An Efficient Implementation of Java's Remote Method Invocation

Jason Maassen Rob van Nieuwpoort Ronald Veldema
Henri E. Bal Aske Plaat

Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
jason@cs.vu.nl rob@cs.vu.nl rveldema@cs.vu.nl bal@cs.vu.nl aske@cs.vu.nl
http://www.cs.vu.nl/manta/

Abstract

Java offers interesting opportunities for parallel computing. In particular, Java Remote Method Invocation provides an unusually flexible kind of Remote Procedure Call. Unlike RPC, RMI supports polymorphism, which requires the system to be able to download remote classes into a running application. Sun's RMI implementation achieves this kind of flexibility by passing around object type information and processing it at run time, which causes a major run time overhead. Using Sun's JDK 1.1.4 on a Pentium Pro/Myrinet cluster, for example, the latency for a null RMI (without parameters or a return value) is 1228 μsec, which is about a factor of 40 higher than that of a user-level RPC. In this paper, we study an alternative approach for implementing RMI, based on native compilation. This approach allows for better optimization, eliminates the need for processing of type information at run time, and makes a light weight communication protocol possible. We have built a Java system based on a native compiler, which supports both compile time and run time generation of marshallers. We find that almost all of the run time overhead of RMI can be pushed to compile time. With this approach, the latency of a null RMI is reduced to 34 μsec, while still supporting polymorphic RMIs (and allowing interoperability with other JVMs).

1 Introduction

There is a growing interest in using Java for high-performance parallel applications. Java's clean and type-safe object-oriented programming model and its support for concurrency make it an attractive environment for writing reliable, large-scale parallel programs. For shared memory machines, Java offers a familiar multithreading paradigm. For distributed memory machines such as clusters of workstations, Java provides Remote Method Invocation, which is an object-oriented version of Remote Procedure Call (RPC). The RMI model offers many advantages for parallel and distributed programming, including a seamless integration with Java's object model, heterogeneity, and flexibility [30].

Unfortunately, many existing Java implementations have inferior performance of both sequential code and communication primitives, which is a serious disadvantage for high-performance computing. Much effort is being invested in improving serial code performance by replacing the original byte code interpretation scheme with just-in-time compilers, native compilers, and specialized hardware [17, 22, 25]. The communication overhead of Java RMI implementations, however, remains a major weakness. RMI is originally designed for client/server programming in distributed (web based) systems, where latencies on the order of several milliseconds are typical. On more tightly coupled parallel machines, such latencies are unacceptable. On our Pentium Pro/Myrinet cluster, for example, Sun's JDK 1.1.4 implementation of RMI obtains a two-way null-latency (the latency of an RMI without parameters or a return value) of 1228 microseconds, compared to 30 microseconds for a user level Remote Procedure Call protocol in C. (A null-RMI in Sun's latest JDK, version 1.2 beta, is even slower.)

Part of this large overhead is caused by inefficiencies in the JDK. The JDK is built on a hierarchy of stream classes that copy data and call virtual methods. Serialization of method arguments is implemented by recursively inspecting object types until primitive types are reached, and then invoking the primitive serializers a byte at a time. All of this is performed at run time, for each remote invocation. In addition, RMI is implemented on top of IP sockets, which adds kernel overhead (and four context switches on the critical path).

Besides inefficiencies in the JDK, a second and more fundamental reason for the slowness of RMI is the difference between the RPC and RMI models. Java's RMI scheme is designed for flexibility and interoperability. Unlike RPC, it allows classes unknown at compile time to be exchanged between a client and a server and to be downloaded into a running program. In Java, an actual parameter object in an RMI can be a subclass of the method's formal parameter type. In polymorphic object-oriented languages, the *dynamic* type of the parameter-object (the subclass) should be used by the method, not the static type of the formal parameter. When the subclass is not yet known to the receiver, it has to be fetched over the network from a remote process and be downloaded into the receiver. This high level of flexibility is the key distinction between RMI and RPC [30]. RPC systems simply use the static type of the formal parameter (thereby type-converting the actual parameter), and thus do not support polymorphism and break with the object-oriented model.

The key problem is to obtain the efficiency of RPCs *and* the flexibility of Java's RMI. This paper discusses a new compiler-based Java system, called *Manta*,[1] which is designed from scratch to implement polymorphic RMIs efficiently. On our Myrinet cluster, for example, Manta obtains a null-latency of 34 μsec, a factor of 36 improvement over the JDK 1.1.4. Table 1 shows two-way null-RMI latencies and throughput of Manta, Sun's JDK (a byte

PPoPP '99 5/99 Atlanta, GA, USA
© 1999 ACM 1-58113-100-3/99/0004...$5.00

[1] A fast, flexible, black-and-white, tropical fish, that can be found in the Indonesian archipelago.

code interpreter), Sun's JIT (a just-in-time byte code compiler), and Panda (a conventional RPC in C), on two different processors and two different networks. The table shows that Manta is substantially faster than Sun's RMI, and close to a Panda RPC. (The difference between Panda 3.0 and 4.0 is explained in Section 3.)

Manta replaces Sun's run time protocol processing as much as possible by compile time analysis. We use a native compiler to generate efficient serial code and specialized serialization routines for serializable argument classes. The generated serializers allow a simpler RMI protocol that avoids type inspection at run time altogether. Since type information is known at compile time, it suffices to carry a simple type-id in the protocol, instead of elaborate type descriptions. In this way, almost all of the protocol overhead has been pushed to compile time, off the critical path. The problems with this approach, however, are how to interface with other Java Virtual Machines (which is required for interoperability) and how to address dynamic class loading (required to support polymorphism). To interoperate with other (non-Manta) JVMs, Manta supports the standard Sun RMI and serialization protocols in addition to its own fast protocols. Dynamic class loading is supported by compiling methods and generating serializers at run time.

