Dude, Where's My Object?

A System for Sharing Java Objects

Team 4
David Euresti
Anant Saraswat
Stefanie Tellex
Lin Wu

Massachusetts Institute of Technology
December 6, 2001

Abstract

This paper describes the Dude, Where's My Object? (DWMO) system, which allows any Java object to be shared over a network. The system uses a modified Java Virtual Machine (JVM) to allow threads running on different machines to access each other's objects through the use of remote procedure calls. The system also uses a modified version of the Java Parallel Virtual Machine (JPVM) [3] library to allow JVMs to be passed object references on initialization, thus facilitating the distribution of necessary objects at the start of a parallel program. By implementing the sharing of objects in the JVM rather than as a new library, DWMO makes the use of remote objects completely transparent to the programmer. The Dude, Where's My Object system represents the first steps towards the goal of being able to take an unmodified threaded Java program and distribute it across multiple machines to achieve higher performance.

Introduction

The ability to make a heterogeneous cluster of machines appear to the user as a single, very powerful machine is perhaps the Holy Grail of distributed computing research. On the surface, Java is an ideal programming language to use to achieve this goal. Since it uses a virtual machine, Java hides the hardware details of different computers in the network from the programmer. In addition, Java already has well-defined mechanisms for writing multi-threaded programs. However, Java does not provide any convenient mechanisms for parallel computing on multiple machines in a cluster. The effective use of Java for parallelism, therefore, requires the existence of an efficient way to share objects over the network. An ideal system would be one in which the programmer could take a multi-threaded program and, with no modification whatsoever, migrate threads onto different machines in a network and see increased performance.

In this paper, we present the Dude, Where's My Object? (DWMO) system, which we believe is an important step on the road to this Holy Grail. The DWMO system allows objects to be shared over the network in a completely transparent fashion. It initializes JVMs on different machines using minimal modifications to an existing parallel virtual machine package, the Java Parallel Virtual Machine (JPVM). Not only does the current version of DWMO provide a useful abstraction, it also yields significant performance improvements over multi-threaded programs, for certain classes of applications. The ultimate goal of DWMO is to implement automatic thread migration so that existing multi-threaded programs can be distributed across multiple machines without requiring any alteration of code whatsoever, thus preserving the original semantics of the program while reaping the benefits of extra processing power.

The DWMO system is implemented as a series of modifications to the Kaffe JVM. DWMO allows different machines to share objects with each other by letting each JVM create *remote references* to objects residing on other machines. To the programmer, no difference exists between a local reference and a remote reference. However, the underlying JVM has been modified so that if a program tries to access the methods and fields of a remote object, that access is intercepted and converted to an RPC call to the machine on which the object actually resides.

Because it is implemented as a modified JVM, rather than as a package used by the programmer, the DWMO system allows the sharing of objects to be completely transparent to the programmer. This is a key distinction between DWMO and other systems that allow the sharing of objects, such as Java’s Remote Method Invocation (RMI) package or Birrell’s Network Objects [1]. RMI, for example, effectively requires that any object that can be accessed remotely be defined as an extension of a certain base remote class. Since Java does not have multiple inheritance, this imposes severe constraints on the program’s design. In addition, remote objects must define both an interface that specifies remote methods as well as an implementation class. This imposes extra work initially, as well as in the long term, since both the interface and the implementation class must be maintained. Finally, RMI arguments and return values are often passed by value, breaking the usual semantics of Java method calls.

DWMO, however, makes no programmer-visible distinction between remote objects and local objects. Classes are defined, and objects are declared, exactly as they would be in a normal program. If an object needs to be accessed remotely, the modified JVM handles the access transparently, requiring no effort on the part of the programmer.

The rest of the paper is organized as follows. Section 2 explains the design of our system in further detail. Section 3 presents the test results with the current implementation. Section 4 describes possible improvements to the system, and section 5 compares the system with other similar work. Finally, we present our concluding remarks in section 6.

