Title: Design of the Kan Distributed Object System
Authors: Jerry James and Ambuj Singh

Contact author: Jerry James
Email: jamesj@acm.org
Telephone: (785) 864–4620
Fax: (785) 864–3226
Address: EECS Department
415 Snow Hall
University of Kansas
Lawrence, KS 66045

Design of the Kan Distributed Object System

Jerry James and Ambuj Singh

August 1999

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California

Abstract

Distributed software problems are often addressed with object-oriented solutions. Objects provide the benefits of encapsulation and abstraction that have proven useful in managing the complexity of sequential code. However, the management of distributed objects is typically by means of complex APIs, such as CORBA or DCOM. The complexity of the APIs is itself a hurdle to the writing of efficient, robust programs.

An alternate approach is to provide the programmer with a simple interface to an underlying object management layer that provides efficient access to objects, reliability, and sufficient power for common distributed programming tasks. We have implemented such a system, called Kan. It has a clear, simple object model with powerful semantics, embodying such concepts as atomic transactions, asynchronous method calls, and multithreading. The model constructs help the programmer avoid common concurrent programming errors, allowing clean expressions of concurrent algorithms. We describe the implementation, and investigate several runtime optimizations that make performance efficient for some classes of applications. These optimizations concentrate on reducing method invocation costs. Local method invocations are optimized with a thread pool, thread inlining, and pointer swizzling. Remote method invocations are optimized with object management routines that adapt to access patterns.

1 Introduction

Distributed systems are increasingly ubiquitous, both due to an ever greater need for computational power and because many applications are inherently distributed. Airline reservation systems are an example of the latter, since the travel agents who use such systems are themselves widely distributed geographically. Modern computing systems must cope with the challenges presented by widely distributed systems, such as asynchrony, large and unstable network latencies, heterogeneous computing nodes, congestion, network failures and partitions, computer failures, and security issues. Furthermore, the increased complexity of software and applications is blurring the traditional boundaries between application domains such as databases, distributed computing, and parallel computing. The same user program needs to manipulate shared information consistently, migrate and access distributed resources, and perform its tasks in a concurrent and efficient way. While there is a large body of research on these topics, much remains to be done.

Part of the challenge of programming distributed systems lies in the complexity of the software itself. Software that copes with all of the above challenges tends to be large and complex. Furthermore, distributed system services need clear APIs to be useful. Many have chosen object-oriented technology as the answer to both challenges. For example, commercial distributed applications commonly use the OMG's CORBA [43] or Microsoft's DCOM [13] to provide interoperability between distributed system services and clients written on different platforms, and even with different programming languages. Objects provide the benefits of

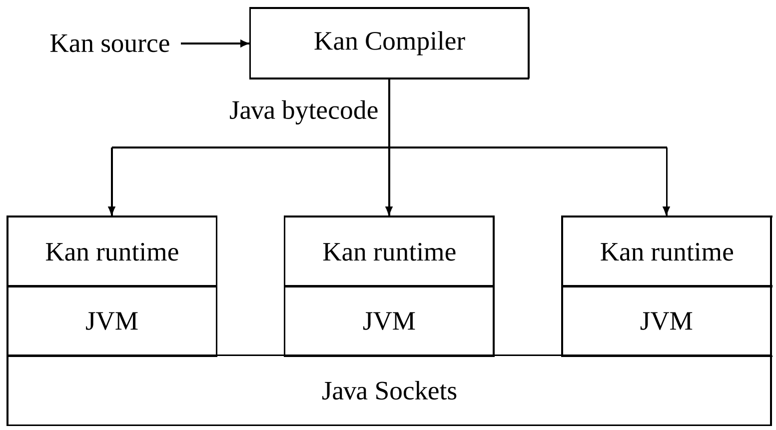

Figure 1: Kan System Structure

encapsulation and abstraction that have proven useful in managing the complexity of sequential code, and provide the modularity needed to make distributed service APIs manageable.

An alternative to large, complex APIs is a simple, powerful interface to an object management layer that provides both efficient object access and reliability. This paper describes the Kan[1] system, a step in that direction. Kan has a clear, simple object model with powerful semantics embodying such concepts as asynchronous method calls (used for expressing concurrency), guards (used for expressing dataflow and synchronization constraints), and nested atomic transactions [39, 40] (used for expressing atomicity). It hides distribution, replication, migration, and faults from the programmer. The model is designed to significantly reduce programming errors caused by incorrect thread synchronization and invalid object consistency. The programmer is freed from the burden of managing object migration, replication and thread migration. The resulting *network transparency* allows the programmer to concentrate on algorithms instead of the details of placing and finding objects. The programmer's view is that of a global namespace in which objects are accessed by concurrent threads. Preserving this simplicity while maintaining efficiency through runtime monitoring is our principal objective. The Kan system borrows the concepts of transactions and consistency from databases, the concepts of synchronization, fault-tolerance, interoperability, and mobility from distributed systems, and concepts of weak consistency and access patterns from parallel computing. The result is a simple, yet powerful, programming model, free from the complexity of modern distributed programming APIs, yet with efficient runtime support.

We chose Java [24] as the base language for our system. The Java language provides us with the ability to write multithreaded applications. However, it is not entirely appropriate for programming distributed systems. For example, Brose, Löhr, and Spiegel show [11, 12] that Java's method-calling semantics, pass-by-value, lead to unacceptable latencies when accessing arrays, and even class instances. They also show that RMI (Java's Remote Method Invocation API) does not solve the problem. Java, even with RMI, does not exhibit access transparency, or identical method calling syntax and semantics for both local and remote objects. Our design allows the programmer to think in terms of nondistributed and distributed, rather than local and remote. The benefit is that the desired scope of an object is generally clear from the program design. For example, user interface objects are nondistributed (or have local scope), since they must interact with the hardware of the computer which the user is operating. However, data containers are distributed (or have global scope), since they must be shared to provide the desired utility.

The basic structure of the Kan system is shown in Figure 1. A source file, using our distributed program-

[1]The word *Kan* is Sanskrit for a small particle, an atom or molecule.

ming constructs, is passed to the Kan compiler. The compiler produces standard Java bytecode, containing calls into the Kan runtime system. That system itself is written in pure Java, so the system and user applications will all run on any standard Java Virtual Machine, using Java sockets for communication.

The functionality of distributed shared objects should be complemented by efficient management in order for a system to be truly useful and scalable. There are a number of performance bottlenecks in supporting distributed objects. First, invoking asynchronous method calls on such objects incur a number of costs. For example, supporting such asynchronous calls implies that a thread is forked for every call. However, thread forking costs can be reduced by maintaining a pool of idle threads. Asynchronous method calls first attempt to fetch such a thread from the pool, forking a new thread only if that fails. A further optimization can be made when an asynchronous method call is made between two objects that are colocated on the same node. In that case, all thread management costs can be avoided if the called method is nonblocking, by having the invoking thread execute the invoked method. A similar optimization is to delay execution of the called method until join time, and use the same thread at that point, if the code between the call and the join is nonblocking. Finally, the overhead of processing global identifiers can be avoided with a *pointer swizzling* technique. When a global object becomes local to a given node, that node swizzles all global identifiers for the object into local references to the object. This allows the object to be called directly from other local objects, bypassing object management costs. All of these optimizations, the thread pool, both forms of thread inlining, and pointer swizzling, have been implemented in the current Kan system.

A second performance bottleneck is that of managing distribution. Simply put, accesses to a remote object are more expensive than accesses to a local object. Our aim is to replicate and migrate objects so as to maximize the number of accesses that are local. To that end, the Kan system assigns a type to every object. The first type of object, *migratory*, is assigned to objects that are accessed by one computing node at a time. Such objects are accessed via some other structure which provides synchronization between nodes. They are often part of a task queue, where nodes fetch a task to work on, perform some work, then produce a result or put the task back on the queue. For example, distributed solutions to the Traveling Salesperson problem often employ such a queue, where partial tours are put on the queue. Each node takes a partial tour off the queue, extends it, and puts the new tours onto the queue. Migratory objects are accessed via a protocol that transfers them entirely to an accessing node. Hence, only the first access to a migratory object by a given node will be remote; subsequent accesses will be local until the object migrates again. The second type of object, *general*, is simply not migratory. Such objects are managed by a dynamically replicating protocol that counts read and write accesses, placing replicas at sites where many reads are issued. The system also features a dynamic object type detector, which assigns the general and migratory object types at runtime, thereby successfully managing objects that change type during the course of an application.

The rest of the paper is organized as follows. In Section 2 we describe our object-oriented model of distributed computing. The architecture of Kan is presented next, in Section 3, along with a rationale for the design decisions. In Section 4, we show the results of our performance testing on Kan, and describe various optimizations used to enhance that performance. In particular, we target thread management (adaptive thread pool, thread inlining, and pointer swizzling), access patterns, and clustering and granularity of object access for optimizations. We discuss related work in Section 5. Finally, we close with conclusions and discussions of future extensions to Kan in Section 6.

2 The Kan Object Model and Language

The Kan object model is similar to other concurrent object models, such as those of Orca [6] and Java [34]. It includes notions of *object*, *class*, *method*, and *thread* that are similar to those appearing in the definitions of such languages as Java [24] and C++ [49]. The Kan programming language itself is Java, with the extensions described in the rest of this section.

2.1 Asynchronous method call

Parallelism in Java programs is expressed via threads. Thread creation is explicit; a new thread is added to the system by creating a *java.lang.Thread* object. While this presents no problem in the context of a single JVM, it does not work as well in a distributed system composed of multiple JVMs with replicated objects. In particular, one wishes to create a new thread in order to execute some method on an object, wherever that object might be located.

Hence, we provide similar functionality in Kan with implicit thread creation via asynchronous method calls. Such calls return an explicit *future*, similar to those used in ABCL/f [57], for example. This is an object that can be used to *join* with the new thread, to test for completion of the called method, and to fetch return values or rethrow exceptional return values. The synchronous method calls with which most programmers are familiar are equivalent to an asynchronous call followed immediately by a join with the forked thread. This equivalence is exploited in our language design.

The *asynch* keyword is written after a method call to indicate that the call should be made asynchronously (see Figures 2 and 3 below). In such cases, a *Future* object is returned to the caller. The future can be used to determine when the called method has completed execution, and to retrieve return values and exceptional results. Method calls without this keyword are made synchronously, as with normal Java method calls.

Method parameters are always passed by value, as in local Java semantics. In particular, since variables of object type store references, then objects are passed by reference, whether the called object is local or remote. That is, there is no local/remote distinction in the model. This ensures that all method calls have a uniform semantics, thereby avoiding the pitfalls identified by Brose, Löhr, and Spiegel [11, 12].

Consider the example code in Figure 2. It sends several values to an adder (or accumulator) asynchronously, then waits to be sure each call has completed. The `add` methods return the new value in the adder after the addition. The program ends by fetching the number returned by the third add operation. Note that the result fetching assumes that the third thread spawned executes last. This might not be the case; asynchronous calls are unordered, so code that assumes a FIFO ordering may not execute as expected. In particular, in this example, the third thread (corresponding to future f_3) may execute *first*. The right approach is to query the *Adder* object a for its final value after the add operations have been joined; e.g., *result* $= a.\texttt{val}();$.

Asynchronous method calls can be used to express the parallelism in distributed and parallel applications. We have found them useful in expressing such applications as a parallel Sieve of Eratosthenes and a distributed web cache, among others. The sieve application divides the range of numbers to sieve among a set of nodes. It then uses asynchronous method calls to fork off one thread per node. Each thread sieves over its selected range, and all higher ranges. In this way, the asynchronous method calls capture the parallelism inherent in this application.

$\textit{Adder}\ a = \textsf{new}\ \textit{Adder}(0);$
$\textit{Future}\ f_1, f_2, f_3;$

$f_1 = a.\texttt{add}(x)\ \textsf{asynch};$
$f_2 = a.\texttt{add}(y)\ \textsf{asynch};$
$f_3 = a.\texttt{add}(z)\ \textsf{asynch};$
$f_1.\texttt{join}();$
$f_2.\texttt{join}();$
$f_3.\texttt{join}();$
$\textit{int result} = f_3.\texttt{resultInt}();$

Figure 2: Model example: adding integers x, y, and z

2.2 Guarded Atomic Actions (Transactions)

Kan includes the notion of a guarded atomic action. The guard is a predicate over the states of the executing thread and the associated object. Since that state may contain object references, we also allow method calls on such objects to be part of the guard. However, such calls must be to "read-only" methods; i.e. such methods cannot change the state of the associated object. This restriction is necessary because a guard may be evaluated multiple times before it is found to be satisfied.

Each guard is associated with a block of code that is executed as a transaction in a state satisfying the guard. Guards are useful for expressing dataflow synchronization requirements. The most common idiom for programming dataflow synchronization expressions in Java is to insert a `while` loop that tests for the condition, wrapped around a `wait`. Then, anywhere that the condition becomes true (or *might* become true), a `notify` is issued (or, if the number of waiting threads might be greater than one, a `notifyAll` is issued) If the programmer forgets a single `notify`, some threads may wait forever, leading to lack of progress at runtime. If the programmer forgets a single `wait`, the program can be incorrect, due to executing critical code when the condition is not satisfied. In Kan, the programmer does not have to worry about where the condition becomes true. The programmer just marks off regions of code where interference might be a problem, and writes dataflow synchronization and mutual exclusion requirements into the guard without worrying about where the requirements will be met. This significantly reduces opportunities for intermittent, difficult to debug synchronization errors.

The guard in our model plays much the same role as in Owicki and Gries' system [44], ensuring that the local state meets some criterion before the following atomic step takes place. However, our construction differs from their **await B then** S construction in allowing method calls inside an atomic step. Like Owicki and Gries, we assume that each transaction is terminating, as a non-terminating atomic action can never have any visible effect on the system state. The guard construct also bears a close relation to that of Orca [7], the key difference being that Orca allows a thread to wait on multiple guards, nondeterministically selecting one if more than one is satisfied. It is also similar to the dataflow synchronization structure of Distributed Oz [51].

The *guard* keyword is used to mark a block of code as containing an atomic transaction, and to block execution of that transaction until a boolean predicate is satisfied. The boolean predicate must be over the local state of the thread and the fields of the object. Method calls are permitted. However, since a guard may be evaluated multiple times, the programmer should ensure that those method calls do not change the state of any object.

We have found guards to be a powerful and useful construct for expressing dataflow synchronization constraints in a distributed whiteboard. In the whiteboard, a shared space can be drawn upon by each participant. The task of the application is to show each participant an up-to-date view of the shared space. One thread on each computing node acts as an observer. In effect, it computes a snapshot of the shared space, then suspends itself on a guard that asserts that the state of the object does not correspond to its snapshot. When that assertion becomes true, it indicates that the shared space has changed state. Therefore, the awakened thread updates the displayed view of the shared region, then repeats the action of taking a snapshot and waiting for another change to the shared space.

2.3 Nested atomic action (nested transaction)

Sometimes more than one object must be accessed in the same atomic step. For that reason, Kan provides *nested transactions* [39, 40]. A nested transaction occurs when one or more threads are forked and joined in the same transaction (i.e., either a synchronous method call is made, or an asynchronous method call is made and the associated future is joined). In such cases, the entire action must appear to take place in one atomic step. Note that a method called as part of a nested transaction may itself consist of multiple

```
public class Bank {
    public static void transfer (float amt, Account a1, Account a2)
        throws InsufficientFundsException {
        guard (true) {
            Future f_w = a1.withdraw(amt) async;
            Future f_d = a2.deposit(amt) async;
            f_w.join();
            f_d.join();
        }
    }
}
```

Figure 3: Transfer funds

transactions, but the total effect must be equivalent to executing all the transactions in isolation; i.e. with no other threads executing in the system. Our nested transactions are very similar to those appearing in the literature on databases, and are related to those featured in Argus [35, 37].

As an example, consider the transfer of funds from account X to account Y. We must ensure that money is not created (the withdrawal from X fails, but the deposit to Y succeeds) or lost (the withdrawal from X succeeds, but the deposit to Y fails). That is, the entire transfer must take place in one atomic step. Consider the code of Figure 3. It starts the withdrawal and deposit actions asynchronously, then ensures that both actions have completed. If the withdraw action cannot be completed due to insufficient funds, then it throws an *InsufficientFundsException*. When the join action takes place on f_w, the exception is rethrown from there, causing an abort of the entire transaction due to a user exception. Otherwise, both actions complete atomically, thereby effecting the desired transfer.

Nested atomic actions are not only useful for banking applications. We have also found them useful for handling the node splitting rules in a distributed B-Tree application. In this case, the problem to be solved is that an insertion has caused a node to overflow, requiring a split and an insert into the parent. If multiple children overflow concurrently, the inserts into the parent must be handled in a consistent way. The nested transaction model gives us the ability to make all of the inserts and splits required for any given element insert appear to take place in a single atomic step, without interference from concurrent insertions into the tree. Therefore, searchers also see a consistent view of the tree, since the intermediate steps are hidden from view.

