Transparent Communication for Distributed Objects in Java

Michael Hicks* Suresh Jagannathan§ Richard Kelsey§ Jonathan T. Moore*
Cristian Ungureanu§

Abstract

We describe a native-code implementation of Java that supports distributed objects. In order to foster the correctness of distributed programs, remote access is syntactically and semantically indistinguishable from local access. This transparency is provided by the runtime system through the implicit generation of remote references to an object when it is passed as an argument or returned from a remote method call. Consistency is achieved through the use of a distributed (and thus scalable) global addressing scheme. Experiments show that application performance is a function of data layout, access algorithm, and local workload. For distributed applications, such as distributed databases, these factors may not be known statically, suggesting the importance of runtime support.

1 Introduction

We consider a new approach to layering a distributed semantics onto Java [GJS95]. Our goal is to provide a system that simplifies the task of writing correct distributed Java programs without seriously compromising generality, scalability, or performance.

Distributed programming in our extended Java requires programmers only to specify an initial home for an instance; once an object is created, the semantics of accessing it remain unchanged regardless of whether the object is local or remote. The system is completely general: no distinction is made *a priori* about which objects may be remote. This means that programmers may leave the implementation of a program's control component unchanged when converting a serial program to a distributed one. Alterations to a program's control structure may be made to improve performance or to express a naturally distributed algorithm, but are not necessary for correctness. In this way we resemble distributed shared memory systems, but at the granularity of objects (presenting a global shared object-space) rather than memory locations.

While simplifying distributed programming in general, we believe this model will be particularly useful in applications where a computation's data access behavior cannot be easily determined statically or where data cannot be easily migrated. For example, a program might be written to search several databases in multiple administrative domains in a conceptually integrated manner. Programs may be written to search or otherwise operate on the data as if it were all available locally, when in fact it may be distributed throughout a large network.

To support a global shared object-space that can scale easily, we use a novel algorithm. The algorithm does not rely on a central registry or name-server to mediate creation of unique identifiers. Nodes involved in a remote communication event dynamically resolve the meaning of remote references in messages through the use of a distributed access table that manages remote references received from other nodes. The runtime system thus allows the collection of nodes involved in a Java computation to grow dynamically without requiring recompilation. The use of a distributed access table reduces the number of administrative messages required to resolve the meaning of remote references compared against an implementation that utilizes a centralized registry.

To improve performance, a native-code optimizing compiler is used to emit fast sequential code. In addition, the compiler is designed to perform distribution and communication-sensitive optimizations to help reduce the overhead induced by a naive implementation of this model. Because the way a program references an object is decoupled from its location, the runtime system can make dynamic decisions on how data should be distributed and how tasks should be scheduled. These decisions can never influence correctness, but may improve performance.

Like Java/RMI [WWR97, Sun97]), our design is based on remote method invocation. To achieve greater generality, however, we do not require programmers to classify instances as being remote or local. Instead, our semantics permits programs to dynamically create *any* kind of instance on any node in a network ensemble. An instance supplied as an argument or returned as a result of a method call can be referenced by any other object in the system regardless of its location in the network.

Like distributed shared memory systems (e.g., [LH89, YC97]), our implementation allows objects to be accessed transparently anywhere in a network ensemble. We differ insofar as we also permit code to be dynamically linked into a running computation (i.e., programs are non-SPMD), and we use remote method call as the underlying communication mechanism: instance methods execute on the node where the object was allocated, and not on the node where the reference is made. Because of our exclusive use of RPC, we do not support caching of mutable data on different nodes; thus, unlike DSM, our implementation does not include a significant invalidation and consistency component.

*Department of Computer and Information Science, University of Pennsylvania {mwh|jonm}@dsl.cis.upenn.edu

§Computer Science Research, NEC Research Institute {suresh|kelsey|cristian}@research.nj.nec.com

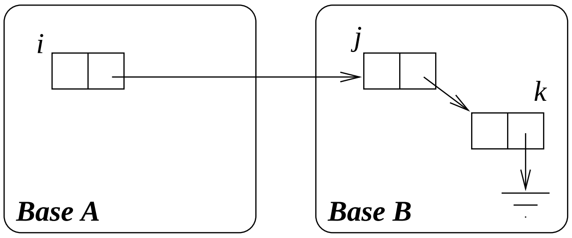

Figure 1: Remote references span bases.

Our use of a conceptually global shared object-space for specifying distributed programs means that local and remote method calls have the same external behavior. This semantics is in sharp contrast to the semantics supported by typical RPC-based implementations [BN84, SB90, BNOWer]. In RPC, local objects are *copied* on a remote call, and are not shared between the caller and callee if the two reside on different nodes. Programs using standard RPC as the communication model must therefore be carefully crafted to avoid unexpected behavior due to unwanted copying and loss of data sharing. Our design avoids this pitfall by implicitly generating remote references for instances used in a remote communication event. In this regard, our system resembles JavaParty [PZ97] and JavaSpaces [JSp99], two systems that also provide programmer-controlled shared-memory abstractions. We differ in the mechanics through which remote references are created and managed, and in the use of a specialized runtime (rather than Java/RMI) to provide transport and marshalling services.

The paper is organized as follows. The next section gives an overview of system assumptions and the programming model. Section 3 provides details on how remote references are implemented. Section 4 discusses the communication protocol. Section 5 gives further details on the implementation. Section 6 provides performance results. Sections 7 and 8 present related work and conclusions.

2 The Programming Model

A distributed Java program consists of a collection of threads and instances distributed on a set of *bases*. A base defines an abstraction of a physical machine and an address space, and is implemented as a process. Typically, there will be a single base running on each machine in a network; however, we make no assumption that there be a one-to-one correspondence between bases and physical nodes: programmers can create multiple bases on a single machine, but a base never spans multiple machines.

Bases have a *representation* as an ordinary Java instance. Each base contains such a *base instance* for every base to which it has established communication. The expression, `getBase(ipAddr,portNum)`, returns the base representative corresponding to the base executing at IP address `ipAddr`, accepting communication events at port `portNum`. The method creates a new base if not present.