2. Design

There are two important components to the design of the DWMO system. The first and most crucial piece is the modified JVM. The representation of objects in the DWMO JVM has been changed to permit references to remote objects. Furthermore, the DWMO JVM has been modified to accept method calls via RPC from remote JVMs, and to intercept all attempts to access remote objects and convert them to RPC calls to remote JVMs. The second piece of the DWMO system is the modified version of JPVM. JPVM is a package that allows the user to create and initialize JVMs on different machines. The JPVM package has been modified so that it can pass object references as arguments to the JVMs it activates, thus creating a simple method of distributing any objects that need to be shared at the start of a parallel program.

There are several open-source versions of the JVM available. The version being used for DWMO is version 1.0.6 of the Kaffe JVM [4]. This particular JVM was chosen because it is very well documented, and there is a large community of users who write modified versions of it. Hence, it was a logical choice for use in a project that requires modification of a JVM.

2.1 Remote References

The first important modification to the DWMO JVM is a new structure for object references. Kaffe, the Java Virtual Machine used in DWMO, is a large C program. In a normal JVM, each object is represented by a struct of type Hjava_lang_Object. This data structure is a header structure that contains a lock, which is used for synchronization purposes, and a dispatch table that has entries for the methods of the object. In Kaffe, the fields of the object are stored at the memory location immediately following the header itself. Each type of instruction that accesses a Java object, such as a method call or a field access, generates calls to functions in the JVM that handle that instruction.

```
typedef struct Hjava_lang_Object {
    struct _dispatchTable*  dtable;
    struct _iLock*          lock;
    int                     remote_memory_address;  //new field added by DWMO
    struct sockaddr_in      hostname;               //new field added by DWMO
    /* Data follows on immediately */
} Hjava_lang_Object;
```

Figure 1: modified HJava_lang_object

In the DWMO JVM, however, the Hjava_lang_Object has been modified to contain two extra fields, as shown in Figure 1. The hostname field contains the information needed to find out where the object resides, specifically the IP address of the machine that is storing it and the port that should be used to communicate with that machine via RPC. The remote_memory_address field contains the address in memory at which the Hjava_lang_Object corresponding to this object resides on its host machine. Whenever a Java program attempts to access a method or field of an object, the DWMO JVM first checks if the object is local by comparing the hostname field to its own IP address and port. If the object is not local, the JVM uses the RPC system, described in section 2.2, to execute that instruction on the remote machine and retrieve the result.

2.2 DWMO RPC

The DWMO JVM automatically converts Java bytecode instructions that act on a remote reference into an RPC to the host JVM of the object being referenced. DWMO's RPC protocol uses the Java Native Interface to execute remote calls. An object is passed by marshalling the hostname and mem_loc fields of its corresponding Hjava_lang_Object into a string of bytes. The class name of the object is also sent. DWMO assumes that all participating JVMs have the same set of class files, so additional type information does not need to be passed. This byte string contains all the information needed to locate and uniquely identify an object in the network. Primitives are passed by value, as with any Java method call. As a performance optimization, Strings are passed by value by simply sending the component characters of the string in sequence. Although Strings are normally passed by reference in Java, passing them by value cannot change the functioning of the Java program, since Java Strings are immutable.

The DWMO system currently implements three RPC calls, shown below:

CALL_METHOD:

This call intercepts the JVM's "invokevirtual" instruction, which handles the invocation of the majority of Java methods. CALL_METHOD takes four arguments: the object reference on which to invoke the method, the list of arguments to that method, the name of the method being called, and its unique signature. The class name of the object (included in the object reference) and name and signature of the method uniquely identify to the JVM which method to call. The method signature is needed in addition to the name because Java allows methods to be overloaded. The RPC returns the primitive or object reference returned by the invoked method.

A Java method can also exit by throwing an exception. Although we have not yet implemented this feature, it is a straightforward extension to the existing system. Instead of returning a primitive or object reference, the RPC would return an exception reference (an object reference that happens to be an Exception), and a flag indicating that an exception has occurred. The local JVM would then handle the exception as it would a local exception, propagating the exception up the call stack of the thread executing the method.

