Bandwidth, Latency, and other Problems of RMI and Serialization

Michael Philippsen and Bernhard Haumacher
University of Karlsruhe, Germany
phlipp@ira.uka.de and hauma@ira.uka.de

Ideally, computational scientists would love to see the write-once-run-anywhere principle to be applicable to parallel and distributed Java programs as well. Unfortunately, it is not.

Although Java's threads are appropriate for programming SMPs, Java's support of DMPs, e.g. clusters of workstations, is insufficient. In a perfect world there would be a single JVM that transparently hides the DMP's distributed nature and provides the illusion of a single address space.

But since there is no such JVM – except for a quite incomplete research prototype [9] and too much Linda-inspired JavaSpaces [4] – existing features of Java have to be used instead. Programmers feel that explicit socket communication is similar to writing assembler code today: It might result in better performance but it is much too error-prone and time consuming to be done. Hence, Java's remote method invocation (RMI) which internally uses object serialization is the only option. However, both object serialization and RMI suffer from severe performance problems that hinder their adoption in scientific applications on DMPs and workstation clusters.

This section is the result of a careful analysis of Java's object serialization and RMI (JDK version 1.2beta3) that has recently been undertaken at the University of Karlsruhe, Germany. More information and some benchmark codes can be found at [7].

The section is structured as follows. Since object serialization is used inside of RMI, we first have an isolated look at it. The view is then broadened to RMI. Repeatedly, the text is organized along a simple pattern: We identify certain design decisions that have guided the development of serialization and RMI, we show that these decisions are grounded on assumptions that in general do not apply to high performance computing in Java, and suggest modifications. Sometimes, benchmark results are provided to illustrate the significance of the individual problems discussed. The remaining text focuses on those performance problems that are most likely to seriously affect scientific programming but are probably acceptable by commercial applications.

1 Performance problems of Object Serialization

Object serialization has mainly been designed to implement persistent objects and to send and receive objects in typical client-server applications.

Unnecessary 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. 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 the underlying assumption is wrong. In contrast to commercial applications, object persistence is not terribly relevant for computational science. The life time of an object is within the boundaries of the overall runtime of the job. When objects are being communicated it is safe to assume, that all nodes have access to the same ByteCode, that might for example be provided by a common file system like NFS. 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.

To alleviate this problem, a minimal encoding of type information must be defined. Although it is significantly shorter to textually send the name of the class (including package prefix), an even shorter representation seems possible.

At Karlsruhe, we have prototypically implemented the textual encoding. It has improved the performance of serialization significantly.

Figure 1 has four bars. The first two bars give the time needed for the serialization's write operation for an object with two ints, two floats and two null pointers. The time is in μs per object on a 167 Mhz Sun Ultra Sparc. The last two columns show the read time. The light grey bases of the bars correspond to the time

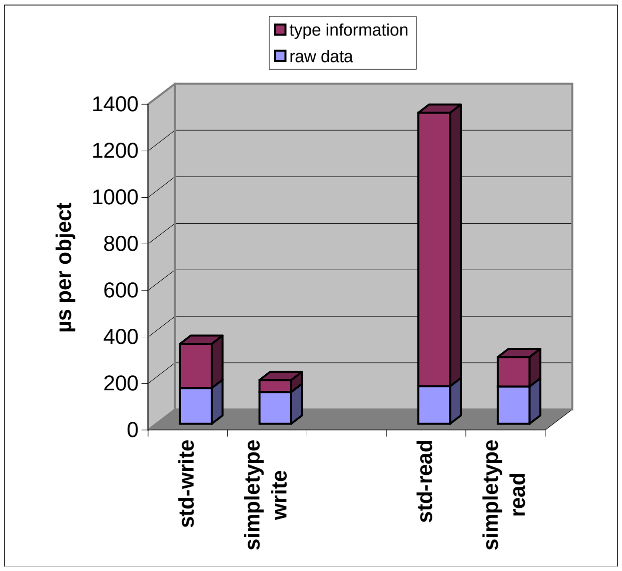

Figure 1: Serialization with slim Type Encoding

needed for the raw data information, the dark parts of the bars indicate the crucial time taken by the type information. It can be seen that it is significantly more expensive to read the type information than to write it. The second and fourth column reflect measurements with an modified serialization that uses a type encoding by class name. In addition, for the read operation, type descriptor objects have been cached instead of being recreated for every single object.