The Manta system combines high performance with the flexibility and interoperability of RMI. In a metacomputing application [10], for example, some clusters can run our Manta software and communicate internally using the fast Manta RMI protocol. Other machines may run other JVMs (containing, for example, a graphical user interface program) and Manta communicates with these machines using the standard Sun RMI protocol. The machines running Manta and the JVM can both invoke each other's methods. Manta implements almost all other functionality required for Java RMI, including heterogeneity, multithreading, synchronized methods, and distributed garbage collection.

The main contributions of this paper are as follows. First, the paper shows that Java RMI can be implemented efficiently and can obtain a performance close to that of RPC systems. On Myrinet, a null-RMI takes 1228 μsec for Sun's JDK, and 34 μsec for our system, only 4 μsec more than a C-based RPC. Second, we show that the efficiency improvement can be achieved without sacrificing polymorphic RMIs and interoperability.

The rest of the paper is structured as follows. The design and implementation of the Manta system are discussed in Section 2. At the time of this writing, most of the system is up and running, though some parts of the interface to Sun JDK RMIs are still being finished; Section 2.4 reports on the implementation status. In Section 3 we give a detailed performance analysis of our system. In Section 4 we look at related work. Section 5 presents conclusions.

2 Design and Implementation of Manta

This section will discuss the design of the Manta system (including the unimplemented parts) and describe the current implementation status of the system. We will focus on the Manta RMI implementation.

2.1 Manta Structure

Since Manta is designed for high-performance parallel computing, it uses a native compiler rather than a JIT. The most important advantage of a native compiler is that it can do more aggressive optimizations and therefore generate better code. To support interoperability with other JVMs, however, Manta also has to be able to process the byte code for the application, and contains a run-time byte-code-to-native compiler.

The Manta system is illustrated in Figure 1. The box in the middle describes the structure of a node running a Manta application. Such a node contains the executable code for the application and (de)serialization routines, both of which are generated by Manta's native compiler. A Manta node can communicate with another Manta node (the box on the left) through a fast RMI protocol (using Manta's own serialization format); it can communicate with another JVM (the box on the right) through a JDK-compliant protocol (using Sun's serialization format). Determining which protocol to use is done with an initial probe RMI, which is only recognized by a Manta application, not by a JVM.

A Manta-to-Manta RMI is performed with Manta's own fast protocol, which is described in detail in the next subsection. This is the common case for high performance parallel programming, for which Manta is optimized. Manta's serialization and deserialization protocols support heterogeneity.

A Manta-to-JVM RMI is performed with a slower protocol that is compatible with Sun's RMI standard. Manta uses generic routines to (de)serialize the objects to or from Sun's format. These routines use runtime type inspection (reflection), and are similar to Sun's protocol. The routines are written in C (as is all of Manta's run time system) and execute more efficiently than Sun's protocol, which is written mostly in Java.

A Manta application must be able to work with byte codes from other nodes, to implement polymorphic RMIs with JVMs. When a Manta application requests a byte code from a remote process, Manta will invoke its byte code compiler to generate the metaclasses, the serialization routines, and the object code for the methods (as if they were generated by the Manta source code compiler).

The dynamically generated object code is linked into the application with the `dlopen()` dynamic linking interface. If a remote node requests byte code from a Manta application, the JVM byte code loader retrieves the byte code for the requested class in the usual way through a shared filesystem or through an http daemon. RMI does not have separate support for retrieving byte codes (see also *http://sirius.ps.uci.edu/~smichael/rmi.htm*). Sun's `javac` compiler is used to generate the byte code at compile time.

Two Manta applications can also implement polymorphism by exchanging the Java source code instead of the byte code. In this case (not shown in Figure 1), the native Manta compiler is invoked during runtime, resulting in better object code than with the byte code compiler. Manta applications must still be able to compile byte codes, however, since a class may originate from a non-Manta JVM, in which case the source may not be available.

The structure of the Manta system is more complicated than that of a JVM. Much of the complexity of implementing Manta efficiently is due to the need to interface a system based on a native-code compiler with a byte code-based system. The fast communication path in our system, however, is straightforward: the Manta RMI protocol just calls the compiler-generated serialization routines and uses a simple scheme to communicate with other Manta nodes.

2.2 Serialization and Communication

RMI systems can be split into three major components: low-level communication, the RMI protocol (stream management and method dispatch), and serialization. Below, we discuss how Manta implements this functionality.

Low-level Communication

Java RMI implementations are built on top of TCP/IP, which was not designed for parallel processing. Manta uses the Panda communication library [1], which has efficient implementations on a variety of networks. On Myrinet, Panda uses the LFC communication system [4, 5]. To avoid the overhead of operating system calls, LFC and Panda run in user space. On Fast Ethernet, Panda is implemented on top of UDP. In this case, the network interface

System	*Version*	*Processor*	*Network*	*Latency* (μs)	*Throughput* (MByte/s)
Sun JDK	1.1.3	300 MHz Sparc Ultra 10	Fast Ethernet	1630	1.0
Sun JIT	1.1.6			1210	4.1
Sun JIT	1.2 beta			1311	3.0
Manta/Panda 3.0				338	7.4
Panda	3.0			328	8.7
Sun JDK	1.1.4	200 MHz Pentium Pro	Fast Ethernet	1711	0.97
Manta/Panda 3.0				233	7.3
Panda	3.0			228	10.3
Sun JDK	1.1.4	200 MHz Pentium Pro	Myrinet	1228	4.66
Manta/Panda 3.0				34	20.6
Manta/Panda 4.0				39	51.3
Panda	3.0			30	55.7
Panda	4.0			31	59.4

