Method Invocation Based Communication Models for Parallel Programming in Java

VRIJE UNIVERSITEIT

Method Invocation Based Communication Models for Parallel Programming in Java

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 17 juni 2003 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Jason Maassen

geboren te Vlissingen

promotor: prof.dr.ir. H.E. Bal

copromotor: Dr.-Ing. habil. T. Kielmann

Ik heb gewonnen!

Contents

List of Figures

List of Tables

Acknowledgments

Hurrah! I've reached the acknowledgments ...

It is very likely this acknowledgments section is the first part of this thesis that people will actually read. It is also the very last section that I will write. I'm almost done...

I really enjoyed life as a Ph.D student. It gave me the opportunity to do very interesting research and it had trips to far-away conferences as an added bonus. But the best part was that I got to work with a great group of people! I would like to use this acknowledgments section to thank them. In their own way, they all contributed to this thesis.

First of all, I would like to thank my supervisors, Henri and Thilo. They always gave me the feeling that I could pursue my own research interests and supported me when this work had to be translated into publications. Thank you, for all your advice, your knowledge, your support, and especially for your patience. I know that writing this thesis took far too long.

Next, I would like to thank all the people who are (or were) part of the Orca research group at the VU. Raoul and Aske, who helped us out in the early years of the Manta project. Rutger and Ceriel, the two programmers who have been of vital importance to the work presented in this thesis. Without their help, we would never have been able to get all (?) of the bugs out of Manta. John and Kees, who have managed to keep the DAS system running for all these years, even when I was one of the last people using it. Thanks to them I was able to present a consistent set of measurements in this thesis. The DAS is old and tired now. You can switch it off. I'm done.

And now for Rob and Ronald, the other two "creators" of Manta. I honestly cannot remember when we first met, but it must have been almost ten years ago, in 1993, when we all started our computer science studies at the VU. We've had an incredible amount of fun since then. Do you guys remember all the hours spent in the PC room, trying to write the fastest 3-D rotating coffin ever ? Remember the stress of the compiler construction course, the LAN-party at Mark's, the holiday in Florida,

and Ronald's infamous "Follow me!" ? We finally ended up doing a joint masters project, which resulted in the first version of Manta. After this, the three of us decided to become PhD students in Henri's group, and became room mates at the university. Although we each had our own research area, we all used Manta as the basis for our research. As a result, we closely cooperated in the development of Manta and often visited conferences together. The annual ASCI conference was our favorite. It had a pool.

Although Ronald has now moved on, I still have the pleasure of working with Rob on a daily basis. I have really enjoyed myself all these years. Thanks guys! Oh, before I forget, *I won!*

I would also like to thank Vladimir Getov, Dick van Albada, Chris Laffra, Koen Langendoen, and Maarten van Steen, who together form the reading committee for this thesis. Thank you for all the useful comments.

Next, I would like to thank my family. My parents always told me that a good education was very important. I agreed, and as a result I've been in school and studying for the last 24 years. They have always supported me during all this time. I am very grateful for that. Also, for the last eight months now, my father has been helping us rebuild our house. Thank you very, very much, we could not have done it without you. I would also like to thank Valesca (my sister) and Dave. They have promised me to take care of the audiovisual part of my defense.

Special thanks goes to my grandfather who taught me how to multiply when I was a very small boy. A little extra help in the beginning goes a long way.

And finally, Paula. I'm afraid that writing this thesis was as hard on you as it was on me. Nevertheless, you've always supported me. I love you.

Jason Maassen
April, 2003

Chapter 1

Introduction

There is a growing interest in using Java for high-performance parallel applications. Java offers a clean and type-safe object-oriented programming model and its support for concurrency, heterogeneity, security and garbage collection make it an attractive environment for writing reliable, large-scale parallel programs. Java supports two forms of concurrency. For shared-memory machines, it offers a familiar multithreading paradigm. For distributed-memory client-server applications, Java provides Remote Method Invocation (RMI) [98], an object-oriented version of Remote Procedure Call (RPC) [11]. The RMI model offers many advantages for distributed programming, including a clean integration with Java's object model, support for heterogeneity, polymorphism, security, and dynamic class loading.

In RMI, communication is expressed by invoking methods on objects that are located on another machine. All parameters to this method invocation are automatically forwarded to the destination machine, which executes the method and returns a result. This *method invocation* based communication bears a close resemblance to the communication between regular Java objects (i.e., objects that are located on a single machine). As a result, RMI integrates cleanly into the Java object model. The parameters and result values of an RMI are forwarded in a well-defined, machine-independent format, allowing RMI to be used in a heterogeneous environment.

In RMI, an actual object parameter to a method may be a *subclass* of the method's formal parameter, a feature known as *polymorphism*.[1] The same holds for (object) result values. When this subclass is not yet known to the receiver, it can be fetched from a file or HTTP server and loaded into the running program. This high level of flexibility is the key distinction between RMI and RPC [107]. RPC systems simply use the static type of the formal parameter (thereby type-converting the actual parameter), thus lacking support for polymorphism.

[1] 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.

We believe that the RMI model is an attractive starting point for supporting high-performance parallel programming in Java. Because the RMI model integrates cleanly into Java's object model, it is reasonably straightforward to convert sequential Java applications to distributed or parallel RMI applications.

Unfortunately, early Java implementations had inferior performance of both sequential code and RMI communication, a serious disadvantage when using RMI for high-performance computing on distributed-memory machines. Much effort has been invested in improving sequential code performance by replacing the original bytecode interpretation scheme with just-in-time (JIT) compilers, native compilers, and specialized hardware [2, 18, 34, 55, 71, 73, 82, 86].