2.3.1 Deadlock

Nested transactions make deadlocks possible. For example, consider a transaction that first locks object X, then attempts to lock object Y. Meanwhile, another transaction has locked object Y and is now attempting to lock object X. This is a classic case of deadlock. In general, unless the user obeys some locking protocol, the system must be prepared to deal with deadlocks.

Guards introduce another scenario in which deadlock can occur. Suppose a transaction has obtained a lock on object X, then goes to sleep on an unsatisfied guard. If the guard cannot be satisfied until some other thread obtains the lock on X, then deadlock has occured. Hence, when executing nested transactions, we cannot wait forever for an unsatisfied guard to become satisfied. After a finite time, we must abort and roll back the transaction. Note that single level transactions can sleep on unsatisfied guards indefinitely, because they will not hold any locks while sleeping.

Our solution to the deadlock problem is to make them invisible to the user. The system will take corrective action, aborting and rolling back one transaction out of a set that is (or might be) involved in a deadlock.

This will allow the remaining transactions to acquire the needed locks and make progress. More information on this topic is provided in Section 3.4.4.

2.4 Consistency Model

When choosing a consistency model, we must balance the needs of the application programmer for a clear, intuitive model with the needs of the system to apply performance-enhancing optimizations. In the case of shared read/write memory, weakly consistent systems enable optimizations while providing the equivalent of a strong memory model for certain common classes of programs. However, the situation is less clear with shared object models. Weakly consistent objects are poorly understood. Indeed weak consistency appears to be incompatible with objects that make method calls on each other up to arbitrary depths. For that reason, shared object systems are often linearizable [28].

However, there is an intermediate ground. Java programmers are already used to programming in a multithreaded environment, marking critical sections of code with synchronized. We extend this programming model to Kan, and guarantee that transactions (or atomic actions) are linearizable with respect to each other. This gives the programmer the freedom to leave code that is known to be free from interference outside of transactions, thereby avoiding the cost of atomic actions. However, if code outside of a transaction executes concurrently with a transaction, we make no guarantees that the effects of the nonatomic code are made visible everywhere (or indeed, anywhere).

Nested transactions are exactly like their counterparts in databases. The entire nested transaction must appear to the rest of the system to take effect in one atomic step. We give a design that accomplishes this in Section 3. Note that our consistency model is equivalent to linearizability iff every method consists of exactly one transaction and there is no nesting.

2.5 The Kan Programming Language

Since we selected Java as the implementation language for Kan, we chose Java as the programming language as well. However, Java does not fully support our distributed object model. In fact, strictly speaking, Java does not support distributed programming at all. In conjunction with RMI (the Remote Method Invocation package), Java supports *remote* programming. However, there is no support for object directories, object replication and migration, etc. in RMI. For these reasons, we have enhanced the Java language both with the keywords described above, and with the following additional features.

In addition to the asynch and guard keywords, Kan uses the global keyword as a modifier for classes. It marks the class as needing special preprocessing by the Kan compiler to make it suitable for distribution. Classes without this keyword are not distributable; care must be taken not to share references to such objects between global, or distributable, objects. Note that if such references are shared, the object may be copied.

Java applications are started at a method named `main` in a class identified by the user. This method must have the signature public static void `main` (*String*[] *args*). A distributed Kan application may need more information. It may need to know about the number of nodes on which it is executing, for example. Information about a distributed application is packed into a *kan.comm.AppInfo* object, which is passed as the first argument of `main`. That is, the `main` method has the signature public static void `main` (*AppInfo app*, *String*[] *args*) for Kan applications. The number of nodes can be determined, for example, by calling *app*.`numNodes`().

The new keyword is used to create objects in Java. In Kan, it can be used in the same manner as in Java. When a distributed object (that is, an object of a class declared global) is created, it is created on a node chosen in a round-robin fashion. The user can also select the node on which an object is initially created[2] using an alternate syntax. In an application running on N nodes, the nodes are identified by the integers zero

[2]The system may choose to move or replicate the object after its creation, however.

```
public class Barrier {
    private int N;          // # of processes
    private int in = 0;     // # entered barrier
    private int out = 0;    // # left barrier

    public Barrier(int num) {
        N = num;
    }

    public synchronized void join() {
        if (++in == N)
            notifyAll();
        else while (in ≠ N) {
            try { wait(); }
            catch (InterruptedException e) {}
        }
        if (++out == N)
            in = out = 0;
    }
}
```

Figure 4: A Java barrier

```
public class Barrier {
    private int N;          // # of processes
    private int in = 0;     // # entered barrier
    private int out = 0;    // # left barrier

    public Barrier(int num) {
        N = num;
    }

    public void join() {
        guard (out == 0)
            in++;
        guard (in == N) {
            if (++out == N)
                in = out = 0;
        }
    }
}
```

Figure 5: A Kan barrier

through $N - 1$. To create a distributed object on node i, the programmer writes a statement of this form: *DistObj dObj* $=$ new(i) *DistObj*$($*args*$)$.

2.6 Example

As a final example of the model and language, consider a barrier over a fixed set of processes or threads. One way to implement a barrier in a shared data environment is to have each thread atomically increment a counter as it enters the barrier. The last thread to enter the barrier sees that the counter equals the total number of threads using the barrier, and signals all threads to exit. The barrier can be made reusable by detecting that all threads have exited the barrier and resetting the state of the barrier at that time. If done in Java, the results might look like the code in Figure 4.

However, this code is incorrect. It demonstrates a common Java programming error. Even though the `join` method is synchronized, interference can still occur between separate uses of this supposedly reusable barrier. On the first use of the barrier, the final thread to enter executes a `notifyAll` and exits the barrier. Suppose it does not lose its timeslice, does all the work of the next phase, and reenters the barrier. Now *in* is $N + 1$, so it waits. Eventually, the last thread from the first use of the barrier exits, and *in* and *out* are reset to zero. But there is a thread waiting in the barrier already, so after all other threads enter the barrier for the second time, *in* will equal $N - 1$, and no thread can exit the barrier.

Waiting and notifying are difficult to do correctly in Java, even within the same method. When corresponding waits and notifies are spread across methods, or even across classes, mistakes become very easy to make. In contrast, consider the Kan version of the barrier, shown in Figure 5. The structure of the barrier is clear, with one transaction for entering the barrier and another for exiting the barrier. The dataflow

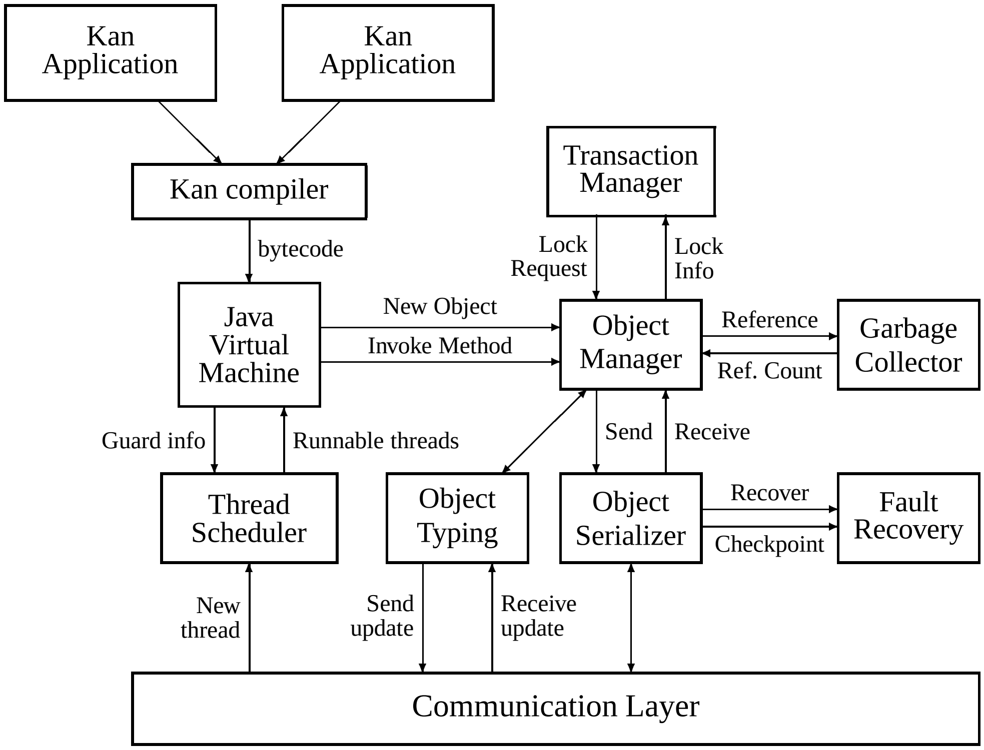

Figure 6: Kan System Organization

synchronization requirements are explicit, instead of being hidden in implicit wait/notify pairs.

3 Kan System Architecture

In this section, we describe the design choices made when implementing the Kan system. The main concern was to correctly implement the model described in Section 2. The secondary concern was to implement the model efficiently, a topic we turn to in Section 4. The major components of the Kan system, and their relationship to one another, are shown in Figure 6. We will cover each of these components in the succeeding sections. All of these components have been implemented, except for the garbage collector, which is under construction (see [8]), and the fault tolerance module, for which we have a paper design.

3.1 Communication

In this section, we describe the lowest layer of the Kan system, the communication layer.

3.1.1 Sockets

At the lowest layer, we use Java sockets for communication. These have the advantage of providing a simple, uniform interface across all Java platforms. Furthermore, socket programming has a long history, so the issues involved in programming them are well understood.

On the other hand, Java sockets are inefficient. Reading and writing Java sockets involves accessing several layers of Java streams, crossing a JNI (Java Native Interface) boundary, then executing in kernel socket code. Furthermore, Java does not support the `select()` function, although most modern Unixes support it. This lack requires us to run one polling thread per socket, thereby paying higher threading and context switch costs for communication than otherwise might have been the case.

Nevertheless, the initial implementation of Kan uses Java threads for their simple, well-understood semantics. Future revisions of Kan will implement network-specific communication layers. This will require native code for each network type, so the socket code will be retained as the default in case no applicable code is available. Active Messages [52], in particular, appear promising as a way of reducing communication costs.

Currently, Kan sets up socket connections between all pairs of nodes at startup. This approach is not scalable. Future work will manage sockets as resources, setting them up on demand, and limiting the total number of sockets open at any one time. This approach will also provide a greater degree of fault tolerance.

3.1.2 Object Streams

Java provides a way of transmitting objects across networks via a mechanism called *serialization*. When an object is serialized, a portable byte stream representation of the object is created. The reverse process, *deserialization*, creates an object from a byte stream.

The interfaces to the serialization and deserialization processes are through two JDK-supplied stream classes, *java.io.ObjectOutputStream* and *java.io.ObjectInputStream*. A stream of each type is attached to every socket connection by Kan, enabling objects to be sent over a network. As with the sockets themselves, this approach has the advantage of simplicity and ease of use, but the disadvantage of poor performance. However, since Kan must transmit user objects of unknown structure over the network, the use of serialization is mandated.

3.1.3 Physical and Logical Nodes

A Kan system is composed of a set of computers communicating over a network. We refer to each computer as a *physical node* of the Kan system. The class *kan.comm.PhysicalNode* contains the socket and object stream information needed for communication over the physical network.

Another view of a node is as a container of user objects, threads, and transactions. However, this is a different kind of entity, as user objects and threads can be moved among the physical nodes of a system. We call these containers *logical nodes*. Initially, there is exactly one logical node per physical node. Currently, the initial configuration remains stable throughout execution of a Kan application. However, when we plan to implement a fault tolerance scheme that remaps logical nodes to different physical nodes due to failures. Also, the current system allows multiple Kan applications to run simultaneously, possibly resulting in logical nodes from different applications residing on the same physical node.

An object of class *kan.comm.LogicalNode* contains information about a single logical node. Each logical node has a globally unique identifier. In the current implementation, the ID is a 64-bit value. The upper 32 bits are the IP address of the physical node initiating the associated application. The lower 32 bits are uniquely assigned by the initiating node, based on a counter kept by each physical node. This scheme will have to be redesigned for other network types (including networks using 128-bit IPv6 addresses).

Each *LogicalNode* object either resides on the local physical node or on a remote physical node. If it is local, it has a reference to a *kan.comm.LocalLogicalNode* object, which contains references to the object and thread containers for that node. Otherwise, it has a reference to a *kan.comm.RemoteLogicalNode* object, which contains interface code for sending messages to such a node.

3.1.4 Messages

The various nodes of a Kan system use objects of class *kan.comm.Message* to send messages to one another. This class has fields holding the originating and destination nodes of a message, as well as an automatically generated sequence number. This number is used to match replies up with the messages to which they are replying. The two main subclasses of *Message* are *kan.comm.SysMessage* (system messages), for communication between physical nodes, and *kan.comm.AppMessage* (application messages), for communication between logical nodes.

Examples of messages that are sent between physical nodes are those regarding application startup and teardown, and fault tolerance traffic. Messages that are sent between logical nodes will be described in more detail below. They include messages for invoking method calls and returning values, creating objects, and object coherence traffic.

The fact that messages exist in a class hierarchy simplifies message handling. We use polymorphic dispatch to tell each message to handle itself, once the message object has been reconstituted. This approach lets us avoid constructing tables of message types inside the communication layer. However, this again trades off speed for convenience, as polymorphic dispatch incurs a runtime cost.

3.2 Distributed Objects

In this section, we describe the management of global objects.

3.2.1 Global IDs

Each object, when created, is assigned a globally unique ID. These IDs are formed of two components: the ID of the creating logical node (which is itself globally unique; see Section 3.1.3), and a counter kept by that node. For objects with home nodes, we arrange to make the creating node the home node. This lets any possessor of a *kan.obj.ObjectID* determine the home node of the associated object with a local operation on the ID.

3.2.2 KanObject

Each global object has a controller, an object of class *kan.obj.KanObject*. This controller tracks replicas, and provides the interface for hiding object typing (see Section 4.4) from the rest of the system. Object typing determines the consistency and locking protocols used to access the object.

3.2.3 Lock

Each object and object replica has an associated local lock. These locks solve the preemptible, guarded readers/writers problem. That is, they provide both shared (read) and exclusive (write) locks, each in both preemptible and nonpreemptible mode, and also ensure that a lock is only held by a thread with a satisfied guard. These locks are purely local constructs. To lock a replicated object, one must also lock all of the replicas. This is considered in more detail in Section 4.4, which describes object typing.

A future extension of locks is to associate several objects with the same lock. This will have the effect of causing the objects to stay together during migrations and replications, thereby clustering related objects.

3.3 Distributed threads

Asynchronous method calls can cross physical node boundaries. Therefore, we need some machinery to enable remote method invocations, and proper transmission of return values and exceptions. This section describes that machinery.

3.3.1 Global IDs

Each thread is assigned a globally unique ID of class *kan.thrd.ThreadID*. These IDs are constructed exactly like object IDs (see Section 3.2.1). In fact, they have a common superclass, *kan.util.ID*, which is also used to construct globally unique IDs for other Kan constructs (see Sections 3.3.3 and 3.4.1).

3.3.2 AsynchCall vs. KanThread

Java threads may carry information that is not easily moved between Java Virtual Machines. In particular, an implementation is allowed to give each thread a native stack, with native machine pointers to local data structures. Such pointers cannot be meaningfully transferred across a network. For this reason, Java threads cannot be serialized and deserialized successfully.

However, we need to do exactly that in order to support the fault tolerance scheme we plan to implement in Kan. When a node fails, the threads that were executing on it must be reconstituted elsewhere. Therefore, we encapsulate the work to be done by a thread in an *kan.thrd.AsynchCall* object. Each *AsynchCall* is run by associating a *kan.thrd.KanThread* (subclass of *java.lang.Thread*) with it. Although we cannot transfer *kan.thrd.KanThread*s between nodes, we can transfer *AsynchCall*s, thereby giving us the desired functionality.

3.3.3 Future

A *kan.thrd.Future* holds the result of an asynchronous method call. Each is assigned a unique ID so that results can be returned across the network successfully. This ID is constructed like all other globally unique IDs considered so far.

A joining thread calls one of the family of `join` methods defined in class *Future*. There is one that returns object types, one for each of the primitive types, and one that returns nothing (i.e., it is of type *void*). Each of them acquires a lock on the *Future* object, then checks whether it is marked as completed. If not, then a `wait`() is executed to put the thread to sleep until the method call returns. If so, then the appropriate value is returned. However, if the called method terminated due to an exception, that exception is rethrown at this time, rather than returning a value.

When a method call terminates, the return value is shipped back to the calling node, if it is not local. There a lock is acquired on the appropriate *Future* object, the return value (or exceptional value) is stored in the *Future* object, and a `notifyAll`() is executed to wake up any sleeping threads. Note that a `notifyAll` is necessary, and not a `notify`, since the user may have passed the *Future* object around, resulting in multiple threads attempting to join.