Base instances provide a `remote` operation which may take a static method call or a constructor call as an argument. For example, assume a class `List` with static method `cons` to add new elements to a list. To create a remote list element, we evaluate the expression:

```
i = B.remote(List.cons(j,k))
```

to allocate a new list element on base `B` whose `car` is `j` and whose `cdr` is `k`. Note that `remote` is a syntactic form, not a method, since its argument is not entirely evaluated at the point of call: although the arguments `j` and `k` are evaluated on the current base, the actual call to `List.cons` takes place on `B`. In the above call, a reference to a remote object is returned if `B` is a base different from the base on which the statement executes; however, the instance `i` returned can be used in any context where an ordinary (non-remote) instance can be used.

An important invariant preserved by the semantics is that equality is preserved even among remote references. Thus, consider the method:

```
public static boolean cmp(Object x, Object y)
  { return x == y; }
```

The call: `b.remote(cmp(o1,o2))` will produce the same answer regardless of the base to which `b` is bound. The communication algorithm described below permits every base to retain a unique view of the remote instances it receives from other bases while ensuring global consistency.

There is no syntactic or semantic difference between accessing a local instance versus a remote one. In this way we present an abstraction in which references may span bases, as shown in Figure 1. Invoking an instance method or referencing a field through a remote reference results in a remote call to the base where the actual instance resides.

The decision to have the syntax and semantics of remote evaluation be indistinguishable from method invocation is deliberate, and is intended to foster program correctness. However, care must be taken to see that this simplification does lead to unacceptable performance cost. Because locality considerations dictate that an instance method should be executed on the base where its associated instance data resides, method calls are implicitly performed locally for local instances and remotely for remote instances. Remote instances can be used in any context where an ordinary (non-remote) instance can be used.

While implicit remote method invocation simplifies the complexity of writing correct distributed programs, we recognize that it may not always be possible to rely on automatic runtime management to achieve acceptable performance. Thus, we provide other primitives to give programmers greater control over distribution decisions. For example, we permit the programmer to specify the evaluation base for constructors and static methods using the `remote` construct described above. Furthermore, a predicate is provided to determine whether an instance is local or remote: `o.isRemote()` for any instance `o` returns `true` if `o` is a remote instance. The system also permits programs to clone any object on a different base; the expression `o.cloneOn(b)` clones instance `o` on base `b`, and returns a remote reference to the remote copy. Note that by the semantics of remote method invocation, the behavior of this operation does not depend on `o` being a local instance.

These predicates permit a programmer to refine programs for better distributed performance. For example, if a program detects a remote reference it might spawn a thread to coordinate the remote computation in parallel with useful

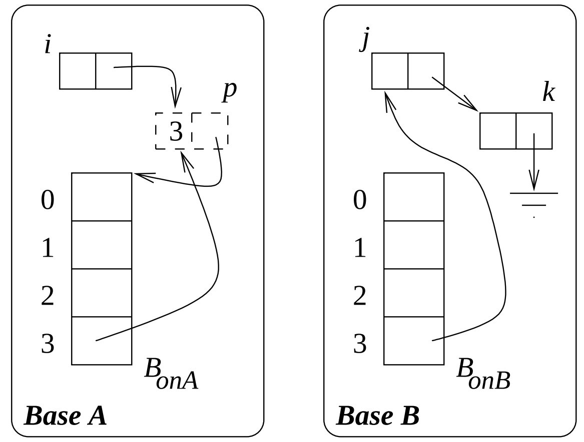

Figure 2: Base caches provide a distributed view of remote references.

local work. Similarly, if a series of operations are to be performed on a remote instance by some object o, a program may choose to clone o on the base where the target instance resides, thus replacing remote operations with local ones. Remote clones provide a convenient way of implementing base-specific objects, and thus are a useful abstraction for implementing machine-dependent system operations such as I/O.

3 Remote References

In this section, we describe the implementation of remote references, which are a fundamental part of our programming model. Since our design imposes no restrictions on which instances may be referenced remotely, the number of remote references generated in a program may be quite large. This suggests the use of a scheme that maintains the global identity of objects in a distributed fashion. Specifically, an object in our system can be identified by a combination of the base on which it resides plus a *local id* that is an integer allocated by the owning base. When an instance is exported—i.e., a remote reference to it is created on another base—it is assigned a local id and entered in a table on the exporting base.

This export table is actually a special instance of a *base cache*, which is a table mapping local ids to instances. A base cache A_{onB} resides on base B and maps local ids which were assigned on A to their instances. Thus, the export table on a base A would be A_{onA}. However, this export table is partially replicated on all the other bases with which A communicates as A_{onB}, A_{onC}, A_{onD}, etc.

A remote reference is represented by a special proxy object which contains the actual instance's local id plus a pointer to the local base cache of the owning base (among other fields which are not important for this discussion, see Section 5). Consider, for example, the situation depicted in Figure 2 where we have expanded the detail of our sample remote reference from the previous section. Here, the instance i on base A contains a remote reference to an instance j on base B. Suppose that B has assigned j a local id of 3 (thus $B_{onB}[3]$ contains a pointer to j). Now i actually contains a pointer to a proxy object p which contains the local id 3 plus a pointer to B_{onA} (since B is the owning base). Furthermore, $B_{onA}[3]$ contains a pointer back to p; further references from B with a local id 3 will resolve to this same proxy object p, thus allowing pointer equality to behave transparently as described at the end of the previous section.

It is important to note that the base caches of a base B which reside on other bases need not be completely filled. Specifically, B_{onA} will only contain entries for those instances on B which have remote references to them on A. In fact, base A maintains its own particular view of the instances allocated on other bases in B_{onA}, C_{onA}, D_{onA}, etc.

Base caches reduce communication overhead in two ways. First, a receiver needs to inquire about the meaning of a local id from the sender only the first time it receives it. Secondly, remote references can be forwarded from one base to another without involving the base on which the referenced instance is allocated, since a proxy object need only contain a local id plus a pointer to the appropriate base cache. In the above example, A could pass the remote reference to j on to another base C simply by sending the local id 3 plus an indication that B is the owning base. C can then consult $B_{onC}[3]$ to see if a local proxy for j already exists, and can create one if necessary. Note that C need not communicate with B to receive the remote reference. By building a distributed cache of references in this manner, we believe our implementation can scale more effectively than an implementation which uses a more conventional centralized registry [WWR97].