The communication overhead of RMI implementations, however, remains a major weakness. RMI was designed for client-server applications running in a distributed and heterogeneous environment, where latencies in the order of several milliseconds are typical. Therefore, the focus of the RMI standard [98] is on supporting heterogeneity, security, versioning, etc., rather than on performance. Parallel applications are often run on homogeneous, tightly-coupled cluster computers, however. These systems consist of several identical machines connected using some high performance network that provides latencies in the order of several microseconds. In such systems, support for heterogeneity, security and versioning, are of minor importance. Communication performance is important, however, since the main objective of running parallel applications on a cluster is to obtain a speedup over the original sequential application. Unfortunately, the performance of many RMI implementations is not sufficient.

A recent Ph.D. thesis by Breg [15] illustrates the severity of this problem. This thesis studies the performance of three parallel applications on five different RMI implementations on a Fast Ethernet cluster. All implementations obtain very poor (or no) speedups on all applications, due to the overhead of the RMI implementations. This overhead can be one or two orders of magnitude higher than that of lower-level models like RPC or the Message Passing Interface (MPI).

A second limitation of RMI is that it only supports *synchronous point-to-point* communication. Remote invocations can only be forwarded to a single destination, and the invoker always waits for a reply before continuing. Although synchronous point-to-point communication is perfectly suited for expressing communication in client-server applications, many parallel applications are difficult to implement efficiently using this limited model. Parallel applications often require *asynchronous communication* (e.g., without a reply), *group communication* (e.g., multicast, broadcast, or scattering and gathering of data), or *object replication* [6] to run efficiently. These primitives can be implemented using threads and (multiple) RMI calls, but we will show that this is cumbersome and less efficient than using low-level communication primitives such as asynchronous or broadcast communication.

A different approach to high-performance parallel programming in Java, is to use an external communication library such as MPI [19, 52, 90]. This increases expressiveness, but at the cost of adding a separate model based on message passing, which

does not integrate well into Java's (method-invocation based) object model. Also, MPI deals with static groups of processes, rather than with objects and threads. MPI's collective communication primitives must be explicitly invoked by all participating processes, forcing them to execute in lock-step, a model which is ill-suited for multi-threaded, object-oriented applications.

In this thesis we will determine how method invocation based communication in Java (such as RMI) can be made suitable for high-performance parallel programming. We will first use an existing RMI implementation to determine the major performance bottlenecks. We will then use this information to design a new RMI implementation, called Manta RMI. Manta RMI is designed from scratch to support parallel programming on (homogeneous) cluster computers. It obtains a high communication performance, while keeping the major advantages of RMI (integration into the object model and support for polymorphism). Using Manta RMI as a basis, we will then extend the RMI model to support alternative, more expressive forms of communication, such as object replication and group communication.

Because Manta RMI is specifically designed for high-performance parallel programming on homogeneous cluster computers, it does *not* adhere to the official RMI specification [98]. Manta RMI has no support for security or versioning, only very little support for heterogeneity, and it does not have Java's run-everywhere property. However, Manta RMI is designed to be source level compatible with Java RMI. This allows any Java RMI application to be converted to Manta RMI by simply recompiling it. By dynamically switching between Manta RMI and a non-optimized standard Java RMI implementation, the Manta platform also has the ability to interoperate with other Java platforms.

This rest of this chapter proceeds as follows. Section 1.1 states our goals and contributions. In Section 1.2, we will give a brief description of our cluster computer, the Manta system that we have used to implement Manta RMI, and the application kernels[2] that we will use to evaluate the performance of our communication primitives. Finally, Section 1.3 gives an outline for the remainder of the thesis.

1.1 Goals and Contributions

The goal of our research is to create a (prototype) Java platform that is suitable for high-performance parallel programming on homogeneous cluster computers. This platform must provide highly efficient communication, preferably using communication models that integrate cleanly into Java and are easy to use. We believe that the RMI model is an attractive starting point for developing such a platform. Therefore, the focus of our work will be on creating a highly-efficient RMI implementation and

[2] An application kernel is not a 'complete' application. It only implements a specific parallel algorithm. Therefore, application kernels are usually smaller and simpler than 'real' applications, but larger and more complex than benchmarks.

extending the RMI model to support alternative, more expressive forms of communication, that both improve the performance and reduce the complexity of parallel applications. We make the following contributions towards reaching this goal:

- We give a description of the RMI model and analyze the performance of an existing RMI implementation to assess its suitability for high-performance parallel programming.
- We have designed and implemented an alternative, high-performance RMI that is suitable for use in parallel applications.
- We introduce *Replicated Method Invocation* (RepMI), a new approach for object replication in Java that is designed for efficiency, ease of use, and a clean integration into the RMI model. RepMI can be used to efficiently express shared data in parallel applications,
- We introduce *Group Method Invocation* (GMI), a highly-expressive extension of the RMI model that provides efficient communication with groups of objects. Using GMI, group and collective communication can be expressed efficiently. Unlike traditional message-passing based systems, this new (method-invocation based) form of group communication integrates cleanly into the Java language and the RMI model.
- We evaluate the performance of our communication primitives using several micro-benchmarks and a set of thirteen application kernels.

1.2 Experimental Environment

We will now give a brief description of the experimental environment we have used to run our experiments and develop our programming models.

1.2.1 The Distributed ASCI Supercomputer

All experiments described in this thesis are run on the *Distributed ASCI Supercomputer* (DAS) a homogeneous cluster of 64 machines, each containing a 200MHz Intel Pentium Pro and 128 MByte of main memory. The system runs RedHat Linux 6.2 (kernel version 2.2.16).

All machines are connected by two different networks: 100 Mbit/s Fast Ethernet and 1.2 Gbit/s Myrinet [13]. The Fast Ethernet network is mainly used for control purposes, such as sharing files over NFS, starting applications, etc. The Myrinet network is reserved for the communication of parallel applications.

Each Myrinet network interface board contains a programmable network processor (a 33 MHz LANai RISC processor), that can be used to run custom network control programs. The boards also contain 1 MByte of SRAM memory, that is used store both

the code and data of the control program (e.g., the message data). The Myrinet boards are connected using 8-port crossbar switches and high-speed links (1.28 Gbit/s in each direction).