Table 1: Two-way Null-RMI Latency and Throughput

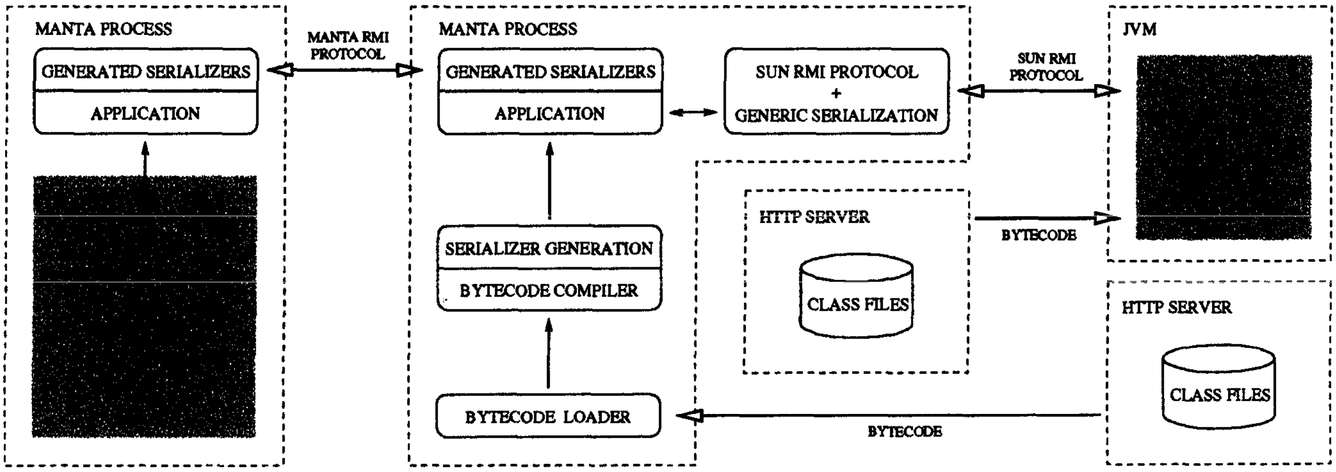

Figure 1: Manta/JVM Interoperability

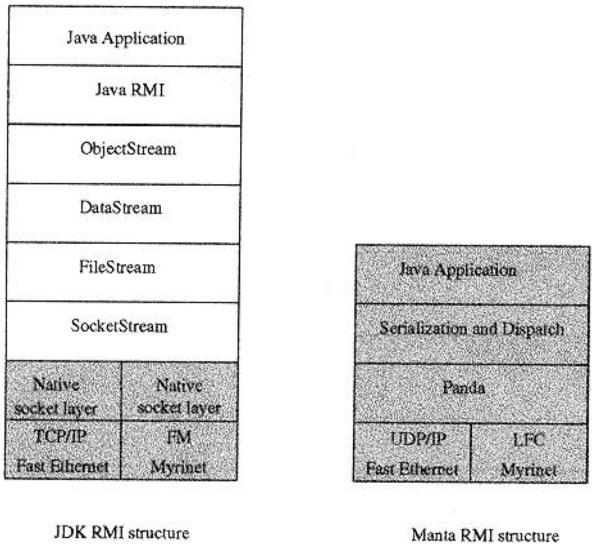

Figure 2: Structure of Sun and Manta RMI; Grey indicates compiled code

is managed by the kernel, but the Panda RPC protocols run in user space.

The Panda RPC interface is based on an *upcall* model: conceptually a new thread of control is created when a message arrives, which will execute a handler for the message. The interface has been designed to avoid thread switches in simple cases. Unlike active message handlers [29], upcall handlers in Panda are allowed to block in a critical section, but a handler is not allowed to wait for another message to arrive. This restriction allows the implementation to handle all messages using a single thread and to avoid context switches for handlers that execute without blocking [19].

The RMI Protocol

The run time system for the Manta RMI protocol is written in C. It was designed to minimize serialization and dispatch overhead, such as copying, buffer management, fragmentation, thread switching, and indirect method calls. Figure 2 gives an overview of the layers in our system and compares it with the layering of the JDK system. The shaded layers denote compiled code, while the white layers are interpreted (or JIT-compiled) Java. Manta avoids the stream layers of the JDK. Instead, RMIs are serialized directly into a Panda buffer. Moreover, in the JDK these stream layers are written in Java and their overhead thus depends on the quality of the interpreter or JIT. In Manta all layers are either implemented as compiled C code or compiler generated native code. Also, Manta applications can call RMI serializers directly, instead of through the (slow) Java Native Interface. Heterogeneity between little-endian and big-endian machines is achieved by sending data in the native byte order of the sender, and having the receiver do the conversion, if necessary. Another important optimization in our RMI protocol is to avoid generating a new thread at the receiving node. The Manta compiler determines for each remote method whether it is guaranteed to execute without blocking (whether it may execute a “wait()” operation). If the method will never block, it is executed without doing a thread context switch. The compiler currently makes a conservative estimation and only guarantees the non-blocking property for methods that do not call other methods.

