Efficient Replicated Method Invocation in Java

Jason Maassen, Thilo Kielmann, Henri E. Bal
Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
jason@cs.vu.nl kielmann@cs.vu.nl bal@cs.vu.nl
`http://www.cs.vu.nl/manta`

ABSTRACT

We describe a new approach to object replication in Java, aimed at improving the performance of parallel programs. Our programming model allows the programmer to define groups of objects that can be replicated and updated as a whole, using totally-ordered broadcast to send update methods to all machines containing a copy. The model has been implemented in the Manta high-performance Java system. Performance measurements on a Myrinet cluster show that the replication mechanism is efficient (e.g., updating 16 replicas of a simple object takes 68 microseconds, only slightly longer than the Manta RMI latency). Example applications that use object replication perform as fast as manually optimized versions based on RMI.

1. INTRODUCTION

Object replication is a well-known technique to improve the performance of parallel object-based applications [3]. Although several different forms of object replication have been proposed for Java [9, 14, 19, 25, 27], no scheme exists yet that transparently and efficiently supports replicated objects in Java and that integrates cleanly with Java's primary point-to-point communication mechanism, Remote Method Invocation (RMI) [28]. Some systems temporarily cache objects rather than trying to keep multiple copies of an object consistent [9, 14, 19, 27]. Some proposals have a programming model that is quite different from the object invocation model of RMI [25]. Also, performance results are often lacking or disappointing. The probable reason for these problems is the inherent difficulty in implementing object replication. In particular, it is hard to find a good programming abstraction that is easy to use, integrates well with RMI, and can be implemented efficiently.

In this paper we introduce a new compiler-based approach for object replication in Java that is designed to resemble RMI as much as possible. Our model does not allow arbitrarily complex object graphs to be replicated, but deliberately imposes restrictions to obtain a clear programming model and high performance. Briefly, our model allows the programmer to define closed groups of objects, called clusters, that are replicated as a whole. A cluster has a single entry point, called the root object, on which its methods are invoked. The compiler and runtime system together determine which methods will only read (but not modify) the object cluster; such read-only methods are executed locally, without any communication. Methods that modify any data in the cluster are broadcast and applied to all replicas. A single broadcast message is used to update the entire cluster, independent of the number of objects it contains. The semantics of such replicated method invocations are similar to those of RMI.

We have implemented this scheme in the Manta high-performance Java system [18, 26]. Updating a simple object replicated on 16 Myrinet-connected machines takes 68 microseconds, only slightly longer than the RMI latency in Manta. We have also implemented two parallel Java applications that use replicated objects, which we use to illustrate efficiency and ease of programming of replicated objects in Manta.

The contributions of the paper are as follows:

- We propose a new model, similar to RMI, that allows closed groups of objects to be replicated.
- We describe a compiler-based implementation of this model as part of the Manta system.
- We analyze the performance of this implementation on a Myrinet cluster, using a micro benchmark and two applications, showing the performance benefits of object replication in Java.

The outline of the rest of the paper is as follows. In Section 2, we describe our approach to object replication. In Section 3, we discuss the implementation in the Manta system. In Section 4, we discuss the implementation and performance of two parallel applications. In Section 5, we look at related work. Finally, in Section 6, we present our conclusions.

2. REPLICATION IN MANTA

The primary goal of our object replication mechanism is to provide a programming model as close as possible to RMI. With RMI, parallel applications strictly follow Java's object-oriented model in which client objects invoke methods on server objects in a location-transparent way. Each remote object is physically located at one machine. Although the RMI model hides object remoteness from the programmer, the actual object location has a strong impact on application performance.

From the client's point of view, object replication is conceptually equivalent to the RMI model. The difference is in the implementation: objects may be physically replicated on multiple processors.

The advantage of replication is that read-only methods (i.e., methods that do not modify the object's data) can be performed locally, without any communication. The disadvantage is that write methods become more complex and have to keep the state of object replicas consistent. For objects that have a high read-write ratio, replication will reduce communication overhead.

Data replication can be implemented in different ways, influencing both performance and the programming model. Many systems that use replication apply an *invalidation* scheme where the replicas are removed (invalidated) after a write method. Our experiences with the Orca language, however, show that for object-based languages an *update* protocol often is more efficient, especially if it is implemented with *function shipping* [3]. With this strategy, a write method on a replicated object is sent to all machines that contain a copy. Then the method is applied to all copies. For object-based systems, this strategy is often more efficient than invalidation schemes. Especially if the object is large (e.g., a big hash table), invalidating it is unattractive, as each machine must then retrieve a new copy of the entire object on the next access. With function shipping, only the method and its parameters are sent, usually resulting in much smaller data transfers than with invalidation schemes or data shipping schemes, which send or broadcast entire objects. Manta therefore uses an update mechanism with function shipping. To update all replicated clusters in a consistent way, methods are sent using *totally-ordered group communication* [3], so all updates are executed in the same order on all machines.

Remote method invocation (RMI) can be seen as a simple form of function shipping to a single, remote object. This is why we call our approach *replicated method invocation*. As with RMI, the arguments to methods of a replicated object have *call-by-value* rather than *call-by-reference* semantics. The same holds for return values. Because methods are executed once per replica, return values as well as possibly raised exceptions will be discarded on all nodes except the one on which the method was invoked.

