Parallel Application Experience with Replicated Method Invocation[*]

Jason Maassen, Thilo Kielmann, Henri E. Bal

Division 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 and evaluate 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 reliable, 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.

We evaluate system performance both with micro benchmarks and with a set of five parallel applications. For the applications, we also evaluate ease of programming, compared to RMI implementations. We present performance results for a Myrinet-based workstation cluster as well as for a wide-area distributed system consisting of four such clusters. The micro benchmarks show that updating a replicated object on 64 machines only takes about three times the RMI latency in Manta. Applications using Manta's object replication mechanism perform at least as fast as manually optimized versions based on RMI, while keeping the application code as simple as with naive versions that use shared objects without taking locality into account. Using a replication mechanism in Manta's runtime system enables several unmodified applications to run efficiently even on the wide-area system.

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 [12, 18, 24, 32, 35], 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) [36]. Some systems temporarily cache objects rather than trying to keep multiple copies of an object consistent [12, 18, 24, 35]. Some proposals have a programming model that is quite different from the object invocation model of RMI [32]. Also, performance results are often lacking or disappointing. The probable reason for these problems is

[*] This is a preprint of an article accepted for publication in "Concurrency and Computation: Practice and Experience", ©2001 John Wiley & Sons Ltd

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 a Remote Method Invocation. 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 clouds, that are replicated as a whole. A cloud 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 cloud; such read-only methods are executed locally, without any communication. Methods that modify any data in the cloud are broadcast and applied to all replicas. A single broadcast message is used to update the entire cloud, 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 [23, 33]. Updating a simple object replicated on 64 Myrinet-connected machines takes 120 microseconds, only about three times the RMI latency in Manta. We have also implemented five parallel Java applications that use replicated objects, which we use to illustrate efficiency and ease of programming of replicated objects in Manta.

Object replication is even more beneficial in wide-area distributed systems with high communication latencies. Parallel computing on a wide-area system is attractive for very compute-intensive applications, since it allows the processing power of multiple computing resources to be combined for a single program. Several well-known examples of such distributed supercomputing applications exist (e.g., SETI@home, RSA-155). We will show how object replication in Java can ease the implementation of efficient parallel applications for distributed supercomputers. To this purpose, we have implemented our object replication scheme on a wide-area system consisting of four cluster computers. Performance measurements show that, despite the high wide-area latencies, three out of the five applications obtain good speedups on this wide-area system, without any changes to their source code. Without object replication, only one application obtains reasonable wide-area performance.

This paper is based on our earlier work as published in [22]. Here, we provide an in-depth evaluation of Manta's replication mechanism, for both a single cluster and a wide-area system. The outline of the rest of the paper is as follows. In Section 2, we propose a new model, similar to RMI, that allows closed groups of objects to be replicated. In Section 3, we describe a compiler-based implementation of this model as part of the Manta system and analyze the performance of this implementation on a Myrinet cluster using a micro benchmark. In Section 4, we describe five applications and show the performance benefits of object replication in Java. Section 5 shows that object replication also allows several (unmodified) applications to run efficiently on a wide-area distributed supercomputer. In Section 6, we look at related work. Finally, in Section 7, we present our conclusions.

2 Replicated Method Invocation

The primary goal of our object replication mechanism is to provide a programming model close 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 [3], however, show that for object-based languages an *update* protocol often is more efficient, especially if it is implemented with *function shipping*. 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 replicas in a consistent way, methods are sent using *reliable, 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 6). 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, where the entire list would be a single object).

Our solution to this problem is to take an intermediate approach and replicate only closed groups of objects, which we call *clouds*. A cloud 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 cloud is implemented using a single broadcast message, independent of the number of objects in the cloud. The entry point of a cloud is called its *root*, and it is the only object that can be accessed by objects outside the cloud. In addition, a cloud can have other objects reachable from the root, called the *node* objects; these node objects, however, cannot be referenced directly from outside the cloud. As a consequence, only methods of the root object can be directly invoked in order to manipulate (read or modify) the cloud. All other method invocations inside the cloud 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 grouped objects (the clouds), we have defined a new and simple programming interface in Manta to express this grouping mechanism. We discuss this interface below.

2.1 Programming interface and example

Object clouds are defined by the application programmer, using two so-called "special" interfaces to mark cloud objects. Although this approach is not completely transparent, it 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 cloud 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.

To illustrate the use of the two special interfaces, Figure 1 shows a simple example of an object cloud, a replicated stack, implemented as a linear list. Whenever a new *Stack* object is created, a new cloud is created using the *Stack* object as its root. By calling the *push* method, *StackNode* objects will be added to this cloud. 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 cloud.

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, clouds are thus passed by reference via RMI, just like ordinary remote objects. Also, method invocations on replicated clouds 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.

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

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

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

Figure 1: A replicated stack

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 and runtime system try to enforce them and produce error messages whenever they detect violations.

No remote references. As a result of our decision to replicate only closed groups (clouds) of objects, cloud objects cannot contain references to remote objects. Also, the methods defined for (the root of) a cloud 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 cloud rather than being replicated themselves. In such a case, the function shipping approach would cause the *nested invocation problem* [25], illustrated in Figure 2. On the left, A's *meth* method calls *incr* on the remote object B. When A gets replicated (shown on the right), 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 cloud (e.g., the reference from A to B is not allowed).

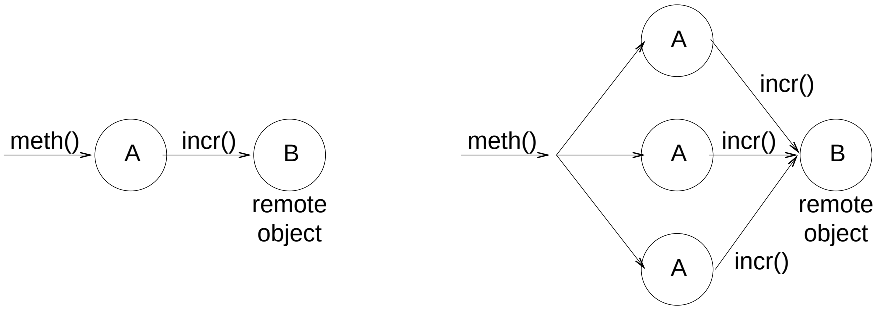


Figure 2: The nested method invocation problem

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 clouds from each other. For the same reason, objects in a cloud may only contain a reference to the root object of the same cloud. Inter-cloud references are not allowed. As all objects in a cloud have to implement either the root or the node interface, and as remote references are not allowed inside clouds, 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 cloud. This would break the call-by-value semantics which enforce node objects to be private copies of their cloud.

