

Efficient Object Serialization in Java

Lukasz Opyrchal and Atul Prakash
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122, USA
E-mail: {lukasz, aprakash}@eecs.umich.edu

Abstract

Object serialization is the ability of an object to write a complete state of itself and of any objects that it references to an output stream, so that it can be recreated from the serialized representation at a later time. Pickling, the process of creating a serialized representation of objects, has been investigated for many years in the context of many different distributed systems. Sun Microsystems introduced a simple and extendible API for object serialization in version 1.1 of the Java Development Kit. Application developers can use this standard serialization in their applications, or they can develop custom versions for different user-defined classes. Many distributed applications that use standard serialization to communicate between distributed nodes, experience significant degradation in performance due to large sizes of serialized objects. We present a number of improvements to the serialization mechanism aimed at decreasing pickle sizes without visible degradation in serialization performance. Through performance results, we show that it produces pickles up to 50% smaller without degrading serialization performance.

1. Introduction

Many of today's distributed systems use objects as the means of communication between nodes. A method for transferring Abstract Data Types is presented in [8] and later extended in [4]. Java's pickling system, described by Riggs in [19], is a direct derivative of those systems. The idea is to be able to capture the state of a Java object in serialized form for transmission or storage. Pickling defines the serialized form to include enough additional information to be able to identify the types of objects and the relationships between objects within a stream [19]. This additional information is necessary in the process of recreating objects from their serialized representation.

Java provides a generic pickling mechanism, which is able to serialize and deserialize any object that implements the `java.io.Serializable` interface. Java

serialization is extensible and customizable [11]. There are many problems that have not been addressed by current implementation (jdk 1.1.6). One of them has to do with performance and specifically with the size of pickles produced by the system. When serialization is used to transfer objects in a distributed application, large pickles adversely affect the performance because of greater network latencies for large messages [2]. The standard algorithm includes some optimization techniques (such as the *referencing mechanism* described in Section 3), but further improvements can be made.

This paper describes improvements to the standard Java serialization algorithm aimed at decreasing pickle sizes. The improvements do not change either the serialization semantics or the API, so the new mechanism can transparently replace the old one (without any changes necessary to applications using serialization). Current implementation cannot read objects serialized by the old algorithm, but it can be easily extended to recognize which algorithm was used during serialization and to call the appropriate read function. The algorithms presented here specifically improve the Java serialization mechanism, but this work can easily be generalized to other systems requiring object serialization.

We also discovered that a large set of applications that use message-based protocols do not take advantage of the optimizing features described in this paper as well as the optimizations included in the standard algorithm. An entirely new scheme is needed to improve serialization performance in this class of applications. This is the subject of our current work.

This paper is organized as follows. Section 2 describes related work. Section 3 briefly describes the design of the standard Java serialization scheme as well as some of the inefficiencies found in that algorithm. Section 4 describes the design of the new serialization scheme, and Section 5 describes its implementation. Experiment results and their analysis are presented in Sections 6 and 7. Section 8 presents our conclusions and directions for future work.

2. Related Work

The problem of transferring abstract data types (ADTs) has existed since the first distributed systems were developed. The first detailed treatment of this problem

* This work is supported in part by the National Science Foundation under cooperative agreement IRI-9216848 and under grant ATM-9873025

<pre> package testing.production.Chat; public class TestChat implements java.io.Serializable { private int intData = 1; public TestData td = new TestData(); public TestChat(){} } package testing.production.Chat; class TestData implements java.io.Serializable { private int intData = 1; TestData(){} } </pre>	<pre> ...sr. testing. production.Chat. TestChat.{?y.7.^ ...I..intDataL.. tdt."Ltesting/pr oduction/Chat/Te stData;xp...sr. testing.product ion.Chat.TestDat a.g..D.N'...I..i ntDataxp... </pre>
---	---

Figure 1: Sample classes and their serialized form

appears in [8]. The authors define a canonical form for transferring of all primitive data types. ADTs are transferred as collections of those primitive data types. The “Network Objects” paper [4] extends this work to include fingerprints, references for repeating objects and special functions for arrays and strings. Attempts have been made to standardize the method for transferring ADTs [16]. The proposed NDR standard defines a standard network format for ADTs and a network representation for all primitive data types. Many of the current object based distributed systems include an API for communicating objects or for persistent storage of objects. An example is the CORBA system and its Externalization Service [15].

