Serialization For Ubiquitous Systems: An Evaluation of High Performance Techniques on Java Micro Edition

Nicholas Palmer, Thilo Kielmann, Henri Bal
Vrije Universiteit
De Boelelaan 1081A
1081 HV Amsterdam, The Netherlands
{palmer,kielmann,bal}@cs.vu.nl

Abstract

Object marshaling, called Serialization in Java, offers a high level of abstraction for information interchange in object oriented systems. It thus reduces the source lines of code required to transmit objects across a network. This abstraction often comes with a runtime and data penalty. In this paper we present an efficient implementation of object marshaling for the Java platform originally used for high performance computing environments. We demonstrate that the same technique is effective on ubiquitous resource constrained platforms, such as Java Micro Edition. We show that by adopting high performance techniques we are able to bring object marshaling to a platform where it is not otherwise possible due to the lack of runtime type inspection. We also demonstrate that performance of this system is better for array oriented data and acceptable for typical application data when compared with a hand coded protocol. This demonstrates the value of bringing techniques from high performance computing to ubiquitous resource constrained devices.

1 Introduction

There are new devices on the market today which are poised to radically change computing as we know it. These "Converged Devices" combine the features of mobile phones and hand held computers into a single package and include advanced abilities such as: global positioning system (GPS) receivers; wireless Internet access; and sensors such as ones for temperature, pressure or acceleration. Often marketed as "Smart Phones", these devices also include what have become commonplace mobile phone features such as: video and still cameras; bluetooth connectivity; audio recording and playback; as well as traditional phone features like messaging and calling. It is widely expected that these devices will become nearly ubiquitous, much as cell phones are today, and are thus a major step towards the vision of ubiquitous computing[15] so often dreamed of. Applications which take advantage of the unique features offered are already being built[1, 2, 5, 6], and the market for these devices is rapidly growing[1]. They are already being hailed as the "next wave in computing"[17]. However, these devices have limited processor power and, more importantly, limited battery power with which to perform computations and communicate data. They are thus constrained in available resources.

In spite of the constraints of the devices, the combination of mobility, (limited) computational power and communication facilities makes these devices ideal for *mobile distributed* computing applications. We[10] and others[16] have argued that the platform offers unique opportunities for the dissemination of important information, such as might occur in a crisis situation. These devices may also be used as distributed sensor networks[8] and in conjunction with other computing resources such as web servers. Certainly there are many novel distributed applications that will arise on this platform. However, as we discovered in our previous work[10], programming distributed applications on these platforms is extremely challenging. Take, for example, the Java Micro Edition (JME) Connection Limited Device Configuration (CLDC) environment from Sun Microsystems which has a huge market share with more than 94 million devices deployed[2]. This huge market sharemakes JME a compelling platform for building applications. However, this environment is missing some key features for building *distributed* applications because it is an object oriented programming environment that lacks a system for conversion of objects to a binary representation, an operation called marshaling.

[1] http://www.palminfocenter.com/news/9247/report-64-million-smartphones-shipped-in-2006/

[2] http://arstechnica.com/news.ars/post/20060815-7514.html

This interface is required to easily exchange objects over the network or store them to persistent storage. This abstraction, called Serialization to Java programmers, exists on both the Java Standard Edition (JSE) platform common to desktop computers and the Java Enterprise Edition (JEE) platform common on servers and is thus well known to distributed programmers on those environments. On JME programmers are required to write a great deal of marshaling code by hand, making it considerably more challenging to build such applications. It is thus vital that Serialization be added to the platform in order to enable it to reach its full potential. What is needed is a system which has the standard Serialization interface familiar to programmers, works on legacy JME devices, and is efficient in terms of CPU and bandwidth consumption.

Our system for efficient serialization is based on Ibis[13, 14] serialization, which we discuss in Section 2. We have made the required modifications to this system in order to make it possible to run it on the JME platform. We describe this solution in more detail in Section 3, analyze the performance of this system on a real phone in Section 4 and conclude with a discussion of our results in Section 5.