A difficult problem with object replication is that a method invoked on a given object can also access many other objects, by following the references in the first object. A write method can thus access and update an arbitrarily complex graph of objects. Synchronizing multiple concurrent write methods on different (but possibly overlapping) object graphs is difficult and expensive. Also, if the function-shipping update strategy is applied naively to graphs of objects, broadcast communication would be needed for each object in the graph, resulting in a high communication overhead. Orca avoids these problems by supporting a very simple object model and disallowing references between objects (see Section 5). A simple solution for Java would be to replicate only objects without references to other objects, but this would be far too restrictive for many applications. For example, it would then be impossible to replicate data structures like linked lists, since these are built out of objects (unlike in Orca).

Our solution to this problem is to take an intermediate approach and replicate only closed groups of objects, which we call *clusters*. A cluster is a programmer-defined collection of objects with a single entry point, that will be replicated and updated as a whole. Hence, a write method on a cluster is implemented using a single broadcast message, independent of the number of objects in the cluster. The entry point of a cluster is called its *root*, and it is the only object that can be accessed by objects outside the cluster. In addition, a cluster can have other objects reachable from the root, called the *node* objects; these node objects, however, cannot be referenced directly from outside the cluster. As a consequence, only methods of the root object can be directly invoked in order to manipulate (read or modify) the cluster. All other method invocations inside the cluster can only be the indirect result of an invocation on the root object.

This model is general enough to express all common data structures like lists, graphs, hash tables, and so on. Also, the model is restrictive enough to allow a simple and efficient implementation, as will be discussed later. As the Java object model has no notion of clustered (or compound) objects, we have defined a new and simple programming interface in Manta to express this clustering mechanism. We discuss this interface below.

2.1 Programming interface and example

Object clusters are defined by the application programmer, using two so-called "special" interfaces to mark cluster objects. This approach is similar to RMI, where the special interface *java.rmi.-Remote* is used to identify remote objects. Root objects are identified by implementing the interface *manta.replication.Root*, while node objects implement *manta.replication.Node*. The use of these interfaces allows the Manta compiler to recognize cluster objects such that replication-related code can be generated (see Section 3). Furthermore, the Manta compiler has to enforce certain restrictions on replicated objects in order to maintain replica consistency, as discussed in Section 2.2.

```
class StackNode implements manta.replication.Node {
        StackNode prev;
        int value;

        public StackNode(int d, StackNode p) {
               value = d;
               prev  = p;
        }
}

class Stack implements manta.replication.Root {
        private StackNode top = null;

        public void push(int d) {
                top = new StackNode(d, top);
        }

        public int pop() throws Exception {
                StackNode temp = top;
                if (temp != null) {
                        top = top.prev;
                } else {
                        // throw exception.
                }
                return temp.value;
        }

        public int top() throws Exception {
                if (top == null) {
                        // throw exception.
                }
                return top.value;
        }
}
```

Figure 1: A replicated stack

To illustrate the use of the two special interfaces, Figure 1 shows a simple example of an object cluster, a replicated stack, implemented as a linear list. Whenever a new *Stack* object is created, a new cluster is created using the *Stack* object as its root. By calling the *push* method, *StackNode* objects will be added to this cluster. Together with the root, these objects form a well-defined closed group. If the methods of the *Stack* class would use objects instead of simple integer values, the call-by-value semantics for parameters and return values ensure that no external references exist to the objects inside the cluster.

Once a replicated *Stack* has been created, a reference to it can be passed to different machines using normal RMI calls. Manta's runtime system on the remote machine will replace this reference by a reference to its local replica, creating a new one if a local replica does not yet exist. From the programmer's point of view, clusters are thus passed by reference via RMI, just like ordinary remote objects. Also, method invocations on replicated clusters are similar to normal remote method invocations, as illustrated by the methods of the *Stack* class. As with RMI, the methods generally have to be synchronized (using Java's *synchronized* keyword); in Manta, write methods of replicated objects are automatically synchronized, read methods are only synchronized if specified in the program.

2.2 Restrictions on replicated objects

In the RMI model, remote method invocation is not completely transparent, and some restrictions are applied on remote objects due to the presence of multiple address spaces. These restrictions also apply to replicated objects in Manta. For example, just as RMI disallows direct access to the fields of a remote object via a remote reference, Manta disallows direct access to the fields of the root object. In addition, Manta has several other restrictions for replicated objects, which are necessary to ensure replica consistency. We discuss these restrictions below. The Manta compiler tries to enforce them, and produces error messages whenever it detects violations.

No remote references. As a result of our decision to replicate only closed groups (clusters) of objects, cluster objects cannot contain references to remote objects. Also, the methods defined for (the root of) a cluster cannot take remote objects as parameters (but only scalar data, arrays, and node objects). Because remote objects are accessed via their remote references, they would be shared by all replicas of a cluster rather than being replicated themselves. In such a case, the function shipping approach would cause the *nested invocation problem* [20], illustrated in Figure 2. On the top, A's *meth* method calls *incr* on the remote object B. When A gets replicated (shown on the bottom), function shipping will invoke *meth* on all replicas, in turn causing all of them to invoke *incr* on B. This in general leads to erroneous program behavior that depends on the actual number of replicas. Manta avoids this problem by replicating closed groups of objects, so it disallows references to remote objects from within a replicated cluster (e.g., the reference from A to B is not allowed).