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, or start a new thread inside the replicated cloud.

To summarize, our model deliberately disallows references between different clouds or between clouds 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 cloud is a closed group of objects that can be replicated efficiently, as discussed in the next section.

However, by increasing the complexity of the runtime system, it would be possible to lift some of these restrictions. For example, references between clouds and remote objects can be allowed if the system can detect that a remote method is invoked from inside a cloud. Only one of the remote invocations would then have to be executed, provided that the result is passed to all replicas of the cloud. A similar solution could be used to implement references between clouds. Although we found that our current replication model is not too restrictive for the applications we investigated, we are planning to lift some of the restrictions in future versions of Manta.

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 [23]. 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 clouds 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. To abort the current invocation, an exception is thrown from the method wrapper. This exception is caught in the first wrapper invoked on the root object, which can then restart the method in write mode. By using the exception mechanism it is possible to restart nested invocations. No extra nesting information has to be kept, other that the information already present for normal exception handling.

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 and reliable, so all Manta nodes are guaranteed to receive all broadcasts and to receive them in the same order.

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. By performing the write operations one after the other, the broadcast order is maintained, and all replicas are kept consistent. This mechanism effectively converts concurrent method invocations into sequential method invocations. Even if multiple threads on the same node concurrently invoke a write method on a replica, the actual execution of the methods will be in some sequential order. 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 [23].

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 the same and can be discarded. The only exception to this is when a machine-specific and fatal exception is thrown (e.g. an OutOfMemoryException). The failing node will broadcast this exception to all other nodes, which will then terminate their program and print the error message contained in the exception.

3.3 Cloud management

Whenever a new root object is created, a new cloud is implicitly created along with it. On the invoking process, the root object is created, and a unique identifier (i.e. the combination of object pointer and Manta node number) is assigned to it. In turn, the new cloud is broadcast to all nodes of the parallel application, using Panda's totally ordered broadcast mechanism. This ensures that clouds are always created on all nodes before any write operation attempts to modify them.

