally, we measured the time needed for the JDK-serialization of the argument. It can easily be seen that the serialization takes at least 25% of the time. The cost of serialization grows with growing object structures up to 50% in our measurements. Benchmarks conducted by the Manta team [11] show similar results.

1.3 Organization of this paper

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 looks at specific performance problems of the JDK-serialization and discusses benchmark results that quantify them. Section 4 sketches the design of the UKA-serialization, our more efficient drop-in replacement for the JDK-serialization. In section 5 we give a detailed performance analysis of the UKA-serialization.

2 Related Work

Some groups have published their ideas on improving the JDK-serialization.

- At Illinois University, an improved remote method invocation has been implemented that is based on an alternative object serialization, see [10]. The authors 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 further improve the performance.
- Henri Bal's group at Amsterdam is currently working on the compiler project *Manta* [11]. Manta has an efficient remote method invocation (35 μs for a remote null invocation, i.e., without serialization) on a cluster of workstations connected by Myrinet. To achieve this result, Manta compiles a subset of Java to native code. 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 the 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 and recommend that the JNI (Java native interface) exploits its knowledge of memory layout. However, our work sticks to Java, avoids native code, and can handle polymorphism and subtyping in arguments.
- There are other approaches to Java computing on clusters, where object serialization is not an issue. For example, in *Java/DSM* [12] a JVM is implemented on top of Treadmarks [4]. 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.
- 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 [5] 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, object serialization will as well become relevant for the Corba world in the future. While Corba currently cannot pass objects by copy, OMG [6] is working with Sun and others to add value classes. Future versions of IIOP (Internet inter-ORB protocol) are likely to apply object serialization techniques [7].

In addition to parallel computing, object serialization is a critical issue in the area of persistent object storage. Since in that area the life time of objects is much longer than in high-performance computing, object recovery needs to be feasible even from newer versions of the JDK and from programs that do not have access to the appropriate ByteCode files. Therefore, this area needs other optimization strategies than the ones discussed below.

3 Improving JDK-Serialization Performance

The JDK-serialization has mainly been designed to implement persistent objects and to send and receive objects in typical client-server applications. These goals have strongly influenced some design and implementation decisions.

3.1 Explicit Marshaling instead of Reflection

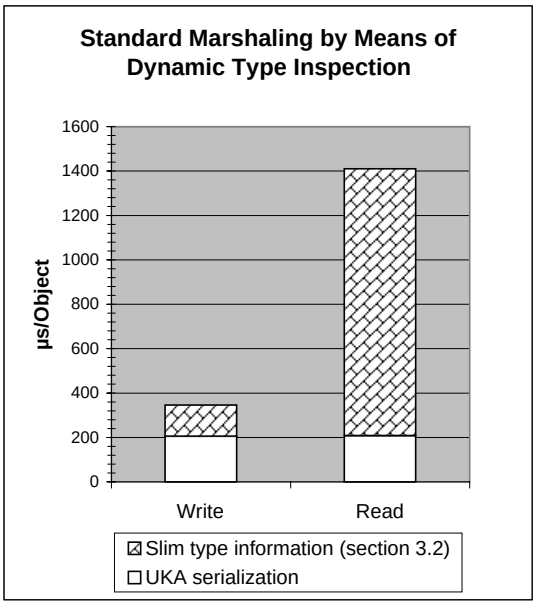
For regular users of object serialization, it is a nice feature that the JDK offers general code that can do the marshaling and unmarshaling automatically by dynamic inspection of the object. For high performance computing however, better performance can be achieved.

Explicit marshaling is much faster when JDK-serialization is used, as a comparison of the full bars from Figure 1 ($\sim$ 346 μs and $\sim$ 1410 μs) and Figure 2 ($\sim$ 120 μs and $\sim$ 490 μs) shows. In Figure 1, dynamic type introspection is used, whereas in Figure 2, the programmer has provided explicit marshaling and unmarshaling routines. Explicit marshaling is still much faster when all applicable optimization of the UKA-serialization are switched on, as a comparison of sizes of the white boxes show.

The following sections 3.2 to 3.4 will discuss the patterned areas of the bars of Figure 2. The UKA-serialization can avoid all of them and even achieves better latency hiding.

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.



The bars show the times needed by the JDK-serialization to write/read an object with 32 `int` values by means of dynamic type introspection.

By switching to a slim type encoding, the UKA-serialization can save the time shown as patterned area. Moreover, with UKA-serialization writing and reading take about the same time.

Fig. 1. JDK-serialization with dynamic type inspection and the effect of all applicable optimizations of the UKA-serialization.

For example, a new version of the class might have a different layout of its instance variables. To cope with these situations, the complete type description is included in the stream of bytes that represents the state of an object being serialized.