Restrictions on the use of special interfaces. Our programming interface does not allow a class to implement both the root interface and the node interface, because that would make it difficult to cleanly separate different clusters from each other. For the same reason, root and node objects may only contain references to node objects. This restriction also rules out references from a node back to the root object of its own cluster. As all objects in a cluster have to implement either the root or the node interface, and as remote references are not allowed inside clusters, classes of root and node objects are not allowed to also implement the remote interface.

No static variables. The use of static variables is not allowed in root and node objects, as static objects may also be accessed and modified from outside the cluster. This would break the call-by-value semantics which enforce node objects to be private copies of their cluster.

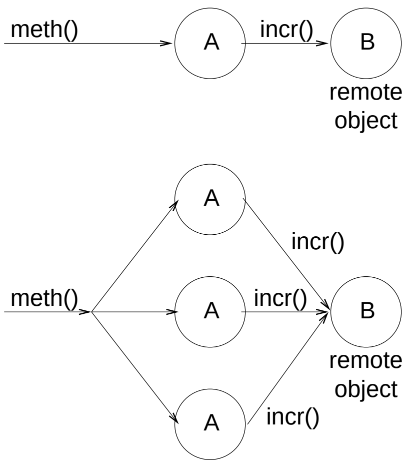

Figure 2: The nested method invocation problem

Only calls to "well-behaved" methods. Inside the methods of the root and node objects, methods of other classes may be called given that they are "well-behaved", deterministically producing identical results on all machines. Their implementation must not depend on static variables or methods, random generators, I/O, or the local time.

To summarize, our model deliberately disallows references between different clusters or between clusters and remote objects. Also, it uses call-by-value semantics for the parameters and result values of replicated method invocations (as RMI does). As a result, a cluster is a closed group of objects, that can be replicated efficiently, as discussed in the next section.

3. IMPLEMENTATION

The implementation of Manta's object replication is partially inside the Manta compiler and partially in the runtime system. Manta uses a static (native) compiler, which translates Java programs to executables [18]. The compiler generates code wrappers for classes implementing the *manta.replication.Root* and *manta.replication.-Node* interfaces, checks the restrictions on both root and node objects, and most importantly, analyses the methods of root and node classes to distinguish between read and write operations. The runtime system establishes object clusters and updates them on all nodes. It also coordinates the execution of method invocations to enforce replica consistency.

3.1 Read/write analysis

The advantage of object replication compared to RMI is that methods which only read objects can be performed locally, without any communication. Only write operations cause communication across the set of replicas. To distinguish between read and write methods, the Manta compiler has to analyze the method implementations. Therefore, the compiler checks if there are any operations in the method that assign values to class variables, or if there are calls to other methods that can be attributed as write methods. If so, the method is classified as a write method, otherwise it is considered to be a read method. Also, if a method may execute a *notify* or *notifyAll* operation, it is a write operation. The implemented analysis is conservative by always classifying methods that contain assignments as write methods, even if the assignments may only be executed conditionally. Furthermore, methods of classes other than for root or node objects are assumed to be free of side effects (see Section 2.2), and can thus safely be ignored in the read/write analysis.

Unfortunately, this analysis cannot be performed completely at compile time. Due to Java's support for polymorphism and dynamic binding, the method to be invoked depends, in general, on the runtime type of the object. Since a read-only method of one class may be overridden by a write method in a subclass (or vice versa), it may not be known until runtime whether a given invocation reads or writes an object. Still, it is important to execute each method in the correct mode (read or write). If a read-only method would be executed as if it were a write method, it would be broadcast, resulting in much overhead. Even worse, if a write method would accidentally be executed as if it were a read-only method, erroneous program behavior would occur. Due to this problem, the final check to distinguish between read and write operations is performed at run time. In Manta, wrappers are generated for all methods of root and node objects in which the current execution mode (read or write) is checked before actually invoking the object's method. If the current invocation is executed in read mode, and the actual method requires write mode, the current invocation is aborted and restarted in write mode. This may, for example, happen during the execution of a method of the root object when another method of a node object is to be called. This restart can be performed safely, because so far only read operations have been executed, and the object state has not changed yet.

3.2 Code generation

The compiler generates method wrappers for all methods of root and node objects in order to maintain read or write mode, and possibly perform restarts. Apart from that, read methods are directly called on the local replica from within the corresponding wrapper.

Write operations are performed in two phases. First, the method wrapper broadcasts a call header and the parameters to all replicas, including itself. The broadcast mechanism we use is part of the underlying Panda layer [3], which handles all communication between Manta nodes. Panda's broadcast is totally ordered, so all machines receive all broadcasts in the same order. This way, all replicas perform write operations in the same order, causing them to be consistent with each other.