Although the replicated clouds are immediately established on all nodes, the application itself views them as being replicated on demand. Only the process on which the cloud was created gets a reference to the new cloud. 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 cloud only on those nodes that actually have a reference to it, and remove the replica when the last reference on a node is removed. This could avoid both memory and processing overhead (due to write updates) caused by objects that are not used on a node. As a drawback, elaborate group management would have to be implemented. The current implementation assumes that the number of replicated objects will be small, and simply replicates all clouds on all nodes. Our previous experience with the Orca shared object system indicates that this approach yields adequate performance for parallel programming [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 Manta node 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.

```
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

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 [30]. 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 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 clouds 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 a simple class with minimalistic read and write methods 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 2.2.14. The nodes are connected via Myrinet [7]. Manta's runtime system has access to the network in user space via the Panda communication substrate [3] which uses the LFC [6] Myrinet control program. Myrinet lacks a hardware broadcast facility, but LFC implements an efficient spanning-tree broadcast protocol inside the Myrinet network interfaces. The system is more fully described in *http://www.cs.vu.nl/das/*.

Table 1 summarizes our results. It presents timings for both the read and write methods. For comparison, we also measured the sequential execution of both methods, and their invocation via Manta's standard RMI mechanism. For the sequential version, we compiled variants of the class that do not implement the replication-related interfaces. All numbers shown in the table are median values over several runs.

Both the sequential and the RMI calls are synchronous; their completion times reported in the table are measured directly at the invoking process. Method invocations on replicated objects, however, are not necessarily synchronized across all replicas. Read operations are performed on the local copy of the cloud; write operations are simply shipped to all remote replicas. A write method returns as soon as the local replica has been updated while all other copies are updated asynchronously. This behavior allows to pipeline several write method invocations, but it also requires elaborate measurement procedures [11, 27]. The completion times for write methods on replicated objects which we report in the table measure the time spent from invoking the method until *all* replicas have been updated.

To measure the time from invoking a write method until a certain replica has been updated, we perform the following test for each replica. The replica reads the local time t_s and performs an RMI call on a special object on the invoking process. The method called on this object then invokes the replicated write method, and returns. On the node with the receiving replica, the write method stores the local time t_e in a global variable. After the initiating RMI has returned, the receiving node takes the time difference $t_e - t_s$ and subtracts 1/2 of the time for an RMI call. The result is the update time for this replica. The completion time for all replicas is the maximum of all values measured this way, and of the completion time at the invoking process, which also contains the time for returning the result.

The sequential method invocations both take about $0.1\mu s$. (The read method is empty while the write method performs a single variable assignment.) The completion times for an RMI are much higher, even when the server object is at the same process (RMI, local). With a local object, the RMIs take 13 and 19 μs, respectively. RMI calls to objects on a remote process take 41 and 45 μs, respectively.

With replicated objects, read methods are performed locally and take only $0.35\mu s$, independent of the number of replicas. With a single replica, the write method takes $30\mu s$, $10\mu s$ more than a local RMI. The same overhead is necessary for updating two replicas compared to a remote RMI. The completion times slowly rise with the number of replicas. For 64 nodes, updating all replicas takes $120\mu s$, less than the time for three RMIs to remote nodes.

Because replicas on remote nodes are updated asynchronously, the invoking write method may return before the remote updates are complete. This allows the invoking process to continue its operation and also to pipeline several write method updates. It is therefore interesting to also measure the inter-operation gap, which denotes the rate at which a process can invoke write operations, depending on the number of replicas. We measured this time by invoking the write method 100 times in a row, and taking the average time per method invocation.

Comparing this gap value with the corresponding RMI completion time indicates the overhead of the replication mechanism faced by applications that pipeline their write methods. With a single replica, write operations can be performed every $24\mu s$, 27% more than invoking an RMI on a local object. With two replicas, the overhead (compared to a remote RMI) is just 15%. As more replicas have to be updated, the overhead grows up to 35% in the case of 64 nodes, which is almost negligible compared to separately invoking methods on 64 remote objects. This increase in overhead is caused by the implementation of the multicast in the underlying communication system [6]. A spanning-tree protocol is used, which requires more messages when multicasting to a larger group. Therefore, the average time required for a single multicast increases with the group size.

In our micro benchmarks, the cost of a read operation on a replicated object is comparable to the cost of its sequential counterpart. Invoking a write operation on 64 machines takes less that 3 RMI calls, a very promising result. In the following section, we investigate the impact of our implementation on five application programs.

4 Applications

We evaluated Manta's replication mechanism with five applications. For each application we followed the general approach to first implement a "naive" version based on shared-object communication where the shared objects are accessed via RMI, but without taking locality into account. For comparison, we manually optimized the communication behavior of these versions using RMI as the only 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. All speedup values are computed relative to the speed of the version

Table 1: Timings of read and write operations and inter-operation gap on a Myrinet cluster (in microseconds), comparing sequential method invocation, RMI, and Manta's replication.