The serialization API appeared in the Java language in version 1.1 of the JDK. Java serialization is described in [19] and [11]. It adds a simple versioning mechanism and user customization to the system described in [4].

Work in data compression is very relevant to this research. We are still investigating feasibility of using text/data compression algorithms to compress certain parts of pickles. A very good description of standard compression algorithms, including LZW, can be found in [5]. Another related area of research is the work on TCP/IP header compression. A very good overview can be found in [10].

3. JavaSoft Implementation

The main advantage of the standard serialization mechanism is its simplicity. It allows serialization of any type of object by making a simple function call to the writeObject() method. Deserializing an object is achieved by a corresponding call to the readObject() function.

During the serialization process, the name of the class and its fingerprint are written to the stream. This is followed by a sorted list of non-transient field names and full names of the superclass and class names of all non-primitive data members. This process is then repeated recursively for all referenced classes. This means that for every non-primitive data member, its full class name is

written to the stream twice. All the values of the different data members are written to the stream following the header information. Fingerprints are included in pickles for class identification, and the sorted list of field names is used by the version control mechanism [11]. This process is described in more detail in [17], [19], and [11]. A sample class and its serialized version is presented in Figure 1 (The text in bold in the above figure indicates the class/package names which are unnecessarily repeated in the pickle).

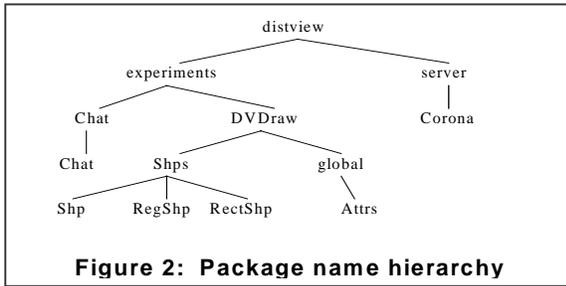
The standard serialization mechanism includes the *referencing mechanism* as one method for optimizing the size of pickles. An object and its class description receive a reference number which is written to the stream in case of multiple references to the same object or class description. This mechanism produces sizable savings in pickle sizes, especially if large numbers of objects or very complex objects are serialized at a time, but there are still places for improvement.

We notice a number of inefficiencies. Most class names are written to the stream twice and package names are often repeated many more times than that. Because class description takes up more than 50% of a pickle, reducing it will have a big impact on the pickle size.

The *referencing mechanism* was designed for streaming protocols. It works best in applications where the serialization stream is open through out the lifetime of the application. That means that message-based applications, which serialize a small number of objects into a buffer and then send it in a message via TCP/UDP/multicast, do not take advantage of optimizations built into the serialization algorithm (namely the *referencing mechanism*). We are exploring solutions to this problem (see Future Work).

4. Design of the Improved Algorithm

The design of the new serialization algorithm was subject to a number of restrictions. First, we decided to keep the same semantics as the standard Java serialization mechanism. That means that the old serialization code



can be replaced with the new implementation without any changes to the applications on top. Changing the semantics of serialization is further discussed in Section 8. Second, we wanted to make sure that the new algorithm does not degrade performance of the serialization process. Performance studies of both algorithms are included in the experiments section below.

The standard Java serialization is already using the referencing mechanism to avoid repeating class descriptions multiple times. In many cases, the pickles are still very large and improvements can be made. We have decided to take advantage of the package name hierarchy used in Java software development. For example, two classes:

```
distview.experiments.DvDraw.Shps.RectShp
```

and

```
distview.experiments.DVDraw.global.Attrs
```

would be serialized with their fully specified names written to the stream. We notice that both classes have common part of their package names and there is no reason to write that common part to the stream multiple times for every serialized object that belongs to a package in this hierarchy. A package name hierarchy is a tree, where the nodes are parts of package names, see Figure 2.