2 Related Work

In order to convert an object into a binary representation it is required to know what data the object is composed of. In the Serialization systems available on both JSE and JEE this information is discovered at runtime via the type inspection interface, called Reflection. This interface is also missing on JME, thus it is not possible to do type discovery at runtime.It is thus required to move the type inspection process to compile time. Fortunately, a great deal of work has been done to build these kinds of systems for high performance computing. It has been argued that the combination of techniques from high performance computing with mobile devices makes a great deal of sense [8], particularly when considering networked applications.

Early work of interest includes Manta[7], which pioneered compile time generation of marshaling code. However, Manta uses native code which can not be used with JME due to the lack of the Java Native Interface (JNI) interface for linking such code to the Java runtime environment. Due to this our solution uses compile time generation of Java bytecode instead of native code for a pure Java solution. Ibis[13, 14] is a follow up project to Manta developed by our group at the Vrije Universiteit targeted at high performance grid programming. Part of the system is the ibis.io package which includes a compile time (java bytecode) serialization generator and a set of classes for performing serialization. This system serves as the basis for our work.

Other work on improving efficiency was done with UKA-Serialization[11, 12], which avoids the native code issues of Manta by subclassing Java's native serialization system in order to improve efficiency of the serialization system. However, it relies on features of the existing serialization implementation since it uses subclasses. Since there is no existing implementation on JME this is not a viable approach to solve our problem. Even if there were an existing system on the phone, their system required modification of the JSE source code to allow their subclasses to operate properly. Due to the fact that these classes are preloaded on to phones and stored in read only memory modification of the existing classes is not possible. Opyrchal and Prakash[9] also did some early work on improving serialization marshaling size by taking advantage of the class hierarchy in Java to reduce the bytes required to send class names in order to only transmit class names once. Our system uses a similar two byte hash of class names in order to reduce the bandwidth overhead of serialization at the expense of backwards compatibility with JSE Serialization. Hessian is a dynamically typed, binary wire protocol[3] designed to have a compact binary form. While the authors claim that this format is ideal for resource constrained environments like JME the current implementation relies on Reflection and so it does not actually work on that platform. We have also investigated the performance of this system some in Section 4.

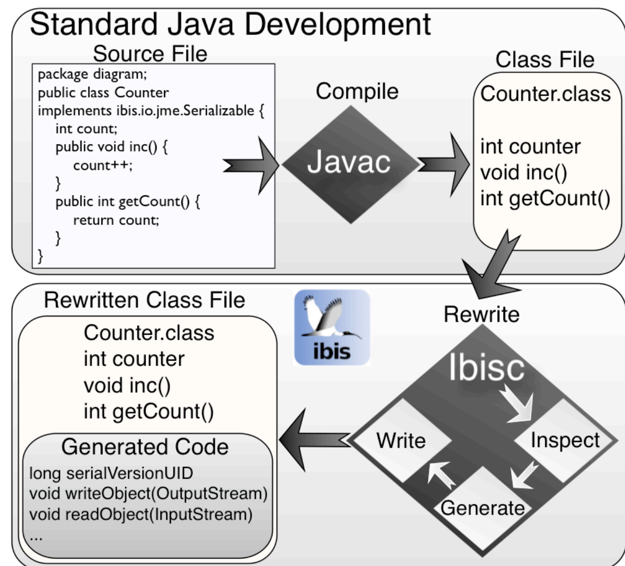

Figure 1: Ibis IO Compile Process

3 Compile Time Serialization

In order to get around the lack of runtime type inspection on the JME platform we have borrowed techniques pioneered for high performance computing. In particular we

have taken advantage of earlier work from our group building a high performance distributed middleware for cluster and grid computing called Ibis[13, 14]. This system uses a compile time serialization code generator (called ibisc) in order to avoid the overhead of runtime type inspection and thus achieve better performance. Because JME lacks the ability to do runtime type inspection this technique is ideally suited for solving our problem. Despite this, because of the legacy of the JME platform the environment differs considerably from the JSE environment the system currently runs on. We were thus required to make extensive modifications to the system in order to make it work properly on these new devices.