3.4 Transactions

Guarded atomic sections of code are implemented with a transactional mechanism. This mechanism handles automatic abort and restart of transactions to avoid deadlock.

3.4.1 Global IDs

Each transaction has a unique transaction ID, constructed just like all the other global IDs described above. In addition, nested transactions carry the ID of the top-level transaction. We define a total order on transaction IDs, which is used in the wound-wait algorithm to decide how to break deadlocks.

3.4.2 Blocking

Some transactions might block midway, due to waiting for a method call to complete, for example. Such transactions might be aborted and rolled back, so undo information must be kept. Other transactions will never block, so we optimize by not keeping the undo information.

Currently, the undo information consists of a copy of the original object. Transactions use this copy to restore the original state of the object in case of an abort. Aborts are represented as exceptions, subclasses of *kan.trans.TransactionAbortException*. Transaction managers install appropriate try-catch blocks to catch aborts and restart the affected transaction.

3.4.3 Nested Transactions

Since transactions are limited to accessing the state of a single object, multiple object atomic actions are accomplished with nested transactions [39, 40]. These occur when a transaction contains one or more method calls that are either synchronous or joined within the scope of the transaction.

Transactions can nest to arbitrarily deep levels. The initiating transaction is referred to as the *root* transaction. When nesting occurs, there is a *top-level* transaction that starts the nesting; the others are called *child* transactions. The terms *ancestor* and *descendant* are defined in the obvious way.

We want the full set of ACID properties to apply to an entire nested transaction hierarchy as a whole. Hence, when a subtransaction completes, its effects cannot be made visible outside of the hierarchy. No effects can be seen until the entire hierarchy is prepared to commit. For this reason, local copies of objects can be tagged as being visible only to subtransactions of some transaction ID. As commits occur, the tags are revised upward in the nested transaction tree. When the top-level transaction commits, the copy visible to that transaction becomes the new, globally visible copy of the object.

Since nested transactions may span multiple nodes, we need a mechanism for ensuring atomic actions across nodes. We choose 2-phase commit for that purpose. When the top-level transaction is ready to commit, it tells all of its subtransactions to prepare to commit. At that time, they all change their preemptible locks to nonpreemptible locks. If any subtransaction is unable to make the lock conversion (due to preemption), it signals an abort. Otherwise, the subtransactions signal that they are ready to commit, and the top-level transaction sends out the final commit message.

3.4.4 Deadlock

The Kan system uses a conservative method for breaking deadlocks, similar to the scheme used by Argus [35, 37]. Nonnested transactions acquire only one lock, so they are not involved in the deadlock breaking system. We use the wound-wait algorithm described by Moss [40]. Accordingly, each transaction is given a timestamp at its creation, which is encoded into its transaction ID. These timestamps do not necessarily correspond to real time; that is, a transaction created later in real time may have an "older" timestamp than some other transaction. Nevertheless, they have the monotonically increasing property that ensures the absence of deadlock.

When a nested transaction T attempts to acquire a lock, it first checks whether some other nested transaction holds the lock. If a younger transaction S (one with a higher timestamp) holds the lock, then T *wounds* S, causing it to abort, roll back, and release the lock. If an older transaction holds the lock, then T *waits*. Waiting transactions are sorted by timestamp so that the oldest waiting transaction is always given the next chance to acquire the lock[3].

It is not necessary to wound a younger transaction immediately, since there may be no deadlock in fact. The Kan system sets a timer on detecting a situation calling for the wound action. If the transaction holding

[3]However, the oldest transaction is not guaranteed to get the lock, since its guard may not be satisfied.

the lock has not released it when the timer expires, it is wounded then. The value of the timer is a tuneable parameter of the system, since a reasonable value will be affected by such factors as network latency and clock speed of the computers involved.

4 Performance of Kan

In this section, we describe the Kan optimizations in more detail, and measure their performance impacts. We use microbenchmarks to assess performance. The measurements presented in this section were taken on a set of 350 MHz Pentium II machines. Each runs the Solaris operating system, either version 5.6 or 5.7[4]. Each machine has 128 megabytes of RAM. Our Java platform is the JDK 1.2.2 reference implementation (also known as the Java SDK 2, version 1.2.2). All Java source files were compiled with optimization enabled. Note, however, that the Java compiler produces identical bytecode with optimization on and off for most of the microbenchmarks in Section 4.1, since they were written in a manner that is intended to defeat optimizations. Finally, all measurements were made with "green threads", a user-level thread system, rather than the heavier-weight native threads. Native threads are managed with operating system support, resulting in degraded performance on uniprocessor machines such as those we used in these tests. Each set of measurements is made twice, once with the Just-In-Time (JIT) compiler enabled, and once with it disabled. All measurements were repeated at least 10 times, and more if needed to obtain a reasonable range of values at a 95% confidence level. Each individual measurement is made by doing the action to be measured tens of thousands to tens of millions of times, as appropriate for the time scale of the action, and dividing the total time by the number of loops. We assume that the looping time is negligible when measuring Kan constructs.

The local network is a 10 Mbps (or "slow") Ethernet. The round-trip latency on the network is 350.859 ± 2.981 microseconds between machines on the same subnet. The latency between two machines that are as far apart as possible, in the network sense, in our local network is 1064.086 ± 10.646 microseconds under conditions of light use.

4.1 Basic Java Costs

In this section, we give the basic costs of using the Java system described above. Several fundamental costs are described and measured, namely those of reading the clock, making method calls, creating and starting threads, doing method lookups with Reflection, synchronization, and the Serialization of objects.

4.1.1 Reading the Clock

First we give the overhead of collecting timing information, in Table 1, so that we know how other measurements are affected by reading the time. First we measure *System*.`currentTimeMillis`(), which is the standard way of getting the current time in Java. This has millisecond resolution, and is based on a call to the Solaris function `gettimeofday`.

Millisecond resolution is too coarse for our purposes, so we implemented our own native code that calls the Solaris function `clock_gettime`, which is available in the POSIX4 library on Solaris 2.6 and the RT (realtime) library on Solaris 2.7. This function has nanosecond resolution (although only microsecond accuracy on the test machines). A number of clocks are potentially available. We use CLOCK_REALTIME, which is the only clock available on our test systems, returning wall clock time. Some systems also support CLOCK_VIRTUAL, which reports only actual CPU usage, taking context switches into account. Our Java interface is through *Timer*.`currentTimeNanos`().

[4] The latter is also known as Solaris 7

Clock function	JIT	No JIT
System.currentTimeMillis	6.093 ± 0.025	6.169 ± 0.062
Timer.currentTimeNanos	5.165 ± 0.005	5.071 ± 0.016
gettimeofday	5.236 ± 0.004	
clock_gettime	4.432 ± 0.015	

Table 1: Clock reading time, in microseconds

Method type	JIT	No JIT
No args, returns void	114.264 ± 0.611	429.611 ± 9.166
1 int arg, returns void	120.567 ± 1.327	501.329 ± 4.064
32 int args, returns void	407.189 ± 3.022	2390.348 ± 5.783
1 ref arg, returns void	135.146 ± 1.739	662.086 ± 0.969
1 ref arg, returns int	178.004 ± 1.307	720.158 ± 2.013
1 ref arg, returns int, final	178.708 ± 1.846	719.954 ± 1.212
1 ref arg, returns int, static	111.492 ± 1.220	609.744 ± 0.857

Table 2: Method calling time, in nanoseconds

Finally, we measured the cost of calling the two C functions, so we can see how much overhead is inherent in the Java Native Interface (JNI), which allows Java programs to call “native” (binary) code. The Java settings make no difference for these results. On our platform, the JNI overhead is approximately 650 to 950 nanoseconds per call to the time functions.

4.1.2 Method Call

Method calls are one of the fundamental activities performed by Java programs. The cost of making a method call is greatly affected by the number of parameters, since each must be pushed onto the stack. Whether there is a return value or not also affects the time. Finally, calls to static methods are faster than calls to instance methods. This is partly due to the fact that instance methods have a hidden parameter, *this*, and partly due to the fact that static method calls are resolved at compile time. The results of our experiments are shown in Table 2, in nanoseconds. A “ref arg” is a reference argument, that is, a parameter of object (or array) type. Note that the final keyword did not impart any significant difference in method calling time.

4.1.3 Threads

Thread creation and startup costs turn out to be a major component of application overhead. Creating a thread involves allocating a stack, and setting up internal parameters. Starting a thread is a slow process, typically involving complicated manipulations of the scheduler state. Table 3 shows the results, in microsec-

Activity	JIT	No JIT
Create 1	4419.167 ± 29.810	283.328 ± 0.718
Create 100	86.539 ± 0.458	72.788 ± 0.392
Start 1	1850.654 ± 48.114	544.878 ± 1.543
Start 100	521.792 ± 2.529	440.341 ± 1.082

Table 3: Thread costs, in microseconds

Activity	JIT	No JIT
instanceof same class	0.323 ± 0.002	0.560 ± 0.013
instanceof subclass	0.326 ± 0.006	0.536 ± 0.003
1 int arg, 1 match, same class	71.216 ± 0.349	60.492 ± 1.190
1 ref arg, 1 match, same class	78.587 ± 0.427	89.367 ± 0.131
1 subclass arg, 1 match, same class	162.728 ± 1.165	133.284 ± 0.405
1 int arg, 1 match, superclass	59.347 ± 1.436	53.956 ± 1.319
1 int arg, 10 matches	70.112 ± 0.309	80.088 ± 1.802
10 int args, 10 matches	147.781 ± 0.210	126.234 ± 0.865

Table 4: Reflection costs, in microseconds

Contention	JIT	No JIT
None	661.965 ± 22.922	938.057 ± 0.310
10 threads	7188.913 ± 106.471	7204.132 ± 69.955
100 threads	7638.570 ± 111.702	7377.141 ± 66.753

Table 5: Synchronization costs, in nanoseconds

onds, of creating a single thread, creating a batch of 100 threads, starting a single thread, and starting 100 threads in a row. Notice that working with larger numbers of threads improves the average time, most notably for thread creation.

4.1.4 Reflection

The Java Reflection package gives a way of searching for classes and methods by name, and of invoking methods found in this way. Reflection is one way of calling a method remotely: simply pass the name of the method to call and look it up on the remote site. The reflection cost involves finding all methods with the appropriate name, discarding all those with the wrong number of parameters, and then searching the rest for a best match for the given parameters.

The table in Table 4 shows the results of our experiments, in microseconds. The first row gives the time for using the instanceof operator when the argument is an instance of the class named on the right side. The second row gives the time when the argument is an instance of a subclass of the class named on the right side. The remaining rows give the time for searching for a *Method* object. First we look for a method taking one integer argument, when there is only one method in the class with the same name in the class on which we invoke the search. Next we repeat the search, but with a method that takes one object (or reference) argument. We repeat that search, but give a subclass of the declared type of the parameter. Next we search for a method taking one integer argument, for which there is a single match, but it is in the superclass of the class on which we invoke the search. Next, we search for a method taking one integer argument in a class containing ten methods with the same name and same number of arguments. Finally, we repeat the search but use a method taking 10 integer arguments.

4.1.5 Synchronization

The synchronized keyword marks a method or block of code as needing to acquire a lock on some object before continuing. This is the only synchronization primitive offered by the Java language. The performance of synchronization has been a major concern for the developers of JVMs, and has been the subject of various attempts at optimization (e.g., [5]) or avoidance (e.g., [10]). In Table 5, we give the cost of entering and

Object type	Class overhead	Per-object	Single object
int	6	4	10
int[100]	17	10	427
Integer	71	10	81
Node	261	144	405
Tree	261	2026	2287
Complex	132	18	150

Table 6: Serialized size, in bytes

exiting a synchronized block with no actions inside the block, in nanoseconds. We measure the cost with three levels of contention: none (only one thread is running the program), ten threads are vying for the lock, and 100 threads are vying for the lock.

4.1.6 Serialization

Java provides a way of converting objects into portable byte streams, and then reconstituting objects from those streams. The conversion of an object to a byte stream is called *serialization*; reconstituting an object is called *deserialization*. We use serialization to send message objects between nodes. The serialization features of Java is not known for its good performance. In fact, it is one of the major bottlenecks in the performance of Kan. In this section, we measure its performance.

We use a variety of objects to measure the performance of serialization:

- A primitive type, *int*, which is a 32-bit quantity.
- An array of 100 *int*s.
- A "wrapped" *int*. This is an object with a single field, of *int* type. It is used to store integers in contexts where an object is required. Its type is *java.lang.Integer*.
- A *Node* object, which is a component in a tree. It has 34 fields. Two of the fields are references to other *Node* objects; they are named *left* and *right*. The other 32 fields all have type *int*. They are named *i1*, . . . , *i32*.
- A tree, which is a balanced binary tree of 15 *Node* objects.
- A "complex" structure. This is an instance of class *ComplexC*, which has one field named *val* of type *int*, and is a subclass of class *ComplexB*. Class *ComplexB* has one field named *ber* of type *int*, and is a subclass of class *ComplexA*. Finally, class *ComplexA* has one field named *num* of type *int*.

In Table 6 we show the number of bytes that are produced when these objects are serialized. The first column shows the number of bytes that are used to represent the class or type of the object. The second column shows the number of bytes that are used to represent a single instance of the class or type. The third column, which is simply the sum of the first two columns, shows how many bytes are produced if a single object of that class or type is serialized.

There are two anomalies in this table that should be noted. First, for the integer array, the third column is not the sum of the first two. In this case, 17 bytes are written to represent the *int*$[100]$ type, 10 bytes are written per array, and the array elements have the cost of single *int*s; i.e., 4 bytes each. Hence, a single array is serialized to $17 + 10 + 4 \times 100 = 427$ bytes. Second, the tree consists of 15 *Node* objects, each of which

Object type	With class information		Without class information	
	JIT	No JIT	JIT	No JIT
int	2.189 ± 0.003	2.100 ± 0.016	1.807 ± 0.004	2.120 ± 0.012
int[100]	149.359 ± 1.467	304.500 ± 0.698	102.397 ± 2.754	299.623 ± 1.379
Integer	71.362 ± 0.705	105.571 ± 0.408	59.681 ± 0.997	100.979 ± 0.234
Node	182.580 ± 5.959	262.198 ± 0.691	151.677 ± 1.843	244.046 ± 0.373
Tree	2010.762 ± 30.021	3339.244 ± 13.913	1917.007 ± 29.781	3287.843 ± 18.774
Complex	72.755 ± 0.637	139.181 ± 0.294	71.795 ± 0.691	136.916 ± 0.156

Table 7: Serialization times, in microseconds

Object type	With class information		Without class information	
	JIT	No JIT	JIT	No JIT
int	5.506 ± 0.027	13.555 ± 0.049	3.588 ± 0.000	13.417 ± 0.028
int[100]	125.999 ± 0.287	281.582 ± 0.778	67.146 ± 0.823	263.665 ± 0.406
Integer	63.991 ± 0.744	104.493 ± 0.169	45.791 ± 0.379	98.528 ± 0.181
Node	158.346 ± 2.506	268.440 ± 1.214	139.453 ± 0.938	249.788 ± 0.300
Tree	2005.962 ± 30.181	3638.721 ± 11.239	1968.107 ± 27.598	3614.023 ± 14.199
Complex	82.748 ± 2.003	168.452 ± 1.336	65.097 ± 0.575	147.952 ± 0.113

Table 8: Deserialization times, in microseconds

takes 144 bytes. However, the per-object size of a tree, 2026, is not $15 \times 144 = 2160$. The reason is that the leaf nodes of the tree have null left and right pointers, and null has a more compact representation than a *Node* reference.

In Table 7, we show how long it takes to serialize these objects, in microseconds. The first two columns show the time it takes when the class information is also written to the serialization stream. The second two columns show the time taken when the class information has already been written to the stream. The difference is the time it takes to write that class information. We serialize to an array of bytes in memory, to avoid filesystem access costs.

In Table 8, we show how long it takes to deserialize these objects, or reconstitute them from an object stream, in microseconds. The first two columns show the time it takes when the class information has not yet been read from the serialization stream. The second two columns show the time taken when the class information has already been read from the stream. The difference is the time it takes to read the class information, and find the associated class. We reference each object type before the timing loop, so that class loading time is excluded from the results shown in the table. We also deserialize from an array of bytes in memory, to avoid filesystem access costs.

4.2 Kan Thread Costs

The asynchronous method calling capabilities of Kan provide a simple, powerful mechanism for introducing concurrency into Kan programs. In this section, we investigate the performance of the Kan asynchronous method calling mechanism, and study the effects of some optimizations.