In the case of the above two classes, when they are both serialized one after another, the full name of the first one is written to the stream, but there is no need to write the full name of the second one. Following the hierarchy tree, we notice a common part

```
distview.experiments.DVDraw
```

which does not need to be written to the stream. A 2-byte code can be assigned to that common part of the name when the first class is serialized and can then be written to the stream, followed by the class specific part of the

```

distview.experiments.DvDraw.Shps.RectShp 201
distview.experiments.DvDraw.Shps         202
distview.experiments.DVDraw               203
distview.experiments                      204
distview                                  205
  
```

Figure 3: Codes Assigned after Serializing RectShp

name, “global.Attrs”. The second class would be encoded in the following way:

“00 FE global.Attrs”, where “00 FE” is the 2-byte code for

```
distview.experiments.DVDraw
```

Whenever a new class is serialized, the package name hierarchy tree is expanded and all new name parts get a new 2-byte code. In the case of the first class above, the following codes could be assigned by the serialization mechanism (see Figure 3).

Next, when the second class is serialized, the part of the name

```
distview.experiments.DVDraw
```

is found on the tree, and instead of this full name, the code is written to the stream. Two new codes are also added to the package name tree:

```

distview.experiments.DVDraw.global.Attrs 206
distview.experiments.DVDraw.global       207
  
```

Figure 4 shows the result of serializing classes from Figure 1 using the new algorithm. Compare this with the serialized version from Figure 1, where text fragments in bold indicated redundant class/package names in the pickle. Figure 4 shows how those redundant parts were replaced by the coding scheme (codes are in bold). The next section describes in more detail the implementation of this scheme.

5. Implementation

We modified the standard Java serialization classes: *ObjectOuptutStream*, *ObjectInputStream*, *ObjectStreamClass*, and *ObjectStreamConstants*. We also introduced a new class to the system. Class *ClassLookup* handles the coding scheme for package hierarchy. *ClassLookup* implements a hash table and an array to map names to encodings and encodings to names. We have decided against an explicit tree representation because the

```

0: AC ED 00 05 73 72 00 20 74 65 73 74 69 6E 67 2E ...sr. testing.
10: 70 72 6F 64 75 63 74 69 6F 6E 2E 43 68 61 74 2E production.Chat.
20: 54 65 73 74 43 68 61 74 7F B5 7B 3F 79 9E 37 F6 TestChat..{?y.7.
30: 5E 02 00 02 49 00 07 69 6E 74 44 61 74 61 4C 00 ^...I..intDataL.
40: 02 74 64 7E 00 BA 00 08 54 65 73 74 44 61 74 61 .td~....TestData
50: FE 67 7F 96 44 A3 4E 27 7F 78 70 00 00 01 73 .g..D.N'.xp....s
60: 72 7E 00 BD 7F 02 00 01 49 00 07 69 6E 74 44 61 r~.....I..intDa
70: 74 61 78 70 00 00 01                                     taxp....
  
```

Figure 4: Class from Figure 2 - serialized form – new algorithm

```

classname - full name of a class (including package name)
rest      - holds parts of classname that have not been encoded
previously
repeat
    if (classname already encoded)
        return code for that class
    else
        ClassLookup.insert(classname)
        r = StripLastNamePart(classname)
        rest = r + rest
while (classname != NULL)
code = -1
return (code, rest)

```

Figure 5: Encoding Algorithm

package hierarchy is a generic tree, and not a binary tree. A generic tree implementation using a binary tree would deteriorate into a linear search if all serialized classes belonged to the same package. A hash table implementation uses more memory than a tree representation would, because it stores full class names for each class instead of taking advantage of the hierarchical tree representation. However, it should perform well because of its constant lookup time.

The class `ClassLookup` defines a hash table and an array for looking up class names and their respective codes. Standard hash table methods are used for lookup. The pseudo-code algorithm for inserting class names and/or their codes into a pickle is presented in Figure 5.