The Myrinet boards are controlled using the LFC Myrinet control program, which is described in detail in [9]. LFC supports unicast, multicast and broadcast communication. To prevent expensive calls to the kernel, LFC provides user-space access to the Myrinet network. To receive messages, an efficient *upcall* [10, 24] mechanism is used. Briefly, whenever a message is received, LFC will invoke an application defined function that handles the message (e.g., copies the message or performs some computation). The main advantage of using upcalls is that it allows application-specific message processing at times that cannot be predicted by the application.

Note that we do not use LFC directly, but use the *Panda* [94] library instead, which is implemented using LFC. Panda will be described in more detail below.

1.2.2 Manta

The communication models that we will describe in this thesis have all been implemented using the *Manta* platform. Manta consists of a highly optimized runtime system and native Java compiler that directly compiles Java source code to native executables.

We have chosen to implement our own Java system (Manta), because existing Java systems (e.g., provided by Sun Microsystems Inc.[3] or IBM Inc.[4]) operate as a *black box*. There is no control over the code generated by the JIT compiler, nor is there any low-level access to the runtime system. These restrictions make it difficult to thoroughly evaluate the performance of current RMI implementations. It also limits the number of possible optimizations, since all optimizations must be expressible in bytecode.

We have therefore chosen to implement our own compiler and runtime system, which will be explained briefly below. The focus of this description will be on the parts of Manta that are not directly related to RMI communication. The communication related parts of Manta will be described in detail in Chapters 2 and 3.

The compiler

Instead of using bytecode interpretation or JIT compilation, the Manta system uses a native compiler. One of the difficulties of building a JIT compiler is that the compiler competes with the application for system resources like CPU cycles and memory. As a result, the compilation and optimizations done by the JIT compiler must not only produce efficient code, but must do so within strict memory and time limitations. A native Java compiler compiles the program ahead-of-time and therefore does not

[3] http://java.sun.com/j2se
[4] http://www-106.ibm.com/developerworks/java

compete for resources with the application. It can take all the time and memory it needs to compile and optimize the program. As a result, a native compiler is easier to implement than an (efficient) JIT compiler. Since our primary interest is in the efficiency of RMI communication in Java (not in building JIT compilers), we chose to use a native Java compiler for Manta.

Using the Manta compiler, Java source code is directly compiled into a native executable. We compile Java source code (instead of Java bytecode) because this allows us to explore the possibilities of modifying the Java language itself. For example, by adding a single keyword, *remote*, we can not only compile applications that use the standard RMI syntax, but also applications that use the syntax introduced by JavaParty [85]. For simplicity we will only use the standard RMI syntax in this thesis.

The runtime system

Next to a compiler, Manta also includes a Java runtime system, which is mostly implemented in C and makes extensive use of the Panda library. It consists of the following parts:

- High performance communication
- Interoperability
- Memory management (including garbage collection)
- Java libraries
- Native support (for I/O, threads, synchronization, etc.)

The first two parts (high performance communication and interoperability) will be explained in detail in the following chapters. Automatic memory management is one of the attractive features of Java. To implement this, the Manta runtime system uses a mark-and-sweep garbage collector, which is described in detail in [64].

A significant part of the Manta runtime system consists of the Java class libraries. These libraries contain a large number of packages ranging from language support (java.lang), I/O (java.io) and network support (java.net) to miscellaneous utilities like hash tables and lists (java.util). These libraries are mostly implemented in Java. However, some 'native' support, implemented in C, is also required (for I/O, threads, synchronization etc.). To implement this, we make extensive use of the Panda library, which we will now decribe briefly.

The Panda library

Panda [94] is a portable library designed to support implementations of parallel programming systems. Panda was originally designed to implement a portable version of

Orca [6], but has also been used in the implementation of other systems, such as MPI, PVM (Parallel Virtual Machine) and CRL (C Region Library).

The two main abstractions that are provided by Panda are *communication* and *threads*. For communication, Panda offers both point-to-point message passing and broadcast communication. Communication in Panda is *reliable*; messages are guaranteed to be delivered to the receiver. Panda also offers *totally-ordered* broadcast communication, where broadcast messages are guaranteed to be delivered *in the same order* to all machines. Totally-ordered broadcast is used to implement object replication.

To minimize the number of memory copies required during communication, Panda supports a scatter/gather interface. This interface does not require the user to provide a fully-assembled message. Instead the user provides Panda with an *I/O vector*, an array of *(pointer, size)* structures, each referring to a piece of data that should be included in the message. Panda then assembles the message while the data is transferred, resulting in a high throughput. Like LFC (described in 1.2.1), Panda uses an *upcall* model for message delivery.

The *threads* abstraction offered by Panda provides the functionality to *create*, *join*, *yield*, and *exit* threads. It is able to support *prioritized* threads and *preemptive* thread scheduling, and provides *locks* and *condition variables*. The Panda library is described in detail in [94].

Panda has been implemented on a large number of systems. For our Manta implementation on the *DAS*, we use two different Panda implementations. The first provides communication over Fast Ethernet (implemented using UDP), and will only be used briefly in Chapter 2. The second Panda implementation provides communication over Myrinet (implemented using LFC) and will be used throughout this thesis.

Both versions use the same user-level thread package [42] to implement the Panda threads abstraction. This abstraction is used in the Manta runtime system to implement the Java thread and synchronization primitives.

1.2.3 Application Kernels

In this thesis, we describe an existing RMI implementation, a high-performance RMI implementation (Manta RMI), and two new models that provide object replication (RepMI) and group communication (GMI). To evaluate the performance of the implementations of these communication models, we use a set of thirteen application kernels. However, not every application kernel is used to evaluate every communication model. Table 1.1 shows for each the communication models which of the application kernels are used to evaluate their performance. We will provide a short description for each of the applications in the following chapters.