The Manta RMI protocol cooperates with the garbage collector to keep track of references across machine boundaries. Manta uses a local garbage collector based on a mark-and-sweep algorithm. Each machine runs this local collector, using a dedicated thread that is activated by the run time system or the user. The distributed garbage collector is implemented on top of the local collectors, using a reference counting mechanism for remote objects (distributed cycles remain undetected).

The Serialization Protocol

The serialization of method arguments is an important source of overhead of existing RMI implementations. Serialization takes Java objects and converts (serializes) them into an array of bytes. The JDK serialization protocol is written in Java and uses reflection to determine the type of each object during run time. With Manta, all serialization code is generated by the compiler, avoiding the overhead of dynamic inspection. Serialization code for most RMI calls is generated at compile time. Only serialization code for polymorphic RMI calls that are not locally available is generated, by the Manta compiler, at run time. The overhead of this run time code generation is incurred only once, the first time the new class is used as a polymorphic argument to some method invocation. For subsequent uses, the fast serializer object code is then available for reuse. The overhead of run time serializer generation is on the order of seconds at worst, depending mostly on whether the Manta compiler is resident, or whether it has to be paged in over NFS.

To accomplish fast serialization with correct Java semantics, the compiler generates special (un)marshall methods. For every method in the method table, a method pointer is maintained here to dispatch to the right (un)marshaller for that method. A similar optimization is used for serialization: every object has two pointers in its method table to the serializer and deserializer for that object. When a particular object is to be serialized the method pointer is extracted from the object’s method table and invoked. On deserialization the same procedure is applied. The serialization and deserialization code is generated by the compiler and has complete information about fields and their types. When a class to be serialized/deserialized is marked “final”, the cost of the virtual function call to the right serializer/deserializer is optimized away, since the correct function pointer can be determined at compile time.

The Manta serialization protocol performs optimizations for simple objects. An array whose elements are of a primitive type is serialized by doing a direct memory-copy into the message buffer, which saves traversing the array. Serialization produces a deep copy. In order to detect duplicate objects, the marshalling code uses a hash table containing objects that have already been serialized. If the method does not contain any parameters that are objects, however, the hash table is not created, which again makes simple methods faster. Also, the hash table itself is not transferred over the network, instead, the table is rebuilt on-the-fly by the receiver. Compiler generation of serialization is one of the major improvements of Manta over the JDK.

2.3 Generated Marshalling Code

Figures 3, 4, and 5 illustrate the generated marshalling code. Generation of meta classes and marshallers is described in more detail in [28]. Consider the “RemoteMonkey” class in Figure 3. The “foo()” method can be called from another machine, therefore the compiler generates marshalling and unmarshalling code for it.

The generated marshaller for the “foo()” method is shown in Figure 4 in pseudo code. Because "foo()” has a String as parameter, which is an object in Java, a hash-table is created to detect duplicates. A special create *thread* flag is set in the header data structure. This is done because “foo” contains a method call (“System.out.println()“). and might therefore potentially do a “wait()“, or block on a similar synchronization statement. When a remote ex-

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RemoteMonkey extends UnicastRemoteObject
  implements RemoteMonkeyInterface {
    int value;
    String name;

    synchronized int foo(int i, String s) throws RemoteException {

        value = i;
        name = s;

        System.out.println("i = " + i);

        return i*i;

    }
}
```

Figure 3: A simple remote class.

ception is thrown, the hash table is reused to detect duplicates in the exception object. The programmer may define his own exceptions in Java, so it is not guaranteed that the thrown exception does not contain a cycle. The *writeObject* call will serialize the string object to the buffer at the current position. The *flushFunction* does the actual writing out to the network buffer. It is also used, together with the *fillFunction*, for fragmentation.

```
marshall__foo(RemoteObject *this, int i, class__String *s) {

    allocBuffers(&inBuffer, &outBuffer);
    hashTable = createHashTable();

    writeHeader(&outBuffer, this,
              OPCODE_CALL, CREATE_THREAD, flushFunction);
    writeInt(&outBuffer, i, flushFunction);
    writeObject(&outBuffer, s, flushFunction, hashTable);

    // Request message is created, now write it to the network.
    flushFunction(outBuffer);

    clearHashTable(hashTable);

    // Receive reply.
    fillFunction(&inBuffer);

    opcode = readInt(&inBuffer, fillFunction);

    if (opcode == OPCODE_EXCEPTION) {

        class__Exception *exception =
             readObject(&inBuffer, fillFunction, hashTable);

        freeBuffers(inBuffer, outBuffer);
        killHashTable(hashTable);

        THROW_EXCEPTION(exception);

    } else {

        int result = readInt(&inBuffer, fillFunction);

        freeBuffers(inBuffer, outBuffer);
        killHashTable(hashTable);

        RETURN(result);

    }
}
```

Figure 4: The generated marshaller for the "foo" method.

Pseudo code for the generated unmarshaller is shown in Figure 5. The header is already unpacked when this unmarshaller is called. Because the *create thread* flag in the header was set, this unmarshaller will run in a new thread started by the runtime system. The marshaller itself does not know about this. Note that the *this* parameter is already unpacked and is a valid reference for the machine the unmarshaller will run on.