A `findCodes()` method in `ClassLookup` first checks if a given class has already been encoded. If yes, a code is returned and is then written to the stream following a special byte 7E (this byte indicates that a class code is next in the stream). Otherwise, a new code is assigned to the new name and inserted into the lookup tables. Next, the package name (the one highest in the hierarchy for the given class) is checked. If it has been encoded before, the code is returned and written to the stream, followed by the remaining package/class name which has not been encoded up to this point (variable `rest`). If the package name is also new, it is inserted into the lookup tables and the next (less specific) part of the package name is considered. This process is repeated until either an encoding is found or there are no more parts of package name to consider. In that case, the full class name is written to the stream.

6. Experiments

This section describes experiments and tests run to evaluate the new serialization algorithm. We ran tests comparing sizes of serialized objects as well as tests comparing performance of both serialization algorithms. Analysis of the results is included in Section 7.

We have designed a number of simple classes in a package hierarchy to be used in the experiments. This hierarchy is shown in Figure 6. Classes `MainChat` and

`OtherClass` contain a small number of object data members (either standard Java classes or very simple user defined classes). Below, we describe a few of the experiments.

6.1 Size Experiments

This experiment compares the sizes of pickles generated by the standard Java serialization and by the new algorithm described in this paper. We tested serialization of simple objects (classes `MainChat`, `OtherClass` and three standard Java classes) and complex objects (classes from the `DistView` toolkit [18]). Both experiments included serializing 1, 2, 3, 4, and 5 different objects. The `DistView` experiment involved serializing different classes from different packages of the application.

We have decided to compare pickles generated by the new serialization algorithm to pickles compressed using gzip compression. Gzip compresses the entire pickle, not just the header part, so it can be used as a benchmark for our algorithm. We ran the same experiments as above but all pickles were compressed as well. We used the standard Java classes (`java.util.zip.GZIPOutputStream` and `java.util.zip.GZIPInputStream`) to compress serialized objects. The results of all the size experiments simple and complex objects are shown in Figures 7 and 8.

The improvement of the new algorithm over the old one is quite significant in all cases (up to 50% savings). Compression works best for large objects or a number of smaller objects. Compression algorithms take advantage of repeating patterns and work best when many similar objects or large objects, possibly with similar data