We have not yet implemented interception of interface method calls (the "invokeinterface" instruction) or calls to special methods such as private/protected or superclass methods (the "invokespecial" instruction). These should be very straightforward given our implementation of invokevirtual. Invoking static methods is a significantly more difficult problem, since every static method call must be broadcast to every JVM in order to preserve the semantics of Java.

GET_FIELD:

This maps to the JVM's "getfield" instruction, which returns the value of a non-static field of an object. This call takes as arguments the object whose field is to be retrieved, the field name, and the field signature. It returns the desired field. At this time, it is not possible to write directly to a field of a remote object, although this is a straightforward extension to the existing system. A mutator method is required to change the value of a field.

GET_ARRAY_LENGTH:

This maps to the JVM's "arraylength" instruction, which returns the length of an Array object. It takes an array object as an argument and returns its length.

The current version of DWMO does not intercept certain other functions, such as array load and store instructions, or instructions to set a field value. However, for the vast majority of these functions, the modifications necessary are very similar to those that have already been made, and will be implemented in subsequent revisions of the system. Only a few special cases, such as access of static members, will require significant effort.

2.2.1 RPC Threading

The DWMO JVM contains a background thread used to send and receive RPC calls. This thread is created using Kaffe's jthread library, a user level threading system, and exists as a java.lang.Thread object within the DWMO JVM. The RPC thread is implemented in C++ using SFS's asynchronous library [7]. Because this library uses static data structures, a single thread must be used for both sending and receiving calls. This thread listens for clients requesting remote operations on the DWMO JVM, and, when requested, also connects to other DWMO JVMs in order to request the execution of remote operations. Unfortunately, the asynchronous library does not mesh well with the user-threading library Kaffe provides; when the asynchronous library blocks on the "select" function, waiting for clients to connect to make an RPC call, it stops all other threads in the JVM from running. To get around this we had to resort to polling; we forced the RPC thread to sleep most of the time, only checking if there was an operation pending every 50 milliseconds. Thus, on average there is an unnecessary delay of 50 milliseconds on each RPC call; it takes on average 25 extra milliseconds for the thread to notice a new request, and 25 extra milliseconds to send the response. A number of solutions to this problem have been proposed for the next version of DWMO. These solutions are presented in section 3 below.

When the asynchronous library receives a request to execute a method, it starts a new jthread in which to execute the method call. This new thread serves two purposes. First, the asynchronous library requires that the callback functions never block. If the method call ran in the RPC thread itself, it could call other remote methods and potentially block, deadlocking the system. Second, it provides a framework in which we will eventually implement more of the semantics of a Java method call. Java provides methods that allow a method to determine what thread it is running in, the priority of that thread, and other related information. This information exists on the JVM that requested this method be called. Eventually, it should be included in the RPC call that invokes methods so that we can make the local thread that actually executes the method look more like the remote thread that called it.

2.3 JPVM

In addition to dynamically creating objects, a distributed program must also have the ability give threads access to certain objects when the threads are initialized. In an ordinary threaded Java program, this would be done by passing the objects in the constructor of the new thread. If, for example, one were to use DWMO to have multiple machines compute the factorial of Integers in a Vector, it would make sense to have all of the threads be initialized with that Vector. One eventual goal of DWMO is to implement fully transparent thread migration, so that these initialization issues are completely hidden from the user. For now, however, initialization is achieved by the use of a modified version of the Java Parallel Virtual Machine (JPVM) package. The normal version of the JPVM package starts multiple JVMs distributed across different machines and provides a message-passing interface that allows those JVMs to communicate. However, the normal JPVM package does not allow the user to pass objects to JVMs, only primitives. Therefore, in order to use JPVM as the thread initializer for DWMO, we had to modify the JPVM package to allow it to send object references as well as primitives to other machines.