Table 1.1 also shows the type of communication that is required to implement each of the kernels. In general, three types of communication can be identified, *point-to-point* communication, *shared data* and *group or collective* communication.

	Communication Models				
Kernel	RMI	Manta RMI	RepMI	GMI	*Communication Style*
SOR	X	X			point-to-point
Radix	X	X			all-to-all
Water	X	X			point-to-point
Barnes	X	X			point-to-point + broadcast
FFT	X	X		X	all-to-all
ASP	X	X	X	X	broadcast
LEQ		X	X	X	gather-to-all
QR		X	X	X	reduce-to-all + broadcast
ACP		X	X	X	shared data
TSP		X	X	X	shared data
LUFact				X	broadcast
Moldyn				X	reduce-to-all
Raytracer				X	point-to-point

Table 1.1: The application kernels.

In point-to-point communication, data is sent from one sender to one receiver. This type of communication can easily be expressed using RMI. However, RMI communication is also *synchronous* (the invoker always waits for a reply before continuing), which may reduce the efficiency of some applications. RMI communication will be the focus of Chapters 2 and 3.

Some applications use shared data to express communication. A data structure is shared between all machines and is frequently read but only occasionally updated. Although the RMI model is perfectly suited to express shared data, using RMI will often result in inefficient applications due to problems with *data locality*. Storing the data in a single remote object, on a single machine, is inefficient because it requires an expensive RMI to read the data from another machine. This problem will be the focus of Chapter 4.

In group or collective communication, multiple machines are involved in a single communication operation. A *multicast*, for example, forwards data to multiple destinations, while a *broadcast* forwards data to *all* participating machines. In a *reduce-to-all* operation, several data items, provided by different machines, are combined to a single value by applying some function (e.g., determining the *maximum*, *minimum*, or *sum* of all items). This value is then returned to all participants. The *gather-to-all* operation is similar, but instead of reducing the data items, it gathers them together. In the *all-to-all* operation, data is exchanged between all participants.

Note that in this thesis, we will use the term *group* communication to denote communication that is initiated by a single sender and has multiple receivers. We use the term *collective* communication to denote communication that is initiated by multiple senders, and has one or more receivers. Using RMI to express group and collective communication is cumbersome and often inefficient. This problem will be the focus of Chapter 5.

1.3 Outline

The outline for the remainder of the thesis is as follows. To provide this thesis with a solid foundation, Chapter 2 describes the RMI model, its relation to object serialization, and implementations and performance of both mechanisms. In Chapter 3 we will describe our high-performance Manta RMI implementation and compare its performance to existing implementations using benchmarks and application kernels. These chapters are based on the work described in [65] and [66]. This work is the result of joint research performed by R. van Nieuwpoort, R. Veldema, and the author. Chapter 4 introduces a new compiler-based approach for object replication in Java that is designed to resemble RMI. This chapter is based on [62]. In Chapter 5 we show how the RMI model can be extended to allow communication with a group of objects. This chapter is based on [63]. Chapter 6 concludes. Related work will be dealt with individually in each chapter.

Chapter 2

Remote Method Invocation

2.1 Introduction

In this chapter we will describe the underlying model, current implementations and general performance of RMI. We will also describe *serialization*, the mechanism used by RMI to transfer method invocation parameters and result values from one *Java Virtual Machine* (JVM) to another. The performance of RMI operations is strongly influenced by the serialization performance.

The description of RMI and serialization in this chapter is based on the implementation provided by Sun's Java Developer Kit version 1.1. Although the implementation of both RMI and serialization changes with each Java release, the general approach has remained the same.

To analyze the performance of the current RMI and serialization implementations and identify specific performance bottlenecks, we will use micro benchmarks and application kernels. As explained in Section 1.2.2, doing this analysis using a JIT-compiler would be difficult. In that case, the performance of an application depends heavily on decisions made by the JIT compiler (e.g., which methods are compiled, when are they compiled and how much time is spent on optimizing them). Since most JIT-compilers operate as a black-box, they don't lend themselves well for measurements. Therefore, we have compiled the Sun JDK 1.1 RMI implementation, the micro benchmarks, and the applications kernels using the Manta compiler. This approach provides us with a predictable performance, since there is no JIT-compiler competing with the application for system resources. It also allows us to do detailed measurements inside the runtime system.

We will postpone the description of related work to the end of the next chapter, which describes our optimized RMI implementation. There, we will describe alternative RMI and serialization implementations and different communication models, instead of RMI, that can be used for parallel programming.

2.2 The RMI model

RMI [98] can be seen as an *object-oriented Remote Procedure Call* (RPC) [11]. It allows methods to be invoked on an object that is located on another JVM. Like a normal Java program, an RMI program consists of one or more threads that manipulate a collection of objects by invoking methods on these objects. In the RMI program, however, these threads and objects do not necessarily share the same address space, but may be distributed over multiple JVMs. Like in RPC, communication in RMI is *implicit*. A thread simply invokes a method on a reference to a *remote object* (i.e., an object located on another JVM). This method invocation will then automatically be transferred to the remote object, executed, and the result value or exception returned to the thread. An example is shown in Figure 2.1.

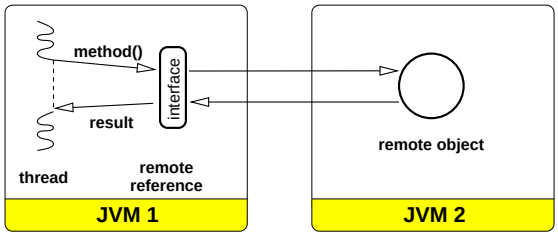

Figure 2.1: Invoking a remote method.