		completion time		gap
	processors	*write*	*read*	*write*
sequential		0.14	0.08	
RMI, local		19.10	13.00	
RMI, remote		44.80	41.00	
replicated	1	30.42	0.35	24.20
replicated	2	54.68	0.35	51.30
replicated	4	61.40	0.35	56.50
replicated	8	71.05	0.35	54.00
replicated	16	78.85	0.35	56.00
replicated	32	111.11	0.35	58.30
replicated	64	119.51	0.35	60.30

running the fastest on a single node.

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 (only when a better solution is found).

In our implementation of TSP, the current best 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 to 64 nodes. The search space consists of 17 cities. For TSP, the fastest program on a single node is the manually optimized version, running for 1613 seconds. All speedup values are computed relative to this time. 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 to read the value of the *Minimum* object, resulting in poor performance and no speedups. The overhead of the very frequent read operations causes a bottleneck at the node owning the *Minimum* object, causing completion times to increase with the number of nodes. For example with 64 nodes, we counted about $8 \cdot 10^8$ 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 on the active TSP worker objects. The read operations can now be performed locally, by reading the value from a variable, even avoiding the overhead of method invocation. Whenever a 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 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 46.8 on 64 nodes. The difference between the two versions 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 0.35 microseconds, while reading a local class variable needs less than 0.1 microseconds. The difference of the total completion time is $8 \cdot 10^8 / 64 \cdot (0.35 - 0.1) \cdot 10^{-6}\ \mathrm{seconds} \approx 3\ \mathrm{seconds}$. In fact, we measured completion times of 31.4 seconds for the manually optimized version and of 34.3 seconds for the replicated version, yielding the speedup values shown in Figure 5.

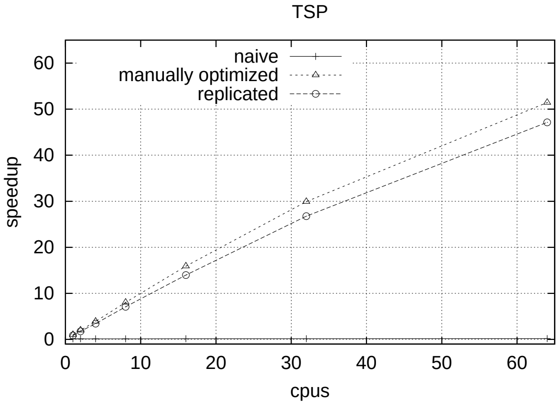


Figure 5: Speedup for the TSP application

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. The processor containing this row must make it available to the other processors by storing it in a remote object or broadcasting it.

In the naive RMI version we have implemented this communication pattern by using a remote object of class *Broadcast*. Each machine has a copy of such an object. 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 *Broadcast* object to block until it is available. 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 the results for a 2000x2000 distance matrix. All speedup values are computed relative the replicated version on one node, which runs for 1072 seconds.

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 27.0 on 64 machines. 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 59.0 on 64 machines.

The replicated ASP implementation uses a single, replicated *Broadcast* object, shown in Figure 6. Whenever a processor writes a row into the *Broadcast* object using the *send* 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 *receive* method. The replicated implementation is as simple as the naive version. Figure 7 shows that it performs even better than

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

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

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

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

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

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

the manually optimized RMI version, achieving a speedup of 61.2 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.

4.3 QR Factorization

QR is a parallel implementation of QR factorization. In each iteration, one column, the *Householder vector* H, is broadcast to all processors, which update their columns using H. The current upper row and H are then deleted from the data set so that the size of H decreases by 1 in each iteration. The vector with maximum norm becomes the Householder vector for the next iteration. To determine which processor contains this vector, an operation similar to a reduce-to-all collective operation (as defined in the MPI standard [10]) is used.

In the naive RMI version, a processor stores the Householder vector in a remote object, similar to the *Broadcast* object of ASP, allowing other processors to read it. The reduce-to-all operation is also implemented by a single remote object. An example of this object can be seen in Figure 8. Each processor submits a value to this object using a remote method, *reduce*. This remote method blocks until all values are submitted, after which the result of the reduction is returned to all

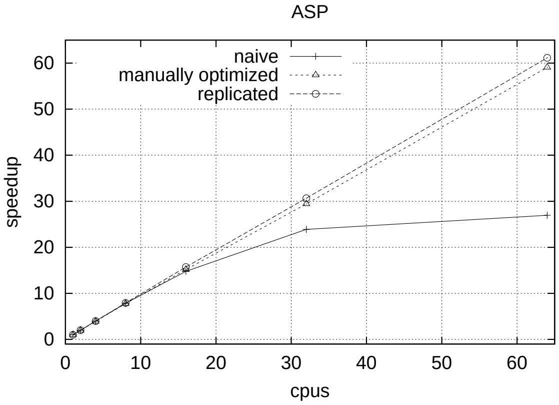


Figure 7: Speedup for the ASP application

processors.

Figure 9 shows the results for a 2000x2000 matrix. All speedup values are computed relative the naive version on one node, which runs for 2628 seconds.

The implementation of both broadcast and reduce-to-all have an impact on the performance of the naive QR, producing a poor speedup of 27.6 on 64 processors.

This speedup can be improved by replacing both the broadcast and reduce object. In the optimized RMI version, as with ASP, the broadcast object is replaced by a *binary tree* broadcast, while the replicated RMI version uses a replicated object. Both versions replace the reduce object by a binary tree. Each machine contains a leaf-node of this binary tree. A value is submitted to this leaf node and sent towards the root of the tree. A reduce between two values is done at every intermediate level. This causes a single value to arrive in the root of the tree, which can then be sent back down to the processors.

Replacing the broadcast and reduce objects improves the speedup of QR on 64 processors to 41.8 in the optimized case, and 44.3 in the replicated case. As with ASP, this difference is caused by the low-level broadcast mechanism used by the replication system.

4.4 The Arc Consistency Problem

The Arc Consistency program (ACP) can be used as a first step in solving Constraint Satisfaction Problems. The program takes as input a set of n variables in domain m and a set of binary constraints defined on some pairs of variables, that restrict the values these variables can take. The program eliminates impossible values from the domains by repeatedly applying the constraints, until no further restrictions are possible.

The program is parallelized by dividing the variables statically among all processors. The solution is stored in a shared n by m matrix of booleans. Each boolean in this matrix describes a single (variable, value) pair. By setting a boolean to *false* the program can restrict the values a variable can take. Every processor repeatedly gets a copy of the shared matrix and restricts the value of its variables as much as possible. The shared matrix is then updated with the restricted values. Because restricting a variable can have an effect on other variables, other processors must