On each node, a separate thread consecutively processes incoming broadcast messages. The call header and the parameters are extracted, and a handler method executes the respective method on the local object replica. For transferring parameter objects, the standard object serialization method from Manta's RMI protocol is used. The serialization code is generated by the Manta compiler and is highly efficient [18].

Finally, when the method completes, its outcome (result object or raised exception) is intercepted by the handler. On the invoking node, the outcome will be forwarded to the original caller. On all other nodes, the outcome is simply discarded.

3.3 Cluster management

Whenever a new root object is created, a new cluster is implicitly created along with it. On the invoking process, the root object is created, and a unique identifier is assigned to it. In turn, the new cluster is broadcast to all nodes of the parallel application, using Panda's totally ordered broadcast mechanism. This ensures that clusters are always created on all nodes before any write operation attempts to modify them.

Although the replicated clusters are immediately established on all nodes, the application itself views them as being replicated on demand. Only the process on which the cluster was created gets a reference to the new cluster. The application code then has to distribute the reference to other nodes using RMI.

A possible optimization of this scheme would be to replicate a cluster only on those nodes that actually have a reference to it. This could avoid some overhead of processing write updates on objects that are not used on some of the nodes. As a drawback, elaborate group management would have to be implemented. Our current implementation simply replicates all clusters on all nodes. Our previous experience with the Orca shared object system indicates that this approach yields adequate performance [3].

3.4 Wait and notify

The execution model for write methods also has to correctly handle synchronization for *wait, notify*, and *notifyAll* primitives. Whenever a broadcast message for invoking a write method is received, the method will not immediately be executed. Instead, each object cluster has a queue for incoming broadcast messages, and a thread waiting for messages to appear in the queue. Whenever a message appears, the thread takes it out of the queue and invokes the respective method. All write methods are therefore executed by a single thread, one at a time, in the order they were received in. This model ensures that all nodes execute all write methods in the same order.

This single-threaded scheme cannot be used for executing write methods that may block while calling *wait*. In this case, no other write methods will be able to run, including the one intended to wake up the blocked method. This problem is illustrated in Figure 3, which presents the code of a *Bin* object, a simple bounded buffer with a single data slot. The *get* method will block until a value has been written into the bin, then it empties the bin, and wakes up other, waiting, methods. The *put* method will block until the bin is empty, it will fill the bin, and then wake up waiting methods. Both *put* and *get* are write methods (they change *filled* and call *notifyAll*), and are therefore broadcast to all replicas. On each node, the corresponding messages are put into the queue. If a *get* would block because the *Bin* object is empty, the thread serving the write method would block and the *put* that was intended to wake up the *get* would never be executed.

A simple-minded solution would be to create one thread for each incoming broadcast message. Unfortunately, the global execution order could then no longer be guaranteed. Instead, we use a solution similar to the *Weaver* abstraction introduced in [23]. A new thread is created whenever the original thread blocks. Although this happens in the same order on each node it still has to be guaranteed that blocked threads also wake up in exactly the same order on all nodes, otherwise the total execution order for write methods would still be violated. Unfortunately, Java's *wait/notify* mechanism does not guarantee any order in which waiting threads will wake up. Manta's runtime system therefore provides specific implementations of *wait, notify*, and *notifyAll* for replicated objects. Here, the execution of *notifyAll* on a root or node object causes waiting threads to be put back into the execution queue in exactly

```
class Bin implements manta.replication.Root {
        private boolean filled = false;
        private int value;

        public synchronized int get() {
                while (!filled) wait();
                filled = false;
                notifyAll();
                return value;
        }

        public synchronized void put(int i) {
                while (filled) wait();
                value = i;
                filled = true;
                notifyAll();
        }
}
```

Figure 3: A replicated *Bin* object

the global order in which they were invoked. The current thread servicing the queue will then detect that the head of the queue contains a blocked thread, wake this thread up, and terminate itself. The woken up thread will then continue to run and wake up the next thread when it terminates. The last thread will not terminate, but continue servicing new calls from the queue. This way, all machines will wake up the threads in the same order and keep the copies of the object clusters consistent.

The solution presented here is specific to the Manta system. In Manta, the implementation of the *wait, notify,* and *notifyAll* methods are aware of object replication. Because the implementations of these methods in the Sun JDK are *final* (i.e., not overloadable), we are not allowed to replace them with replication aware versions, making it harder to implement our scheme in a non-Manta Java system. A solution would be to offer alternative methods with different names, or to use a preprocessor to replace calls to *wait, notify* and *notifyAll* at compile time.

3.5 Performance evaluation

To evaluate the performance of Manta's replication mechanism, we implemented the *Stack* class from Figure 1 and compiled it with our Manta system. Our experimentation platform, called the *Distributed ASCI Supercomputer* (DAS), consists of 200 MHz Pentium Pro nodes each with 128 MB memory running Linux 2.0.36. The nodes are connected via Myrinet [5]. Manta's runtime system has access to the network in user space via the Panda communication substrate [3] which uses the LFC [4] Myrinet control program. Myrinet lacks a hardware broadcast facility, but LFC implements an efficient spanning-tree broadcast protocol inside the Myrinet network interfaces. The DAS system is more fully described in *http://www.cs.vu.nl/das/*.