Therefore, we suggest that there will be another protocol version for the serialization to be used by RMI in cluster settings. This extension is completely under the hood and does not affect any API.

Currently the only way to experiment with specifically tuned implementations of object serialization is to make sure that the new implementation precedes the default one in the CLASSPATH variable. In combination with RMI, this approach faces several obstacles. Since it is unclear/undocumented, which features of the object serialization are actually used by RMI, it requires quite an amount of re-engineering to develop a serialization that works in RMI. Moreover, there is no straightforward way to force the registry to use the modified CLASSPATH.[1]

We therefore suggest that RMI is designed and documented so that it can easily work with specific implementations of object serialization.

[1]It is undocumented that the `main` routine of the class `sun.rmi.registry.RegistryImpl` is to be invoked in a JVM.

Implementation Deficiencies. If object serialization is used to write persistent state information to disk or to send that state over a slow network connection, the overall performance is dominated by the I/O-bottleneck. Hence, the implementation of `java.io.ObjectInputStream` and `java.io.ObjectOutputStream` is not thoroughly optimized.

When high speed communication networks are used, the time spent in generating the byte stream representation for a given object (or the other way round) becomes an issue with respect to bandwidth and latency.

At Karlsruhe, we identified the following implementation deficiencies in JDK 1.2beta3.

- It must be avoided that the byte representation is copied from buffer to buffer. As in other communication protocols, a zero-copy implementation must be the ultimate goal.
- The handling of floats and doubles must be improved. The conversion of these primitive data types into their equivalent byte representation is (on most machines) a matter of a type cast. However, in the current implementation, the type cast is implemented in the JNI and hence requires various time-consuming operations for check-pointing and state recovery.
- Finally, the serialization of float arrays (and double arrays) currently invokes the above mentioned JNI routine for every single array element. We found that it is much faster to enter the JNI only once for the whole array (or at least for the section that still fits in the available buffer size) and return a byte array.
- As other benchmarks have shown, e.g. [1] object creation takes too long. Since object output and object input should be overlapped to reduce latency, object input is not supposed to take much longer than object output.

Here are the results of our performance measurements that demonstrate that a drop-in replacement of serialization can improve performance significantly.

Figure 2 demonstrates two ways to improve the serialization. First of all, one can use or generate explicit `writeExternal` routines. If these routines are coupled with improved buffering, performance can be improved significantly. In addition, it is visible that it is significantly more expensive to write float values into the stream than to write ints. This is caused by the overhead of the JNI entries.