4 Communication

Because references may be passed as arguments to remote method invocations or returned as results, we need a way to transmit and resolve remote and local references. Here we describe the protocol that is used to resolve references that are transmitted between bases. For a given instance j of class c, created on base B and being transmitted by base A, j is represented as a triple $(id(B_{onA}), id(instance(c)), id(j))$, where $id(x)$ refers to the local id of some instance x and $instance(c)$ is an instance of `java.lang.Class` that represents c (we will just use c for brevity). Note that if $A = B$, then the instance is local; if the instance has never been transmitted before, then it is assigned a local id at marshalling time. As should be clear given our earlier description of remote references, the first and third elements of this triple are needed are needed to construct a proxy or resolve the reference on the remote base. We additionally require the class's id for our implementation; we describe its use in more detail in Section 5.

To illustrate how this information is used, Figure 4 shows three bases A, B, and C, and their associated caches. As in our previous examples, instance i resides on A, while instances j and k reside on B. At the outset, A has assigned local id 1 to B_{onA}. At this time B_{onA} is only partially complete—some class instance c is filled in slot 2, but slots 1 and 3 are empty (while they are present in B_{onB} with instances k and j, respectively).

The first row illustrates a message from B to A contain-

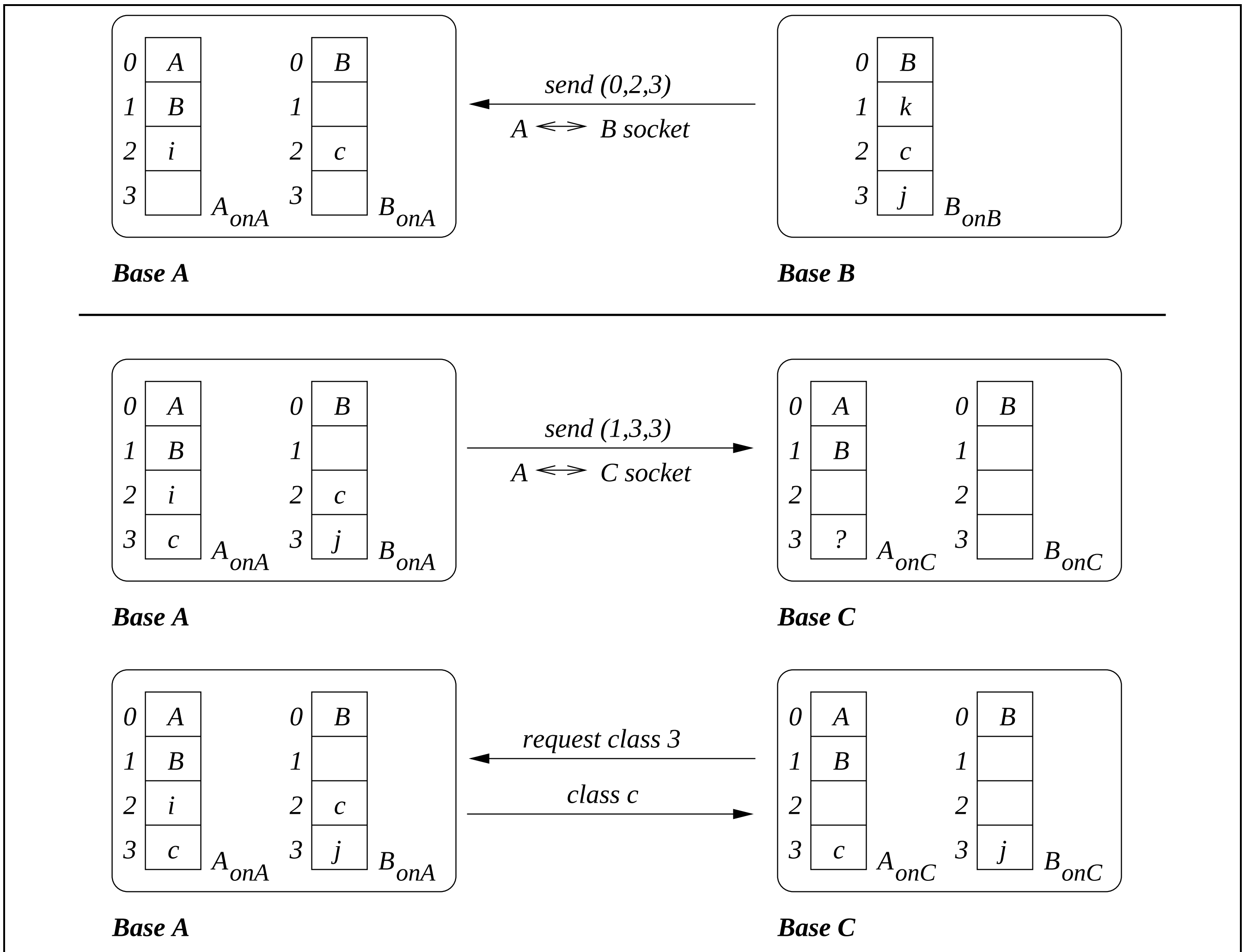

Figure 3: Communication among bases. The reference to j in B_{onB} is the instance itself, while in B_{onA} and B_{onC} it is to a proxy. All references to caches (A, B) and classes (c) are to instances. The vectors shown are the relevant base caches.

ing the representation for j: the local id for B_{onB} is 0, a local id for class c is 2, and the local id for i is 3. Since communication is point-to-point, the receiver (in this case A) knows the base of the sender (B), and thus knows the base cache to use to resolve local ids it receives (B_{onA}). It first looks up the base cache local id 0, which resolves to B_{onA}, j's owner. It then looks up j's local id 3 in that cache; since j is not present, a remote reference must be created. To do this, the instance's class c must first be resolved from id 2 in the *sender's* cache, B_{onA}. A then creates a proxy for j based on this information and then stores it at index 3 in its cache for B.