Table 1 summarizes our results for the *push* method, writing a stack object, and for the *top* method, reading a stack. For comparison, we also measured the sequential execution of the methods, and their invocation via Manta's standard RMI mechanism. For the sequential version, we compiled variants of the *Stack* classes that do not implement the replication-related interfaces. The time for the sequential *push* ($3.1\,\mu s$) is dominated by the creation of a *StackNode* object. For the RMI version, the *Stack* class implements the *java.rmi.Remote* interface. Invoking both methods on such an object locally (within the same process), adds about $3\,\mu s$ to the pure sequential times. Calling a remote object on a different machine adds about $50\,\mu s$.

The replicated *Stack* has been tested using up to 16 machines. The *top* read-only method can be performed locally, independent of the number of replicas. It completes much faster than via a local RMI, within only $0.5\,\mu s$. Broadcasting the *push* write method to two machines takes slightly longer than a single RMI ($60\,\mu s$), and increases by less than $3\,\mu s$ each time the number of machines is doubled. With a single process, the Panda layer avoids the actual network communication, saving 50% of the broadcast overhead, compared to using two processes. The times shown for the replicated *push* method denote the time from the method invocation until all processes have completed the operation.

In this micro benchmark, the cost of a read operation on a replicated object is comparable to the cost of its sequential counterpart. The write operation takes only slightly longer than a single RMI call. These results are very promising. In the following section, we investigate the impact of our implementation on two application kernels.

Table 1: Completion times of *Stack* operations on a Myrinet cluster (microseconds), comparing sequential method invocation, RMI, and Manta's replication

	cpus	*push*	*top*
sequential		3.1	0.1
RMI, local		6.1	2.7
RMI, remote		55.3	49.2
replicated	1	29.1	0.5
replicated	2	60.0	0.5
replicated	4	62.3	0.5
replicated	8	65.3	0.5
replicated	16	68.0	0.5

4. APPLICATIONS

We evaluated Manta's replication mechanism with two applications. For both, we followed the general approach to first implement a “naive” version that is based on shared-object communication where the shared objects are accessed via RMI. For comparison, we manually optimized the communication behavior of these versions exclusively using RMI as communication mechanism. Finally, we implemented versions of the “naive” codes that replicate their shared objects. For all three versions of an application, we compare performance and source-code complexity.

4.1 The Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. We use a branch-and-bound algorithm, which prunes a large part of the search space by ignoring partial routes that are already longer than the current best solution. The program is parallelized by distributing the search space over the different nodes. Because the algorithm performs pruning, however, the amount of computation needed for each sub-space is not known in advance. The program therefore uses a centralized job queue to balance the load. Each job contains an initial path of a fixed number of cities; a node that executes the job computes the lengths of all possible continuations, pruning paths that are longer than the current best solution.

The TSP program keeps track of the current best solution found so far, which is used to prune part of the search space. Each node needs an up-to-date copy of this solution to prevent it from doing unnecessary work, causing it to frequently check the currently best solution. In contrast, updates to the best solution happen only infrequently.

In our implementation of TSP, the solution is stored in an object of class *Minimum*. We have implemented three different versions of the *Minimum* class, using a remote object, manually optimized remote objects, and a replicated object.

Figure 5 shows the speedups for the three versions with 1 up to 64 nodes. All speedup values are computed relative to the speed of the manually optimized version, running on a single node. The *naive* RMI version implements the *Minimum* class using a remote object, stored on one of the nodes. The other nodes receive a reference to this *Minimum* object. An expensive RMI is needed in order to read the value of the *Minimum* object, resulting in poor performance and no speedups. The overhead of the very frequent read operations actually causes a bottleneck at the node owning the *Minimum* object, causing completion times to increase, rather than to decrease, with the number of nodes. For example with 16 nodes, we counted about $1.5 \cdot 10^8$ incoming RMI requests on the node owning the *Minimum* object.

To prevent this prohibitive overhead, the *optimized* RMI version manually replicates the current minimum value to class variables of the active TSP worker objects. The frequently occuring read operations can now be performed locally, by reading the value from a variable, even avoiding the overhead of method invocation. Whenever one node finds a better solution, it performs an RMI call to a remote *Minimum* object. This object has a vector of references to all TSP worker objects, which also act as *remote* objects. While processing a *set* operation, the *Minimum* object in turn performs a *set* RMI on all TSP worker objects, updating their minimum values. Using this optimization, TSP achieves a speedup of 51.8 on 64 nodes. However, the implementation of the *Minimum* class becomes much more complicated as it needs remote references to all TSP worker objects. Furthermore, the worker objects also have to provide a method that can be invoked remotely which somewhat contradicts the "naive" design.

```
class Minimum implements manta.replication.Root {
        private int minimum = Integer.MAX_VALUE;

        public void set(int minimum) {
                if (minimum < this.minimum) {
                        this.minimum = minimum;
                }
        }

        public int get() {
                return minimum;
        }
}
```