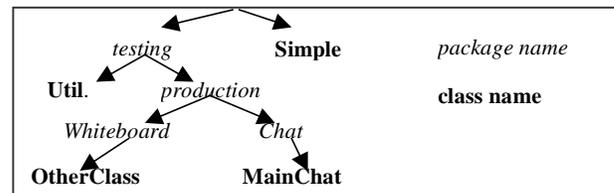


Figure 6: Classes used in experiments

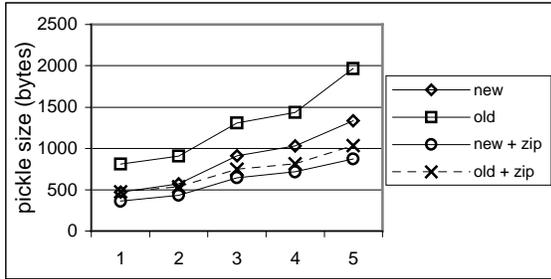


Figure 7: Pickle Sizes for Simple Objects

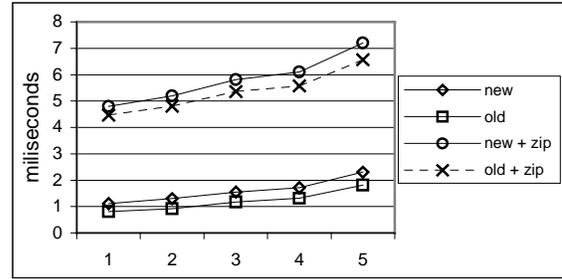


Figure 9: Serialization Performance on Windows NT - Simple Objects

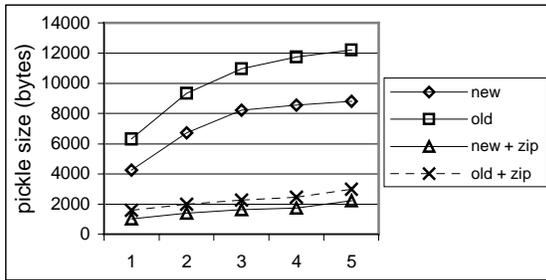


Figure 8: Pickle Sizes for Complex Objects

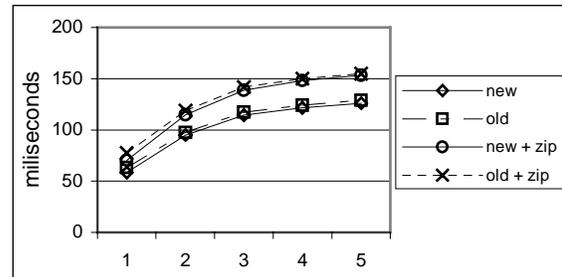


Figure 10: Serialization Performance on Windows NT - Complex Objects

members and data patterns, are serialized. The compression ratio is much worse when only a few small objects are serialized. This experiment also shows that combining compression with the new algorithm yields the best results (in terms of pickle size).

The next two sections show why gzip-type compression is not always a good choice for decreasing pickle sizes of serialized objects.

6.2 Time Experiments

This experiment measures the performance of object serialization. We wanted to make sure that the new algorithm does not introduce delays into the serialization process. Performance tests were performed on Windows NT and Solaris operating systems [17]. Both machines had Sun Microsystems' Java Development Kit 1.1.6 installed (with and without the modifications described in this paper). For more details on these tests, see [17].

We repeated the "size" experiments multiple times to get more accurate performance measurements. Execution times for simple objects are shown in Figure 9 (for Windows NT).

The times shown measure the number of milliseconds to open a serialization stream, call the `writeObject()` function to serialize the object, and to close the stream. As we can see, performance of the new serialization algorithm is comparable to that of the standard Java version. The compressed version is three to four times slower. We obtained similar results for large objects as well (Figure 10). The experiments indicate that compression introduces large delays into the serialization process. Next section discusses the experiments results.

7. Results Analysis

Our experiments indicate that the new serialization algorithm produces smaller pickles than the original scheme (up to 50% smaller). Performance tests show that performance of the new algorithm is comparable to that of the old algorithm. The experiments also show, that even though compressed pickles are much smaller, the time delay introduced by a compression algorithm may be prohibitive for some applications. We have estimated latencies (using experimental TCP performance data) for serializing an object and sending it over the network [2]. The estimates were done for the new and the old algorithms as well as the gzip compression. The results confirm the fact that performance of both (old and new) serialization algorithms is comparable. Those results also confirm the fact that the large latency introduced by the compression algorithm dominates the benefits of producing smaller pickles [17]. Additionally, the compression algorithm waits for the entire object to be serialized before it starts compression, which eliminates the stream feature where receiver can start reading data before the entire object is serialized. This increases the latency of a compression-based scheme even more. If pickle size is of the biggest concern, the combination of compression and the new algorithm yields the best results.

We have achieved one of our design goals of producing smaller pickles without degradation in performance. We, therefore, can conclude that the new serialization algorithm presented in this paper is a good

replacement for the standard Java serialization scheme. Standard compression algorithms can greatly reduce pickle sizes but the delay that they introduce makes them unsuitable for interactive distributed applications.

Both serialization algorithms were designed for stream communication. This means that the most efficient communication scenario would be to open a serialization stream on top of a socket stream and always communicate over that stream. Unfortunately, message-based applications do not take advantage of optimizations built into serialization algorithms. We intend to explore this issue of optimizing serialization for message-based protocols in our future work.

8. Concluding Remarks and Future Work

Object serialization is used in many distributed systems as the means of communicating data. Our experiments show that standard Java serialization can produce large pickles, which in turn can degrade the performance of such distributed systems. We developed a new serialization algorithm that decreases the size of serialized objects while implementing the same semantics as the old algorithm. Through a number of experiments, we determined that it produces smaller pickles (up to 50% smaller) without degrading performance. We claim that the new mechanism is a good choice for the standard Java serialization scheme. The new algorithm supports the same API and semantics as the old one, so it can be used transparently in place of the old algorithm.