The second row shows the equivalent message being sent from A to C, consisting of the triple $(1, 3, 3)$. There are two differences from the previous message. The first is that the local id for B's cache is now 1, rather than 0, since 1 is the local id for B_{onA} (rather than B_{onB} as before). The second difference is that since A is sending the message, it refers to its local copy of class c, which has not been exported yet. It therefore allocates local id 3 for the class instance and notes this in A_{onA}. Note that there may never be remote references to base caches or classes since each base has its own copy of each. Therefore, as we have seen here, each base may assign a base cache or class instance a different local id.

When C receives the message, it will not be able to create the proxy for j because it fails to resolve the class local id 3. C thus sends a message to A asking for clarification about this local id. When the reply is received in the form of the class's name, C is able to match the name to its local copy of the class structure c. It then updates its cache for A, so that $A_{onC}[3]$ points to c. C may now create a proxy instance for j and update B_{onC} accordingly. Subsequent messages to C that refer to j can be processed without any additional administrative messages.

5 Implementation

This section presents details about the implementation of our distributed Java. We first present some information about the compiler and runtime system, followed by a description of bases, classes and instances, remote invocation, and threads.

5.1 Compiler

The compiler and runtime system are written in Java (60K lines of code). The runtime system also contains about 6K lines of C code, mostly consisting of native methods for performing I/O or 'unsafe' operations, like thread stack initialization. Each Java class is compiled into a shared object which is dynamically loaded if necessary during the execution of the program; currently, these objects are loaded from each node's local file system.

5.2 Bases

In our current implementation, each base communciates with other bases via a duplex socket using TCP/IP. However, we expect to relax this constraint to allow connections between bases to be dynamically constructed and removed, thus improving overall scalability of the system. Moreover, the specification of the message layer can be easily modified to support, for example, hierarchical (tree-based) connections.

5.3 Classes, Instances, and Proxies

Instances, instance proxies, classes, and class proxies are the four main components of our implementation. These elements are pictured in Figure 4.

Instance proxies have much the same representation as local instances. A local instance has a pointer to class data, and contains slots to hold a local id (if allocated), a hashcode, a mutex, and any instance fields. An instance proxy is simpler—it contains a pointer to a *class proxy* (described below), a pointer to its 'owning' base cache (that is, the cache representing the base on which the local version of the instance was allocated), a local id, and reference counts used by the distributed garbage collector. The collector currently uses a scheme based upon weighted reference counts [JL96] for collecting remote references, although distributed cycles could be collected by applying techniques like those found in [LPS98].

The figure also depicts our implementation of classes, and their corresponding *class proxies*. Proxies contain a pointer to their corresponding class, and also contain a pointer to a table of method stubs. When an instance method is invoked, the class pointer is followed to the method table. If the 'instance' is local, then the actual code will be found in the method table; in the case that the 'instance' is actually a proxy then the class proxy's method table will be used. The stubs found in this table dispatch calls to the actual method code on other bases: the stub marshals arguments for the call, initiates the remote method invocation, and then unmarshals and returns any result. Notice that a given method m and its stub representation reside at the same offset in the corresponding method tables. Thus, the calling sequence for m is unchanged regardless of whether the method will be evaluated locally or remotely. This strategy avoids the need for testing the location of an object when doing a method dispatch.

Unfortunately, a similar trick is not possible for field references. In the absence of any optimizations, each field reference is preceded by a compiler-generated check to determine whether the instance is local or a proxy; this is done by following the class pointer and checking a flag in the class. If the object is a proxy, some stub code is invoked to remotely acquire the field value. The only occasion where the check is clearly unnecessary is when the field is referenced via `this` (either explicitly or implicitly): since methods are evaluated on the base where their associated object resides, an expression of the form `this.x` always refers to a local field or method. Compiler optimizations can be used to reduce the amount of checking necessary, and to prefetch or bundle remote field accesses.

Classes also contain slots for GC- and class-specific data, static fields, static methods, and static method stubs (for use in B`.remote` expressions). To preserve the invariant that a computation's location does not alter its correctness, mutable class data (such as mutable class fields) must be consistent. This is acheived by requiring that such data reside on only one base in the system. This base is indicated by the class's *owning base* field; accesses to such data require checks similar to those instance fields. Immutable data (such as code and immutable fields) may be copied.

Having a designated base for storing mutable class data complicates class initialization, especially given the semantics imposed by Java regarding the order in which classes are initialized. Every class c that needs to be initialized must first initialize its superclass, d. The superclass decides which node first requested initialization of c, making that node the *owner* of c. Only the owner is allowed to contain the mutable data of the class. All other nodes that require initialization of c receive a message informing them that the class is initialized, and the base where the owner resides. The owner of `java.lang.Object` is the node that initiated the computation.

5.4 Remote Method Call

As mentioned, the compiler generates for each method a *send-stub* that marshals any arguments, sends a message to the remote base containing the arguments and other pertinent information (such as the method index), and finally unmarshals any returned result. Primitive types (i.e., integers, floats, etc.) are marshalled in the standard way, while instances (local or proxy) are marshalled as described in Section 4 (along with additional reference counts, not described in this paper).