4.2.1 Kan and Java RMI

The following steps are one obvious way of supporting the semantics of a Kan asynchronous method call:

1. Create a *Future* object to hold the method results, and assign it a unique ID.

Method type	Local		Remote	
	JIT	No JIT	JIT	No JIT
Void, void	425.877 ± 1.167	494.176 ± 4.663	2697.359 ± 7.374	4406.865 ± 44.523
1 int, void	428.845 ± 1.399	491.748 ± 1.720	3493.557 ± 23.673	5791.551 ± 11.388
32 ints, void	441.200 ± 1.086	507.777 ± 0.943	6627.224 ± 108.093	11396.879 ± 19.264
int[32], void	427.352 ± 2.314	491.322 ± 1.991	3342.132 ± 13.550	5432.092 ± 20.557
1 Node, void	430.730 ± 0.406	489.430 ± 0.773	3626.030 ± 9.047	6299.875 ± 21.053
1 Tree, void	431.037 ± 0.813	492.294 ± 2.076	5400.828 ± 43.591	9457.182 ± 25.846
1 Node, int	432.788 ± 1.449	498.200 ± 2.013	4828.872 ± 5.956	8534.941 ± 33.449
1 Tree, int	433.987 ± 2.213	499.307 ± 0.964	6615.240 ± 11.882	11658.575 ± 13.692

Table 9: Kan async method calling time, in microseconds

Method type	JIT	No JIT
Void, void	1315.047 ± 11.967	1653.024 ± 23.128
1 int, void	1376.068 ± 39.148	1704.321 ± 34.385
32 ints, void	2078.981 ± 7.157	2964.234 ± 5.952
int[32], void	1822.955 ± 8.861	2380.704 ± 6.786
1 Node, void	2116.841 ± 7.291	3258.233 ± 23.282
1 Tree, void	4070.608 ± 7.000	6519.519 ± 69.966
1 Node, int	2149.841 ± 12.548	3330.587 ± 51.431
1 Tree, int	4127.767 ± 29.583	6539.649 ± 70.854
Kan style args	4694.192 ± 16.485	8745.290 ± 87.538

Table 10: Java RMI times, in microseconds

2. Determine whether a local copy of the called object exists.
3. If no local copy exists, then:
   (a) Marshal the arguments, method name, object ID, and *Future* ID.
   (b) Send an invocation request to some copy of the object.
   (c) Unmarshal the arguments and other information on the remote node.
4. Create a thread to make the method call.
5. Using reflection, make the method call.
6. On completion of the call, if the call was remote:
   (a) Marshal the result, whether normal or exceptional.
   (b) Send the result and the *Future* ID back to the originating node.
7. Store the result in the *Future* object and allow any pending join actions to complete.

We implemented this scheme in Kan. The results are shown in Table 9 for a variety of method signatures. Each signature is described as the parameter types, followed by a comma and the return type.

For comparison purposes, we also measured the costs of making a remote method call with Java's Remote Method Invocation (RMI) package, using the same method signatures. The results are shown in Table 10. These numbers are significantly lower than those for Kan, in Table 9. Part of that is due to the

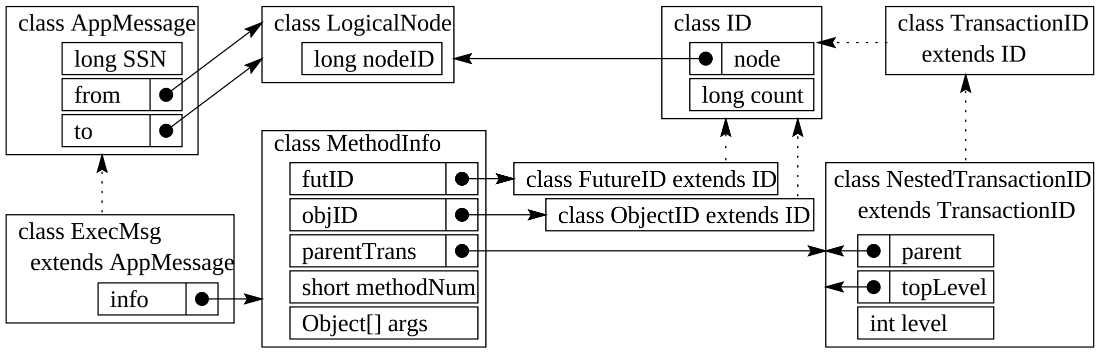

Figure 7: Kan ExecMsg structure

fact that Kan does more than RMI. RMI gives the ability to access *remote* objects, that is, those at known network locations; however, it gives no support for *distributed* objects, that is, those that might migrate or be replicated. Such support must be built on top of RMI.

Consider the data structure shown in Figure 7. It shows the structure of a method invocation request (an *ExecMsg*) in Kan. The solid lines indicate object references, and the dotted lines indicate the superclass relation. Each class involved in the sending of an *ExecMsg* is shown, along with its fields. The *LogicalNode* class is a node identifier as described in Section 3.1.3. The *ID* class is a generic superclass of all identifier types, as described in Section 3.2.1; it contains a logical node reference and a unique counter. The *AppMessage* class is a generic superclass of messages sent in the context of some user application (as opposed to system messages, such as those in support of fault tolerance). Such messages are described in Section 3.1.4. They contain a unique identifier (the *SSN*), and identifiers for the sending and receiving nodes.

The *ExecMsg* itself contains a *MethodInfo* object, which tells how to invoke the desired method. It contains a *FutureID*, which identifies the target *Future* for this method call (see Section 3.3.3). The *MethodInfo* also contains an identifier for the object on which the method is to be invoked. If this method call is being made in the context of a nested transaction, that information is passed along as well. Finally, there is an identifier for the method to invoke, and an array containing all of the method arguments.

We tested the performance of Java RMI on a graph of objects with identical structure. The result is shown in the last line of Table 10. Note that it took significantly longer to make this method call than with the other method signatures. In fact, Kan has better performance on a null method than RMI in this case. This means that RMI not only lacks some tools for distributing objects, but that such tools can be implemented on top of Java sockets more cheaply than they can be on top of RMI. This further limits the effectiveness of RMI for providing distributed solutions.

To see where the time is spent, we broke a remote method call (on a method with no parameters) down into steps. Table 11 shows the names of the methods inside of Kan through which execution flows during a remote method call invocation. Note that the total is about 1 millisecond more than the times shown in Table 9. This is due to the necessity of passing timing information through the system. In particular, the message sending costs are inflated in Table 11, since an array of 8 *long*s is added to each message. An analysis of these figures shows that several hundred microseconds is being spent on the creation and starting of a new thread for each asynchronous method call. If we could remove that cost completely, nearly half of the remaining time (about 138 microseconds) would be paid in reflection and context switch costs. If we were able to eliminate those costs, we would still be 2 orders of magnitude worse than the native Java method calling costs reflected in Table 2. To deal with these costs, we implemented three optimizations, which we describe and measure in the remainder of this section.

Kan action	JIT	No JIT
KanSystem.`invokeMethod`	27.200 ± 0.126	36.004 ± 0.079
ThreadScheduler.`createThread`	29.776 ± 0.078	45.094 ± 0.151
Serialize & send *ExecMsg*	2204.444 ± 4.380	3700.190 ± 8.809
ExecMsg.`handle`	29.076 ± 0.121	36.848 ± 0.407
ThreadScheduler.`makeLocalCall`	41.172 ± 0.484	50.529 ± 0.584
AsynchCall.`execute`	8.683 ± 0.026	9.892 ± 0.139
ThreadScheduler.`done`	6.435 ± 0.027	4.543 ± 2.773
Serialize & send *DoneMsg*	899.991 ± 5.321	1324.733 ± 16.784
DoneMsg.`handle`	29.501 ± 0.132	34.857 ± 0.085
ThreadScheduler.`localDone`	22.683 ± 0.100	17.737 ± 1.945
Joining overhead	69.753 ± 0.342	67.717 ± 0.205
Total	3418.940 ± 6.989	5409.849 ± 22.112

Table 11: Kan remote method calling time breakdown, in microseconds

4.2.2 Adaptive Thread Pool

Thread creation and startup costs can be largely eliminated with a simple structure. We maintain a thread pool, to which unused threads are returned while idle. If there is an available thread in the pool, we avoid the cost of Java thread creation, and we pay the startup cost only once per thread. However, there is a cost associated with maintaining such a thread pool. Each thread has a stack; therefore, inactive threads in the pool are holding onto memory resources. Therefore, we have to balance our desire to have a large pool (to severely curtail or eliminate the possibility that a thread will be created) with our need to have enough memory for the user application to run.

We therefore implemented an *adaptive* thread pool, as originally described in [53]. The pool tries to maintain a size that matches the application's current needs. It monitors the number of idle threads in the pool across a fixed number of pool operations, and then *steps* the pool size up or down if needed. Between steps, if a thread is needed and the pool is empty, then a thread is created. If a method call completes and the pool is not full, the thread is added to the pool. If the pool is full, the thread is discarded. At each step, if threads were created during the previous interval, then we step up the pool size. If the pool was never empty during the previous interval, then we step down the pool size, as long as it does not fall below a fixed minimum. The step frequency, step size, and minimum pool size are all tunable parameters of the system.

Table 12 shows the number of active threads, idle threads in the pool, and the pool capacity for a run of the system in which the program forks 200 threads, then immediately joins all of them. The minimum pool size is 50 threads; the interval is 40 pool actions, and the step size is 20% of the original pool size. Note that the application made 180 method calls before beginning to join any, and then made the remaining 20 calls after system activity had tailed off somewhat. The pool capacity tracked the speedy rise in demand, and tracked somewhat less closely the fall-off in demand.

We measured the performance of our simple method calling microbenchmark with the thread pool active. As shown in Table 13, the times are lower for both local and remote method calls (as compared to Table 9. To discover where the time savings was most evident, we also reran our time breakdown experiment. The results, as shown in Table 14, show that the cost of setting up a handling thread for a *DoneMsg* on the receiving side are significantly reduced, and the *ThreadScheduler*.`makeLocalCall` method represents the remainder of the savings.

Thread pools have been implemented in many systems, for the good reason that they can always be implemented more cheaply than the cost of starting a new thread. In some cases, the threading system itself does pooling underneath. Even so, a thread pool like Kan's adds minimal overhead, so the Kan-level pool

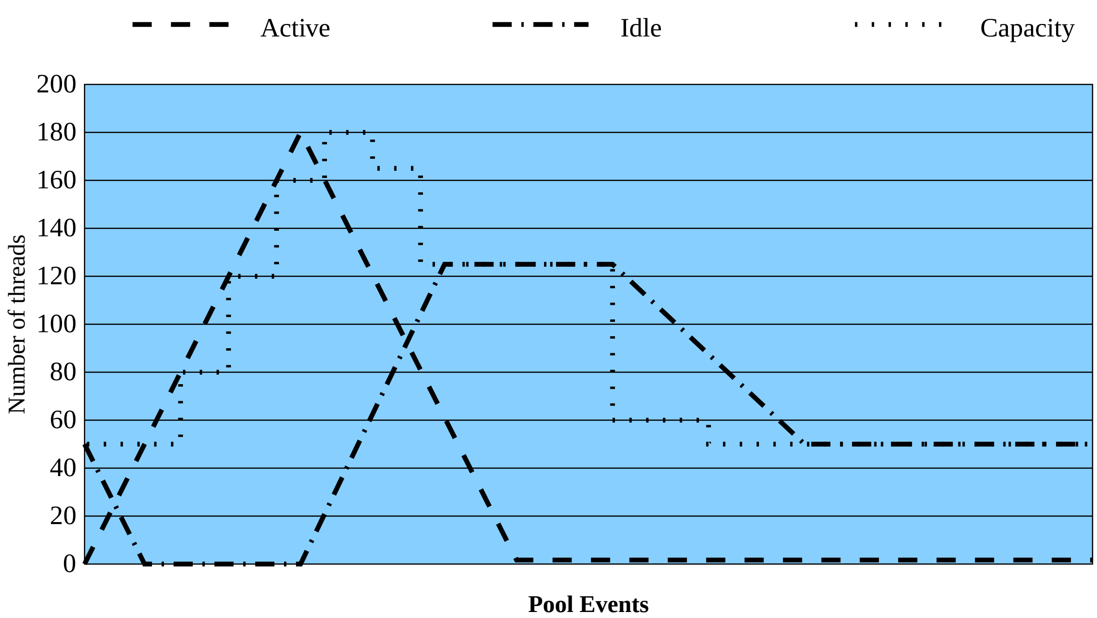

Table 12: Thread Pool Behavior

Method type	Local		Remote	
	JIT	No JIT	JIT	No JIT
Void, void	157.572 ± 0.387	207.784 ± 0.741	2614.537 ± 8.001	4218.637 ± 9.228
1 int, void	159.844 ± 0.681	210.136 ± 0.863	3381.118 ± 18.204	5637.269 ± 10.367
32 ints, void	183.463 ± 3.365	231.509 ± 1.010	6441.988 ± 27.833	11262.247 ± 87.319
int[32], void	161.433 ± 0.216	210.253 ± 0.588	3260.032 ± 68.430	5263.548 ± 9.450
1 Node, void	161.296 ± 0.686	208.641 ± 0.856	3602.406 ± 101.405	6140.749 ± 8.553
1 Tree, void	160.737 ± 0.326	211.577 ± 1.020	5329.746 ± 32.701	9382.398 ± 50.755
1 Node, int	164.719 ± 0.272	214.404 ± 0.756	4722.381 ± 9.435	8393.015 ± 20.906
1 Tree, int	164.620 ± 0.879	216.159 ± 0.648	6512.642 ± 34.961	11510.338 ± 11.233

Table 13: Thread pool performance, in microseconds

Kan action	JIT	No JIT
KanSystem.`invokeMethod`	27.372 ± 0.151	36.103 ± 0.100
ThreadScheduler.`createThread`	34.341 ± 0.355	46.077 ± 0.256
Serialize & send *ExecMsg*	2460.399 ± 14.946	4039.675 ± 33.255
ExecMsg.`handle`	32.770 ± 0.446	36.662 ± 0.145
ThreadScheduler.`makeLocalCall`	27.516 ± 1.189	35.956 ± 0.091
AsynchCall.`execute`	8.964 ± 0.212	9.544 ± 0.029
ThreadScheduler.`done`	6.481 ± 0.086	6.349 ± 0.011
Serialize & send *DoneMsg*	509.286 ± 18.856	826.961 ± 38.457
DoneMsg.`handle`	31.335 ± 0.217	34.975 ± 0.170
ThreadScheduler.`localDone`	21.625 ± 0.058	25.123 ± 0.224
Joining overhead	69.001 ± 0.325	69.727 ± 0.369
Total	3297.462 ± 13.774	5233.583 ± 26.137

Table 14: Kan remote method calling time breakdown, in microseconds

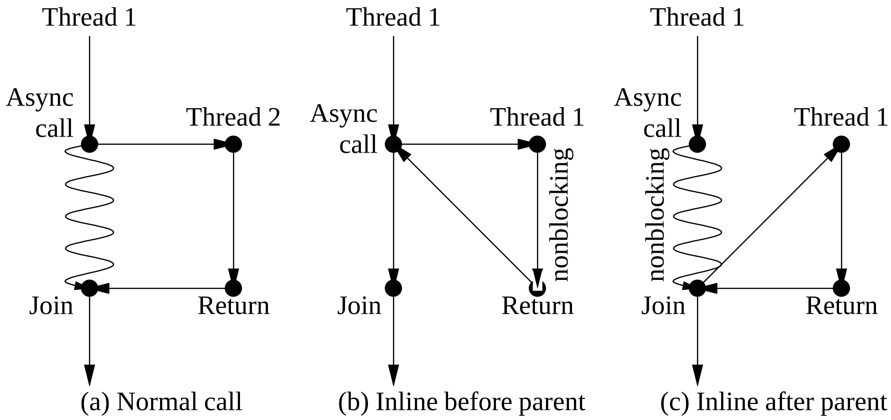

Figure 8: Thread Inlining

does little harm in such a case. In our case, the JDK 1.1 version of the thread pool showed more dramatic effects than the JDK 1.2 version, apparently due to reduced thread startup costs in 1.2. However, even with the reduced benefits, the thread pool is still able to reduce the cost of making asynchronous method calls in many cases and never increases that cost by a significant amount. Hence, the thread pool is always active in Kan.

4.2.3 Thread Inlining

The idea behind thread inlining is to avoid context switch costs. For local method calls, even if the user has asked for an asynchronous method call, if we know that the called method is short and will not block, it may be more efficient to execute it as a synchronous method call. That is, we inline execution of the child into the parent thread. On the other hand, if the parent method takes few steps between forking the asynchronous method call and joining with it, it may be more efficient to execute those few steps first, then make a synchronous call.

These two forms of thread inlining are represented graphically in Figure 8. In Figure 8(a), we show a normal method call. When the asynchronous call is made, a second thread executes the called method while the parent thread continues with its activities, then eventually joins with the child thread. In Figure 8(b), the called method is nonblocking. Hence, the original thread makes the method call immediately. Upon returning, it continues with its activities. The join then becomes a no-op, since the child method already finished executing. In Figure 8(c), the calling method is nonblocking between the call and the join. Hence, execution of the child method is postponed; the call is a no-op. When the join is reached, the parent then executes the child method synchronously.