```
unmarshall__foo(class__Object *this) {
    allocBuffers(&inBuffer, &outBuffer);
    hashTable = createHashTable();

    int i = readInt(&inBuffer, fillFunction);
    class__String *s = readObject(&inBuffer,
                                  fillFunction, hashTable);

    result = CALL_JAVA_FUNCTION(foo, this, i, s, &exception);

    hashTable = clearHashTable();

    if(exception) {

        writeInt(&outBuffer, OPCODE_EXCEPTION, flushFunction);
        writeObject(&outBuffer,
                exception, flushFunction, hashTable);

    } else {

        writeInt(&outBuffer, OPCODE_RESULT_CALL, flushFunction);
        writeInt(&outBuffer, result, flushFunction);

    }

    // Reply message is created, now write it to the network.
    flushFunction(outBuffer);

    freeBuffers(inBuffer, outBuffer);
    killHashTable(hashTable);

}
```

Figure 5: The generated unmarshaller for the "foo" method.

2.4 Implementation status

Our work on an efficient Java RMI started out as an attempt to make a fast version of JavaParty [24]. JavaParty does not use a registry, and uses a syntax that differs slightly from Sun RMI. It uses the keyword `remote` for classes that can be called remotely. For example, the RemoteMonkey class declaration from Figure 3 would be written as `public remote class RemoteMonkey {`. The Manta system is now being extended to also support the standard Sun RMI syntax. Further extensions needed to interoperate with Sun JVMs are support for the Sun RMI registry, support for the Sun RMI wire protocol, and the ability to work with byte code files. Some of these extensions are already working, although the efficient Myrinet implementation of polymorphic remote method calls currently only works for the JavaParty syntax.

The Manta compiler and fast RMI protocol are operational and have been used to run several applications. The compiler currently generates code for the Pentium and Sparc architectures. The Manta run time system supports several networks (including UDP/IP networks and Myrinet). On Myrinet, the user-level communication system we use (LFC) offers no protection, so the Myrinet network can be used by a single process only. (This problem can be solved with other Myrinet protocols that do offer protection [4].) The Fast Ethernet implementation uses a kernel-space UDP/IP protocol and does not have this problem.

In addition, we are finishing the implementation of the dynamic byte code compiler, which includes the ability to generate serialization routines from byte codes. We have implemented dynamic source code compilation, which can be used for polymorphic RMIs between two Manta nodes (see Section 2.2). The linking of dynamically generated object code works on Linux and Solaris. On BSD 3.0 (one of the operating systems used for our Myrinet cluster) it does not work because of a bug in BSD 3.0's implementation of `dlopen()`. Interoperability with Sun RMI, including polymorphic RMI, poses the largest engineering challenges. At the time of this writing, we have run a small mixed Sun/Manta RMI application, and we have run a small application compiled by the Manta byte code compiler. Currently, interfacing with Sun JVMs, and the ability to use the Sun RMI syntax over the fast Myrinet protocol, are being finished.

3 Performance Measurements

In this section, the performance of Manta is compared against the Sun JDK 1.1.4. Experiments are run on a homogeneous cluster of Pentium Pro processors. Each node contains a 200 MHz Pentium Pro and 128 MByte of EDO-RAM. All boards are connected by two different networks: 1.2 Gbit/sec Myrinet [6] and Fast Ethernet (100 Mbit/sec Ethernet). The system runs the BSD/OS (Version 3.0) operating system from BSDI and RedHat Linux version 2.0.36. Timing differences between BSD and Linux are small to negligible. Except where otherwise noted, the numbers reported are from runs on BSD. Both Manta and Sun's JDK run over Myrinet and Fast Ethernet. We have created a small user-level layer that implements socket functionality in order to run JDK RMI over Myrinet, on top of Illinois Fast Messages (FM) [23]. FM's round-trip latency is 4 μs higher than that of LFC.

3.1 Latency

For the first benchmark, we have made a breakdown of the time that Manta spends in remote method invocations, using zero to three (empty) objects as parameters, and no return value. The measurements were done by inserting timing calls, using the Pentium Pro performance counters. These counters have a granularity of 5 nanoseconds. The results for Manta over Myrinet (using Panda 3.0) are shown in Table 2.

The simplest case is an empty method without any parameters, the null-RMI. On Myrinet, a null-RMI takes about 34 μs. Only 4 microseconds are added to the latency of the Panda RPC, which is 30 μs. When passing primitive data types as a parameter to a remote call, the latency grows with less than a microsecond per parameter, regardless of the type of the parameter (this is not shown in the table). When one or more objects are passed as parameters in a remote invocation, the latency increases with several microseconds. The reason is that a table must be created by the run time system to detect possible cycles and duplicates in the objects. Separate measurements show that almost all time that is taken by adding an object parameter is spent at the remote side of the call, deserializing the call request (not shown). The serialization of the request on the calling side, however, is affected less by the object parameters.