Figure 4: Replicated implementation of the *Minimum* class

The implementation of the *replicated* version of TSP is almost identical to the naive (original) RMI version. The only difference is that the *Minimum* class is marked as being a root object instead of a remote object (see Figure 4). Because the object is replicated on all nodes, all changes are automatically forwarded and each node can locally read the value of the object. Figure 5 shows that, although the replicated implementation is just as simple as the naive RMI implementation, its performance comes close to the manually optimized RMI version, achieving a speedup of 45.2 on 64 nodes. This speedup is slightly inferior to the manually optimized RMI version. The difference originates in the overhead of reading the minimum value. With 64 nodes, for example, we counted the total number of invocations of the *get* operation to be about $8 \cdot 10^8$. As shown in Table 1, invoking a local read operation on a replicated object takes 0.5 microseconds, while reading a local class variable needs less than 0.1 microseconds. The difference of the total completion time is $8 \cdot 10^8/64 \cdot (0.5 - 0.1) \cdot 10^{-6}$ seconds ≈ 5 seconds. In fact, we measured completion times of 32.6 seconds for the manually optimized version and of 37.4 seconds for the replicated version, yielding the speedup values shown in Figure 5.

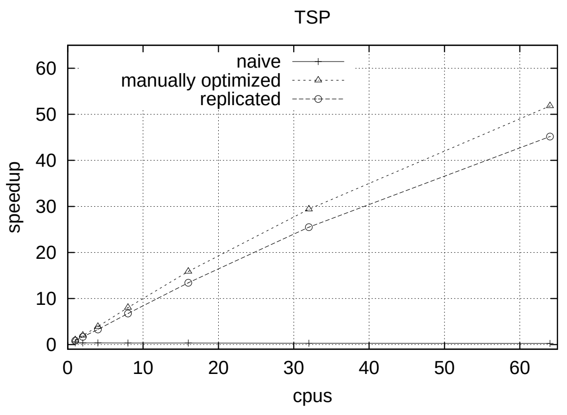

Figure 5: Speedup for the TSP application

4.2 All-pairs Shortest Paths Problem

The All-pairs Shortest Paths (ASP) program finds the shortest path between any pair of nodes in a graph, using a parallel version of Floyd's algorithm. The program uses a distance matrix that is divided row-wise among the available processors. At the beginning of iteration k, all processors need the value of the kth row of the matrix.

We implemented this communication pattern using a replicated object of class *Matrix*. The processor containing the row for the next iteration stores it into this object, allowing other processors to read the row. It is possible for a processor to request a row which has not been produced yet, causing the call to the *Matrix* object to block until it is available. As with the TSP problem, we have implemented three versions of ASP, using a remote object, an optimized remote object, and a replicated object.

The naive RMI version implements the *Matrix* class using a remote object. Each machine has a copy of such an object, used by other machines to retrieve its rows using RMI. Because each machine has to fetch each row for itself, each row has to be sent across the network multiple times, causing high overhead on the machine that owns the row. For instance, if 64 nodes are used, each row is sent 63 times. Figure 7 shows that the naive RMI version performs well up to 8 nodes. On more nodes, the overhead for sending the rows becomes prohibitive, limiting the speedup to 28.5 on 64 machines. Again, all speedup values are computed relative to the speed of the manually optimized version, running on a single node.

To prevent the overhead of sending the rows multiple times, the optimized RMI version uses a *binary tree* to simulate a broadcast of a row. When a new row is generated, it is forwarded to two other machines which store the row locally and each forwards it to two other machines. As soon as the rows are forwarded, the machines are able to receive a new row, allowing the sending of multiple rows to be pipelined. The forwarding continues until all machines have received a copy of the row. Using this simulated broadcast, the optimized RMI version performs much better, achieving a speedup of 58.9 on 64 machines.

```
class Matrix implements manta.replication.Root {

        private int[][] tab;
        private int size;

        public Matrix(int n) {
                tab = new int[n][];
        }

        public synchronized int [] get_row(int i) {
                while (tab[i] == null) {
                        try {
                             wait();
                        } catch (Exception e) {
                             // Handle the exception.
                        }
                }
                return tab[i];
        }

        public synchronized void put_row(int i,
                                         int [] row) {
                tab[i] = row;
                notifyAll();
        }
}
```

Figure 6: Replicated implementation of the *Matrix* class.

The replicated ASP implementation uses a single, replicated *Matrix* object, shown in Figure 6. Whenever a processor writes a row into the *Matrix* object using the *put_row* method, the new row is forwarded to all machines, using the efficient broadcast protocol provided by Panda and LFC. Each processor can then locally read this row using the *get_row* method. The replicated implementation is as simple as the naive version. Figure 7 shows that it performs even better than the manually optimized RMI version, achieving a speedup of 60.9 on 64 machines. This is due to Panda's broadcast which performs better than the RMI-based broadcast tree. In addition, by using Panda's broadcast, parameter objects only have to be serialized once per broadcast, rather than multiple times in the application-level forwarding tree.

As with TSP, the implementation of the *replicated* version of ASP is very similar to the naive implementation. In contrast, the optimized version contains a large amount of extra code to implement the binary tree, making the source code more complex and more than twice as big as the naive version.