As shown in [53], implementing thread inlining introduces some overhead into the system, due to the necessary bookkeeping. In fact, if the parent makes only 1 or 2 method calls, the costs of inlining outweigh the savings. However, if the parent thread makes many method calls, then the savings can be substantial. We show the effects of making both 50 inlined child calls and 100 inlined child calls in Figure 9. For each, we made two kinds of calls. The 2-level calls are a single parent thread invoking multiple nonblocking children in parallel. The serial calls consist of a sequence of threads, each of which invokes the next.

As with the thread pool, we found that the benefits of thread inlining were reduced after switching to JDK 1.2. Indeed, uncontrolled inlining could potentially remove all concurrency in an application, resulting in a serial execution. For these reasons, it is not always appropriate to invoke the thread inlining code. Future work on Kan will include compile-time analysis to identify candidates for thread inlining, and to push some

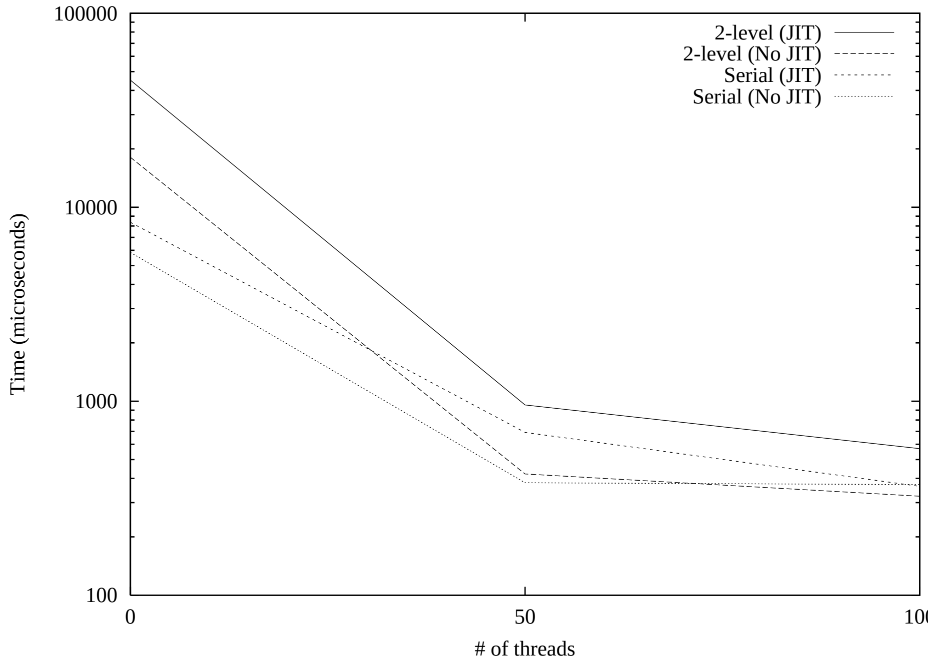

Figure 9: Thread Inlining Performance

of the cost of inlining from run-time to compile-time.

4.2.4 Pointer Swizzling

The idea behind pointer swizzling [55] is that a nonlocal object must be accessed via a global pointer, but that a local object can be accessed via a direct local pointer. Even local accesses via a global pointer are often expensive enough to make direct access desirable. Whenever an object becomes local (via replication or migration), local copies of its global identifier are changed, or *swizzled*, into direct pointers. Whenever an object ceases to be local, local direct pointers are changed into global identifiers. Pointer swizzling has been used in many database systems, to convert disk identifiers into memory addresses, and has also been used in a few programming languages, such as E [54].

In Kan, we use pointer swizzling to avoid using reflection on local synchronous method calls. Instead, we directly call the method. To make a direct method call, not only must the object be local and the call synchronous (to avoid blocking the parent thread), but the call cannot be part of a nested transaction. Because nested transactions may be aborted and rolled back, Kan takes special actions during such transactions to enable it to restore the states of modified objects (see Section 3.4). A direct method call bypasses those special actions.

Pointer swizzling eliminates almost all the cost of making a local Kan method call, leaving about 5.6 microseconds of overhead on top of the Java method call, as compared to the times shown in Table 2. The performance figures are shown in Table 15. Note that the JIT tends to cause slightly degraded performance, as is the case with several other microbenchmarks in this section. When objects are colocated, this optimization provides dramatic improvements in performance. To maximize the impact of this optimization, future work on Kan includes the development of compiler analyses to determine a call graph on the runtime objects, allowing the system to intelligently colocate objects so as to maximize local accesses.

Method type	JIT	No JIT
Void, void	5863.988 ± 8.926	5493.310 ± 5.781
1 int, void	5782.294 ± 107.449	5722.300 ± 46.449
32 ints, void	6003.146 ± 115.627	6354.307 ± 41.959
int[32], void	5854.223 ± 50.400	5663.918 ± 24.147
1 Node, void	5915.389 ± 24.561	5775.324 ± 11.757
1 Tree, void	5877.456 ± 49.694	5787.981 ± 13.828
1 Node, int	6024.161 ± 21.243	5805.999 ± 8.911
1 Tree, int	6138.235 ± 19.349	5864.142 ± 10.906

Table 15: Pointer Swizzling Performance, in nanoseconds

4.2.5 Final Comparison

In this section, we show the effects of all of our optimizations on method calling. In Figure 10, we show the relative costs of making method calls to methods without parameters or return values, using the systems and optimizations described above. These are the figures garnered while using the JIT, as listed in the tables earlier in this section. As illustrated in this figure, the optimizations we applied led to local method calls paying an additional cost of less than 6 microseconds over Java method calls. This is a constant amount of overhead. Methods that have parameters, return values, and nontrivial bodies will have execution times in the tens of microseconds or more, making the difference between a Java call and an optimized Kan call of little consequence.

Remote method calls in Kan are only slightly worse than their RMI counterparts, but provide more functionality. Indeed, when the same objects are transmitted with RMI, RMI takes longer than Kan to make the same method call. This suggests that RMI is not the best solution to distributed problems when replicated or migrating objects are desirable features.

4.3 Transaction Costs

Transactions provide a powerful mechanism for ensuring the atomicity of actions that begin in a known state. However, that power comes at a price. Simple local transactions can be quite cheap, especially if contention for the object is low. However, deeply nested transactions spanning multiple nodes can be quite expensive. The cost of a transaction is affected by the following factors:

- Locality: local transactions are significantly cheaper than transactions that span nodes.
- Number of objects: each object touched by a transaction must be locked.
- Depth: each level of a nested transaction must coordinate with the levels above it to provide linearizability of the whole. Increasing depth brings increasing costs.
- Type of transaction: this is described in the next paragraph below.
- Access type: reading transactions acquire shared locks, while writing transactions acquire exclusive locks.
- Rollbacks: aborted transactions and rollbacks represent wasted work that consume resources without providing any benefit.

To support rollbacks, we make a copy of each object touched before the transaction commences, as described in Section 3.4. One optimization we implemented to reduce costs is to differentiate between three

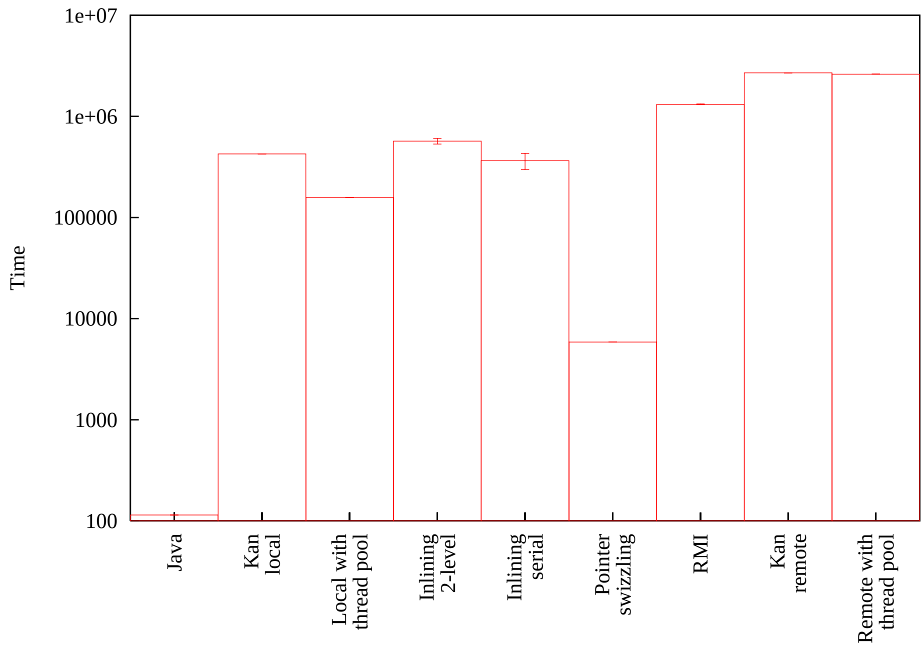

Figure 10: Method calling comparison, in nanoseconds

Access type	Blocking		Nonblocking	
	JIT	No JIT	JIT	No JIT
Read	2.248 ± 0.041	3.860 ± 0.046	2.254 ± 0.032	3.862 ± 0.120
Write	1.922 ± 0.041	3.229 ± 0.069	1.911 ± 0.046	3.252 ± 0.049

Table 16: Transaction type test, in microseconds

kinds of transactions: nonblocking, blocking, and nested (in order of cost). A nonblocking transaction is one which does no joins and throws no exceptions, so we can be sure that it will complete normally. We optimize in that case by skipping the object copy, since it will never be used. A blocking transaction is a single-level transaction that joins with some future or might throw an uncaught exception. In that case, we make a copy. Both nonblocking and blocking transactions are single-object transactions, so they can never be involved in a deadlock. Therefore, we we optimize again by excluding such transactions from the wound-wait algorithm (described in Section 3.4.4).

We performed several experiments, to see the effects of varying the cost factors listed above. The first experiment was to determine the effectiveness of our copying optimization. For this test, we executed calls on a method consisting of a single transaction. We tried all four combinations of read/write and blocking/nonblocking transactions, and measured the total time (including the method call). The results are shown in Table 16. This shows the effect of skipping the copy on a very small object. In this instance, the difference is not statistically significant. However, the gap increases with object size, as the cost of copying increases. We varied the object size to see the effects on writing transactions, as shown in Figure 11. Since the nonblocking transaction is performing the same actions every time, its cost does not change. However, the cost of making an object copy rises with object size. We cannot measure the in-memory size of a Java object directly; the object sizes in the figure represent the serialized size of an object, which, in general, is larger than the actual memory footprint of the object.

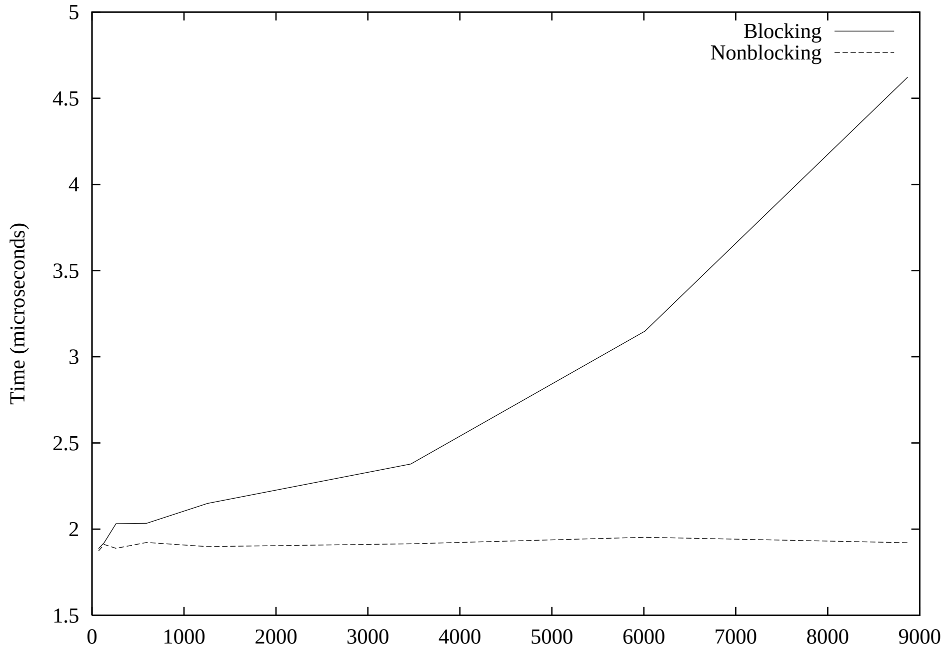

Figure 11: Object copy optimization

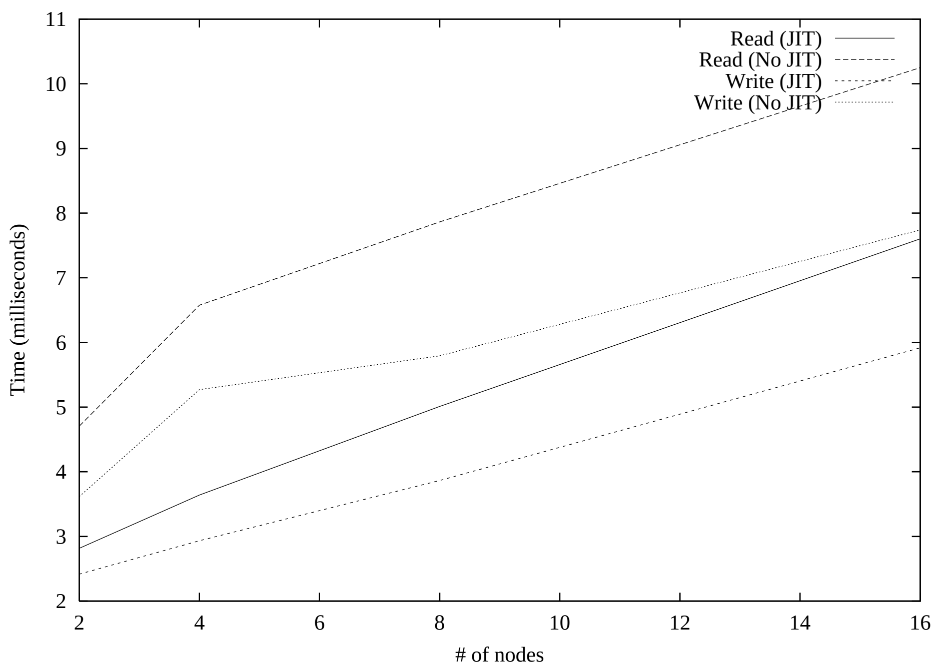

Figure 12: Transaction locality test

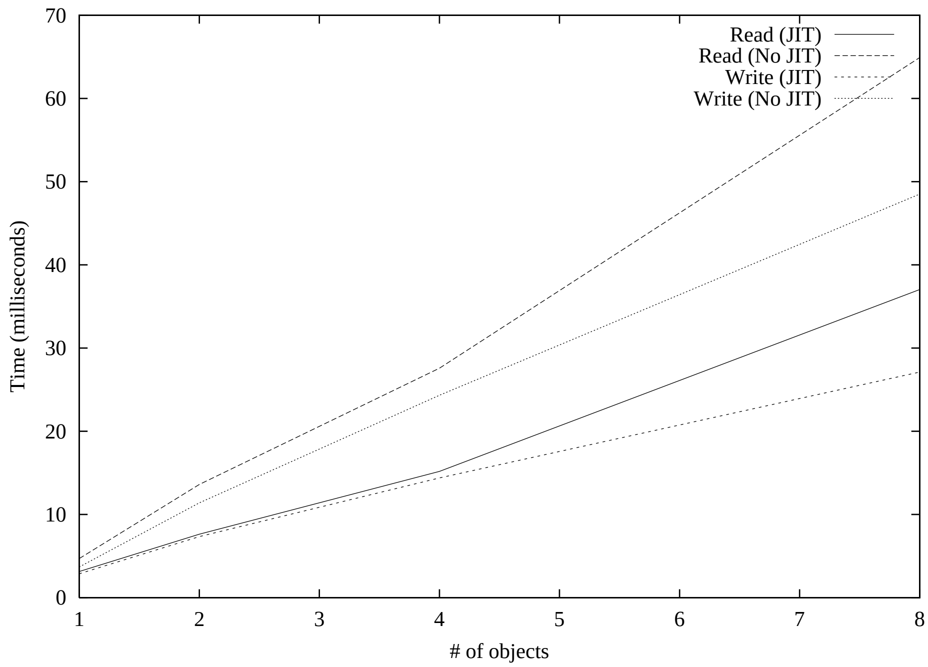

Figure 13: Transaction object number test, 2 nodes

The next experiment was to determine the effects of nonlocal objects on transaction performance. For this test, we spread the objects to be accessed by a nested transaction across varying numbers of nodes. (Note that a nested transaction is blocking by definition, since the wound-wait algorithm may abort it.) Each is a 2-level nested transaction that operates on one object per node. The results are shown in Figure 12. The cost of a non-local transaction rises in nearly a straight line for all four variations. This is a result of the configuration of our LAN, where access to any remote node costs about as much as access to any other. The cost reflected here is that of accessing remote objects at all; the precise location of those remote objects is of little consequence. The results would be very different on a nonuniform network.