To compare the overhead of the JDK and Manta, we have performed the same breakdown of these two systems on Pentium Pros connected by Fast Ethernet. We use Fast Ethernet rather than Myrinet for the comparison between the JDK and Manta, so we can run the JDK "out-of-the-box" (without making any changes to it). The results are given in Table 3. On the JDK and the JIT, the communication times still include a small amount of Java overhead. Pipelining effects in the communication layers complicate measurements, which is why the timings in the columns in the table do not add up exactly to the measured overall run time. The timings on Fast Ethernet are less consistent than on Myrinet, which may be the cause of small discrepancies in the table.

A null-RMI for Manta over Fast Ethernet takes 233 microseconds, while a JDK RMI takes 1711 microseconds; Manta thus is 7.3 times faster. The table shows how expensive Sun's serialization and RMI protocol (stream and dispatch overhead) are, compared to Manta. With 3 object-parameters, for example, the total difference is a factor 60 (2036 μs versus 34 μs).

Part of the overhead of the JDK 1.1.4 is caused by the usage of an interpreter. To determine the impact of a JIT compiler we have also run tests with the Sun JIT 1.1.6 just-in-time byte code compiler. We were unable to run Sun's JIT on BSD/OS or Linux; we used UltraSparcs-10 (running Solaris) for these tests instead. (Other JITs, such as Kaffe, do not yet support RMI.) The results are shown in Table 4. As can be seen, the performance gap between the JIT and Manta is lower than between the JDK and Manta, but the gap is still large. Part of the difference in the communication times is due to Manta using Panda, which runs on UDP, whereas Sun RMI uses TCP. Also, the difference between Manta and the JIT in serialization and RMI-protocol overhead is still large. With 3 object-parameters, for example, the difference is a factor 25 (1170 μs versus 46 μs).

Finally, we also measured the time to create a new thread for an incoming invocation request, which Manta uses for methods that potentially block. On the Pentium Pro, starting a new thread for an invocation costs 16 μs with the Manta run time system, so a remote call that is executed by a new thread costs at least 50 μs (on Myrinet). Our optimization for simple (nonblocking) methods thus is useful.

3.2 Throughput

The second benchmark we use is a Java program that measures the throughput for a remote method invocation with an array of a primitive type as argument, and no return type. The reply message is empty, so the one-way throughput is measured. In Manta, all arrays of primitive types are serialized with a memory copy, so the actual type does not matter. The resulting measurements were shown in Table 1 in Section 1.

The table also shows the measured throughput of the Panda RPC protocol, with the same message size as the remote method invocation. Two versions of Panda are shown. The basic version, with which almost all measurements in this paper are performed, is Panda 3.0. On Myrinet we have also performed measurements with Panda 4.0, which supports a scatter/gather interface. This scatter/gather interface makes it possible to remove some copying of user data from the critical path, resulting in an improved throughput. Unfortunately, dereferencing the scatter/gather vector involves extra processing, which increases the latency somewhat. Panda 3.0 achieves a throughput of 55.7 MByte/s on Myrinet, which is much higher than the throughput for Manta (20.6 MByte/s). The difference is due to two extra memory copies that Manta RMI needs for serialization (at the sending side) and deserialization (at the receiving side). Since memory-copies are expensive on a Pentium Pro [8], they decrease throughput significantly. For larger array sizes, the memory-to-memory copies have a larger impact on the performance. On Panda 4.0 the extra copying is avoided, and we achieve a throughput of 51.3 MByte/s, compared to 59.4 MByte/s of the raw Panda 4.0.

For the Sun JIT throughput is significantly less, and even more so for the JDK. On UltraSparcs-10 with Fast Ethernet, Manta obtains a throughput of 7.4 MByte/s, the JIT obtains 4.1 MByte/s, and the JDK obtains only 1 MByte/s.

3.3 Manta versus Sun Protocol

Sun's RMI protocol contains type information overhead; Manta's RMI protocol is substantially leaner. We have measured the com-

	empty	1 object	2 objects	3 objects
Serialization	-	7	13	19
RMI Overhead	4	5	5	5
Panda	30	32	33	33
Total	34	44	51	57

Table 2: Breakdown of Manta RMI on Pentium Pro and Myrinet; times are in μs

	Manta				Sun JDK			
	empty	1 object	2 objects	3 objects	empty	1 object	2 objects	3 objects
Serialization	-	11	17	24	-	667	879	1088
RMI Overhead	5	10	9	10	907	947	942	948
Communication	227	232	235	243	799	795	797	862
Overall	233	254	262	278	1711	2409	2619	2899

Table 3: Breakdown of Manta and Sun JDK 1.1.4 on Pentium Pro and Fast Ethernet; times are in μs

munication traffic of the two protocols. The result is shown in Table 5. The table shows the number of bytes for a null RMI, for an RMI with a single integer argument, with a 100 element array of integer argument, and one with a single object containing an integer and a double as argument. The table also shows the communication times in microseconds, on a 200 MHz Pentium Pro over Fast Ethernet, including the serialization and RMI protocol processing overhead, for Manta and the Sun JDK 1.1.4.

The JDK protocol sends only moderately more data than the Manta protocol, yet the JDK spends a considerable amount of time in processing and communicating the data. Most of this time is spent in analyzing type information and managing streams. Manta shows that this run time overhead can be reduced significantly.

3.4 Application Performance