When a remote invocation message sent from base A is received on base B, a new thread of control is created on B to handle it. Once the class of the method being invoked is known, a *receive-stub* at the specified method index is invoked. Receive stubs correspond one-to-one to send stubs, performing the inverse operations. For example, the receive stub for a constructor first allocates space for the instance, unmarshals the arguments (how many and their types are determined at compile-time, as is the case with the send-stub), and finally invokes the local constructor. The newly created instance is marshalled and returned. During the marshalling process of this new instance, reference counts and a local id will be allocated, filling an entry in B_{onB}. If an exception is thrown during remote invocation, it is

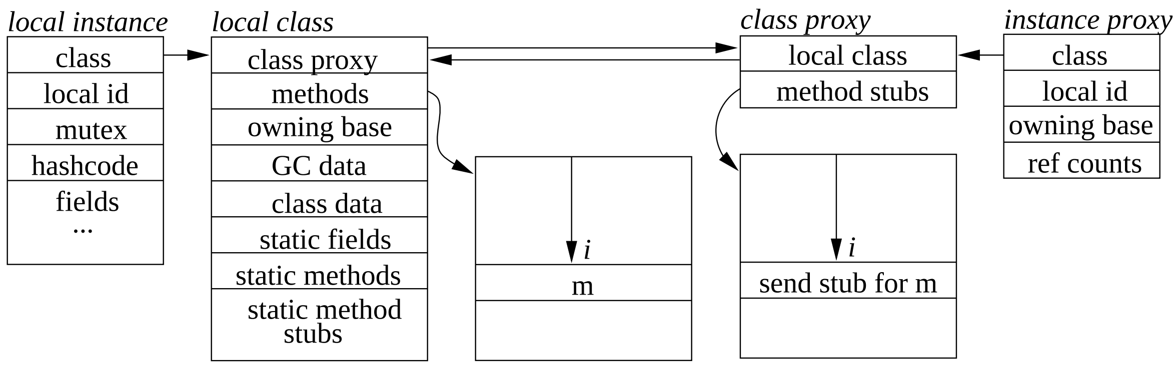

Figure 4: Local and remote object layout.

marshalled, returned, and rethrown on the sender.

5.5 Threads

While our thread model follows Java's semantics, our implementation is necessarily quite different. Ordinarily, an instance of `java.lang.Thread` will be implemented as a single data structure that contains a stack, status fields, etc. However, in a distributed environment, a single thread of control may be physically dispersed among many bases. Remote method invocation is an obvious example: the thread created on behalf of a remote method call is intuitively part of the sender's thread of control.

To make remote method invocation efficient, our implementation supports lightweight threads, allowing potentially thousands of threads to run on a particular base. Thread stacks consist of a series of chained, fixed-sized segments; new segments are added on stack overflow.

When a thread t initiates a computation on a remote base, a thread t' is spawned on that base to handle the computation. Thread t' serves as a delegate for t in the sense that it may reacquire locks held by t. This is safe and necessary due to the fact that t is blocked, awaiting the result being computed by t'; failure to allow t' to behave as if it were t may lead to deadlock.

Because delegates are an implementation mechanism to handle remote method invocation, they are not visible to the program. Indeed, from the programming model's perspective, delegates and threads have the same identity. Delegates implicitly belong to the same thread group as their parent thread, but are not included in a thread group's thread count. If a delegate t raises an exception it cannot handle, control reverts back to the thread (or delegate) that instantiated t.

Threads synchronize using an implementation of thin locks [BKMS98]. Thin locks are particularly useful in our implementation since most threads have short lifetimes, and are often used in single-threaded computations with little or no contention on shared resources. Remote method invocation is an obvious example.

6 Benchmarks

In this section we present the performance of remote method calls (RMC), a primary component of any distributed application using our system, and of a simple list-processing benchmark, to provide insight about our system's suitability for distributed database-style applications.

6.1 Experimental Setup

Our benchmarking cluster is made up of four dual 300 MHz Pentium-II's with split first level caches for instruction and data, each of which is 16 KB, 4-way set associative, write-back, and with pseudo LRU replacement. The second level cache is a unified 512 KB and operates at 150 MHz (we were unable to find any additional details about the second level cache). These machines receive a rating of 11.7 on SPECint95 and have 256 MBs of EDO memory. The machines run Linux kernel version 2.0.30 with multi-processor support. Each is connected to a single fast (100 Mb/s) Ethernet, switched by a 3Com SuperStack 3000.

Elapsed times were measured using the Pentium cycle counter. Our timing infrastructure imposes about 0.84 μs overhead for each call to the clock.

6.2 Remote Method Costs

To measure basic remote method invocation costs, we created a benchmark which allocates a remote instance and then invokes a 'null' instance method 10,000 times. For each invocation, we must marshal the "`this`" pointer; since the instance is itself remote, the "`this`" pointer is a remote reference. No value is returned, so no additional marshalling or unmarshalling is required. No garbage collections occur during the benchmark.

On average, each remote method invocation takes 0.94 ms. This time is further broken down into its components in Figure 5; note that the time indicated in the figure is slightly larger due to the overhead of additional timers. All of the indicated components are measured directly, while the "socket time" is simply the remaining unaccounted time; this includes wire time, the time to receive the message(s) and notify the receiving thread, and additional runtime system overhead, *e.g.* synchronization.