We are currently exploring serialization mechanisms designed for distributed applications based on message passing (instead of streaming protocols). Such applications require a different approach to optimize sizes of serialized objects.

Another direction for future work is to modify the semantics of the serialization algorithm to improve performance. We are exploring the effects of removing fingerprints and field names from pickles. In addition, we will evaluate the possibility of changing the API to give developers a choice of serialization semantics.

9. References

- [1] K. Baharat and L. Cardelli, *Migratory Applications*, DEC Systems Research Center Technical Report 138, February 1995.
- [2] M. D. Bailey, *An Exploration of Java-Based Application Performance*, Report Submitted in Fulfillment of EECS Requirements for Ph.D. Candidacy, 1997.
- [3] A. Birrell and B. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, Vol.2, No. 1, February 1984.
- [4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber, *Network Objects*, Digital Equipment Corporation Systems Research Center Technical Report 115, February 1994.
- [5] M. Crochemore and T. Lecroq, *Text data compression algorithms*, In Handbook on Algorithms and Theory of Computation, edited by M. Atallah, CRC Press, Boca Raton, 1997.
- [6] D. Hagimont, P.Y. Chevalier, J. Mossiere, and X. Rousset de Pina: *Object Migration in the Guide System*, ECOOP'95 Workshop on Mobility and Replication, Aarhus, Denmark, August 1995.
- [7] R. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rassmussen, *Corona: A Communication Service for Scalable, Reliable Group Communication Systems*, In Proc. of the ACM Conference on Computer Supported Cooperative Work (CSCW 96), Boston, MA, Nov. 1996.
- [8] M. Herlihy and B. Liskov, *A Value Transmission Method for Abstract Data Types*, ACM Transactions on Programming Languages and Systems, vol. 4, no. 4, October 1982.
- [9] P. Homburg and M. van Steen and A.S. Tanenbaum: *Distributed Shared Objects as a Communication Paradigm*, Vrije Universiteit, Second Annual ASCI Conference, Lommel, Belgium, June 1996, Pages 132-137
- [10] V. Jacobson, *Compressing TCP/IP Headers for Low-Speed Serial Links*, RFC 1144, February 1990.
- [11] JavaSoft, *Object Serialization Specification*, <http://www.javasoft.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html>, 1997.
- [12] C. Jeness, *Object Serialization*, Golden Code Development Corporation, <http://www.fivepoints.com/ajug/info/tech/serial/serial.html>.
- [13] J. H. Lee, A. Prakash, T. Jaeger, and G. Wu, *Supporting Multi-Applet Workspaces in CBE*, In Proc. of the Sixth ACM Conference on Computer-Supported Cooperative Work, November 1996.
- [14] M. Mira da Silva and M. Atkinson. *Higher-order Distributed Computation over Autonomous Persistent Stores*. In Proc. of The Seventh International Workshop on Persistent Object Systems, Cape May, New Jersey, USA, May, 1996.
- [15] Object Management Group, *Externalization Service Specification*, CORBA Services: Common Object Services Specification, 1997.
- [16] The Open Group, *Transfer Syntax NDR*, DCE 1.1: Remote Procedure Call, 1997.
- [17] L. Opyrchal, *Efficient Object Serialization in Java*, Directed Study Report, <http://www.eecs.umich.edu/~lukasz/Papers/oral.ps>, September 1998.
- [18] A. Prakash and H. S. Shim, *DistView: Support for Building Efficient Collaborative Applications using Replicated Objects*, In Proc. of the 1994 ACM Conference on Computer-Supported Cooperative Work, ACM Press, October 1994, pp. 153-164.
- [19] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat, *Pickling State in the Java System*, In Proc. of the USENIX 2nd Conference on Object Oriented Technologies and Systems, June 1996.