In addition to the low-level latency and throughput benchmarks, we have also used three parallel applications to measure the performance of our system. The applications are Successive Overrelaxation (a numerical grid computation), Traveling Salesperson Problem (a combinatorial optimization program), and Iterative Deepening A* (a search program). For TSP we used a 15 city problem, for SOR a 2048 x 512 matrix, for IDA* we solved a random instance of a sliding tile puzzle (with solution length 56). The applications are described in more detail in [20]. We have implemented the programs with Sun RMI 1.1.4 (on Fast Ethernet) and Manta/Panda 3.0 RMI (on Fast Ethernet and Myrinet). Figure 6 shows run times, in seconds, for the serial program, and for runs of the parallel program, on 1 and 16 processors. Note the different scale for the 16 processor run. The programs are run on the Pentium Pros on BSD.

Performance differences between Sun and Manta can be attributed to differences in serial execution speed (interpreter versus compiler) and to differences in the RMI run time system, which is why we show the speed of the serial code in addition to single processor performance of the parallel code. Furthermore, we have run the serial codes with the Kaffe just-in-time compiler, to give some idea of how Manta compares to a JIT.[2]

The measurements show that Manta is substantially faster than the Sun 1.1.4 JDK. Both for the serial and the parallel run times the difference is large, about an order of magnitude. These particular applications/problem sizes generate a communication pattern that is relatively coarse grain. Manta's performance advantage is therefore mostly due to higher speed of the serial code of the Manta compiler. For finer grain communication, the advantage of Manta's faster RMI implementation will become more prevalent. Even so, the relative difference in performance between Sun JDK and Manta is larger on 16 processors than on 1 processor, indicating that Manta's faster RMI subsystem does make a difference.

4 Related Work

Many projects for parallel programming in Java exist (see, for example, the JavaGrande web page at *http://www.javagrande.org/*). Titanium [32] is a Java based language for high-performance parallel scientific computing. It extends Java with features like immutable classes, fast multidimensional array access and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium into C. It is built on the Split-C/Active Messages back-end. The JavaParty system [24] is designed to ease parallel cluster programming in Java. In particular, its goal is to run multi-threaded programs with as little change as possible on a workstation cluster. JavaParty is implemented on top of Java RMI, and thus suffers from the same performance problem as RMI. Java/DSM [33] implements a JVM on top of TreadMarks [16], a distributed shared memory system. No explicit communication is necessary, all communication is handled by the underlying DSM. No performance data for Java/DSM were available to us. Breg et al [7] study RMI performance and interoperability. Krishnaswamy et al [18] improve RMI performance somewhat by using caching and UDP instead of TCP. Sampemane et al [27] describe how RMI can be run over Myrinet using the socketFactory facility. Gokhale et al [11] discuss high-performance computing issues for CORBA. Hirano et al [12] provide performance figures of RMI and RMI-like systems on Fast Ethernet.

Our system differs by being designed from scratch to provide high performance, both at the compiler and run time system level. For the non-polymorphic RMI part, Manta's compiler-generated serialization is similar to Orca's serialization [2]. The buffering and dispatch scheme is similar to the single-threaded upcall model [19]. Small, non-blocking, procedures are run in the interrupt handler, to avoid expensive thread switches. Optimistic Active Messages is a related technique based on rollback at run time [31]. Instead of kernel-level TCP/IP, Manta can use Panda on top of LFC, a highly

[2] The parallel codes cannot be run since Kaffe does not yet support RMI. We tried Kaffe version 0.92 and 1.0b3, on Linux and BSD. SOR and IDA* worked with Kaffe 1.0b3 on Linux, TSP worked with Kaffe 0.92 on BSD. We were unable to get other combinations to work. Kaffe's long run time for IDA* is due to its slow garbage collector.

	Manta				Sun JIT			
	empty	1 object	2 objects	3 objects	empty	1 object	2 objects	3 objects
Serialization	-	16	23	34	-	304	404	432
RMI Overhead	9	10	9	12	708	733	767	738
Communication	327	333	330	330	500	473	496	511
Overall	337	359	364	377	1210	1513	1670	1685

Table 4: Breakdown of Manta and Sun JIT 1.1.6 on Sparc Ultra 10 and Fast Ethernet; times are in μs

	Manta				Sun JDK			
	empty	1 integer	int [100]	1 object	empty	1 integer	int [100]	1 object
Bytes	28	32	456	56	64	68	488	97
Microsecond	233	234	313	254	1711	1750	3322	2725

Table 5: Manta Protocol versus Sun JDK 1.1.4 Protocol on Pentium Pro and Fast Ethernet

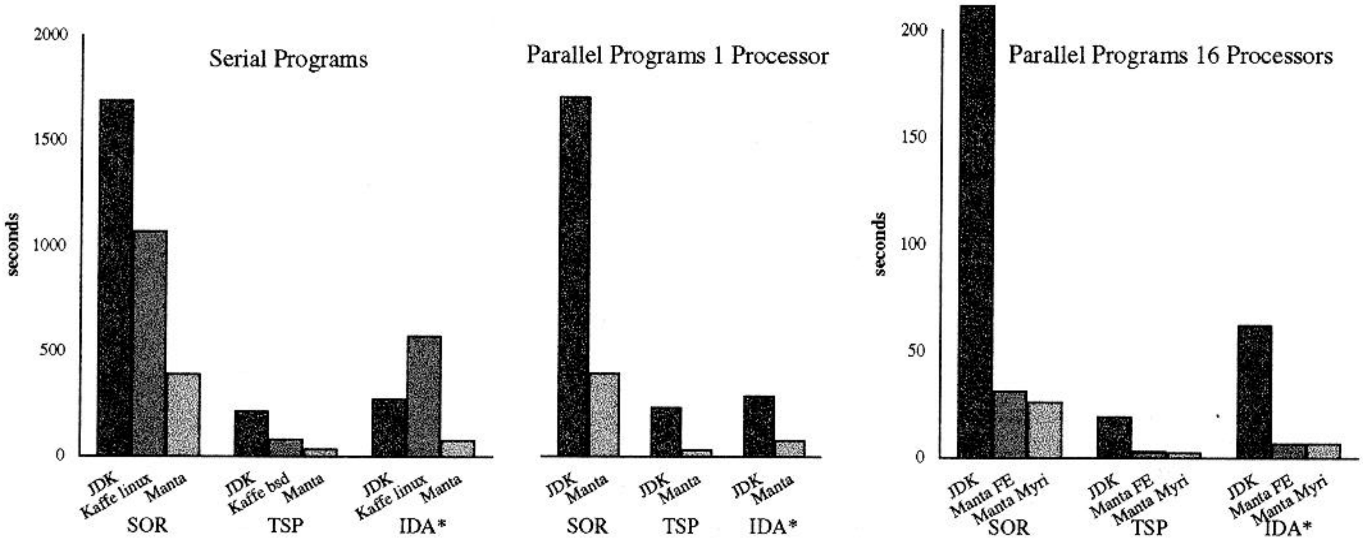

Figure 6: Application Run Time

efficient user-level communication substrate [5]. Lessons learned from the implementation of other languages for cluster computing were found to be quite useful. These implementations are built around user level communication primitives, such as Active Messages [29]. Examples are Concert [15], CRL [13], Orca [1, 2], Split-C [9], and Jade [26]. Other projects on fast communication in extensible systems are SPIN [3], Exo-kernel [14], and Scout [21].

5 Conclusion

We have built a new compiler-based Java system (Manta) that was designed from scratch to support efficient Remote Method Invocations on parallel computer systems. Performance measurements show that Manta's RMI implementation is substantially faster than the Sun JDK and JIT. For example, on Fast Ethernet, the null latency is improved from 1711 μs (for the JDK) to 233 μs, on Myrinet from 1228 μs to 34 μs, in both cases only a few microseconds slower than a C-based RPC. The gain in efficiency is due to three factors: the use of compile time type information to generate specialized serializers; a more streamlined and efficient RMI protocol; and the usage of faster communication protocols.