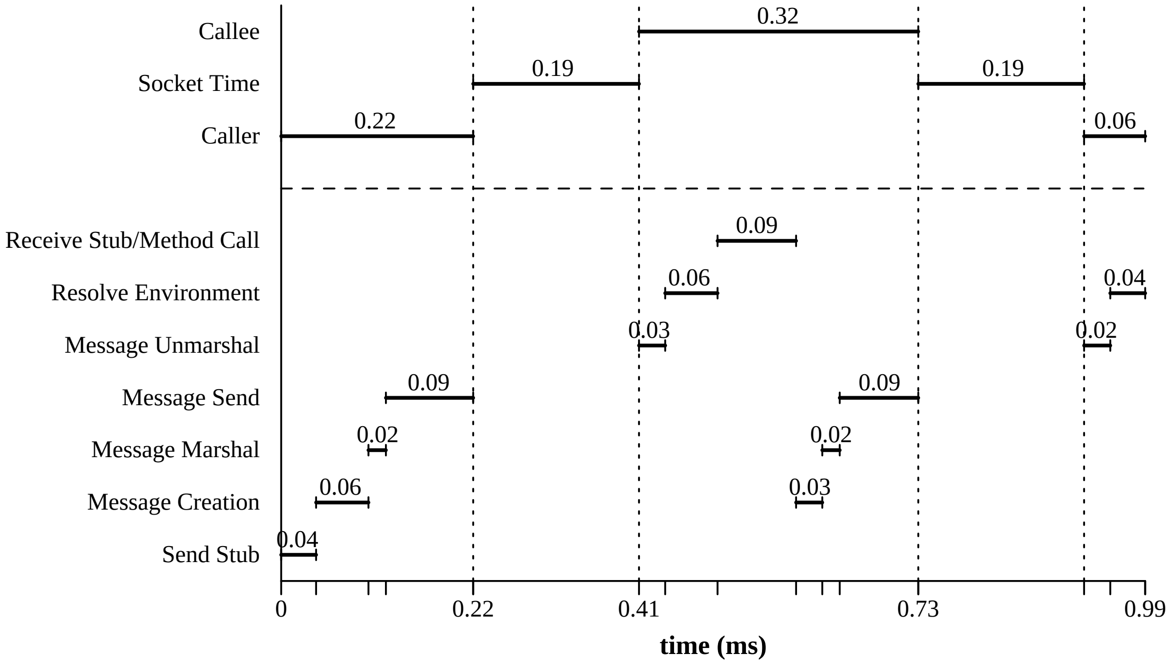

Figure 5: Breakdown of remote method invocation costs

The largest fraction of the total cost is the socket time, at 38% (0.38 ms). Based on other measurements (described below), we know that at least 44% of this time (0.17 ms) is above the basic communication costs for our platform. The next largest portion of the total RMC time (18%) is the cost of sending a message, at 0.18 ms. We therefore believe that tuning the I/O subsystem and the base communication implementation will greatly improve overall performance.

Of the remaining costs, message creation, marshalling, and unmarshalling constitute the largest overhead (18%, or 0.18 ms), followed by the time in the send/receive stubs and the remote method itself (13%, or 0.13 ms, and the time to set up the stub environment (10%, or 0.10 ms). The time spent in the stubs includes the time to marshal and unmarshal the `this` pointer. This constitutes about 30% of stub processing time (0.04 ms).

To put these measurements in context, we ran other similar benchmarks. To determine the basic cost of our messaging infrastructure, we measured the time to send a "null message" to another base and receive an acknowledgment. To measure basic transmission/receipt costs and I/O subsystem overhead we performed a similar test by sending out and receiving back appropriately message-sized buffers through a `java.lang.Socket`. Finally, we ran this same benchmark written in C to determine the lower bound for basic network operations on our platform. The results of these tests are given in Table 1.

Working from the left, we see that our implementation of thread-safe sockets adds an additional 0.13 ms to the C-based measurement. While this number may be improved, we see that larger gains may be had by optimizing the messaging system (the difference between columns 2 and 3), as well as optimizing the operations specific to RMC (the difference between columns 3 and 4).

Table 1: Remote method invocation overheads

C socket	Java socket	null message	RMC
0.21 ms	0.34 ms	0.68 ms	0.94 ms

We are actively tuning the performance of our system to good effect: our current times improve on our initial implementation by more than a factor of two. One key source of improvement has been to eliminate wasteful allocations. This has the two-fold benefit of lowering in-band latency and reducing the frequency of GC. For instance, by reducing the number of allocations required per message, we were able to cut message creation and marshalling times nearly in half. We have also reduced allocation by making selective use of caching; thus, rather than spawning a new thread for each RMC, we allow completed delegate threads to be cached for later use. Message processing costs have decreased as we have improved the speed of runtime system operations in the critical path. For example, by moving from a naive locking implementation to thin locks [BKMS98], we reduced RMC times by about 4%. We expect similar improvements by reducing I/O subsystem costs and context-switch times. Finally, since most of our communication infrastructure is written in Java, we can expect further improvements to occur as our compiler technology improves.

6.3 List Benchmark

We now consider a simple benchmark which iterates over a distributed list of size n spread across b bases, performing a configurable amount of work w on each element, re-

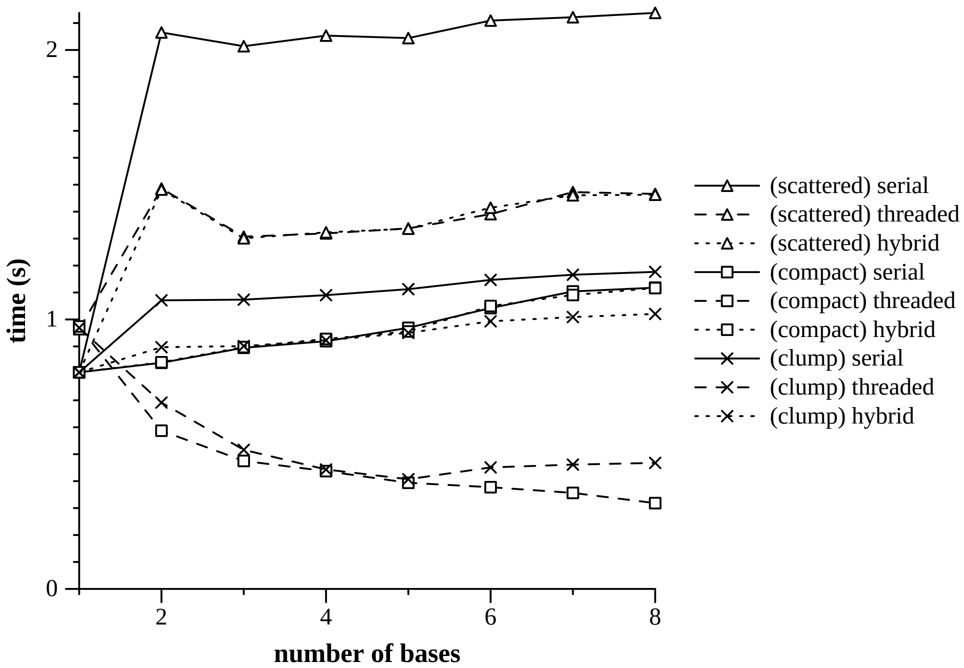

Figure 6: List benchmark times with workload $w = 1.0$ ms

turning a final result to the caller. This benchmark is designed to model a distributed database, as motivated earlier. However, by varying the traversal algorithm to exploit parallelism, the benchmark may also approximate applications designed to run on tightly coupled clusters of machines [ACPT95].