In order for the JPVM to pass an object as an argument, it must have access to the hostname and memory location of that object. It cannot use the marshalling code that exists for doing RPC, since JPVM's marshalling must be done at the Java level. In order to enable marshalling, the java.lang.Object class was modified to add three new methods:

```
public native int getMemLoc(); // returns the memory address of the object
public native short getPortNumber(); // returns the port used by object's host JVM
public native long getAddr(); // returns the IP address of the object's host JVM
```

These new methods are used by the JPVM to allow objects to be passed to other JVMs. In order to send an object, the JPVM converts the results of getMemLoc(), getPortNumber, and getAddr(), and the class name of the object into a byte stream. (The class name is already available in Java via the getClass method.) On the other end, it extracts the class, memory location, port number, and IP address from the buffer and uses the static method Object Dwmo.getReference(long HostAddress, short Hostport, int MemoryLocaiton, Class class) to create a remote reference to the object.

3. Performance

Table 1 shows the performance of a no argument, no return value method call in different environments. Remote method calls take several orders of magnitude longer to run than local ones because of the high latency of the communication process. Therefore, DWMO is better suited to applications that have a high ratio of processor activity to network communication, so that the cost of the communication can be amortized away.

JVM	Speed
DWMO - interpreter	70 milliseconds
Kaffe – interpreter	0.00106 milliseconds
DWMO - Just In Time compiler	0.000014 milliseconds

Table 1: Performance results for method calls in different environments

Figures 2 and 3 show code fragments that illustrate how DWMO can be used to share objects. Figure 2 shows code in which a master thread running on one machine shares its System.out object with a worker thread. System.out is the object that allows Java to communicate with the terminal. The master program packs its System.out into a jpvmBuffer and sends it to the worker program, which prints to it. The output of the worker thread will actually appear on the master machine, since the worker is calling a method on a remote object.

Code to send System.out to the slave JVM (using JPVM to bootstrap.):

```
//Code running on the master JVM:
// Pack System.out and send it to the worker
jpvmBuffer buf = new jpvmBuffer();
buf.pack(System.out);
jpvm.pvm_mcast(buf,tids, num_workers, 1000);
```

Code to receive System.out and use it (using JPVM to bootstrap.):

```
//Code running on the worker JVM
jpvmMessage m = jpvm.pvm_recv();
PrintStream systemout = (PrintStream) m.buffer.upkobj();
systemout.println("I am the worker thread, calling a method” +
“on master's System.out object."); //this text will appear on the output of the master
```

Figure 2: Code to share System.out

Figure 3 shows master and worker code fragments that demonstrate how DWMO can be used to share files. Files can be read or written in Java by binding them to FileInputStream and FileOutputStream objects, respectively. These objects can be passed through the system by DWMO, thus enabling file sharing. In this example, the master program creates a FileInputStream object for a new file and sends it to a worker thread. This allows the worker thread to read the file.

```
//Code on the master JVM
public static FileInputStream createFile() throws IOException {
        File f = new File("/tmp/dwmo-master." +
                        System.currentTimeMillis());
                        f.createNewFile();
```

```
        FileOutputStream o = new FileOutputStream(f);
        o.write("I, the master, create a file\n".getBytes());
        o.close();
        return new FileInputStream(f);
}

// Send the file to the worker
jpvmBuffer buf = new jpvmBuffer();
buf.pack(createFile());
jpvm.pvm_mcast(buf,tids, num_workers, 1000);

//Code on the worker JVM
jpvmMessage m = jpvm.pvm_recv();
FileInputStream in = (FileInputStream) m.buffer.upkobj();
FileOutputStream out = new
FileOutputStream("/tmp/dwmo-slave." + System.currentTimeMillis());

int i;
while ((i = in.read() )!= -1) {
   out.write(i);
}

out.close();
in.close();  // note that the worker can close the master's FileInputStream when finished.
```

Figure 3: Code to share a file

As both of these examples show, once the worker threads receive remote objects from JPVM, all accesses to those objects are completely programmer-transparent. The initialization using JPVM is the only step that requires special effort on the part of the programmer, and for larger programs that effort is comparatively insignificant. In future versions of DWMO that implement thread migration, even the minimal effort required by JPVM will be unnecessary, thus making every aspect of the system completely transparent.