RMI is originally designed for flexible distributed (client/server) computing, and allows subclasses to be downloaded into a running program. Sun's implementation handles serialization, dispatch and buffer management at run time. It is designed for flexibility, not speed. Our system uses compile time information to make the run time protocol as lean as possible, so that processing it will be fast. Flexibility is achieved by recompiling classes and generating serializers as and when they are needed. Our implementation is designed for speed, yet preserves the polymorphism of RMI.

We find that with the right combination of user level messaging, compile time type information, and run time compilation, Java's RMI can be made almost as fast as a C-based RPC implementation while retaining the flexibility of RMI, making Java a viable alternative for high performance parallel programming.

6 Acknowledgements

This work is supported in part by a USF grant from the Vrije Universiteit. Aske Plaat is supported by a SION grant from the Dutch research council NWO. We thank Kees Verstoep for writing a Java socket layer for Myrinet on FM. Ceriel Jacobs and Rutger Hofman implemented and debugged a substantial part of the Manta system. We thank Raoul Bhoedjang for his keen criticism on this work. We thank Michael Philippsen for providing us with JavaParty, and for helpful discussions. We thank Thilo Kielmann for discussions on polymorphism in distributed object oriented languages and for his feedback on an earlier draft of this paper. We thank the anonymous referees for helpful feedback.

References

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

[2] H.E. Bal, R.A.F. Bhoedjang, R. Hofman, C. Jacobs, K.G. Langendoen, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, February 1997.

[3] B. Bershad, S. Savage, P. Pardyak, E. Gun Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, 1995.

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

[5] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Int. Conf. on Parallel Processing*, pages 381–390, Minneapolis, MN, August 1998.

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

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

[8] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 214–224, Seattle, WA, June 1997.

[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing*, 1993.

[10] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[11] A. Gokhale and D Schmidt. Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks. In *17th International Conference on Distributed Computing Systems*, pages 401–410, Baltimore, MD, 1997.

[12] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *ACM 1998 workshop on Java for High-performance network computing*, February 1998. Online at http://www.cs.ucsb.edu/conferences/java98/.

[13] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-performance All-Software Distributed Shared Memory. In *15th ACM Symp. on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, December 1995.

[14] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating Systems Principles*, pages 52–65, 1997.

[15] V. Karamcheti and A.A. Chien. Concert - Efficient Runtime Support for Concurrent Object-Oriented. *Supercomputing '93*, pages 15–19, November 1993.

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

[17] A. Krall and R. Grafl. CACAO -A 64 bit JavaVM Just-in-Time Compiler. *Concurrency: Practice and Experience*, pages 1017–1030, November 1997. Online at http://www.complang.tuwien.ac.at/andi/.

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

[19] K. Langendoen, R. A. F. Bhoedjang, and H. E. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, 5(2):28–38, April–June 1997.

[20] J. Maassen and R. van Nieuwpoort. Fast Parallel Java. Master's thesis, Vrije Universiteit, Amsterdam, August 1998. Online at http://www.cs.vu.nl/albatross/.

[21] D. Mosberger and L. Peterson. Making Paths Explicit in the Scout Operating System. In *USENIX Symp. on Operating Systems Design and Implementation*, pages 153–168, 1996.

[22] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa, a mixed offline compiler and interpreter for dynamic class loading. In *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, OR, June 1997.

[23] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[24] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, November 1997. Online at http://wwwipd.ira.uka.de/JavaParty/.

[25] T.A. Proebsting, G. Townsend, P. Bridges, J.H. Hartman, T. Newsham, and S.A. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, Portland, OR, 1997.

[26] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[27] G. Sampemane, L. Rivera, L. Zhang, and S. Krishnamurthy. HP-RMI : High Performance Java RMI over FM. University of Illinois at Urbana-Champaign, Online at http://www-csag.cs.uiuc.edu/achien/cs491-f97/projects/hprmi.html.

[28] R. Veldema. Jcc, a native Java compiler. Master's thesis, Vrije Universiteit, Amsterdam, August 1998. Online at http://www.cs.vu.nl/albatross/.

[29] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *The 19th Annual Int. Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

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

[31] D.A. Wallach, W.C. Hsieh, K.L. Johnson, M.F. Kaashoek, and W.E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 217–226, Santa Barbara, CA, July 1995.

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

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