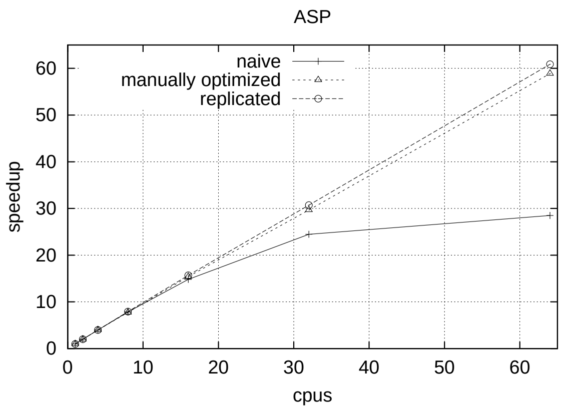

Figure 7: Speedup for the ASP application

5. RELATED WORK

Our approach to object replication in Manta follows the same function-shipping update strategy as in the Orca system [3] and also uses the same underlying communication system (Panda). Still, there are many important differences with Orca. Orca was designed specifically to allow object replication. In particular, its object model is very simple: it supports only methods on *single* objects and it does not even allow references between objects. Hence, Orca is not object-oriented, and its programming style is closer to Distributed Shared Memory [3]. Orca programs read and write one object at a time, much like DSM programs read and write memory locations one at a time. Java and Java RMI support a quite different (object-oriented) programming model, and were not designed with object replication in mind. Implementing replicated objects in Java therefore is much harder. We introduced a clustering concept to allow replication of object graphs (something that cannot even be expressed in Orca, since it lacks references between objects). Also, synchronization in Orca is much more restrictive than in Java and only allows methods to block initially, but not halfway during their invocation. We addressed this problem by imposing a consistent ordering for Java's *wait*, *notify*, and *notifyAll* primitives. Another difference between Java and Orca is that Orca can do the read/write analysis of methods entirely at compile time, as Orca does not support polymorphism. For Java, the analysis has to be done partially during runtime.

An alternative to replication is to use a Distributed Shared Memory (DSM) system. Several DSM systems for Java exist, which provide a shared memory programming model instead of the RMI model, while still executing on a distributed memory system. In these systems, no explicit communication is necessary, all communication is handled by the underlying DSM. Java/DSM [30] and DOSA [11] implement a JVM on top of the TreadMarks DSM [13].

Hyperion [19], Jackal [27], and cJVM [2] are examples of Java systems that *cache* objects. In these systems, a processor can get a temporary copy of an object. These copies are invalidated (deleted) at synchronization points by broadcasting an invalidation message to all copies. In Manta, on the other hand, the replicas of an object are continuously kept coherent, by broadcasting write methods in a totally-ordered way. These broadcast messages already contain the information to refresh the replicas, eliminating the need for further communication. In combination with function shipping, update messages are typically very short, comparable to the size of invalidation messages. Also, a replication scheme can benefit from the availability of an efficient low-level broadcast mechanism (LFC, in our case). The actual performance of the two schemes of course also depends on application-specific communication characteristics.

The VJava [17] system offers caching using a scheme called ObjectViews. With ObjectViews, threads can have different *views* of a shared object. The system can determine at compile time if it is safe to access the object concurrently through two different views. It uses this information to reduce the number of invalidation messages sent.

The Java system described in [14] also supports object caching, and uses a reliable multicast protocol to send invalidation messages. The performance of this system, however, suffers from the inefficiencies of the RMI system (Sun JDK 1.1.5) on which it is based. For example, reading a locally cached copy of an object (i.e., without any communication) costs 900 microseconds (measured on a Sun Ultra 2). In comparison, Manta can update 16 remote copies of an object in 68 microseconds.

The Javanaise system [9] uses clusters of objects in a way similar to Manta, but relies on object caching. Processors can fetch read-only copies of a cluster from a centralized server. Those copies will be invalidated when a processor requests write permission on the cluster, causing considerable overhead with updating large clusters. Manta's replication mechanism is thus much more efficient. In the clustering mechanism of Javanaise, a *cluster* object (corresponding to Manta's *root* object) serves as the entry point to the cluster. Programmers have to annotate its methods as read or write operations, a task automatically performed by the Manta compiler. Finally, Javanaise has no notion of *node* objects and any serializable object can be part of a cluster, burdening a significant part of guaranteeing replica consistency onto the programmer.

There are many other research projects for parallel programming in Java [1, 6, 7, 10, 12, 14, 15, 21, 22, 24, 29, 30]. Most of these systems, however, do not support object replication or caching. Several systems (e.g., [12, 22]) support object migration. The Kan Java-based distributed system [16] supports recovery, object migration, and replication as means for achieving fault tolerance. Manta's focus is on implementation efficiency for parallel applications.

With the Message Passing Interface (MPI) language binding to Java [8], communication is expressed using message passing rather than remote method invocations. Processes send messages (arrays of objects) to each other. Additionally, MPI defines collective operations in which all members of a process group collectively participate; examples are broadcast and related data redistributions, reduction computations (e.g., computing global sums), and barrier synchronization. Object replication roughly corresponds to MPI's broadcast operation. An integration of MPI's other collective operations into Java's object model could complement expressiveness and efficiency of object replication.