In addition to providing a convenient method to share objects, DWMO provides significant performance increases over conventional multi-threaded programs, for certain classes of applications. DWMO provides the best performance increases for applications in which each thread is computationally intensive. In an application in which threads spend most of their time on blocking operations, such as waiting for a disk read to complete or for a connection to be made, there is no real need for extra processing power. However, in applications where each thread is computationally intensive, it makes sense to distribute the threads over multiple processors. We tested DWMO with two such applications, a WorkQueue application and the Bailey-Borwein-Plouffe Pi Algorithm. In both cases, we found significant performance increases with DWMO.

The Bailey-Borwein-Plouffe Pi Algorithm [2] computes the nth hexadecimal digit of pi without knowing the previous n-1 digits. Using this algorithm, we constructed two versions of a program that computed the first 2000 hexadecimal digits of pi, one which was a multi-threaded program running on a single JVM, and one that used DWMO on two machines. The single JVM version took 3 minutes and 37 seconds to run, whereas the DWMO version took only 1 minute and 41 seconds to run, an improvement of 53%.

The WorkQueue application involves a more abstract test. A WorkQueue is a queue in which each entry indicates a type of "work" to perform and how long to perform that work. Work elements are pulled off the queue in order and processed. The idea behind this kind of queue is that one can simulate an application that may involve a list of more complex tasks that can be performed independently. For example, a multi-threaded web server might put incoming requests into a queue, so that child threads could pull off and process a new request as soon as they are done with their current one. In this case, the "work" is to compute String.toLowerCase on a given String a certain number of times. In this test, we constructed four different versions of the program. There were single-JVM versions with one, two, and three threads, and a DWMO version that had two threads on two different machines. In the multi-threaded and DWMO versions, the queue is shared; as soon as a thread finished processing one entry, it pulls the entry from the head of the queue and processes it. The results are shown in Table 2.

Program Version	Run Time(min:sec)
1 JVM, 1 thread	1:02.97
1 JVM, 2 threads	1:06.93
1 JVM, 3 threads	0:57.63
DWMO with 2 JVMs	0:40.23

Table 2: Test Results for WorkQueue

These results show that our system is capable of reaping the performance benefits of additional CPUs. In addition, in both cases, the code in the DWMO version is not much more complex than the threaded version. The same general architecture was used; the program code and design were unaffected by the distributed nature of DWMO. Additional code complexity came from the extra JPVM initialization required to distribute initial shared objects, as well as from the need to work around JVM instructions we have not yet implemented for remote objects. In a completed DWMO system, with implementations of all relevant JVM instructions and transparent thread migration, the added code complexity would disappear completely.

4. Proposed Improvements

Although the current version of DWMO is a first good step towards the Holy Grail, many issues still need to be addressed. Firstly, the DWMO JVM must be able to convert any JVM instruction into an RPC, not just the most commonly used ones. Secondly, initialization of JVMs on machines in the network must be made totally transparent to the user. Finally, the RPC process must be made as fast as possible, since the transparency sought by DWMO is useless unless it also provides performance benefits.

Currently, the most serious performance problem in DWMO is the way in which the RPC system interacts with Kaffe's internal thread system. Of the two potential solutions mentioned in section 3, we have already tried the second option, but have found that the Kaffe JVM does not support pthreads under FreeBSD. Therefore, the first option is probably the better candidate for inclusion in any revision of DWMO.

While the modified JPVM used by DWMO is useful, it is not meant as a permanent fixture of the system. Even though the programming overhead it requires is small, that overhead still exists and ideally should be removed. The best way to do this would be to further modify the DWMO JVM so that it is capable of finding a machine on the network, activating a JVM on that machine, and then using thread migration to start a program on the remote JVM. We believe that this is a tractable problem. However, because of massive investment of time and effort we think thread migration would require, we felt it was beyond the scope of this paper.

At this time, DWMO does not attempt to replicate objects on multiple machines. It is likely that such a step would improve performance, but only if the benefits gained from exploiting locality of reference could offset the costs of implementing coherence protocols and garbage collection. Future versions of DWMO might incorporate replication of objects if it is determined that performance might improve as a result.