The compilation process is outlined in Figure 1. What is important to note is that by moving the type inspection from runtime to compile time we are able to get around the lack of the Reflection API required to do runtime type inspection that is missing on JME. The compile time serialization system works on compiled class files using the Byte Code Engineering Library[3] (BCEL) to rewrite class files with new methods and data members. First, our compiler inspects the class file to determine if it should be modified to include code for serialization. Just as in standard Serialization, we use a marker interface to mark objects which may be passed across the wire. When our compiler sees this marker interface in the class file it checks what data members are in the class and adds new methods to the class for writing these members to a data stream. Thus the type inspection that would be handled by the Reflection API in a standard Serialization system is handled by our system at compile time through inspection of the Java class files and generation of code. By working directly with the bytecode the system avoids the need for access to source code and thus it is possible to serialize classes from third parties that do not provide source code.

After our compiler has inspected the bytecode for the class and determined the data members it generates new methods which can be used at runtime to write the members of the class to the stream or read the class in from the stream. The resulting class file is rewritten to the file system in place of the original class file. The system is effectively a code generator but instead of generating source code it generates bytecode directly using the bytecode from the existing class file to guide the code generation process. This considerably simplifies implementation since we can rely on the existing Java compiler to handle parsing of the source code while still having access to the required information stored in the Java class files that we need.

[3]http://jakarta.apache.org/bcel/

4 System Analysis

There are several important aspects of the system that we consider in the examination of performance. First and foremost, we are interested in the complexity that our abstraction hides from the programmer. We detail this complexity in Section 4.1. We are also concerned about the runtime costs associated with our system which we examine in Section 4.2. Finally, we are interested in the additional costs of moving the data across the wire, which we detail in Section 4.3.

In order to characterize the performance of the system along these three axes we have written a series of benchmarks. These benchmarks each write a particular type of data to a stream and measure the time it takes to write the data as well as the size of the data written. The benchmarks can be divided roughly into two categories: array benchmarks and application benchmarks. The first category consists of benchmarks which write arrays of basic data types which we do for bytes, integers and doubles and also for Strings. We have also written a group of benchmarks designed to model more typical application data in order to examine programming complexity.

We have also implemented a version of our system which uses the Hessian[3] encoding for basic data types. This encoding attempts to save transmission space by storing a smaller representation when possible. For example if an integer can be represented with a byte the byte is used. Unfortunately if the integer needs the full 4 bytes of data storage then Hessian encodes this into 5 bytes, using the first byte as a flag to indicate the number of bytes being sent for this integer is 4. Thus we have implemented three versions of tests where this is possible, one which sends data which the Hessian encoding can compress the most, one which it can not compress at all in this way and a third with a random data. Through these tests we demonstrate that the compact nature of the Hessian data encoding is appropriate for applications where wire compactness is more important than runtime performance and data can be represented using a smaller type then specified a majority of the time, i.e. where the programmer has used an integer but the program stores a byte or short value in the integer field more often than numbers requiring the full width of the integer.

The next set of benchmarks test the performance of data structures commonly found in applications. The first of these, the Mesh test, represents a grid of points with a double assigned to each point in the grid as might be found in various simulations. The second represents a typical event system where each event has a sequence number, timestamp and message associated with it. Our next benchmark, the Graph benchmark, tests how the system handles a data structure composed of a large number of pointers to other objects and highlights issues with encoding objects which

have circular references. The final benchmark tests the impacts of polymorphism on the encoding system. Polymorphism refers to the many forms of an object usually realized using subclasses and in Java interfaces as well and adds complexity for marshaling systems by splitting data between base and subclasses. For each benchmark we have also written a hand coded implementation that writes the data in the most compact way possible in order to be able to characterize the complexity that use of this system saves in Section 4.1 as well as to have a baseline on the minimum amount of space required to transmit the object.

In order to evaluate the real world performance of the system we have built, we have run all benchmarks on a Nokia N80, a perfect example of the "Converged Devices" we wish to target. Each benchmark was run 50 times and averaged in order to characterize the effects of random data we use in some benchmarks as well as deal with the inherent inaccuracies of the microsecond timer available in the virtual machine and the random chance of the device needing to process other signals. We analyze the results of this effort in the next three sections.