We fix the amount of work performed on each element to be roughly 1.0 ms, the baseline cost for a remote method call. The benchmark is run under three kinds of data layout: a scattered layout, where the list is interleaved across all the bases such that each "next" pointer is a remote reference; a compact layout, where the list is divided into evenly-sized clumps spread among the bases (so only one out of n/b elements is a remote reference); and a case in which clumps are smaller: one out of every four elements is a remote reference. For each of these data layouts, we consider three traversal algorithms: serial, threaded, and hybrid, described below. In all cases, we use a 1000 element list and present measurements varying b from 1 to 8 bases (at most 1 base per processor). Times presented are the median of 11 runs.

Figure 6 shows the results of these different runs. The x-axis is the number of bases involved in the computation, and the y-axis is the time to completion. The top three lines (with triangular hashes) show the three algorithms on the scattered data layout. The "serial" algorithm simply moves sequentially down the list. With only one base, the list is entirely local and so moving to the next element is cheap, but with two or more bases, every link in the list is a remote reference, increasing the total time. Because "serial" is single-threaded, the results are relatively insensitive to the number of bases.[1] The "threaded" algorithm, however, spawns a thread at each element to do the 1.0 ms of work and then moves on to the next element. As expected, the run times decrease as parallelism increases when moving from two to three bases. However, since the work time is approximately the time to perform an RMC, the improvement with parallelism quickly diminishes, and the per-base overheads begin to dominate.

The best algorithm, in this case, is the "hybrid" algorithm. Here, the algorithm examines each link in the list to determine if the next element is local or remote, and only spawns a thread if the next element is remote. Thus, for the one-base case, all links are local, no threads are created, and the algorithm performs similarly to the serial algorithm. For all other cases, however, the algorithm performs similarly to the threaded algorithm, thus achieving the optimum in all cases.

The other two data layouts illustrate different trends. Looking at the compact distribution, we see that the serial and hybrid algorithms are indistinguishable; this is because the advantage of the hybrid algorithm offsets its overhead due to less opportunity for parallelism. For the clumped distribution, the hybrid case improves on the serial one because it is able to leverage parallelism on one out of four nodes. The best performing with two or more bases for both clumped and compact layouts is the threaded algorithm; for the most part the compact case performs better. This result conforms to intuition because there are fewer remote references to traverse.

While not presented here, measurements taken for workloads significantly smaller or greater than the workload used in Figure 6 do corroborate intuition: as workload increases so does the benefit of parallelism and vice versa. With smaller workloads, performance is largely driven by data layout (and thus the number of required RMC's) and not

[1] In general, there is a slight upward slope to the lines. We suspect that this is due to the overhead of running two bases on one (dual-processor) machine, since all kernel operations must occur serially.

by algorithm.

The failure of the hybrid algorithm to outperform the threaded version is somewhat surprising since the hybrid implementation takes location information into account. We could recode the algorithm so that rather than computing w serially when the next element is local, we instead find out in total how many adjacent nodes are local, and then fork a thread to compute all of their workloads serially, having the current thread directly perform an RMC to the next base. This would reduce both thread overhead (a given thread potentially performs more work), and makes more effective use of parallelism. This alternative specification is essentially an application of lazy task creation [MKH90]. Nonetheless, this experiment indicates that modifying programs with location-dependent information may not always lead to noticeable performance improvement.

To summarize, with $w = 1.0$ ms the serial algorithm is the best for one base (whatever the layout), the hybrid is best for an extremely scattered layout, and the threaded one is the best in other cases. These benchmarks clearly indicate that overall performance is a function of algorithm, data layout, and workload. Since data layout is the most difficult to predict and can have dramatic impact on performance, it would be useful to provide some sort of feedback mechanism to communicate distribution information to the runtime system so that it can move data around to best fit the chosen algorithm. Since the easiest algorithms to write are the serial ones (which are well-supported by the transparency of our system), this suggests that objects which are frequently referenced together should be migrated to (or allocated on) the same base. Forcing the programmer to encode this policy directly into his programs would be redundant and error-prone, hiding the true nature of the computational element. This motivates the idea of having the runtime do this work automatically; we intend to explore appropriate policies and feedback mechanisms that would assist the runtime in this respect.

7 Related Work

There have been numerous efforts exploring the use of Java as a high-performance distributed language [FF97, Jav98]. These efforts can be broadly classified into three categories. The first two explore improvements to a Java/RMI implementation [WWR97, Sun97] either by providing a high-level veneer that removes some of RMI's complexity [PZ97], or by providing more efficient lower-level protocols [KWB+98]. The third considers extensions of Java with special distribution primitives and semantics [GS97, KBW97, JSp99], possibly adapting abstractions from other concurrent languages [CG89, GBD+94, KK93] into a Java framework. Besides these extensions, there has also been work incorporating *agents* [KB98, KZ97], a locus of data and control spanning multiple machines, into Java. We differ from these efforts in two important respects. First, our communication model is not based on RMI or other kinds of client/server models and thus, for example, does not distinguish between local and remote classes or references. Instead, calls to constructors or static methods are evaluated remotely with the compiler and runtime system establishing remote references as needed. Second, the implementation of both the compiler and runtime are optimized for this programming model; the system does not leverage functionality from existing Java implementations.

We share important similarity with Java Party [PZ97], a distributed Java implementation that is also not explicitly client/server based. Classes (including threads) whose instances may be remote are declared as such. The compiler and runtime system take care of issues related to locality, data mapping, and communication. Java Party thus supports even greater location transparency than provided in our implementation. However, because it is currently implemented as preprocessor that uses RMI as a target, it also inherits some of RMI's shortcomings. For example, programmers must still take care to distinguish between remote and local method invocation as the argument passing convention between the two are different. While the programming model presented by Java Party bears resemblance to the model outlined here, the implementation techniques between the two systems are quite different since we rely exclusively on our own native-code compiler and specialized runtime.

Our design is also similar in spirit to Obliq [Car95], another distributed object-based language. Like our system, remote references in Obliq are also implicit. Obliq distinguishes between methods and procedures, the former being associated with code that modifies object state. Methods are always executed on the machine where its corresponding object lives. Procedures may execute on the machine where the invocation is performed. In both systems, program behavior is not dependent on how objects are distributed in a network. In addition, our implementation differs in important and obvious ways from Obliq's, which is implemented as an interpreted language using Modula-3's network objects [BNOWer].