In the RMI model, references to remote objects are called *remote references*. Unlike normal references in Java, remote references may only refer to a special type of interface, a *remote interface*. As a result, RMI does not allow every type of object to be referenced from another JVM, but only objects that implement a remote interface. We will describe how a remote interface can be created in Section 2.3.

Since the parameters to an RMI may be transferred between JVMs, remote references have different parameter passing semantics than normal references. When a method is invoked using a remote reference (i.e., a remote method invocation), the parameters passed to the method are always passed *by-value*. The remote object receiving the method invocation will thus receive a *copy* of the parameters, regardless of the actual location of the object. The same holds for any result value (or exception) returned by the method.

When a remote reference is passed as a parameter to an RMI (or returned as a result), it will also be passed *by-copy*. On the receiving JVM, a new remote reference will be created that refers to the same remote object as the original remote reference. This remote object will remain at its original location.

When a remote object is passed as a parameter to an RMI, however, it will be passed *by-reference*. The remote object parameter will not be forwarded to the desti-

nation machine. Instead it will be replaced by a remote reference, which refers to the remote object parameter, at the original location.

All parameters to an RMI are forwarded using a machine-independent format, thereby supporting communication in a heterogeneous environment. Like normal method invocations, RMIs are *synchronous*. Therefore, thread that invokes the remote method must wait for a result before it can continue.

One of the most distinctive features of RMI is how it handles the types of the parameters of a method invocation [107]. In RPC, the type and size of the method parameters are determined statically, *at compile time*. This allows RPC to directly copy parameters into a network message and send it off to the destination. On the receiving side, the parameters can be copied from the message into pre-allocated buffers or used directly from the message buffer itself.

In RMI, however, the type and size of the method parameters are determined *at runtime*. This allows RMI to support truly object-oriented method invocations, where an actual object parameter to a method can be a *subclass* of the method's formal parameter. For example, if a method has a formal parameter of type *java.lang.Object*, it can receive any type of object as an actual parameter. This feature is known as *polymorphism*. Because of this polymorphism, the size and type of the parameters to an RMI cannot be determined statically. The RMI runtime system must therefore determine the type of the parameters at runtime, in order to forward them correctly.

Since the receiver of an RMI cannot know the exact type of the parameters that are forwarded to it, type information is included in the message. When a parameter is received of a class that the receiver has not seen before, it can fetch the bytecodes for this class from a file or HTTP server, and load them into the running program.

In addition to supporting polymorphism, RMI also supports linked data structures as parameters. For example, when the root object of a tree is passed as a parameter to an RMI, the entire tree is automatically forwarded to the destination.[1] The forwarded data structure may even contain cycles. To implement the parameter forwarding, RMI uses *Object Serialization*, which will be explained in Section 2.4.2.

Because RMIs have an object as a target, not a machine or a process, RMI applications tend to be *finer grained* than RPC-based programs. Figure 2.2 shows an example. There, JVM 2 has three remote references of the same type, *Collection*, which refer to remote objects on JVMs 1 and 3. By invoking the *Collection.add* method on the different remote references, not only a destination JVM is selected but a specific remote object on that JVM. Note that a single JVM may contain multiple remote objects that export the same remote interface (as shown in the example).

Another feature of RMI is also shown in this example. Although the remote references used on JVM 2 are all of the same type (the *Collection* interface), the actual remote objects have different types (*VectorCollection* and *ListCollection*). Because a remote reference represents a certain *interface*, it hides the actual type (or implementation) of the remote object. This allows different remote object implementations to

[1] A special *transient* modifier is provided that can be used to prevent data from being serialized.

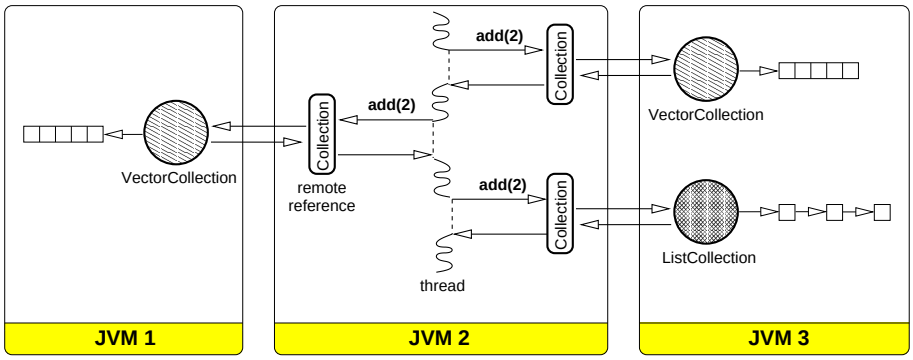

Figure 2.2: Multiple remote objects with the same interface.

be interchanged or used concurrently without any changes to the application.

As we will explain below, no extensions to the Java language are necessary to implement RMI. Instead, remote objects can be created using normal Java interfaces. This approach does require the use of a special RMI compiler that generates code to handle the communication needed for RMI.

These features, polymorphism, dynamic bytecode loading, and support for linked data structures, make RMI highly flexible and give it an expressive model. We believe that this makes RMI an attractive starting point for supporting high-performance parallel programming in Java. Unfortunately, due to its flexibility, RMI also requires a complex runtime system. In the following sections, we will first briefly describe the RMI syntax, and then describe a typical implementation of the RMI runtime system[2], and its performance.

2.3 The RMI syntax

In this section, we will use a simple example application to illustrate how a distributed application can be created using RMI. We will briefly show how to define a remote interface and a remote object, and how they can be used to create a simple client-server application.