Next we varied the number of objects accessed by each nested transaction. We ran a 2-level nested transaction, where the top-level transaction spawned $N - 1$ children, each accessing a different object, where N is 2, 4, 8, and 16. The results are shown in Figures 13 and 14. For two nodes, the cost of increasing the number of objects is once again nearly a straight line. This is due to the necessity of acquiring a lock on each such object. However, for 16 nodes the picture has changed. Now, larger numbers of objects apparently result in reduced cost in some cases. This effect is the result of an optimization in Kan. When a nested transaction spans multiple nodes, the 2-phase commit messages are batched together by node. Thus, expensive network communication rises most significantly with the number of nodes involved in a transaction, and with the number of objects only to a lesser extent.

Next we assessed the impact of deeply nested transactions. Such transactions hold many object locks simultaneously. They also increase the overhead for managing object consistency within the nested transaction itself. We varied both the depth and the number of nodes across which the accessed objects were spread. The results are shown in Figure 15. After some rapid growth in latency going from 2 to 4 nodes, the growth curve slackens off. Even with transactions nested to 8 levels deep, Kan scales nicely to 16 nodes with the JIT in effect, the limit to the number of homogeneous machines we have on our LAN.

Finally, we assessed the impact of rollbacks on transaction throughput. For this test, we wrote a transaction that always fails with a user exception, giving the user the opportunity to handle the abort. We cause

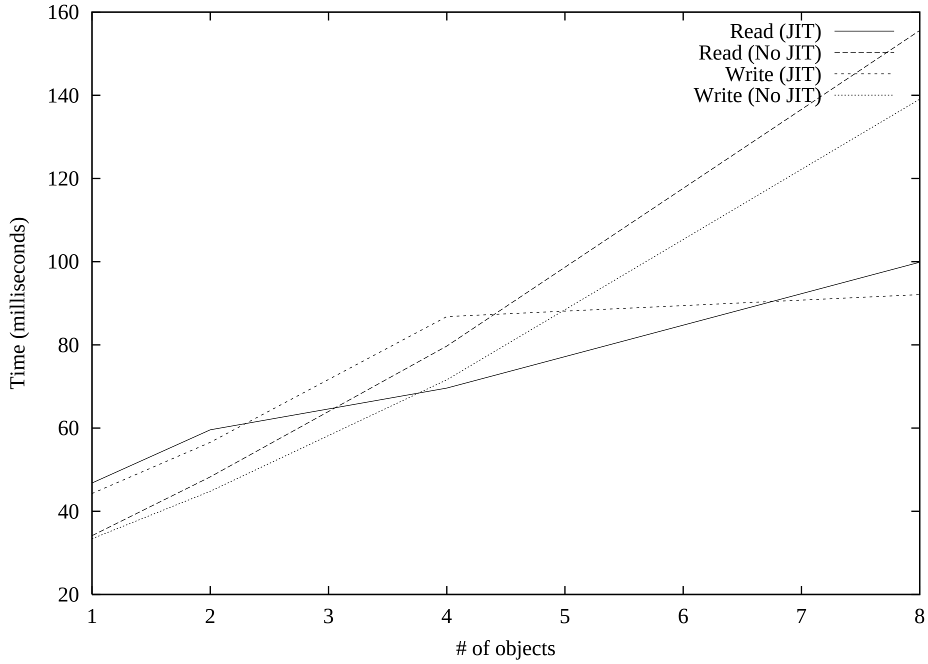

Figure 14: Transaction object number test, 16 nodes

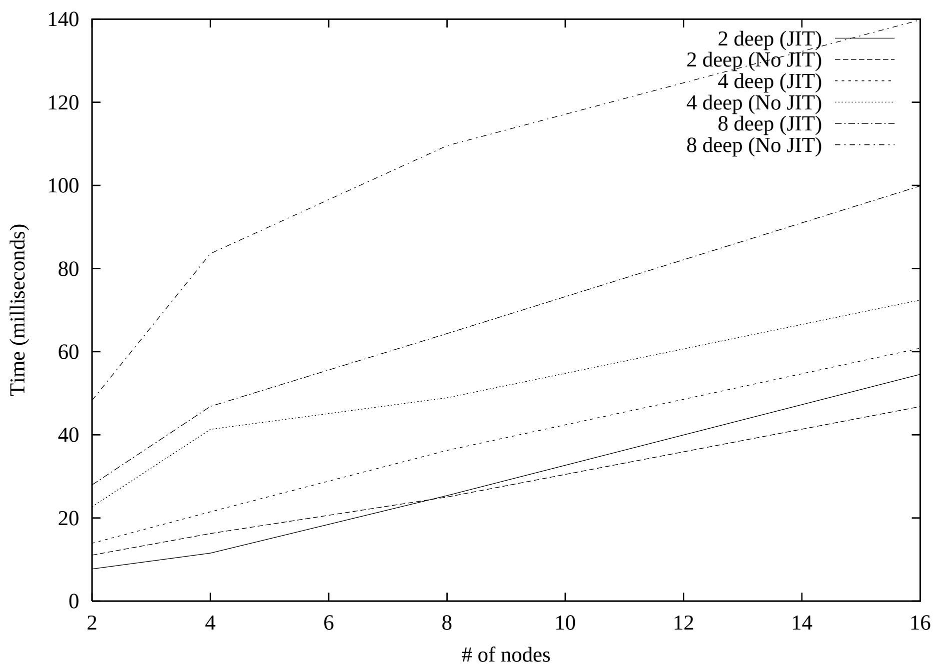

Figure 15: Transaction depth test

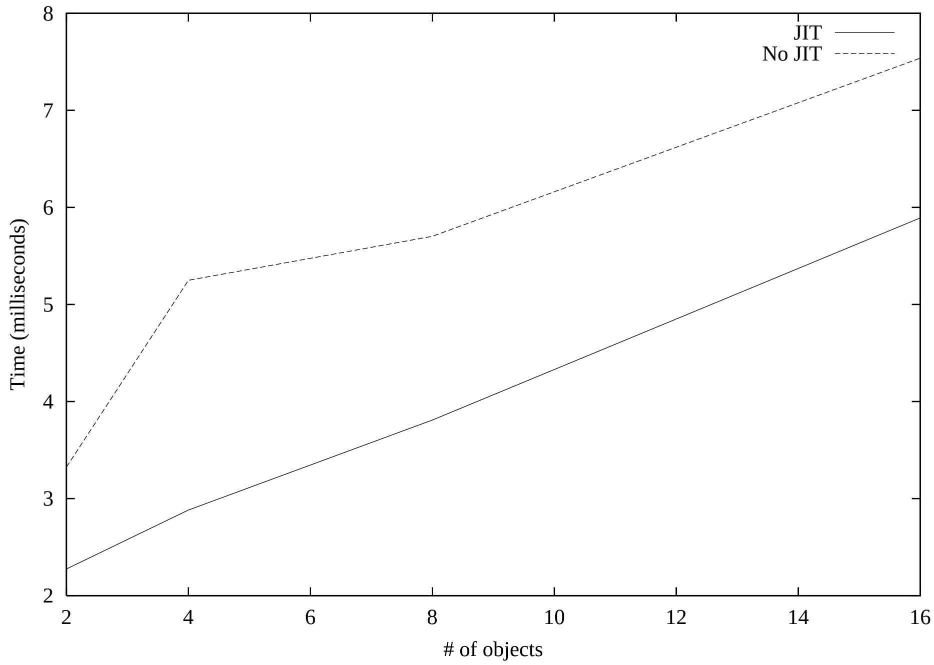

Figure 16: Transaction abort test

the abort inside of a 2-level nested transaction, which has already accessed N objects, N equal to 1, 2, 4, 8, or 16. This test measures the time to acquire a lock on the object, signal the abort, restore the original state of the object, and release the lock. In fact, restoring the original state is an extremely cheap operation, since we copied the entire object before beginning. Therefore, the cost of an aborted transaction is always cheaper than that of a committed transaction on the same object. For that reason, the results of this experiment are very similar to those obtained in the locality test of Figure 12. The results of our abort test are given in Figure 16. Once again, we do not see an upturn in the curve up to 16 nodes, indicating that Kan is well suited for LANs of such size.

4.4 Object Types

Reducing distributed object management costs rests largely on this principle: maximize the number of accesses that are local. An optimal scheme would place a copy where the object is about to be read, and would reduce the number of copies to one before every write. In reality, optimal schemes are hard to approach, because different objects are accessed in different ways.

The Munin [16] shared memory system managed shared variables depending on user access patterns, thereby yielding substantial gains in efficiency. In this section, we describe how Kan manages shared objects in a similar fashion to raise the percentage of method calls that are local. Currently, Kan implements only two object typing schemes, general and migratory. In the future, we plan to investigate other object types that can be implemented in Kan.

4.4.1 Migratory Objects

Some objects are accessed by only one node at a time, due to external synchronization. As an example, consider a distributed quicksort, in which an array of numbers is divided into subarrays which are assigned

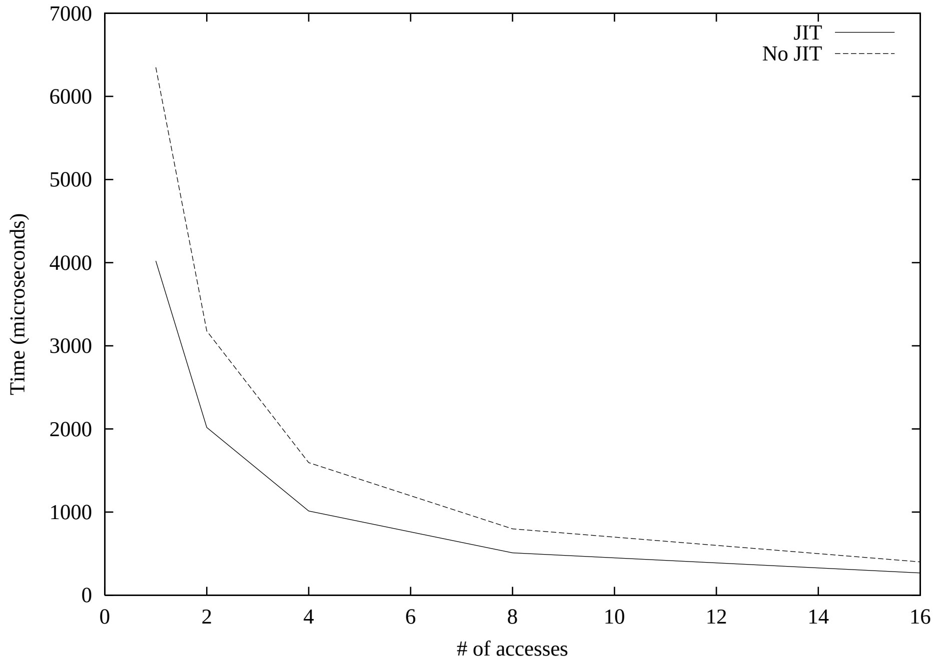

Figure 17: Migratory object typed as migratory

to various nodes for sorting. While the subarray is being sorted, only one node is accessing it. Afterward, it is passed back to the parent node for merging with the other sorted subarrays. The merge process then also takes place on a single node. Objects that are accessed by a single node at a time in this manner are called *migratory*.

Migratory objects are managed with an owner-based protocol, which was independently developed by us and Herlihy and Warres [27]. A remote access causes the transfer of the entire object to the accessing node, which becomes the new owner. Objects also have a home node, which is only used for locating them. A non-local access to a migratory object passes through the home node, and then on to its final destination. When a migratory object is moved, it notifies the home node of the new location of the object. The notification is asynchronous, so there is a possibility that the home node will forward a message on to a node that no longer owns the object. For this reason, each node maintains a forwarding pointer when a migratory object moves.

We measured access times for migratory objects that the system has correctly identified as migratory. Such objects suffer three network latencies when the first access is made; one to the home node, one from the home node to the current owner, and one to transfer the object to the requesting node. However, subsequent accesses are purely local, and therefore extremely cheap. In Figure 17, we show the average cost of an access, based on the average number of accesses made by each node before the object is transferred to the next node. As the number of such accesses rises, the cost of migrating the object is amortized over a greater number of operations, leading to better average performance.

On the other hand, incorrectly typing a migratory object as a general object causes the system to miss out on opportunities for making a larger number of local accesses. If many of the accesses are writes, the system may not even replicate the object, requiring all accesses to cross the network. This can lead to a great deal of unnecessary overhead, as we show in Figure 18. Here, a migratory object is accessed with the general protocol. Its accesses are approximately 50% reads and 50% writes. The cost of maintaining this object is significantly higher than it would have been had we used the migratory protocol. In fact, the cost

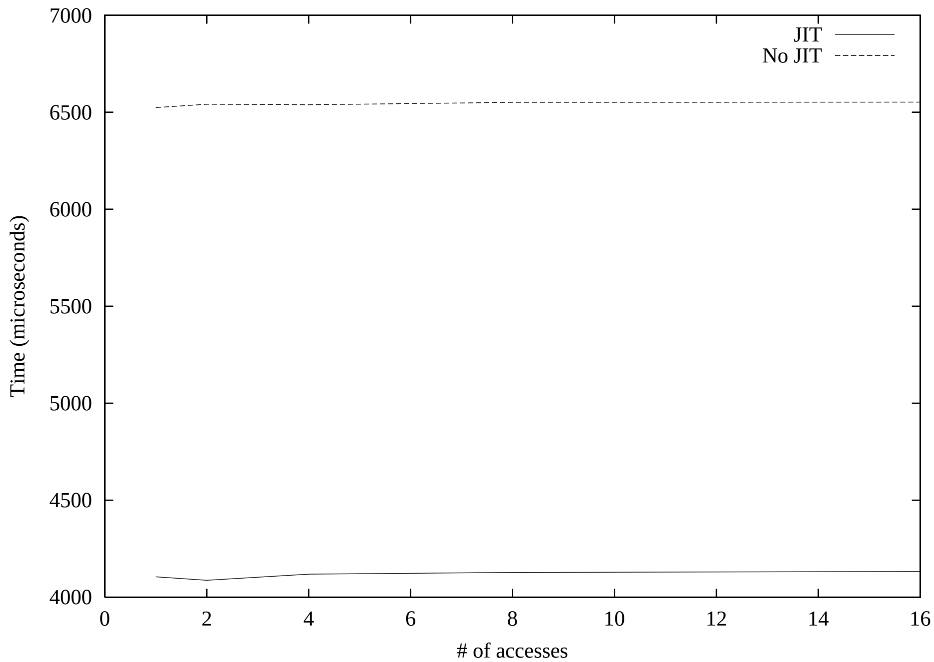

Figure 18: Migratory object typed as general

1:	$\boxed{x.\texttt{set}(1)}$	$\boxed{y.\texttt{read}() == 0}$
2:	$\boxed{y.\texttt{set}(2)}$	$\boxed{x.\texttt{read}() == 0}$

Figure 19: Nonlinearizable history

is approximately equal across the number of accesses, since all accesses are equally expensive.

4.4.2 General Objects

General objects are those that have not been classified as any other type. They are managed with a replicating home-based protocol, developed by Lee [32] from a replication scheme described by Wolfson, Jajodia, and Huang [56]. Objects are permanently located at a home node, chosen when the object is created. Temporary replicas may exist at other nodes. Read accesses are local, if a local copy exists; otherwise, they are sent to the home node. Write accesses always go to the home node, which serializes them and sends them to the replicas.

Furthermore, to ensure linearizability, the writing node must wait until all replicas have been updated. Otherwise, a scenario like the following can occur. Suppose that x and y are integer objects, initially containing zero. Suppose further that node 1 is the home node of x, and also holds a replica of y, and that node 2 is the home node of y, and also holds a replica of x. Consider the history of Figure 19. First node 1 writes to x and node 2 writes to y, but neither waits until the replicas have been updated. Before that update takes place, each reads its replica of the other object. Since read operations are purely local, each reads the initial value, zero. The resulting history cannot be linearized (or even serialized, for that matter).