```
remote class Reduce {

        double maximum_norm;
        int processors, calls, returns;

        Reduce(int p) {
                processors   = p;
                calls        = 0;
                returns      = 0;
                maximum_norm = 0.0;
        }

        public synchronized double reduce(double norm) {

                double result;

                maximum_norm = max(norm, maximum_norm);

                calls++;

                while (calls < processors) {
                        // wait for the last call.
                        wait();
                } else {
                        // last call wakes up the rest.
                        notifyAll();
                }

                result = maximum_norm;

                if (returns++ == processors) {
                        // last return resets the Reduce object.
                        calls         = 0;
                        returns       = 0;
                        maximum_norm  = 0.0;
                }

                return result;
       }
}
```

Figure 8: Implementation of the *Reduce* class.

be notified of these updates. This is done by using a shared *Work* object. Whenever a processor updates a variable in the shared matrix, all variables affected by it (due to a constraint) are marked as *dirty* in the work object. The processor that owns the dirty variable must then check if the values of the variable can be restricted further. All three versions of the program use a remote object to implement the Work object, but each implements the shared boolean matrix differently.

Figure 10 shows the results for 2000 variables, each with a 150 possible values. All speedup values are computed relative the naive version on one node, which runs for 533 seconds.

The naive RMI implementation of ACP uses a remote object to store the shared boolean matrix. Because each processor makes a copy of the shared object at the beginning of an iteration and sends all the updates to the matrix in a single RMI, the amount of communication is limited. This results in a reasonable speedup of 41.7 on 64 processors.

In the optimized RMI implementation each processor contains a private copy of the boolean matrix. A binary tree broadcast, similar to the one introduced in ASP, is used to forward updates

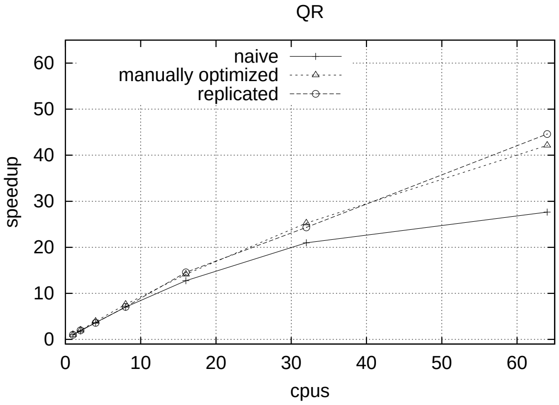


Figure 9: Speedup for the QR application

to the other processors. An extra thread is needed on every processor to apply these updates to the local copy of the boolean matrix. Because the local copy of the matrix is now always up-to-date, it saves the time needed in the naive RMI version to make a copy. This improves the speedup on 64 processors to 53.2.

The replicated RMI version uses a replicated object to store the shared boolean matrix. This object is basically identical to the remote object used in the naive RMI version, except that it is replicated. This allows the object to use the efficient low-level broadcast offered by the replication system, improving the speedup even further to 54.7 on 64 processors.

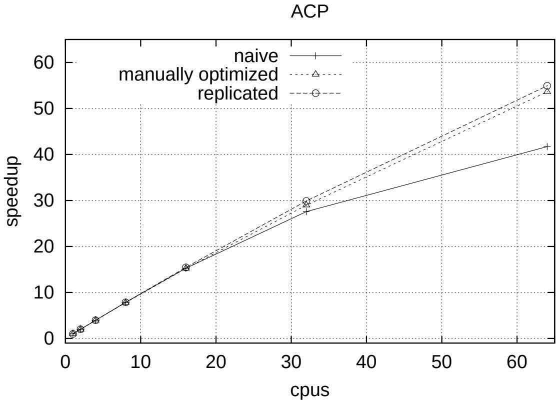


Figure 10: Speedup for the ACP application

4.5 Linear Equation Solver

Linear equation solver (LEQ) is an iterative solver for linear systems of the form $Ax = b$. Each iteration refines a candidate solution vector x_i into a better solution x_{i+1}. This is repeated until the difference between x_{i+1} and x_i becomes smaller than a specified bound.

The program is parallelized by partitioning a dense matrix containing the equation coefficients over the processors. In each iteration, each processor produces a part of the vector x_{i+1}, but needs all of vector x_i as its input. Therefore, all processors exchange their partial solution vectors at the end of each iteration. This is similar to a gather-to-all collective operation as defined in the MPI standard [10].

Besides exchanging their vectors, the processor must also decide if another iteration is necessary. To do this, each processor calculates the difference between their fragment of x_{i+1} and x_i. A reduce-to-all collective operation [10] is used to process these differences and decide if the program should stop.

Figure 11 shows the results for a 1000x1000 matrix. All speedup values are computed relative the replicated version on one node, which runs for 1944 seconds.

In the naive RMI version, both the gather-to-all and reduce-to-all operations are implemented by a single remote object, *Gather*. After an iteration, each processor sends its part of vector x_{i+1} and the value to be reduced to this object and waits for the result of the reduce-to-all. The *Gather* object assembles the entire x_{i+1} vector and reduces all the values to a single result. If required, all processors retrieve a copy of the x_{i+1} vector to use in the next iteration. This communication pattern causes the processors to synchronize at each iteration, which limits the speedup to 9.9 on 64 processors.

In the optimized RMI version the vector fragments and values to be reduced are broadcast using a binary tree, similar to the one introduced in ASP. Each processor can then locally assemble the vector and reduce the values. Unlike the previous programs, in which one processor was broadcasting, in LEQ all processors are required to broadcast data. This requires a large number of RMIs to complete the communication, causing more overhead than in the previous programs. For example, on 64 processors, 4032 RMIs are needed per iteration, while ASP only needs 63 RMIs per iteration. Due to this overhead the speedup of the optimized RMI version is only 12.9 on 64 processors.

As with the other applications the replicated RMI version is similar to the naive RMI code, but uses a replicated object. The replicated *Gather* object can profit from the efficient low-level broadcast used by the replication system. Instead of 4032 RMIs, only 64 broadcast messages are required to complete the communication. This results in a much better speedup of 42.1 on 64 processors.

4.6 Discussion

In the previous sections we have shown five applications, each implemented in three different ways. Although the naive version of an application usually has a poor performance compared to the other versions, it is also the simplest implementation. Shared data are identified by the programmer and encapsulated by a remote object without taking locality, RMI overhead, or network topology into account. When any of these issues are taken into account, as is done in the optimized versions, the code size of the implementation can increase significantly. Table 2 shows the code size of the applications relative to the naive version. These numbers are produced by stripping program source of comments and whitespace, and then counting the number of bytes required for the entire program. The average number is calculated by adding the sizes of all naive implementations, all optimized implementations, and all replicated implementations, and comparing those to each other.

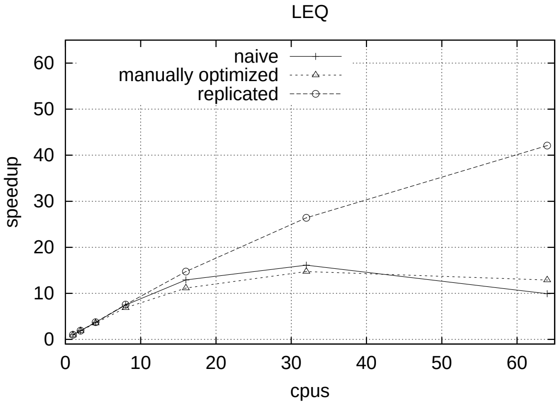


Figure 11: Speedup for the LEQ application