2.3.1 Defining a remote object

To create an RMI application, we first have to create a remote interface that defines which methods may be invoked from a different JVM. Figure 2.3 shows an example, where an interface *Print* is defined that contains a single method. To turn

[2]This description is based on the Sun JDK 1.1 RMI implementation.

```
interface Print extends java.rmi.Remote {
   public void print(String t) throws java.rmi.RemoteException;
}
class Printer extends java.rmi.server.UnicastRemoteObject
              implements Print {

   public void print(String t) throws java.rmi.RemoteException {
      System.out.println(t);
   }
}
class Server {
   public void static main(String [] args) {
      ... initialize RMI here ...
      Naming.bind("MyPrinter", new Printer());
   }
}
class Client {
   public void static main(String [] args) {
      try {
         ... initialize RMI here ...
         Print rp = (Print) Naming.lookup("MyPrinter");
         rp.print("Hello world!");
      } catch (java.rmi.RemoteException e) {
         ... handle the exception here ....
      }
   }
}
```

Figure 2.3: An example RMI application (pseudocode).

Print into a remote interface, it must extend the interface *java.rmi.Remote*. Although *java.rmi.Remote* does not define any methods, it serves as a *marker interface* that allows the RMI compiler and runtime system to recognize remote interfaces. Also, all methods of a remote interface must declare to throw a *java.rmi.RemoteException*, that is used to report communication problems and to forward application exceptions from one JVM to another.

In Figure 2.3, the *Printer* class illustrates how a remote object is defined. By providing an implementation for the *Print* interface, *Printer* becomes suitable for receiving remote invocations. *Print* also extends the class *UnicastRemoteObject*, which contains some basic functionality for remote objects (e.g., an RMI-aware implementation of *equals* and *hashCode*).

As this example illustrates, no extensions to the Java language are necessary to implement RMI. Instead, a special RMI compiler (e.g., *rmic*) is used that recognizes objects that implement a remote interface and generates extra code to implement the RMI communication. This will be explained in more detail in Section 2.4.1.

2.3.2 Binding remote objects

Before an object can receive remote invocations, it must first be *bound*. Binding the object serves two purposes. First, it allows the RMI runtime system to set up any data structures and network connections it needs to allow the object to communicate. Second, a name is associated with the object. Using that name, the location of the object is then published in an *RMI registry* (the RMI naming service). Other JVMs can communicate with this registry to look up any information they need to create a remote reference to the exported object.

The example of Figure 2.3 shows two classes *Server* and *Client*, which are run on two separate JVMs. The *Server* first creates a new *Printer* object and binds it to the name *"MyPrinter"* using the *Naming.bind* method. The *Client* can then find the *"MyPrinter"* object on the server and create a remote reference to this object by using *Naming.lookup*. As shown, this remote reference can then be used to do RMI calls on the *Printer* object.

2.4 The RMI implementation

A distributed application using RMI can be divided into several layers, as shown in Figure 2.4. We can make a distinction between *Java code* and *native code*. The application and most of the libraries it uses are implemented in Java. This Java code runs on top of a Java Virtual Machine layer, which consists of a runtime system (responsible for memory management, I/O, network support, etc.) and an interpreter and/or JIT-compiler. The JVM may use *native libraries*, like a communication library or a thread package. Both the JVM and native libraries are considered native code, since they are usually implemented in C.[3]

The Java code can be divided into four separate layers, *Application*, *RMI*, *Serialization* and the *Network Library*. The RMI layer can be divided further into two separate parts, *Generated Code* and the *RMI Runtime*. Note that the application can use other Java libraries than just RMI (e.g., for I/O or a graphical user interface). This is not shown in Figure 2.4 since the focus of this chapter is on RMI.

The network library consists of a thin Java layer that provides an interface to a (possibly kernel-based) TCP/IP implementation written in C. It contains a Java class representing a *socket* which, like all other I/O in Java, provides a streaming communication model. Since the network library provides a rather straightforward Java interface to TCP/IP sockets, it will not explained in detail. In the rest of this section, we will describe a typical implementation of the two RMI layers and serialization (this description is based on the Sun JDK 1.1 implementation). We will start by explaining the generated RMI code, then explain serialization, and finally briefly look at the RMI runtime system.

[3] A notable exception to this is the *Jalapeño* [18] system, which implements a JIT compiler in Java.

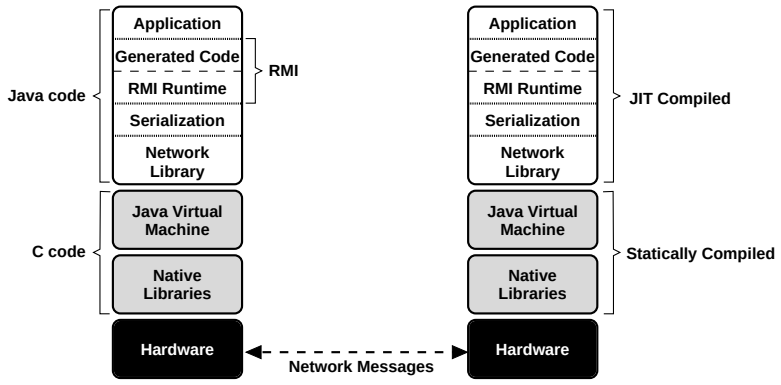

Figure 2.4: Layers of a distributed RMI application

2.4.1 Generated RMI code

Compiling the example application of Figure 2.3 using a normal Java-to-bytecode compiler (e.g., *javac*) does not immediately result in a distributed RMI application. Only when the RMI compiler *rmic* is applied, will the necessary RMI communication code be generated. The RMI compiler generates two extra classes for every remote object, a *stub* and a *skeleton*. The stub provides a special compiler-generated implementation of the remote interface (i.e., *Print*), which is able to forward method invocations to another JVM. This *marshaling code* allows the stub to be used as a remote reference to the object.

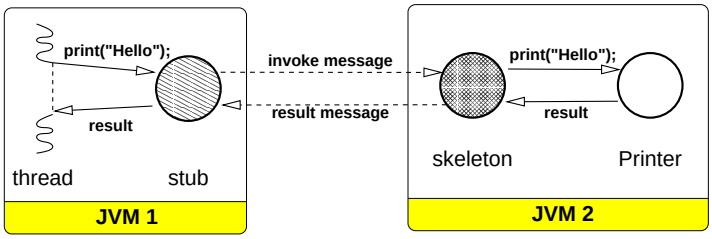

Figure 2.5: An invocation on a stub.