Replicas are created at nodes that issue many read requests. An analysis given by Lee in [32] shows that a replica is desirable when 3 times the number of local reads exceeds the number of remote writes. However,

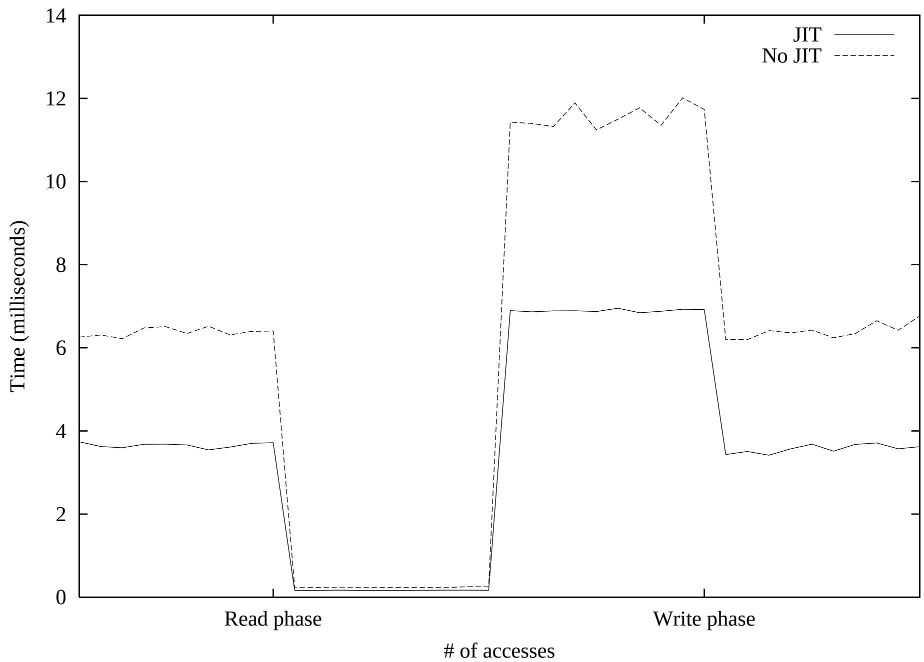

Figure 20: Replication of a general object

we want to avoid the network traffic necessitated by continually sending access counts to the home node. The scheme we use is to count writes at the home node (since all write operations are serialized by the home node anyway), and report the current write access count to each replica as it is updated. Read counts are maintained at each copy. The home node notices when the read count for a node is high enough to warrant a replica, and sends that replica. Each replica then watches its own access count, and removes itself if the number of local reads becomes too low. To avoid thrashing, where a replica is repeatedly created and removed, we do not examine the replication scheme on every access. Instead, we choose a period (which is a tunable parameter of the Kan system), and adjust the replication scheme after that many accesses to the object.

To show the effects of replication, we access a general object in phases, where each phase consists of solely read accesses or solely write accesses. A replica is created part way through the read phase, after the next replication period elapses, resulting in cheap local accesses for the rest of that phase. When the write phase begins, the cost of the operations initially goes up, since the system is now keeping 2 copies of the object consistent. Eventually, after another replication period elapses, the system removes the replica, thereby cheapening the writes in the remainder of that phase. Our results are shown in Figure 20.

As with migratory objects, incorrect typing can have enormous impacts on the performance of the system. We forced a general object (one which is accessed randomly by the nodes of the system) to be managed with the migratory protocol. The result is that extra object movement costs are paid on nearly every access. In fact, almost all accesses result in the sending of several messages. First, the home node is contacted to find the current whereabouts of the object. Then the home node forwards the message to the last owner it knew about. However, many messages have to be forwarded since the object is constantly moving. Once a request finally reaches the object, the object migrates to the requesting node, and the access is performed. The results are shown in Figure 21. Note the general upward trend. This is a result of an increasing backlog, increasing the size of the request queue transmitted with the object.

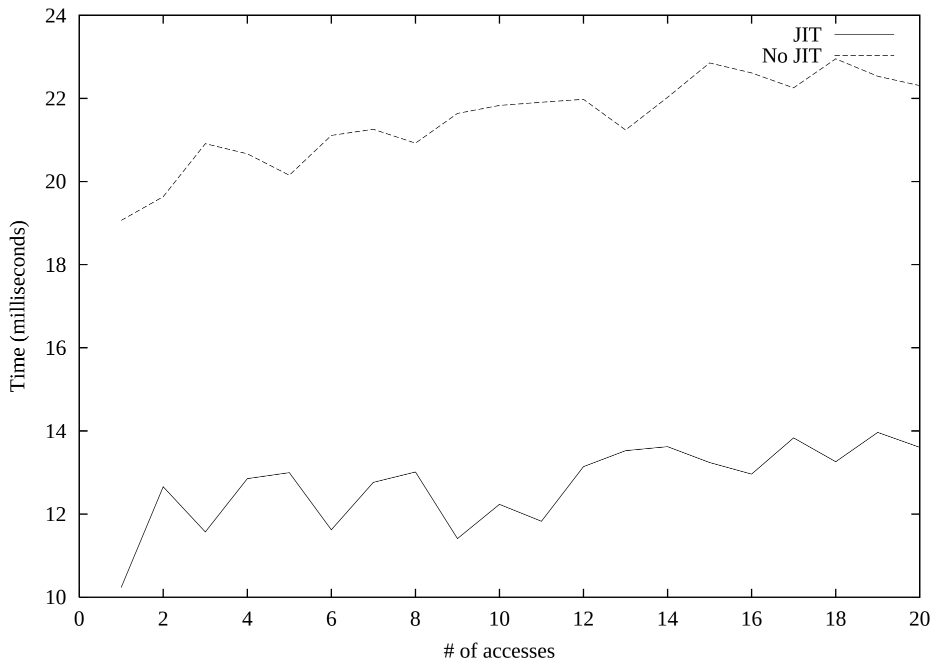

Figure 21: General object typed as migratory

4.4.3 Type Detection and Conversion

Detecting the type of an object statically (i.e., at compile time) is difficult. In some cases, it is impossible, since the type of the object may depend on user actions. For example, in a distributed editor, shared files may be encapsulated as objects. Their types depend on whether those sharing them access them sequentially or concurrently.

Therefore, we use a dynamic (i.e., runtime) type detection scheme. Objects are initially assigned the migratory type. Access statistics are gathered as in the description of general objects above. However, during each time period, we also record whether any concurrent accesses were detected. This happens if a request arrives while another is being serviced. At the end of each time period, we recompute the object type with this algorithm:

If there are no replicas then
    The object is migratory iff there were no concurrent accesses
Else if there is exactly one replica then
    The object is migratory iff there were no local accesses on the home node
Else
    The object is general

If the object type changes, then we must switch between an owner-based and a home-based protocol. Note that switching from a home-based to an owner-based protocol implies that there is at most one replica. In that case, the holder of the replica (or the home node if there is no replica) becomes the owner. In the other direction, we maintained a home node for migratory objects for locating purposes, so we simply drop the owner information.

4.5 Applications

To determine the impact of the Kan optimizations on real applications, we implemented several and ran with the optimizations enabled and disabled. The thread pool provides uniform improvements, of varying degrees, depending on the number of concurrently executing threads spawned by any one application. Those that spawn only a few at a time always found a thread waiting in the pool, resulting in the greatest overall benefit. On the other hand, some parallel applications, such as matrix multiplication, tend to fork a number of threads initially, then run the entire application with just that set of threads. Such applications see a small reduction in the startup time, but receive no benefit from the thread pool during the main part of the execution.

Pointer swizzling was found to be effective for a distributed quicksort application. An array of numbers is to be sorted. At each step, a number, called the *pivot*, is chosen. Then the array is divided into subarrays, one containing all numbers less than the pivot, one containing all numbers equal to the pivot, and one containing all numbers greater than the pivot. Quicksort is then recursively called on the first and third subarrays. The results are joined together in order to form the final sorted array. The distributed version simply acts on a distributed array, with the stride a parameter that can be chosen by the user. Pointer swizzling helps near the bottom of the sorting tree, as the subarrays to be sorted become small enough to fit entirely within a block assigned to some node. With a small number of nodes, the recursive calls quickly reach a level where all further calls are local. There, executions with pointer swizzling enabled run in as little as 4% of the time of those with pointer swizzling disabled. As the number of nodes rises, the optimization has lesser effect. With 16 nodes, the optimized execution time was approximately 60% of the unoptimized execution time.

Object typing helps with a distributed B-Tree application. B-Trees, in their various forms, are useful data structures for finding data that is too large to fit into a computer's memory. We adapt them to another use: find data that is distributed across a system. We use a modified B^*-Tree, called the $\mathrm{B}^{\mathrm{link}}$-Tree [33], for this purpose. Each node in the tree contains a link pointer to its sibling to the right (if any). When searching through the tree, if we find that the highest key in the currently visited node is lower than the search key, we follow the link pointer to the right to continue the search. Node splitting is effected by locking the node to be split, creating the new node, setting the right links appropriately, redistributing the keys, inserting the newly created node into the parent node, and then releasing the lock on the child. This design allows searchers to continue their operations while node splitting is taking place. The tree nodes may be either migratory or general, depending on dynamic searching and inserting behavior of the users. We simulated both kinds of behavior. We found that the adaptive typing code was able to reduce the cost of multiple searches for the same key by replicating the nodes from the root down to the node containing the key. On the other hand, if the distributed system machines took turns heavily accessing a small range of keys, the nodes holding the keys in those ranges tended to be identified as migratory, and would move to the site of activity, reducing the cost of such accesses. In the first case, performance improved by about 16% over the execution without replication. In the second case, performance improved by about 24%, since even the expensive writes were purely local operations.

5 Related Work

The choice of primitives adopted in Kan were motivated by a number of concurrent object-oriented languages and systems proposed recently. We discuss some of those languages and system in this section. We treat the Java-based systems separately from the others.

5.1 Java-based Languages and Systems

A number of recent products (e.g., Sun's JDBC) provide a means of connecting Java programs to databases via a standard SQL interface. The Java program has access to a stable persistent store, and can perform database manipulations and queries. In contrast, Kan does not have persistent objects (but see Section 6), but gives a full general-purpose programming language for constructing object queries and manipulators.

5.1.1 The Aleph Toolkit

The Aleph Toolkit [25] provides common primitives and functionalities needed by distributed systems. Its intended use is as a substrate for heterogeneous distributed systems. Aleph programs can start threads on remote processors and join with them, share objects among threads running on different processors (with synchronization and caching handled by the system), and execute simple single-site transaction. Only single-node transactions are supported, without guards or nesting. The coherence model is based on transactional memory [26, 48]. It guarantees sequential consistency, and provides read, write, optimistic read, and optimistic write accesses.

The code organization and general approach of the Aleph Toolkit are similar to the lower layers of the Kan system in many respects, although it does not contain the optimizations discussed in Section 4. The Arrow directory protocol [27] used by the Aleph Toolkit is the same protocol we use to control migratory objects (see Section 4.4.1).

5.1.2 cJVM

One approach to distributing Java programs is to distribute the JVM itself, and run unmodified Java programs on that JVM. This is the idea behind cJVM [4], a JVM for homogeneous clusters of computers on a high-speed network. It was designed to support servers, by distributing the server load across the cluster. Hence, it performs best on applications with a large number of independently executing threads. Like Kan, cJVM transparently replicates objects to improve availability.

5.1.3 Do!

Do! [31] aims to automatically generate distributed code from multithreaded program source code. The Java language is not extended, but the user is given an API for providing hints to the compiler about appropriate mappings of threads and objects to distributed system nodes. The generated code uses standard Java RMI for communication. It uses a runtime library that supports the creationa nd manipulation of remote objects.

5.1.4 JavaParty

JavaParty [46], like Kan, is an extension to Java. It gives transparent remote objects, bypassing the complexity of RMI. It also transparently migrates objects for greater availability. Currently, migration is triggered at runtime when access patterns indicate that it is needed, or it is explicitly invoked by the programmer. Future developments to JavaParty include compiler analysis to statically determine when migration is helpful. Also like Kan, JavaParty compiles down to standard Java bytecode, allowing JavaParty programs to run on any JVM. It also supports easy integration of standard Java class files, compiled externally to the JavaParty system.

The JavaParty project has produced improved Serialization [45] and RMI [42] implementations. However, their solutions are not portable across JVMs, so we do not use them in Kan.

5.1.5 Javelin

Javelin [20] is an attempt to harness the raw processing power available on the Internet. The goal is to run large-scale coarse-grained parallel applications by dividing the computation among a large set of machines connected to the Internet. The machines with an application to be executed are the *clients*. The machines with processing power to spend on an application are the *hosts*. Bringing clients and hosts together is the job of the *brokers*. The entire system is based on a web browser interface. Users who wish to participate as either clients or hosts point their web browsers to a broker and select the appropriate link. Fault tolerance consists of restarting portions of the application that were assigned to faulty processes.

The original Javelin system used Java applets communicating over Java sockets, which use TCP/IP. A later revision of the system, called Javelin++ [41], changed to using Java applications communicating over Java RMI, and made some other changes relating to scalability.

5.1.6 Manta

Manta [50] was built by the same group responsible for Orca (see Section 5.2.11 below). Based on their experience with JavaParty, they implemented an RMI that is improved still farther over that of JavaParty [38]. This RMI's efficiency arises in large part from abandoning the official Sun protocol in favor of a more compact, but less versatile, protocol. Hence, a Manta system has to detect whether it is connected to another Manta system, allowing it to use the compact protocol, or not, forcing it to use the standard Sun RMI protocol.

Much of Manta's performance improvement derives from their implementation of a native compiler, and the whole-program analysis used by that compiler. Furthermore, the compiler takes special actions when compiling RMI code so that JNI (Java Native Interface) calls are avoided. In fact, communication is inlined into the code, increasing the speed and responsiveness of the system still further.

As with JavaParty, Manta's approach is not portable across JVMs, so is not used in Kan.

5.1.7 Nile

The goal of the Nile [47] project is to provide a self-managing, fault-tolerant, heterogeneous system composed of hundreds of commodity workstations, with access to a distributed database whose size is on the order of hundreds of terabytes. The component workstations are distributed across the North American continent. Nile is intended to be easily maintainable, scalable, and provide useful services past its development phase. It is structured to run embarrassingly parallel applications; i.e., those with independent parallel subtasks, such as web indexers. It is written in Java for heterogeneity.

CORBA is used as a data management layer. The database itself is widely distributed, with replication providing some degree of fault tolerance. The failure of a job is automatically detected, and the job is restarted if the failure can be repaired or worked around. The basic operation of Nile is to divide the application into subparts and distribute those subparts to the constituent computing nodes, then collect and collate the results. If a subpart fails, recovery consists of assigning the subpart to a new computing node.

5.1.8 Parallel Java

Parallel Java [29] is an extension to the Java language to support parallel constructs. It is based on earlier work on a C++ extension, called Charm++ (see Section 5.2.4 below). The parallel extensions provide for the creation of remote objects via proxies, with automatic load balancing. Objects with a port on every node are called *object groups*, and allow for easy expression of algorithms requiring global coordination, such as barriers. Parallel Java is part of a larger effort, named Converse, which is aimed at providing multilingual parallel support. That is, Parallel Java programs can interact with parallel libraries written in other languages supported by Converse.

Like Kan, Parallel Java aims to run on any JVM. Therefore, both systems use features of Java not known for good performance, namely Serialization and Reflection. These features provide portable means of accessing remote objects, but are more general than needed for either project.

5.1.9 ProActive PDC

ProActive PDC [15] (formerly known as Java//) is a library for Parallel, Distributed, and Concurrent programming in Java. The idea is to run the same program as a sequential application, a multithreaded single-node application, and a distributed application. Rather than alter the Java language, ProActive PDC is entirely API-based, needing no special compiler or JVM. The programmer provides hints to the system through its API, and also uses the API to get implicit futures (using wait-by-necessity), continuations (a transparent delegation mechanism), and active objects.

Running a program on the different kinds of platforms supported by ProActive PDC is achieved through object composition. The user defines a sequential object, which the system then composes with a proxy and a so-called *body*. The proxy turns local calls into messages, which are decoded by the body. Futures and continuations are provided by creating specialized subclasses of user objects which contain the appropriate code.

5.2 Other Languages and Systems

5.2.1 ABCL

ABCL [57] is an object-oriented parallel language based on Common Lisp in which objects communicate by passing messages over an asynchronous network. There is an associated specification language, ABCM, in which ABCL programs are represented as collections of communicating concurrent objects. Objects are passive entities, taking action only upon receipt of a message. Objects are also allowed to dynamically create new objects. The model is similar to Kan's, although the language design is quite different. ABCL/f, a variant of ABCL, provides asynchronous method calls that return futures, similar to Kan.

5.2.2 Argus

Argus [35, 37] is an experimental language and system that supports the construction and execution of distributed programs. It is an extension of the programming language CLU.

Argus provides actions, or atomic processes, which are serializble. That is, the overall effect of executing multiple concurrent actions is as if they had been executed in some sequential order. Actions are recoverable; the overall effect of an action is "all-or-nothing". A two-phase commit protocol is used to ensure that either all of the changes made by an action occur or none of them do. Nested actions are provided, based on a multi-reader single-writer protocol. Nested actions obey the following locking rules:

- When a write lock on an object is obtained, make a copy of the object's state in a new version, and work on the new version.
- When the action commits, the new version is retained, and the old version discarded.
- A subaction's locks are given to its parent action when it commits.
- When a top action commits, its locks are discarded and its effects become visible to other actions.
- If an action aborts, the action's locks and the new versions are discarded, and the old version retained.