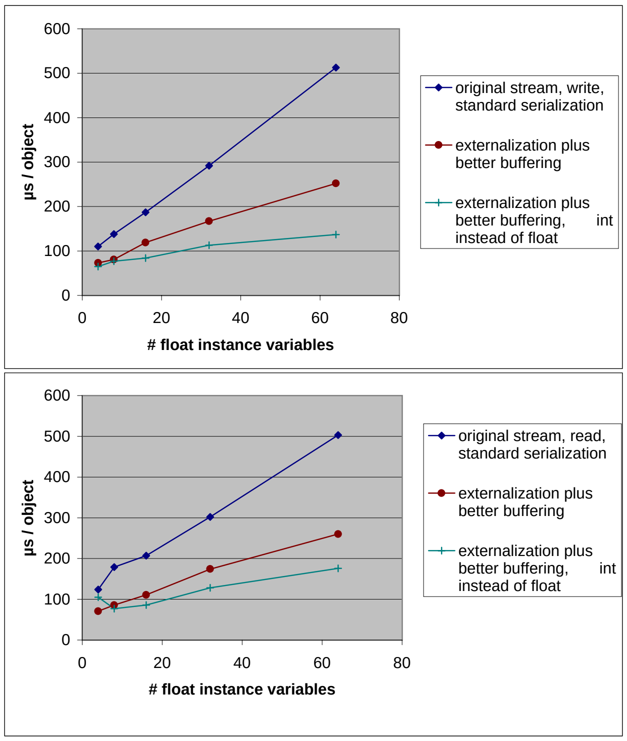

Figure 2: Serialization of floats

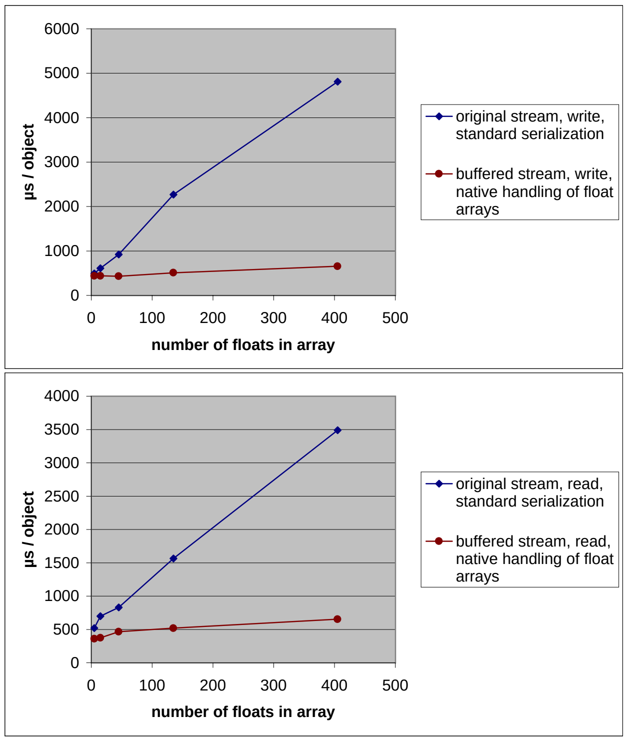

Figure 3: Serialization of float arrays

In case the object that is to be serialized contains an array of floats, the performance penalty for repeated JNI entries is outrageous, see Figure 3. Our measurements show that by doing the conversion once for the whole array, both write and read performance can be increased.

2 Performance problems or RMI

RMI has been designed for client-server applications over instable and slow networks.

High performance parallel programs are different since they are symmetric and are executed on systems that have fast and reliable networks. If there will be no such thing as a JVM that hides the architectural details of a DMP (ultimate goal), a symmetric and transparent RMI is on the wish-list (second best). In a symmetric and transparent RMI, there would be no need for a registry, moreover all network exceptions would be reduced to `RuntimeException` so that one must not distort otherwise clear programs with exception handling. The JavaParty system [6, 3] proves that this approach is feasible.

Since even this second best solution is unlikely to be implemented in the standard JDK, it is absolutely essential, that at least the existing RMI is tuned for performance. Recent work [2, 8] demonstrates that it is possible to implement RMI-like functionality in Java with better performance. But instead of relying on non-standard packages it is more reasonable to improve RMI.

Connection Management. In typical client-server applications, a server does not know whether a client will continue to use the object it serves. Hence, RMI monitors existing socket connections and closes them after they have not been used for a while. The connections are re-opened on demand.

Although this approach seems reasonable for client-server-situations it is too costly for high performance use of RMI, where the participating nodes are known beforehand.

- RMI implements a caching mechanism to check whether there already exists a connection to a particular host. If one is found, it is used. Otherwise, a new connection is opened. Every single network access travels through this hash-table lookup.

- Moreover, to monitor existing connections and to re-use ones that haven't been used for a while, RMI creates special monitor threads. Thread creation is costly.

- Finally, for every remote method invocation, RMI re-transmits class type information, although standard serialization has a mechanism to abbreviate the type information to hash values in case of existing connections. See the discussion of Figure 1 in section 1 to understand the severness of that problem.

We therefore suggest that RMI is extended to allow the user to register a set of participating JVMs (or nodes) upon initialization. These connections will be kept open for the duration of the program. Hence, repeated hash-table lookups, costly monitor threads, and repetition of type information can be avoided.

Implementation Deficiencies. It is difficult to understand the bad performance of RMI, when most of the necessary source code is not available. Therefore, the performance problems we have identified must be a small subset of the whole picture.

Similar to object serialization, RMI has been designed with the assumption that network I/O is slow and the central bottleneck. Since this is not necessarily true in the presence of high-performance networks, certain areas of RMI must be improved.

- RMI creates too many threads. For incoming remote method invocations there are listener threads. A new listener takes over if the fist serves an incoming call. Watchdog threads monitor the state of connections. Additional threads are used for unreferenced objects and for the distributed garbage collection.

  Although the `run` method of class `Thread` can only be executed once, a pool of worker threads can be implemented that execute certain jobs in endless loops. Performance measurements have shown that recycling of threads is faster than recreation.