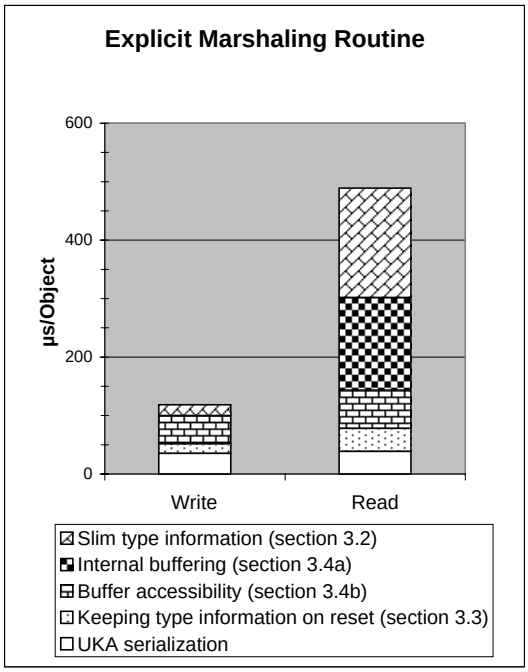
For parallel Java programs on clusters of workstations and DMPs this is not required. The life time of all objects is shorter than the runtime of the job. When objects are being communicated it is safe to assume that all nodes have access to the same ByteCode, for example through a common file system. Hence, there is no need to completely encode and decode the type information in the byte stream and to transmit that information over the network.

The UKA-serialization uses a textual encoding of class names and package prefixes. Even shorter representations are possible. Simplifying type information has improved the performance of serialization significantly. See the patterned areas of the bars of Figures 1 and 2 that are marked 3.2.

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.[2] The current RMI implementation achieves

[2] As we have mentioned in the Related Work section, caching techniques could be used to often avoid retransmission of objects whose states have not changed.



The full bars show the times needed by the JDK-serialization to write/read an object with 32 `int` values with explicit marshaling routines.

The patterned areas show the individual savings due to the optimizations discussed in sections 3.2 to 3.4. Slim type encoding has more effect when dynamic type inspection is used (see Figure 1); but techniques 3.3 and 3.4 can only be used with explicit marshaling.

By switching on all optimizations in the UKA-serialization, only the times of the lower white boxes remain. These are much smaller than what can be achieved with dynamic type inspection.

Fig. 2. JDK-serialization with explicit marshaling and the effect of all applicable optimizations of the UKA-serialization.

that effect by creating a new JDK-serialization object for every method invocation. An alternative implementation could probably 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.

To alleviate this problem, the UKA-serialization offers a new `reset` routine that only clears the object hash-table but leaves the information on types unchanged. The dotted area of the bars of Figure 2 that are marked with 3.3 show how much improvement the UKA-serialization can achieve by providing a second reset routine. (Unfortunately, the current official RMI implementation does not use it.)

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.

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.
Because it is cumbersome to deal with buffer overflow conditions in the explicit marshaling routines, we are working on a tool that automatically generates proper routines.

The patterned area marked 3.4a in Figure 2 shows the effect of external buffering. The patterned area 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 2(3.4a), 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.

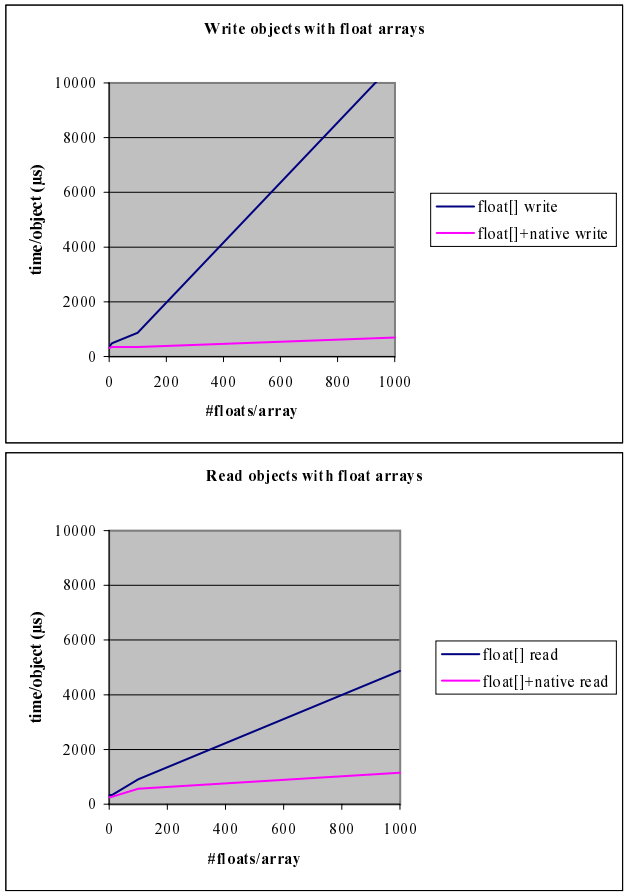


Fig. 3. Serialization of float arrays (same benchmark setup).