4.1 Programming Complexity

One of the chief features we are interested in our serialization system is the programming complexity. This is because we believe that building distributed applications needs to be simplified considerably in order to unleash the power of the JME platform on these new devices. In terms of code complexity Serialization is a clear winner. The more complex the object model the more complex the hand written serialization code becomes. To serialize any object our implementation requires only three statements:

```
ObjectOutputStream os =
    new ObjectOutputStream(
        new BufferedOutputStream(out));
os.writeObject(data);
os.flush();
```

Contrast this with the code required for the hand written protocol for the Event benchmark:

```
DataOutputStream os =
    new DataOutputStream(out);
out.writeInt(events.data.length);
for (int j=0; j<events.data.length; j++) {
    os.writeInt(events.data[j].sequenceNumber);
    // writeUTF doesn't handle null strings
    if (events.data[j].message != null) {
        os.writeBoolean(true);
        os.writeUTF(events.data[j].message);
    } else {
        os.writeBoolean(false);
    }
    os.writeDouble(events.data[j].timestamp);
}
os.flush();
```

Note that the code required to write the data structure in this way grows linearly with the number of data members while the code required for serialization is constant. Also note that the hand written code is fragile in the face of changes to the data structure. Any change to the data structure requires that the programmer remember to add to the serialization code as well. Given that the number of lines of code and complexity of a program has been found to be the best predictor of the number of faults in the program [4] it makes sense to use a system that has such an attractive constant code length interface across all objects and code which is of constant length and can be reused in a large number of applications. This is just good software engineering practice. Note that there are no circular references in this data structure. A data structure with possibly circular references requires even more complex code as is clear in our Graph benchmark which is composed of a network of objects with possible cycles and nulls. The code required to serialize this object already has considerable complexity for the application programmer and we have not implemented optimizations to the hashing which hurt runtime performance of this benchmark so much:

```
public void write(OutputStream out) {
    DataOutputStream os =
        new DataOutputStream(out);
    Hashtable written = new Hashtable();
    writeGraph(os, written, this);
    os.flush();
}
private void writeGraph(DataOutputStream stream,
        Hashtable written, GraphData graph) {
    if (graph == null) {
        out.writeInt(0);
        return;
    }
    int hashCode = graph.hashCode();
    out.writeInt(hashCode);
    Integer hashCode = new Integer(hashCode);
    if (!written.containsKey(hashCode)) {
        written.put(hashCode, graph);
        writeGraph(out, written, graph.north);
        writeGraph(out, written, graph.south);
        writeGraph(out, written, graph.east);
        writeGraph(out, written, graph.west);
    }
}
```

Finally, it is important to realize that programs which take advantage of the polymorphism available in object oriented systems require even more complexity on the programmers part to handle the polymorphism. For the sake of brevity we don't detail what is required to handle this polymorphism, however we ask that you note that it is generally accepted that good software practice is to develop a working system first and then optimize the places that are performance issues. By using the serialization system we have built it is possible to quickly develop a working system and to optimize how particular objects are sent later us-

ing the hooks within serialization provided for exactly this purpose.

4.2 Runtime Overhead

The runtime tests measure the time it takes our implementation of object serialization to marshal the object and write it to the data stream. This is thus a measure of the efficiency of the marshaling system itself as well as the efficiency of the data encoding system. It is clear in Figure 2 that the overhead for writing arrays of data is actually better than the best hand written protocol. (Note that error bars show actual minimum and maximum times recorded.) This is due to the fact that our system avoids data copying whenever possible while the native implementation makes needless copies of data in the encoding process. Our performance for Strings is not as good because strings in Java are an immutable array of bytes. However, all byte arrays are mutable. Thus when we need access to the bytes that make up a string we have to either iterate over each character and encode it or we have to request a mutable byte array at which point a copy of the string is made. The native implementation can use a no-copy implementation that takes advantage of the fact that the code used to write Strings is known not to mutate the underlying byte array. This causes our Event benchmark to be lower in performance because it includes a String that has to be encoded.