The table shows that the optimized versions of an application are always bigger then the naive version. The increase in size varies from 2 % with TSP to 104 % with LEQ. On average the optimized implementation is 23 % bigger that the naive implementation. The table also shows the advantage of our replication system. Although the replicated versions of the application usually have the best performance, very little code has to be added. Of the five applications, three show a minor increase in code size of 2 to 9 %, while the other two applications show no increase at all. On average, the implementation using replication is only 2 % bigger than the naive implementation.

Table 2: Code sizes of the applications relative to naive version, in %.

	naive	*optimized*	*replicated*
TSP	100	102	100
ASP	100	177	109
QR	100	127	109
ACP	100	115	100
LEQ	100	204	102
Average	100	123	102

While implementing the replicated versions we found that replication is used for three different purposes: sharing data, broadcasting, and collective communication. Table 3 shows an overview of the communication patterns of the different applications. TSP and ACP use replication to share data that are read very frequently, but written infrequently and at irregular intervals. In these applications the actual overhead of a read method can decrease performance. Although this overhead is small, the large number of invocations can cause significant overhead. As a result, the optimized version of TSP performs better than the replicated version. In ACP, every processor makes a copy of the shared matrix before each iteration, preventing this overhead.

ASP and QR use replication to implement a broadcast. By writing data into a replicated object, it is broadcast to all processors, which can then read the data locally. By using replication,

Table 3: Communication patterns of the applications.

	pattern
TSP	shared data, updated asynchronously
ASP	broadcast in each iteration
QR	broadcast and reduce-to-all in each iteration
ACP	shared data, updated asynchronously
LEQ	gather-to-all and reduce-to-all in each iteration

the programs can take advantage of the efficient low-level broadcast provided by the replication system. Also, as shown in Table 2, the implementation of a broadcast using replication is much simpler than implementing a binary tree broadcast using only RMI. Although the implementation of a broadcast is quite simple, care must be taken to ensure that the *receive* method (see Figure 6) can be executed locally. If the receive method contains a single assignment on a class variable (e.g., to count the number of receives) it is labeled as a *write operation* by the compiler (see Section 3.1) and can no longer be executed locally. Each receive method invocation would then be broadcast to all processors, reducing the efficiency of the implementation.

In LEQ, replication is used to implement collective communication. A replicated object is used to simultaneously perform an gather-to-all and reduce-to-all operation. Each processor writes the data to be gathered and reduced into a replicated object, then locally tries to read the result of the reduce-to-all operation. This read blocks until all processors have written their data. After all data are written, the result of the reduce-to-all is returned and the result of the gather-to-all can be read locally. As with broadcast, implementing the collective communication using replication resulted in smaller and more efficient code.

5 Object Replication in Wide-area Systems

So far, we have evaluated Manta's object replication mechanism using a tightly-coupled system, namely a Myrinet-based workstation cluster. Due to the highly efficient implementation, our replication-based applications run on such a system about as fast as their manually optimized counterparts. Simultaneously, the application source code remains as simple as with the naively implemented version where shared objects have a single instance, accessed via remote method invocation (RMI).

In this section, we evaluate the performance of replicated method invocation on a wide-area system in which multiple workstation clusters are coupled in order to accumulate their computing capacity. The challenge with such systems is that the wide-area links between clusters have orders of magnitude higher latency and lower bandwidth, compared to the intra-cluster connections. Manta's replication mechanism enforces strong consistency amongst object replica. Our goal is to verify whether this mechanism can be used efficiently on wide-area, multi-cluster systems. Therefore, we first describe our wide-area computing platform. Then, we present the micro-benchmark results, in analogy to Section 3.5. Finally, we present and discuss the performance of our five applications from Section 4 on the wide-area system.

5.1 The Wide-area DAS System

The DAS consists of four clusters, located at different universities in The Netherlands. The clusters are connected by dedicated wide area links. Fig. 12 shows the structure of the overall system. Panda gives access to LFC for communication within a cluster, and to TCP between clusters. In addition, we extended Panda's broadcast to efficiently exploit the given cluster topology, similar to our MagPIe library [17]. Here, each broadcast message is sent to each cluster exactly once, and is locally distributed using LFC's spanning tree.

One of the DAS clusters has 128 processors, and has been set up to allow easy experimentation with different WAN latencies and bandwidths, by adding delay loops to the networking subsystem [17]. This wide-area emulation is part of Panda and thus is transparent to software layers on top of it, like Manta. We use this emulation system for the performance experiments reported in this paper. We emulate 4 clusters, and between each pair of them a wide-area latency of 10 ms, and a wide-area bandwidth of 1 MB/sec. In comparison, the (one-way) latency over Myrinet is about 23 μs and the (Manta-level) throughput is about 50 MB/sec, so there is almost two orders of magnitude difference between the local and wide-area network of the DAS. We emphasize that all our results have been measured on a real parallel machine, using the same software on the compute nodes as in the real wide-area system. Only the wide-area links are simulated over some of the Myrinet links. To achieve this, the software on each cluster's gateway (an additional, dedicated Panda node) uses a different runtime option.

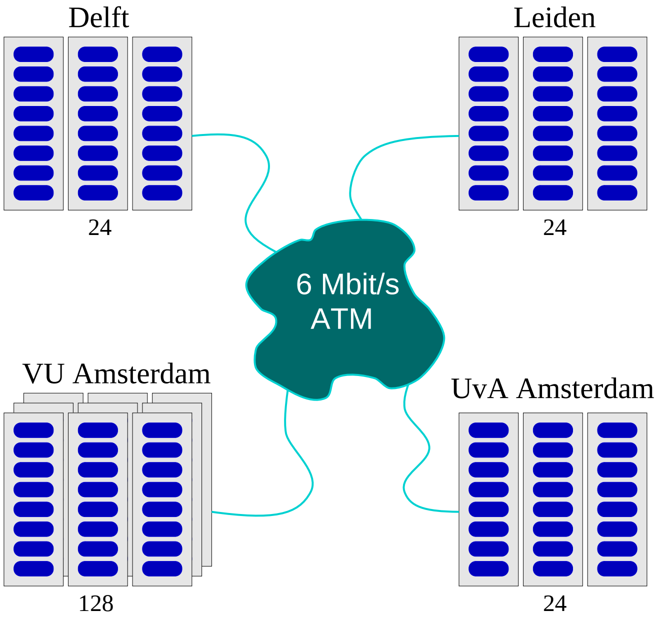


Figure 12: The wide-area DAS system.

5.2 Micro Benchmarks

Table 4 compares the timings of read and write operations on replicated objects with the standard RMI mechanism. Unlike Table 1, local accesses are left out as their results are the same. We focus on comparing Manta's replication mechanism using 4 clusters of 16 processors each with RMI calls where client and server are located in different clusters.

Panda implements the total ordering of broadcasts with a centralized sequencer node, located at one of the network interfaces. In order to broadcast a method invocation, a node first performs a remote procedure call to the sequencer from which it receives a sequence number which is in turn attached to the broadcast message. As can be seen from Table 1, this mechanism works fine within a Myrinet cluster. With multiple clusters, the broadcast performance strongly depends on whether or not the sequencer is in the same cluster as the broadcasting node. In Table 4, we thus present timings for both cases.