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 3. This, however, cannot be done in pure Java and is left for JVM vendors to fix.

4 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 in section 3. 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 sections below, we discuss in a increasingly detailed way why straightforward approaches fail and why subclassing the JDK-serialization imposes major problems. Section 4.5 then shows a design that works.

4.1 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.

4.2 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.[3]

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.

4.3 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.[4]

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,

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

[4] 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.

for certain constellations of `instanceof`-usage this approach does not work at all. See section 4.4 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 section 4.5.

4.4 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 `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.)

4.5 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 section 4.2.

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`.[5] And 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.

5 Results

We have designed the UKA-serialization according to the approach sketched in section 4.5. The implementation includes the optimizations that are discussed in sections 3.1 to 3.4. Further optimizations would need help from the JVM vendors.

5.1 Improvement of Serialization

Section 3 has discussed the individual problems of the JDK-serialization and has provided solutions. Because of all the details, the quantitative result might have passed unnoticed.

Instead of 1410 μs for reading (JDK-serialization of a certain type of object by means of dynamic type inspection) the UKA-serialization takes just 39 μs, which amounts to an improvement of about 97%. For writing, the improvement is in the order of 90%. See the bold line of Table 2. Similar improvements occur for different types of objects. 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

[5] 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 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.

which holds four `ints`. Note that reading and writing now often take about the same time and therefore allow for balanced overlapping and hence better object transfer times. Overlapping is only relevant for large object graphs that are sent in several packets.

μs per object	32 `int`		4 `int`, 2 `null`		tree(15)	
	w	r	w	r	w	r
JDK serialization	346	1410	169	596	1192	1889
UKA-serialization	35	39	19	28	201	354
improvement %	**90**	**97**	**89**	**95**	**83**	**81**
explicit marshaling	228	920	81	308	396	647
slim type encoding	19	187	16	159	72	213
internal buffering	0	159	6	19	0	330
buffer accessibility	48	65	30	49	502	291
two types of reset	16	40	17	33	21	54

Table 2. Improvements for several types of objects.

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.

The lower half of the table shows the contributions of the individual improvement techniques.

5.2 Improvement of RMI

To use the UKA-serialization with RMI, we have applied the retrofitting tool to Sun's RMI class files. Wherever a regular `ObjectOutputStream` has been created in the original code, the upgraded versions used the corresponding UKA-objects. Due to latency hiding effects the improvement (36%–51%) is even better than expected. See Table 3.

μs per object	32 `int`		4 `int`, 2 `null`		tree(15)	
RMI `ping(obj)`	4193	100%	2890	100%	5330	100%
Socket `send(obj)`	2790	67%	1541	53%	3990	75%
`Serialize(obj)`	1756	42%	765	26%	3081	58%
RMI with UKA	2043		1840		2900	%
improvement %		**51 %**		**36 %**		**46 %**

Table 3. Ping times (μs) of RMI, socket communication, and JDK-serialization.

6 Conclusion

The UKA-serialization demonstrates that it is possible to implement object serialization in Java with improved performance by at least a factor 5. Some additional help from the JVM vendors could speed things up even further. The UKA-serialization which is pure Java and freely available [3] can be used as a drop-in replacement for the JDK-serialization. A retrofitting tool is provided that upgrades existing class files to work with the UKA-serialization classes. When used with RMI, our synthetic benchmarks improve by 36% to 51%. Benchmark results with full applications will follow.

In the future, we will work on a tool that automatically generates the explicit marshaling routines that right now must be provided by hand. Moreover, we are attacking the remaining cost of a remote method invocation by using high speed networks and by optimizing the RMI layer.

Acknowledgments

We would like to thank the JavaParty team. Matthias Jacob did excellent work on the performance comparison (Java versus Fortran) on a geophysics application. Christian Nester and 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 the JDK-serialization.

References

1. 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.
2. Java Grande Forum. http://www.javagrande.org.
3. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
4. 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.
5. 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.
6. OMG. http://www.omg.org.
7. OMG. *Objects by Value Specification*, January 1998. ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf.
8. Sun Microsystems Inc., Mountain View, CA. *Java Object Serialization Specification*, November 1998. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf.
9. 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.

10. 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.
11. 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.
12. Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.