To ensure consistency, the following locking and version management rules are obeyed for a subaction S on object X:

- Acquiring a read lock: All holders of write locks on X must be ancestors of S.
- Acquiring a write lock: All holders of read and write locks on X must be ancestors of S. If this is the first time S has acquired a write lock on X, push a copy of X on the top of its version stack.
- Commit: S's parent acquires S's lock on X. If S holds a write lock on X, then S's version becomes S's parent's version.
- Abort: S's lock and version (if any) are discarded.

Argus guarantees that parent actions never run concurrently with their children. When a top action attempts to commit, the two-phase commit protocol is used to ensure that the new versions of all objects modified by the action and all its committed descendants are copied to stable storage. Deadlocks are broken by a timeout-based resolution mechanism.

5.2.3 Compositional C++

Compositional C++ (CC++) [18] is a superset of C++, adding the keywords sync, global, par, parfor, atomic, and spawn. It provides support for explicit parallelization of programs by way of declaring blocks of code as parallel, in either a synchronous or asynchronous manner. Loops can be declared as parallel as well. Synchronization is provided in the form of atomic methods. The C++ language is extend to provide global pointers to CC++ objects. Each main process is itself an object, which allows processes to operate on one another. The consistency guarantee is cache consistency [3, 23], or coherence, a fairly weak guarantee.

CC++ is a parallel language, in constrast to Kan, which is largely a distributed language. However, the CC++ and Kan share the ability to specify blocks of code as atomic, construct global pointers to objects, and spawn new threads at the language level.

5.2.4 Charm++

In Charm++ [30], the basic unit of computation is the *chare*. A chare can be located on a specific node, or it can be a *branch office* chare, with local components on every node. A number of common modes of information sharing are supported by means of shared variables. Several types are available, including read-only variables, accumulator variables, monotonic variables, write-once variables, and distributed tables. In general, an object can choose its own method ordering protocol through system calls that indicate the messages which the object is willing to receive. It does so by expressing the method protocol dependencies as a directed acyclic graph. However, there is no intra-object concurrency possible and no values can be returned in response to a message (i.e., concurrent methods all have return type *void*). The consistency guarantee is broadcast consistency.

5.2.5 Cilk

Cilk [21] is an algorithmic, multithreaded language that compiles to ANSI C. The runtime system guarantees predictable performance. It features an architecture- and language-independent checkpointing facility based on source-to-source translations. Code is structured into procedures. Each procedure consists of one or more nonblocking threads. The dependencies among threads form a rooted DAG. Shared data consistency is defined on this DAG [9]. Load sharing takes place through randomized work stealing, in which a node with nothing to do asks a random neighbor for a task. There is an online data-race detector, which is intended to be used as a debugging tool.

The Kan model technically allows restricted threads of the sort employed by Cilk. However, the actual implementation cannot use such threads, since threads can enter arbitrary (local) Java code. Furthermore, Kan consistency is defined at the object level, for which the DAG approach is inappropriate, since it requires breaking encapsulation to make all dependencies explicit.

5.2.6 COOL

COOL [17] supports concurrent method execution in an object that adheres to the multiple reader-single writer protocol. This is done by specifying a method to be either of type mutex (writer) or non-mutex (reader). The COOL runtime system then ensures this consistency model by using read and write locks. Intra-object concurrency is supported. Condition variables and monitors are used to implement inter-object communication. While these provide synchronization, they are unable to pass values back (that is, concurrent methods cannot return values).

5.2.7 Concert

Concert [19] supports distributed objects with an aim towards providing fine-grained parallelism. Object-based concurrency control and encapsulation, and a dynamic concurrency model are provided. The thrust of the project is to use aggressive whole program compilation, interprocedural optimization, and an efficient runtime system which works in concert with the compiler optimizations.

Like Kan, Concert provides a global namespace for objects, object-level concurrency control, and a powerful programming model. Also like Kan, Concert's aim is to give the programmer a powerful, flexible set of tools that are supported efficiently by the system. Concert also lets the programmer explicitly control the locality and concurrency control features of the system using pragmas, a feature that Kan lacks.

5.2.8 Distributed Oz

Distributed Oz [51] is a distributed version of the higher-order concurrent constraint language Oz. Oz objects combine stateful data abstraction with mutual exclusion and synchronization, including a dataflow synchronization construct that is similar to Kan's `guard` statement. It was inspired by concurrent logic programming, which led to the inclusion of logic variables and constraints in the language. It targets symbolic processing and problem-solving applications. Distributed Oz adds several features, such as a language construct for specifying object mobility patterns.

5.2.9 Emerald

Emerald is a strongly-typed pure object-oriented language. Objects can migrate between nodes. Objects can be declared *immutable*, which simplifies sharing. There is both inter-object and intra-object concurrency. Objects can be declared as *monitors*, which simplifies handling intra-object concurrency. Objects with a process (executing in parallel with the monitor) are active; those without a process are passive. All method calls are synchronous; new threads of control arise by creating a new active object. A garbage collector reclaims unreferenced objects.

5.2.10 Obliq

Obliq [14] supports distributed object-oriented programs through migrating threads. A computation can roam over the network, while maintaining connections between its constituent parts. Objects are local to some computing node, but threads migrate. Hence, a distributed computation consists of the migration of a thread over the necessary set of objects.

The Obliq language is based on Modula-3, and is a lexically-scoped untyped interpreted language. It contains a notion of hierarchical spaces; an Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet.

5.2.11 Orca

Orca [7] provides globally accessible objects (actually abstract data types), which are manipulated by way of operations. The objects are not shared in the usual sense. Sharing of objects arises through passing references to the objects into processes when they are created. Operations on global objects are atomic; that is, they act as though a lock were held on the object for their entirety. The system distinguishes between read locks and write locks, allowing multiple readers to proceed concurrently. Operations affect a single object only. Continuations are available to support concurrent operations.

Guard expressions can be given, which block an operation from beginning until they are satisfied. Multiple guards can be given for an operation, each with an associated program block. When any guard is true, one is selected nondeterministically, and its associated code is executed.

Orca uses a combination of compile-time and run-time techniques to determine user access patterns. It then switches between a fully replicated scheme, a single-copy scheme, and a migratory scheme to try to yield the greatest possible efficiency.

The system is built on top of Panda, a portable communication system. It is layered to provide heterogeneity, by allowing the system administrator to tune the Orca runtime system to the underlying architecture.

The Orca language, which compiles to ANSI C, is used on the Orca system. This language provides a class-like construct (abstract data types) from which object instances are created at runtime. However, the language is object-based rather than object-oriented; it does not support inheritance or dynamic binding, and there is non-object data in the system. The language is type-secure. That is, all language rule violations are detected by the compiler or the runtime system (e.g., out-of-bound array references or references to deallocated memory).

Orca advertises sequential consistency, but it actually provides the stronger condition of linearizability, via its combination of locks and totally ordered broadcast. Consistency is maintained by:

1. Replicating read-only objects to any node that asks for a copy;
2. Not replicating certain objects, so that all operations on them are via RPC; and
3. Making all communication go through a totally ordered multicast, so that operations are seen at all replicas in the same order.

The replication strategy is chosen by keeping read/write access counts. The ratio that represents the switchover point is chosen based on RPC and broadcast costs; hence, it varies with each platform. Only two options are considered: full replication and no replication. The project members state that experiments showed that this strategy produced better performance on their system than dynamic replication.

Fault tolerance is provided only for noninteractive parallel applications. The strategy is to take periodic globally consistent checkpoints, and roll back to the last such checkpoint on a failure. The globally consistent checkpoints are coordinated with Orca's totally ordered broadcast facility. A coordinator process periodically sends a take checkpoint message. On delivery of this message, each process saves its local checkpoint.

5.2.12 Thor

Thor [36] is a large-scale object-oriented database that supports reliable and available persistent objects. It hides distribution, caching, and disk management from the programmer. Thor defines an ABI (Application Binary Interface), so that programs written in different languages can operate on the same objects. Thor supports mobile and replicated objects, and provides a transactional mechanism for objects.

6 Conclusion and Future Work

The Kan system provides the programmer with powerful tools for writing parallel and distributed applications. These tools include asynchronous method calls, and nested atomic transactions with guards. In this paper we have shown that, in spite of the power of the tools provided, reasonable performance can be provided in a system that scales well on a LAN of up to 16 nodes. We have also shown that this performance can be obtained on top of Java sockets, bypassing RMI. In fact, using RMI to provide the same distributed object services results in greater latencies. Coupled with RMI's distinction between local and remote method call semantics, implementors of distributed systems face a tradeoff between the ability to interface with other systems employing RMI and providing replicating and migrating objects more cheaply than RMI is able to do so.

Performance of Kan is enhanced with several optimizations that are applicable to other systems of this kind. The thread pool is an oft-implemented construct that reduces the cost of creating threads. Even this simple construct provides a measurable increase in performance. We have also implemented a thread inlining feature, which turns asynchronous method calls into synchronous method calls to avoid thread context switch costs. This optimization is not universally applicable, since in the extreme it would serialize an application. However, it can provide substantial savings when a parent thread spawns many short-lived children. Pointer swizzling nearly eliminates the overhead of managing global objects when those objects are local to the caller, giving close to Java method call performance. Finally, the typing of objects allows the system to reduce the number of remote method calls made. As the number of local calls climbs, the other optimizations have a greater chance to make an impact, resulting in still further improvements in performance.

In the future, we intend to further develop Kan and explore further performance-enhancing optimizations. For example, object typing is based on the principle that one should maximize the number of method calls that are local. Extending this principle, we find that many software objects are related, in the sense that they make many method calls on one another. In such cases, it is desirable to ensure that the related objects are always colocated. Such colocation gives the method calling optimizations a greater opportunity to have an effect, resulting in still further boosted performance. The analysis necessary to determine when objects are related is a topic of future study.

We also plan to investigate ways of making Kan scalable. Scalability is currently limited by such constructs as the all-to-all socket connections established by the communication layer. While such all-to-all connections are fine for LANs such as the one we used for our experiments, they become unrealistic as the number of nodes involved climb into the hundreds and thousands. Instead of maintaining such connections continuously, we plan to manage sockets as resources, closing inactive sockets as needed to make room for connections between newly communicating nodes.

References

[1] ACM SIGPLAN. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 17–19 June 1998. ACM Press.

[2] ACM SIGPLAN. *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, CA, USA, 12–14 June 1999. ACM Press.

[3] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *SPAA '93*, pages 251–60, Velen, Germany, 30 June–2 July 1993.

[4] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *ICPP '99*, Aizu-Wakamatsu, Fukushima, Japan, 21–24 September 1999. To appear.

[5] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI '98* [1], pages 258–68.

[6] Henri E. Bal. *Programming Distributed Systems*. Prentice-Hall, New York, 1991.

[7] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[8] Sandeep Bhatia. Distributed garbage collection for a reliable messaging system. Master's thesis, University of California, Santa Barbara, Computer Science Department, August 1999.

[9] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *IPPS '96*, pages 132–41, Honolulu, HI, USA, 15–19 April 1996. IEEE Computer Society Technical Committee on Parallel Processing and ACM SIGARCH, IEEE Computer Society Press.

[10] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. To appear in OOPSLA '99.

[11] Gerald Brose, Klaus-Peter Löhr, and André Spiegel. Java does not distribute. In Christine Mingins, Roger Duke, and Bertrand Meyer, editors, *TOOLS Pacific '97*, pages 144–52, Melbourne, Australia, 24–27 November 1997.

[12] Gerald Brose, Klaus-Peter Löhr, and André Spiegel. Java resists transparent distribution. *Object Magazine*, 7(10):50–2, December 1997.

[13] Nat Brown and Charlie Kindel. *Distributed Component Object Model Protocol — DCOM/1.0*. Microsoft Corporation, January 1998. Internet RFC draft 2.

[14] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[15] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–61, September-November 1998.

[16] John B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–27, September 1995.

[17] Rohit Chandra, Anoop Gupta, and John Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[18] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In Utpal Banerjee et al., editor, *Languages and Compilers for Parallel Computing '92*, volume 757 of *Lecture Notes in Computer Science*, pages 124–44, New Haven, CT, USA, 3–5 August 1992. Springer-Verlag.

[19] Andrew A. Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert system. In Hermann Hellwagner, editor, *HIPS '97 (IPPS '97 Workshop)*, pages 15–25, Geneva, Switzerland, 1 April 1997. 11th International Parallel Processing Symposioum; IEEE Computer Society Technical Committee on Parallel Processing; ACM SIGARCH.

[20] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–60, November 1997.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98* [1], pages 212–23.

[22] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90*, pages 15–26, Seattle, WA, USA, 28–31 May 1990. Revised in Stanford CSL Technical Report CSL-TR-93-568.

[23] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991.

[24] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[25] Maurice Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In Anand Sivasubramaniam and Mario Lauria, editors, *CANPC '99*, volume 1602 of *Lecture Notes in Computer Science*, pages 137–49, Orlando, FL, USA, 9 January 1999. Springer.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, pages 289–300, San Diego, CA, USA, 16–19 May 1993. IEEE Computer Society Technical Committee on Computer Architecture, ACM SIGARCH, IEEE Computer Society Press.

[27] Maurice Herlihy and Michael P. Warres. A tale of two directories: Implementing distributed shared objects in Java. In *Java Grande '99* [2], pages 99–108.

[28] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–92, July 1990.

[29] L. V. Kale, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel Java with global object space. In Hamid R. Arabnia, editor, *PDPTA '97*, volume 1: Computer Science Research, Education, and Applications, pages 235–44, Las Vegas, NV, USA, 29 June 1997. Computer Science Research, Education, and Applications Tech.

[30] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object-oriented system based on C++. In Andreas Paepcke, editor, *OOPSLA '93*, pages 91–108, Washington, DC, USA, 26 September–1 October 1993. ACM SIGPLAN, ACM Press.

[31] Pascale Launay and Jean-Louis Pazat. Generation of distributed parallel Java programs. In David Pritchard and Jeff Reeve, editors, *Euro-Par '98*, volume 1470 of *Lecture Notes in Computer Science*, pages 729–32, Southampton, UK, 1–4 September 1998. Springer.

[32] Suk Yong Lee. Supporting guarded and nested atomic actions in distributed objects. Master's thesis, University of California, Santa Barbara, Computer Science Department, July 1998.

[33] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–70, December 1981.

[34] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[35] Barbara Liskov. Argus (distributed program language and system). In Barbara Simons and Alfred Z. Spector, editors, *Fault-Tolerant Distributed Computing*, volume 448 of *Lecture Notes in Computer Science*, pages 108–14. Springer-Verlag, Berlin, Germany, 1990.

[36] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in Thor. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD '96: Management of Data*, pages 318–29, Montreal, Quebec, Canada, 4–6 June 1996. ACM SIGMOD, ACM Press.

[37] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[38] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. In *PPOPP '99*, pages 173–82, Atlanta, GA, USA, 4–6 May 1999. ACM SIGPLAN, ACM Press.

[39] J. Eliot B. Moss. Nested transactions and reliable distributed computing. In *2nd Symp. on Reliability in Distributed Software and Database Systems*, pages 33–9, Pittsburgh, PA, USA, 19–21 July 1982. IEEE Computer Society, ACM, University of Pittsburgh, IEEE Computer Society Press.

[40] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press Series in Information Systems. MIT Press, Cambridge, MA, USA, 1985.

[41] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: Scalability issues in global computing. In *Java Grande '99* [2], pages 171–80.

[42] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Java Grande '99* [2], pages 152–9.

[43] The Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

[44] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[45] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, 12 April 1999. Held in conjunction with IPPS/SPDP '99.

[46] Michael Philippsen and Matthias Zenger. Javaparty—transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[47] Aleta Ricciardi, Michael Ogg, and Fabio Previato. Experience with distributed replicated objects: The Nile project. *Theory and Practice of Object Systems*, 4(2):107–15, 1998.

[48] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[49] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 3rd edition, 1997.

[50] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *Java Grande '99* [2], pages 8–14.

[51] Peter van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–51, September 1997.

[52] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *ISCA '92*, pages 256–66, Gold Coast, Queensland, Australia, 19–21 May 1992. ACM SIGARCH, IEEE Computer Society Technical Committee on Computer Architecture, IEEE Computer Society Press.

[53] Jing Wang. Thread optimizations in concurrent object oriented languages. Master's thesis, University of California, Santa Barbara, Computer Science Department, September 1998.

[54] Seth J. White and David J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In Li-Yan Yuan, editor, *VLDB '92*, pages 419–31, Vancouver, Canada, 23–27 August 1992. Morgan Kaufmann Publishers, Inc.

[55] P. R. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *Computer Architecture News*, 19(4):6–13, June 1991.

[56] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[57] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press Series in Computer Systems. MIT Press, Cambridge, MA, USA, 1990.