As a standard RMI call is synchronous, both read and write operations take 20.2 milliseconds, dominated by the wide-area round trip latency. For the read operation, replication now becomes extremely efficient as all read operations are performed on the local replica, roughly at the cost of a local method invocation. When the sequencer is located in the same cluster as the broadcasting node, write operations on 64 processors distributed across 4 clusters complete in about 10.4 milliseconds, corresponding to the wide-area latency. (There is no need for sending reply messages back across the wide-area network as method results are taken from the local node.) When the sequencer is located in a remote cluster, the same write operation completes after 30.5 milliseconds, composed of the round trip time to the sequencer and the subsequent broadcasting to the remote clusters. The inter-operation gap shows a similar difference. With the sequencer in the local cluster, the gap between two write operations is 1 millisecond, dominated by wide-area bandwidth slowing down message sending. With the sequencer in a remote cluster, the gap is 21.5 milliseconds, because for each broadcast a new sequence number has to be fetched synchronously from the sequencer node.

Obviously, applications can be slowed down significantly if they have to execute many write operations in clusters not hosting the sequencer node. In earlier work with wide-area applications written in the Orca language [4], we augmented Panda with a facility to migrate the sequencer node between clusters. Whenever applications invoke several write operations in the same cluster in a short period of time, it is very beneficial to first migrate the sequencer to this cluster and then perform the write operations. In the Orca work, we triggered sequencer migration on the application level. However, most applications show at least some "update locality". So, for Manta, we augmented the runtime system to always migrate the sequencer before executing write operations. While this is the default behavior, a runtime option can resort to a static sequencer that remains on one node throughout the whole application run. The application results presented below have been obtained with sequencer migration turned on.

Table 4: Timings of read and write operations using multiple Myrinet clusters (microseconds), comparing RMI and Manta's replication.

					completion time		gap
	clusters	×	processors	sequencer	*write*	*read*	*write*
RMI, remote	2	×	1		20275.20	20272.70	
replicated	4	×	16	local	10394.00	0.35	1023.70
replicated	4	×	16	remote	30519.00	0.35	21582.70

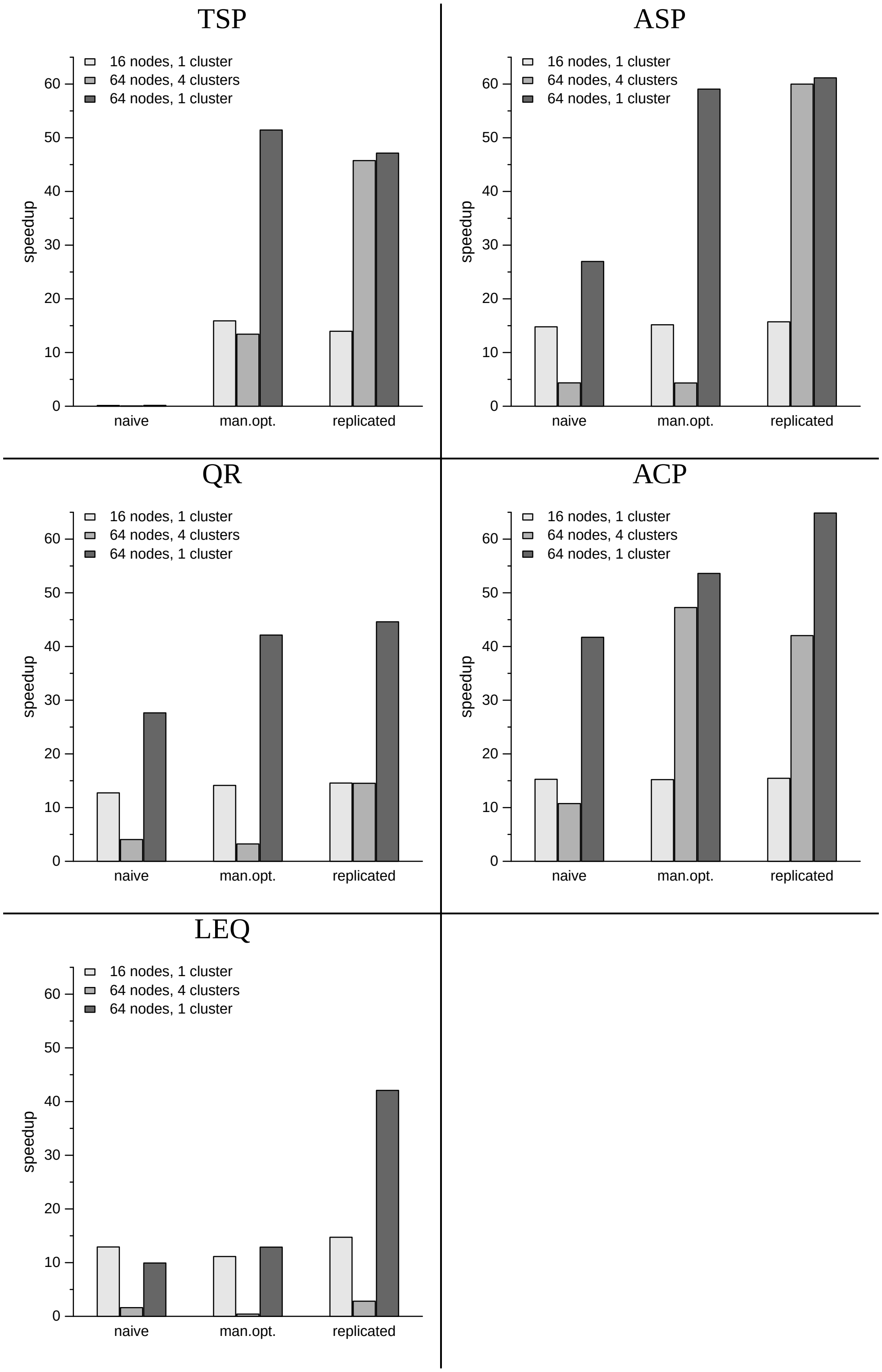
5.3 Application Results

Figure 13 shows the results of executing our five applications on 4 clusters of 16 processors each, compared to executing them on a single cluster, both with 16 and with 64 processors. The comparison with the single-cluster runs gives lower and upper bounds for the expected speedups when using multiple clusters. As in Section 4, all speedups are computed relative to the sequential execution of the fastest of the three versions we compare for each application (naive, manually optimized, or replicated). The manually optimized versions do not implement any additional optimizations for wide-area systems. Wide-area optimizations on the application level are beyond the scope of this paper. We discuss such optimizations elsewhere [5, 29, 34].

Traveling Salesperson Problem The naive (RMI) version of TSP is much slower than its two counterparts. Due to its high overhead for reading the minimum value from a remote object, it can not use multiple processors efficiently, neither on a single nor on multiple clusters. The manual optimization of the RMI version was to change the minimum value to become a variable on each node, and to broadcast updates to it using a spanning tree. For single-cluster systems, this yields good speedup values. With multiple clusters, however, the speedup using 4×16 processors drops below the speedup of 16 processors, making this version useless for wide-area systems. The reason is that the spanning tree ignores cluster boundaries, causing each update message to be sent, unnecessarily, multiple times over the same wide-area link. Manta's runtime system (the Panda layer) solves this problem as it is aware of the actual cluster topology. So, each broadcast message is sent to each cluster exactly once, and is locally distributed using LFC's spanning tree, resulting in minimal utilization of the slow wide-area links. Consequently, the replicated version achieves a speedup of 45.8 using four clusters, which is almost as good as with a single cluster of 64 processors (47.1).

All-pairs Shortest Paths Problem With the naive version of ASP, each processor has to fetch every row of the matrix for itself. This approach already limits achievable speedups on a single cluster. When using a wide-area system, the matrix rows are transferred once per processor across each wide-area link, reducing the speedup even further. The manual optimization uses a spanning tree broadcast for reducing the necessary communication volume. This significantly helps for the single-cluster case. With wide-area systems, however, the spanning tree ignores cluster boundaries, as is the case with TSP. Hence, matrix rows are still sent multiple times over each wide-area link. Using replication, Manta's wide-area aware broadcast allows to achieve a speedup of 60.0 with four clusters, almost as good as the speedup of 61.2 which we achieve with 64 processors in a single cluster.

QR Factorization ASP communicates exclusively by broadcast operations, allowing multiple broadcasts to be pipelined. Thus, application performance is dominated by the inter-operation gap rather than by the overall completion time of the broadcasting write operations. QR is different, as it does a broadcast and a reduce-to-all operation, the latter causing all processes to synchronize in every iteration. Although the replication-based broadcast minimizes the wide-area communication of this part of the application, the reduce-to-all still dominates the achievable overall speedup. The synchronizing nature of the reduce-to-all causes all processes to wait at least for the completion


TSP

ASP

QR

ACP

LEQ

Figure 13: Application speedups, comparing four clusters with 16 processors each to one cluster with 16 nodes and one cluster with 64 nodes.

time of a wide-area broadcast, once every iteration. Additionally, the binary-tree implementation of the reduce-to-all is non-optimal for multiple clusters, as communication crosses cluster boundaries several times, up and down the tree. As a result, QR achieves a speedup of only 14.5 on 4×16 processors, which is about the same as with 16 processors (14.6), making this application unsuitable for wide-area systems. Nevertheless, QR could achieve better results when using a wide-area optimal reduce-to-all operation like the one implemented in our MPI library MagPIe [17].

Arc Consistency Problem ACP is another broadcast-only application. Due to the low communication volume, even the manually optimized binary-tree broadcast achieves reasonable speedup of 47.3 on 4×16 processors. The replication-based implementation performs slightly worse (getting a speedup of 42.0). The reason is that ACP has less update locality, so the sequencer has to be migrated frequently, causing the slightly inferior speedup.