- Java's object serialization is capable of handling "replacement objects". Instead of itself being represented in a stream of bytes, an object can tell the serialization process that a different object has to be serialized instead. With object replacement switched on, it is more difficult for the serialization to encode graphs of objects. Cycles in the original graph may or may not be present in a graph that has some replacement objects. Hence, an additional layer of lookup is required to avoid endless loops.[2] Replacement objects hence significantly slow down object serialization in general.

  Unfortunately, RMI uses replacement objects. Instead of transmitting a stub object, this object calls for the serialization of a replacement object that implements a system-wide remote reference. Although this is a neat use of "replacement objects", it is overkill.

  There are two options to improve performance. First, RMI should refrain from using replacement objects and instead encode the remote reference directly, e.g. by writing the fields into the stream. Second, object serialization should be extended to offer a slim replacement facility for objects that can be guaranteed not to participate in cycles. This would avoid secondary lookup.

Custom Transport. RMI is designed to be used with standard TCP/IP sockets. In Java 1.1.x there are two general approaches to make RMI use a different network. The first approach uses a different native socket implementation without Java noticing it. This requires two steps. First, a native implementation of the underlying sockets must be created and loaded as a dynamic library at the start of the program. Second, the RMI registry must be patched to use that special purpose library as well. The other alternative is to create a subclass of `java.lang.socket` and simulate socket functionality based for example on a UDP-transport. However, as Krishnaswamy et al. show in [5] this requires source code modifications.

This clearly demonstrates a design flaw in RMI. RMI needs to be able to work smoothly with custom socket implementations, in particular, neither native code and patches should be required. We suggest that RMI supports a pluggable serialization that can be selected at runtime.

In Java 1.2beta3 the situation seems to be different, although we did not have a chance to test that. The new socket library offers a way to provide a custom implementation of socket, that no longer needs to be a subclass of `java.lang.socket`. By means of a socket factory, this specific socket class is supposed to be usable in RMI. However, it needs to be studied how well that concept works in practice.

Even with the newly introduced way to use custom socket implementations, RMI is still tied to sockets. Many existing high speed communication networks however, offer much more efficient packet based

[2]Whereas the primary hash-table lookup is implemented efficiently, the current lookup for replacement objects needs improvement.

protocols. Some even offer broadcast, pre-fetch and post-store mechanisms. Currently, there is no way to make RMI use these features. Instead, it is required that a conforming socket layer is implemented on top of the low lever network features. Since RMI's protocol itself is packet-based, this additional socket layer is too costly.

We suggest that RMI is modified so that it can make use of packet based networks and of other network features. At least it should be documented how that might be achieved in the current implementation.

3 General Suggestions

Source Code Availability. Several of the suggestions made above are incomplete and need to be thought about in more detail. This can only be done in cooperation with JavaSoft's RMI architecture group since a significant part of the RMI source code is not available. We therefore suggest that the remaining source code of RMI is provided – a step that will increase the number of people working on an improvement of RMI.

Synchronization Problems. RMI's implementation of synchronized remote methods is useless. There are two problems. First, concurrent threads that invoke a synchronized remote method are sequentialized at the stub level, i.e., on the client side. Second and more important, thread ids change whenever a call leaves a JVM. Whereas Java does allow a thread to lock a particular lock repeatedly, for example by calling another synchronized method of an object from within a method that is already synchronized, there are scenarios in RMI, where this does no longer work. Consider for example a situation, where from within a synchronized method, a remote method is invoked, that in turn calls back another synchronized method of the first object. RMI guarantees a deadlock in this case, whereas the same code would work smoothly in standard Java. The problem is that a new thread will execute the call-back. And since this new thread does not have the lock, it cannot acquire it.

We suggest that the notion of thread id is introduced in RMI. For an incoming call, the service thread must be able to indicate in a transparent way it's original id to the locking mechanism.

References

[1] 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(to appear), 1998.

[2] S. Hirano. Horb – the magic carpet for network computing. http://ring.etl.go.jp/openlab/horb/, 1996.

[3] JavaParty. wwwipd.ira.uka.de/JavaParty.

[4] JavaSpaces. Sun Microsystems, Mountain View, 1997. java.sun.com/products/javaspaces.

[5] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java Remote Method Invocation (RMI). In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, 1998.

[6] M. Philippsen and M. Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[7] JavaParty Team. Benchmarking RMI and serialization. wwwipd.ira.uka.de/JavaParty/rmibench.

[8] G.K. Thiruvathukal, L.S. Thomas, and A.T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(to appear), 1998.

[9] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11), November 1997.