An example is shown in Figure 2.5. There, a stub on JVM 1, implementing the *Print* interface, is acting as remote reference to a *Printer* object on JVM 2. Any method invocations on the stub are forwarded to the skeleton, which applies them to the *Printer* object and returns the results.

```
class Printer_Stub extends Stub implements Print {
   void print(String p0) throws java.rmi.RemoteException {
      try {
         RemoteCall call = newCall( ..., PRINT_METHOD_NUMBER, ... );
         ObjectOutput out = call.getOutputStream();
         out.writeObject(p0);
         invoke(call);
         done(call);
      } catch ( ...) { // exception handling }
   }
}
class Printer_Skel implements Skeleton {
   void dispatch(Remote o, RemoteCall c, int methodNumber, ... ) {
      Printer server = (Printer) o;
      switch (methodNumber) {
         case PRINT_METHOD_NUMBER:
           try {
              String p0;
              ObjectInput in = c.getInputStream();
              p0 = (java.lang.String) in.readObject();
              server.print(p0);
              c.returnResult(OK);
           } catch ( ... ) { // exception handling }
           ...
      }
}
```

Figure 2.6: The generated stub and skeleton classes of *Printer* (pseudocode).

Figure 2.6 shows pseudocode for the stub and skeleton. When the *print* method is invoked on the stub, a *RemoteCall* object is created to represent the RMI. After writing the necessary information (i.e., the method number and parameters) into this object, the *invoke* method forwards the RMI to the destination skeleton and waits for a reply. Even though the *print* method does not return a result, a result message will still be send. This messages is used to indicate to the client that server has finished executing the method and that no exceptions have been thrown.

The skeleton serves as a bridge between the stubs and the remote object. As shown in Figure 2.6, it contains a *dispatch* method that is used to handle incoming RMI messages. When an RMI call arrives, the RMI runtime system delivers it to the skeleton using a *RemoteCall* object. The skeleton code will then extract the parameters and invoke the method on the local object. Any return value, including exceptions, is returned to the waiting stub.

To forward a method invocation to another JVM, the RMI runtime system must be able to transfer method parameters (and a result value) from one JVM to another. This is done using *serialization*, which we will explain in the following section.

2.4.2 Serialization

Serialization [97] is used to convert objects into bytes and vice-versa. It allows objects to be stored in a file or to be transferred to another JVM (e.g., by using a network message). For an object to be *serializable*, it must implement the *java.io.Serializable* interface. This interface does not define any methods, but serves as a marker that the object may be converted to bytes. Objects that do not implement this interface cannot be serialized. Figure 2.7 shows a simple example of a serializable class, *Data*. This class contains a single integer field, *i*, and a *toString* method which can be used to print its value.

```
class Data implements java.io.Serializable {
    int i = 42;
    public String toString() {
        return "Value is " + i;
    }
}
```

Figure 2.7: Serialization example.

Implementation of serialization

Serialization, as defined in [97], is implemented in two Java classes in the *java.io* package. An *ObjectOutputStream* can be used to convert an object into bytes, an *ObjectInputStream* can convert these bytes back into an object. Like all I/O in Java, serialization is based on a *streaming* model, which makes it easy to use. For example, by simply connecting an *ObjectOutputStream* to a *SocketOutputStream*, objects can be written to a network connection, while connecting to *FileOutputStream* allows objects to be stored in a file.

Figure 2.8 shows the steps involved in serializing an object. When an object is written to an *ObjectOutputStream*, the stream first checks if the object is indeed serializable (i.e., implements the *java.io.Serializable* interface), and throws an exception if it is not. The distinction between serializable and non-serializable objects is important since some objects may contain data that may not be saved to disk or transferred to another JVM, such as machine specific data (e.g., file descriptors) or security related information (e.g., passwords).

The next step in serialization is *cycle detection*. Serialization does not only support the conversion of single objects, but can also convert linked data structures like lists, trees, and graphs. For example, when the first object of a list is serialized, one of its data fields will refer to the second object, which is then also serialized, etc. As a result, the entire list will (recursively) be converted into bytes.

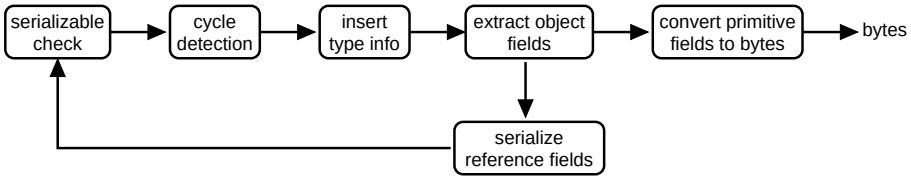

Figure 2.8: Converting an object into bytes

These linked data structures pose a problem, because they can contain *cycles* and *doubles* (i.e., objects that are reachable through multiple paths in the graph). An example of both is shown in Figure 2.9. For this reason, cycle detection plays an important part in the serialization of complex data structures. The *ObjectOutputStream* assigns a numerical identifier to every object is serializes, the *object handle*. All objects and their handles are stored (e.g., in a hash table), allowing the cycle detection mechanism to check if it has seen an object before. If it encounters an object for the second time, it is not serialized, but instead replaced by its handle. This prevents the *ObjectOutputStream* from serializing an object more than once or being caught in an endless loop, continuously serializing the same cycle of objects.

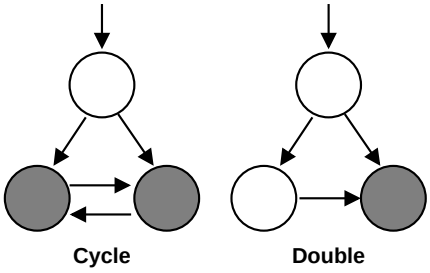

Figure 2.9: Linked data structures with cycles and doubles.