5. Related Work

The original motivation for using RPC calls to implement sharing in DWMO comes from the *network objects* system developed by Birrell, et. al. [1] A network object is a special type of object in the Modula-3 programming language that can be accessed remotely. When a client receives a reference to an object, the code that unmarshals the reference also automatically generates a set of stubs. Method calls on this object actually call the stub functions, which in turn perform RPC calls to the machine that actually owns the object in question. However, in addition to being implemented in Modula-3 instead of Java, network objects impose significant programming overhead on the user. The programmer must explicitly designate a certain classes as being sharable, and all objects must be pure objects, since network objects do not support field accesses. In addition, all network objects must extend a common superclass. Since Modula-3 does not support multiple inheritance, this constrains the design of the program and gives up some of the power of object-oriented programming.

There are many other systems in use today that provide for object sharing in Java, but many of them require special effort on the part of the programmer or are not well suited the same applications targeted by DWMO. Java Remote Method Invocation (RMI), for example, requires very careful design of classes on the part of the programmer. He has to decide if arguments should passed by value or by reference, design an interface to the sharable objects, and maintain interface files for each of those objects. RMI is well suited to applications in which the server and client do not trust each other or when the division between client and server is clear. However, few multi-threaded programs use this client-server approach.

JavaParty [6] implements a new Java class modifier, "remote," which the programmer uses to share objects over a network. Once a class has been declared “remote”, it can be accessed transparently from any of the JVMs participating in the system. In addition, unlike DWMO, JavaParty does not require a custom JVM. Remote objects are implemented via RMI; it uses a preprocessor to automatically generate the RMI infrastructure required. The goal of DWMO, however, is to make the process of sharing as transparent to the programmer as possible, which means that having to use a new keyword is undesirable.

The system most similar to DWMO in terms of programmer interface is the Java/DSM system [8]. In Java/DSM, the JVM is modified so that the programmer need not use any special structures to access shared objects. Instead, the JVM allocates all objects in a shared memory space and uses the Treadmarks DSM system to share that space. However, Java/DSM is not completely programmer transparent, since the programmer is required to know the location of each thread.

6. Conclusion

Dude, Where's My Object? allows objects to be shared between machines in a network without programmer-visible distinction between local and remote objects. The system does require some extra effort to initialize the different JVMs, but this effort is relatively minor and relies on a small set of modifications to a commonly used parallel virtual machine library. Most importantly, though, our test results show that DWMO can provide tremendous performance improvements for certain classes of applications.

We are convinced that, given more time, all of the current issues the system faces can be resolved. With successful thread migration, the minimal programmer effort required to use DWMO will be completely eliminated. With a more efficient RPC system, DWMO will be able to provide performance improvements to a wider variety of applications. And with the implementation of interception for the remaining JVM instructions, users will be able to harness the full power of Java for parallel programming in a network.

At this time, we cannot claim to have found the Holy Grail, the ability to make a cluster of heterogeneous machines behave as a single very powerful machine. However, we believe that the Dude, Where's My Object? system is an important step in the right direction.

Bibliography

[1] Andrew Birrell, Greg Nelson, Susan Owicki, Edward Wobber. Network objects. SOSP 1993.

[2] Steven Finch. The Miraculous Bailey-Borwein-Plouffe Pi Algorithm.[1]http://www.mathsoft.com/asolve/plouffe/plouffe.html. 2001.

[3] Java Parallel Virtual Machine. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[4] Kaffe. http://www.kaffe.org.

[5] Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification Second Edition. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html. May 1999.

[6] Michael Philippsen, Matthias Zenger.JavaParty - Transparent remote objects in Java. Concurrency: Practice & Experience. 9(11) p. 1225-1242.November, 1997.

[7] SFS. [3]http://www.fs.net/.

[8] Weimin Yu, Alan Cox. Java/DSM: A platform for heterogeneous computing. Concurrency: Practice & Experience. 9(11) p. 1213-1224. November, 1997.