6. CONCLUSIONS

In this paper, we presented a new and efficient approach to object replication in Java. We adopted our previous work on the Orca shared object system [3] which combines an update protocol with totally ordered broadcast and function shipping. For integrating Orca's replication mechanism into Java's object model, we introduced a notion of closed groups of objects, called *clusters*, which serve as the unit of replication. Furthermore, we added support for Java's polymorphism as well as for the synchronization mechanism based on *wait* and *notify*, which may cause replicated method invocations to block in the middle of their execution. Our goal was to keep the programming model as close as possible to standard RMI. To achieve this, objects are declared to become replicated by implementing one of two new special interfaces.

Our implementation partially is inside the Manta compiler, and partially in the runtime system. The compiler performs consistency checks, generates code for replicated method invocation, and analyses the methods of cluster objects to distinguish between read and write operations. The runtime system establishes and updates object clusters on all nodes of a parallel application. It also coordinates the execution of method invocations to enforce replica consistency. Read operations on replicated objects can be performed locally (without communication) and take about 0.5 microseconds on our platform. Write operations for updating 16 replicas take 68 microseconds, only slightly longer than a single RMI call.

We have shown that our approach provides efficient object replication for Java with a programming model close to standard RMI. Object clusters allow complex data structures to be replicated without sacrificing runtime performance. We evaluated our system with two application kernels and showed that Manta's object replication model actually allows implementation of straight-forward, shared-object applications, while yielding performance close to manually optimized versions based on individual RMI calls.

We are currently evaluating our replication model. Although the model is restrictive, it was strong enough to express the two application kernels described in Section 4. Furthermore, we are currently evaluating our system with other, more irregular applications, as well as on top of our wide-area Java platform [26].

Acknowledgments

This work is supported in part by a USF grant from the Vrije Universiteit. The DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, and Ceriel Jacobs for their contributions to this research. We thank Kees Verstoep and John Romein for keeping the DAS in good shape.

7. REFERENCES

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proc. of the 1999 Int. Conf. on Parallel Processing*, Aizu, Japan, Sept. 1999.

[3] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[4] R. Bhoedjang, T. Rühl, and H. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[5] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[6] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Santa Barbara, CA, Feb. 1998.

[7] S. Brydon, P. Kmiec, M. Neary, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. In *ACM 1999 Java Grande Conference*, pages 171–180, San Francisco, CA, June 1999.

[8] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998.

[9] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *Proc. Middleware'98*, The Lake District, England, Sept. 1998.

[10] T. Haupt, E. Akarsu, G. Fox, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, and C.-H. Youn. The Gateway System: Uniform Web Based Access to Remote Resources. In *ACM 1999 Java Grande Conference*, pages 1–7, San Francisco, CA, June 1999.

[11] Y. Hu, W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel. Runtime Support for Distributed Sharing in Strongly Typed Languages. Technical report, Rice University, 1999. Online at http://www.cs.rice.edu/˜willy/TreadMarks/papers.html.

[12] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, CA, June 1999.

[13] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 Usenix Conference*, pages 115–131, San Francisco, CA, Jan. 1994.

[14] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient Implementations of Java RMI. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, NM, 1998.

[15] P. Launay and J.-L. Pazat. The Do! project: Distributed Programming Using Java. In *First UK Workshop Java for High Performance Network Computing*, Southampton, Sept. 1998.

[16] S. Y. Lee. *Supporting Guarded and Nested Atomic Actions in Distributed Objects*. Master's thesis, University of California at Santa Barbara, July 1998.

[17] I. Lipkind, I. Pechtchanski, , and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *Proc. of the 1999 Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 447–460, October 1999.

[18] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[19] M. W. Macbeth, K. A. McGuigan, and P. J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proc. CASCON'98*, pages 40–54, Missisauga, ON, 1998. Published by IBM Canada and the National Research Council of Canada.

[20] K. Mazouni, B. Garbinato, and R. Guerraoui. Building Reliable Client-Server Software Using Actively Replicated Objects. In *Proc. International Conference on Technology of Object Oriented Languages and Systems (TOOLS)*, Versailles, France, Mar. 1995. Prentice Hall.

[21] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 153–159, San Francisco, CA, June 1999.

[22] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997. Online at http://wwwipd.ira.uka.de/JavaParty/.

[23] T. Rühl and H. E. Bal. Synchronizing operations on multiple objects. In *Proceedings of the 2nd Workshop on Runtime Systems for Parallel Programming*, Orlando, FL, Mar. 1998.

[24] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5/6), 1999.

[25] B. Topol, M. Ahamad, and J. T. Stasko. Robust State Sharing for Wide Area Distributed Applications. In *18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 554–561, Amsterdam, The Netherlands, May 1998.

[26] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, CA, June 1999.

[27] R. Veldema, R. A. F. Bhoedjang, and H. Bal. Distributed Shared Memory Management for Java. Technical report, Vrije Universiteit Amsterdam, Nov. 1999.

[28] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

[29] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. In *ACM 1998 workshop on Java for High-performance network computing*, Feb. 1998. Online at http://www.cs.ucsb.edu/conferences/java98/.

[30] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, Nov. 1997.