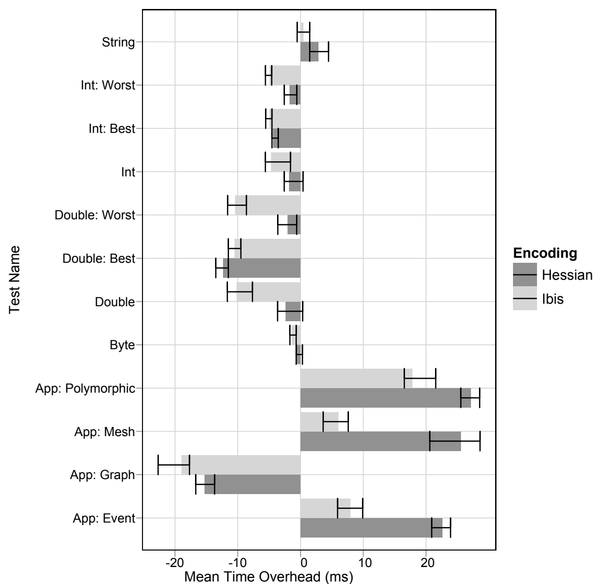

Figure 2: Time Overhead As Compared to Hand Coded

For most of the benchmarks with more realistic application data objects tests our system introduces some overhead when compared with the hand written protocol. This is again in part due to the performance of the string implementation, since our implementation writes class names to the stream. As detailed in Figure 2 the performance is better than the hand written protocol for the Graph test. This is because our implementation avoids the overhead of creating Integer objects to store these hash keys by using a custom Hash class. We have not done the same in the hand written protocol in order to try to show that the complexity of even this naive implementation is very large in Section 4.1. This demonstrates that the best performance isn't always achieved with a hand written protocol that isn't very carefully crafted and that the results are very application specific. This overhead is a significant factor in our poorer performance on the other three application benchmarks. The hand coded version of these three benchmarks do not worry about object references at all while our system can not make the same assumption and so incurs overhead to manage the object references. Note also that our application benchmarks do not include arrays of data which would bring down the comparative overhead due to our superior performance with arrays.

4.3 Data Overhead

For our target platform it is also important to consider the data overhead incurred by the system because every byte sent on the wire consumes limited battery power to transmit it. In terms of the amount of data written, serialization does include some overhead to store the class and version information as well as manage references as mentioned above. This is in part due to the fact that it writes the class names to the stream as well as version information that is not written in any of the hand coded protocols and has to deal with possibly circular references. The only application which does this is the Graph application where our system is very competitive. However as detailed in Figure 4.3 this overhead is not excessive in any way and for some applications the Hessian encoding offered by our system may actually save bytes at the cost of some runtime complexity as detailed in section 4.2.

Note that the random data used in the Int and Double benchmarks performs on par with the worst case for the Hessian encoding, however the best case performs significantly better. Based on the marketing of Hessian we did not expect this to be the case but it makes intuitive sense. In a normally distributed set of integers only a minority of the integers are going to be small enough to be represented in 2 bytes or less because Hessian really only saves space for such integers. It is thus important to choose the encoding used on a per application basis after testing with real world data. Some applications may benefit greatly from the Hessian encoding while in other applications it may not be worth the added runtime complexity. The advantage of our