Emerald [BHJ+87, JLHB88] is a distributed object-based language that provides support for mobile objects. Like Obliq, Emerald has no class/instance hierarchy. However, similar to the implementation described here, method invocation in Emerald obeys the same semantics regardless of whether the invocation is remote or local, and processes are typically unaware of their location. Emerald's thread system shares much in common with the thread implementation used in our system. Emerald also provides other calling protocols that we have chosen not to include in our model, and provides extensive support for mobility. While Emerald does have access tables to map remote references to their locations, they are quite different from base caches which provide a partial, consistent view of the references exported by other bases.

The idea of a distributed cache as a mechanism to implement a global address space was inspired by the implementation used in Kali [CJK95], a distributed extension of the Scheme [RKR98] programming language which uses a similar structure to map remote Scheme objects. Kali's fundamental unit of distribution is a closure: code along with data could be freely transmitted among distributed nodes. In the system described here, objects remain resident on the node where they were created. Thus, Kali programmers must take care to ensure that unwanted copying of shared data does not occur. Our distributed Java semantics guarantees that a distributed Java program will exhibit the same behavior regardless of how data is partitioned among nodes.

CORBA [MZ96] and ILU [JSLJ97] are two well-known object-oriented glue languages that can be used to connect sequential components into a distributed program. The sequential components can be written in a variety of languages and components written in different languages can be freely intermixed within a single distributed program. Because these systems use classical form of remote procedure call, the values that may be sent between components are immutable ones: numbers, characters, sequences of values, and so forth. The only references that can be sent over the wire are references to the glue language's global objects.

Our implementation is also distinguished from Just-in-Time [CFM+97] and Way-Ahead-of-Time [PTB+97] compilers for Java insofar as our compiler translates to native code, and fully integrates distributed remote method invocation functionality. We share similar goals with [GFHM98], which proposes to use Java for high-performance computing using native-code compiler technology and fast native libraries for message-passing and distribution. However, the mechanism through which these goals are achieved is quite different since we propose to achieve distributed functionality and efficiency by fully integrating communication and messaging into the compiler and runtime.

8 Conclusions

Our programming model is intended to foster correctness of distributed Java programs, while still providing hooks to improve efficiency when necessary. Our runtime system helps support abstraction by allowing remote references to move freely among machines irrespective of where the actual objects referenced reside. With reasonable compiler optimizations and further streamlining of the runtime, we expect technology of this kind to be an attractive vehicle for distributed programming that permits highly efficient distributed programs suitable for execution on diverse distributed platforms to be easily written, debugged, and maintained.

References

[ACPT95] Thomas Anderson, David Culler, David Patterson, and The Now Team. A Case For NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Data Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[BKMS98] David Bacon, Ravi Konru, Chet Murthy, and Mauricio Serrano. Think Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design*, pages 258–268, June 1998.

[BN84] A. D. Birrell and B.J. Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BNOWer] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *14th ACM Symposium on Operating Systems Principles*, 1993 December.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 286–298, New York, 1995. ACM.

[CFM+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java Just in Time. *IEEE Micro*, 2(72):36–43, May 1997.

[CG89] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.

[FF97] Geoffrey Fox and W. Furmanski. Overview of Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modelling. *Concurrency: Practice & Experience*, 9(6), 1997.

[GBD+94] Al Geist, Adam Beguelin, John Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine.* MIT Press, 1994.

[GFHM98] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-Performance Parallel Programming in Java: Exploiting Native Libraries. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[GJS95] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Sun Microsystems, Inc., 1995.

[GS97] Paul Gray and Vaidy Sunderam. IceT: Distributed Computing and Java. *Concurrency: Practice & Experience*, 9(7), 1997.

[Jav98] *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection.* John Wiley, 1996.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, January 1988.

[JSLJ97] Bill Jansen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU 2.0alpha12 Reference Manual.* Xerox Corporation, 1997.

[JSp99] Javaspaces white paper, 1999. available from `java.sun.com/products/javaspaces`.

[KB98] Arie Keren and Amnon Barak. Adaptive Placement of Parallel Java Agents in a Scalable Computing Cluster. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[KBW97] L.V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and Implementation of Parallel Java with Global Object Space. In *Proceedings of the Conference on Distributed Processing Technology and Applications*, 1997.

[KK93] L.V. Kalé and S. Krishnan. Charm++: A Portable Concurrent Object-Oriented System Based on C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993.

[KWB+98] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient Implementations of Java Remote Method Invocation (RMI). In *Proceedings of the Usenix Fourth Conference on Object-Oriented Technologies and Systems*, 1998.

[KZ97] J. Kiniry and D. Zimmerman. A Hands-on Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–30, 1997.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[LPS98] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design*, pages 152–161, June 1998.

[MKH90] Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[MZ96] Thomas Mowbray and Ron Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects.* Wiley, 1996.

[PTB+97] Todd Proebsting, Gregg Townsend, Patrick Bridges, John Hartman, Tim Newsham, and Scott Watterson. Toba: Java for Applications, A Way Ahead of Time (WAT) Compiler. In *COOTS'97*, 1997.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(7), 1997.

[RKR98] William Clinger Richard Kelsey and Jonathan, Eds. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices Computation*, 33(9):26–75, September 1998.

[SB90] M.D. Schroder and M. Burrows. Performance of Firefly RPC. *ACM Transations on Computer Systems*, 8(1):1–17, 1990.

[Sun97] Java Remote Method Invocation (RMI). see `http://java.sun.com/products/jdk/1.2/docs/guide/rmi`, 1997.

[WWR97] Ann Wollrath, Jim Waldo, and Roger Rigs. Java-Centric Distributed Computing. *IEEE Micro*, 2(72):44–53, May 1997.

[YC97] Weimin Yu and Alan Cox. Java/DSM: A Platform for Heterogenous Computing. *Concurrency: Practice and Experience*, 9(7), 1997.