Linear Equation Solver LEQ faces the same problems as QR. Because the synchronization per iteration is caused by a gather-to-all (instead of QR's reduce-to-all), the communication volume is much higher. With LEQ, each processor has to send data to every other processor. This significantly increases the communication volume. Even worse is the fact that this communication pattern has no update locality at all, because all processes have to send simultaneously. On 4×16 processors, the replicated version gets a speedup of 2.8, which is better than the naive and the manually optimized versions. Still, this application is unsuitable for wide-area systems. With a static (non-migrating) sequencer node, LEQ achieves a speedup of 4.6, which is better, but still insufficient. LEQ could only perform better on wide-area systems when using wide-area optimized collective communication operations like the ones from our MagPIe library [17].

5.4 Discussion

Manta's object replication mechanism can significantly improve application speedup on wide-area systems, as long as application performance is dominated by broadcasting write operations. Whenever applications depend on different communication patterns like the synchronizing reduce-to-all or gather-to-all, those operations also need to be optimized for wide-area systems in order to achieve good speedups. For the MPI message passing standard, we already developed wide-area optimized versions of these operations as part of our MagPIe library [17]. We are currently working on integrating similarly implemented operations into our Manta system.

6 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.

In previous work, we investigated the suitability of Orca's replication mechanism for clustered wide-area systems [5, 29]. There, we implemented wide-area optimizations on the application level, causing significant changes to the applications. In [34], we presented parallel Java applications for wide-area systems. There, communication was exclusively based on Manta's RMI mechanism; wide-area optimizations had been implemented again on the application level. In this work, applications use object replication instead of RMI. This approach allows us to implement wide-area optimizations in Manta's runtime system, thus simplifying application code.

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 [38] and DOSA [14] implement a JVM on top of the TreadMarks DSM [16].

Hyperion [24], Jackal [35], 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 [21] 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 [18] 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 64 remote copies of an object in 120 microseconds.

The Javanaise system [12] uses groups of objects (there called *clusters*) 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, 8, 9, 13, 15, 18, 19, 26, 28, 31, 37, 38]. Most of these systems, however, do not support object replication or caching. Several systems (e.g., [15, 28]) support object migration. The Kan Java-based distributed system [20] 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 [10], 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. Some applications like LEQ, however, need additional communication patterns in order to perform efficiently. In [17], we presented optimizations of MPI's collective operations for wide-area systems. We are currently integrating these collective operations into Java's object-oriented programming model.

7 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 reliable, 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 *clouds*, 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 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 cloud objects to distinguish between read and write operations. The runtime system establishes and updates object clouds 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 0.35 microseconds on our platform. Write operations for updating 64 replicas take 120 microseconds, only three times 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 clouds allow complex data structures to be replicated without sacrificing runtime performance. We evaluated our system with five application kernels and showed that Manta's object replication model allows implementation of straight-forward, shared-object applications, while yielding performance close to manually optimized versions based on individual RMI calls.

We have also studied the usefulness of object replication in wide-area distributed systems. Due to the high latencies in such systems, replicating objects with a high read/write ratio is very beneficial. Three out of the five applications run efficiently on a four-cluster wide-area system if object replication is used, whereas only one application obtains good speedups without replication.

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 Grégory Mounié for his comments on the paper. We thank Kees Verstoep and John Romein for keeping the DAS in good shape.

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. E. 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, Feb. 1998.

[4] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. F. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *12th International Parallel Processing Symposium (IPPS'98)*, pages 784–790, Orlando, FL, April 1998.

[5] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. F. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *Proc. 12th International Parallel Processing Symposium IPPS'98*, pages 784–790, 1998.

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

[7] 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, Feb. 1995.

[8] 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.

[9] 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.

[10] 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.

[11] B. R. de Supinski and N. T. Karonis. Accurately Measuring MPI Broadcasts in a Computational Grid. In *Proc. IEEE Symp. on High Performance Distributed Computing (HPDC-8)*, 1999.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 131–140, Atlanta, GA, May 1999.

[18] 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.

[19] 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.

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

[21] 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.

[22] J. Maassen, T. Kielmann, and H. E. Bal. Efficient Replicated Method Invocation in Java. In *ACM 2000 Java Grande Conference*, pages 88–96, San Francisco, CA, June 2000.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] N. Nupairoj and L. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing*, pages 212–226, San Antonio, TX, Feb. 1997.

[28] 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/.

[29] A. Plaat, H. E. Bal, R. F. H. Hofman, and T. Kielmann. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. *Future Generation Computer Systems*, 2000.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming using the Remote Method Invocation Model. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[35] R. Veldema, R. A. F. Bhoedjang, and H. Bal. Distributed Shared Memory Management for Java. In *Proc. sixth annual conference of the Advanced School for Computing and Imaging (ASCI 2000)*, pages 256–264, Lommel, Belgium, June 2000.

[36] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July–September 1998.

[37] 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.

[38] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, pages 1213–1224, Nov. 1997.