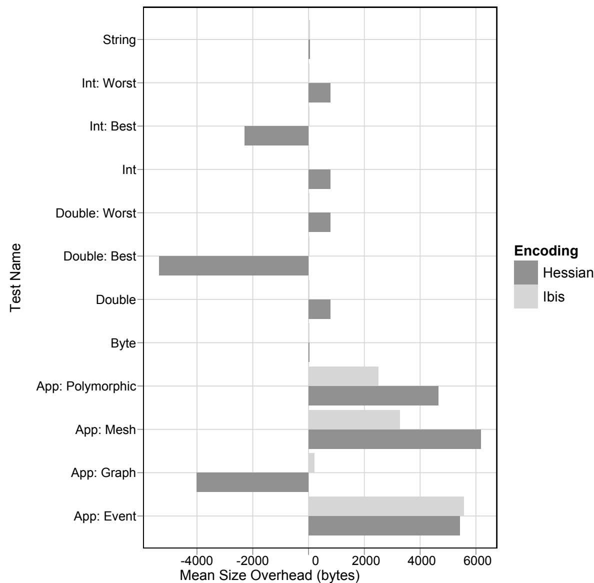

Figure 3: Data Overhead As Compared To Hand Coded

system is that it is trivial to test both and select the right one for a given application by using our factory to generate serialization streams and changing the type of stream generated by the factory in just one place.

5 Conclusion

In this paper we have presented an efficient implementation of object marshaling for the Java Micro Edition platform originally used for high performance computing environments. We have demonstrated that the same techniques are effective for resource constrained devices. In particular we have shown that compile time type inspection is able to bring a Serialization interface to a platform which lacks runtime type inspection and that the runtime and data overhead introduced by this implementation is small enough to be acceptable. We have further shown that Hessian encoding can save considerable encoding space for applications with specific data usage patterns at the expense of some runtime complexity. Considering that computation consumes less energy than communication this may be desirable for many applications. As a whole, this work demonstrates the value of bringing techniques from high performance computing to resource constrained devices where performance is important due to the constraints of the hardware. Our future work will focus on bringing other techniques for distributed computing in high performance environments to resource constrained platforms such as JME and the new Android platform in order to unleash the potential of this new distributed platform.

References

[1] R. Ballagas, J. Borchers, M. Rohs, and J. Sheridan. The Smart Phone: A Ubiquitous Input Device. *Pervasive Computing, IEEE*, 5(1):70–77, 2006.

[2] A. Cheok, A. Sreekumar, C. Lei, and L. Thang. Capture the flag: mixed-reality social gaming with smart phones. *Pervasive Computing, IEEE*, 5(2):62–69, 2006.

[3] S. Ferguson and E. Ong. Hessian 2.0 serialization protocol.

[4] L. L. Gremillion. Determinants of program repair maintenance requirements. *Commun. ACM*, 27(8):826–832, 1984.

[5] L. Iftode, C. Borcea, N. Ravi, and P. Kang. Smart Phone: an embedded system for universal interactions. *Distributed Computing Systems, 2004. FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of*, pages 88–94, 2004.

[6] J. Krikke. Samurai Romanesque, J2ME, and the battle for mobile cyberspace. *Computer Graphics and Applications, IEEE*, 23(1):16–23, 2003.

[7] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[8] L. McKnight, J. Howison, and S. Bradner. Guest Editors' Introduction: Wireless Grids–Distributed Resource Sharing by Mobile, Nomadic, and Fixed Devices. *Internet Computing, IEEE*, 8(4):24–31, 2004.

[9] L. Opyrchal and A. Prakash. Efficient Object Serialization in Java. *Publication No. XP-002242373*.

[10] N. Palmer. Enabling distributed applications for smart phones. Master's thesis, Vrije Universiteit, Amsterdam, 2007.

[11] M. Philippsen and B. Haumacher. More efficient object serialization. *IPPS/SPDP Workshops*, pages 718–732, 1999.

[12] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency Practice and Experience*, 12(7):495–518, 2000.

[13] R. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. Bal. Ibis: an efficient Java-based grid programming environment. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 18–27, 2002.

[14] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H. Bal, R. Van Nieuwpoort, J. Maassen, G. Wrzesinska, et al. Ibis: a flexible and efficient Java-based Grid programming environment. *Concurrency and Computation Practice and Experience*, 17(7-8):1079–1107, 2005.

[15] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, 1991.

[16] S. Winter, G. Iglesias, S. Zlatanova, and W. Kuhn. Gi for the public: the terrorist attack in london, December 2005.

[17] P. Zheng and L. Ni. *Smart Phone and Next Generation Mobile Computing*. Morgan Kaufmann, 2006.