Serialization does not only save the state of the object, but also any type information required to later recreate the object. In the next step (*insert type info*), the *ObjectOutputStream* checks if it has previously seen an object of this type. If not, a type description is saved, which consists of the fully qualified name of the class (e.g., *"java.lang.String"*), a *version number*, a description of the fields in the object (their type and name) and, if required, a type description of the parent class. The version number (a signature checksum that is calculated using a description of the object's fields and method) is saved to ensure that the receiving JVM has a compatible implementation of the class. Note that this implementation does not have to be exactly the

same. Using the description of the fields, the receiving JVM may be able to extract the object, even if there are differences between the local and serialized versions. This feature, know as *versioning* can be used for backwards compatibility.

When the type information is saved, the *ObjectOutputStream* assigns a *class handle* to it. If, at a later point in time, another object of the same class is encountered, only this handle is stored instead of the entire class description. No bytecodes are included in the type description. The implementation of the class is not transferred, only a description of its implementation.[4]

To obtain the class descriptions, the *reflection* mechanism of Java can be used. This mechanism allows all required information about a class to be retrieved *at runtime*. For example, using *java.lang.Class* and the *java.lang.reflect* package, it is possible to get a description of all the fields in a certain class of object, including their type, name and modifiers (e.g., *public*, *private*, *static*, etc.). Similarly, a description of the methods in a class can be retrieved, and information about the class hierarchy (i.e., which interfaces are implemented by a certain class, what is its parent class, etc.).

Next, the *ObjectOutputStream* saves the state of the object. One by one, the values of each of the fields in the object are extracted. The values of primitive fields (e.g., *int* or *long*) are then converted to bytes (using operations like *and*, *or* and *shift*) and written to the underlying stream. Fields of type *double* or *float* cannot be converted directly. Instead, the native methods *Double.doubleToLongBits()* and *Float.floatToIntBits()* are used to convert them into *long* and *int* values, respectively. These values can then be converted into bytes. For reference fields, the serialization mechanism is invoked recursively to serialize the object referred to.

Not all fields of an object will be automatically serialized. For example, fields that are *static* (i.e., global variables) will be skipped. Java also has a special modifier, *transient*, that indicates that a field *must not* be serialized. This gives the programmer more control over the serialization process.

To get almost complete control over the (de-)serialization process, two special methods, *writeObject* and *readObject*, may be defined in a class. When the presence of these methods is detected, the object state will not be saved automatically. These methods will be invoked instead. This allows the programmer to implement his own serialization and decide what fields should be serialized. This mechanism can also be used to store extra information with the object (e.g., for authentication or security purposes).

Implementation of Deserialization

Compared to serialization, *deserialization* of an object is relatively straightforward. An *ObjectInputStream* reads bytes from an underlying stream, such as a file or a socket. First, it will read a class description and check if the class and version are

[4] Every *ObjectOutputStream* stores its own type and cycle information. If multiple *ObjectOutputStreams* are used to transfer the same set of objects they do not share any information, not even if they all transfer these objects to the same destination.

known to the JVM. Then, it creates a new object and reads the values for each of the object's fields from the stream. The Java reflection mechanism can be used to write these values into the object. To allow the *ObjectInputStream* to reconstruct a serialized graph that may contain cycles or doubles, it assigns an *object handle* to every object it encounters and stores these objects in a table. When it tries to deserialize the value of a reference field, it may read a handle instead of an object. Using the table, it can then look up the associated object and store a reference to it in the field. When the stream contains a new object, the deserialization routine is invoked recursively.

There is one issue, however, that make the implementation more complicated. When a new object is created, this cannot be done using a *new* call, because the class of the object is unknown at compile time. Although a *class* object contains a *newInstance* method that can be used to dynamically create objects of that particular class, this method cannot be used either. A call to *newInstance* will not only create an object, but also invokes the object's constructor, which may have side effects. Therefore, deserialization uses a special native function *allocObject*, that is provided by the Java Native Interface (JNI).[5] Using this function, objects can be created without invoking any of their constructors, thereby preventing any side effects.

Serialization Protocol

As explained above, serialization may store a considerable amount of extra information next to the actual object state. Figure 2.10 shows the bytes produced when serializing a *Data* object. In total, 33 bytes are required to store the serialized *Data* object.

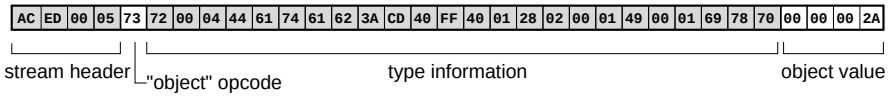

Figure 2.10: A serialized *Data* object

Of this 33 bytes, 24 are used to store type information about the *Data* class. Because the class and field names are stored as text, the size of the type information depends on the length of these names. Only four bytes are used to store the actual value of the object (i.e., a single integer value). The other five bytes are used for the stream protocol

Fortunately, the type information is only saved once for each type of object written to the stream. Adding a second *Data* adds 10 bytes to the stream instead of 29. Of these 10 bytes, two are stream protocol opcodes, four are used as a class handle (to

[5] The JNI defines how native (e.g., C) functions can be used from Java, and provides support for using Java objects and methods in native code.

refer to type information that was saved earlier) and four store the actual data. This illustrates that the overhead of the stream protocol and type information can be significant. To further investigate this, we use a simple application that serializes 256 integer values using eight different approaches: serializing 256 separate *int* values, an array of 256 *ints*, 256 objects each containing one *int*, an array of 256 of such objects, a single object containing 256 *int* fields, a single linked list, a double linked list, and finally, a balanced binary tree.[6]

The results are shown in Table 2.1. For each approach, the *payload* (i.e., the amount of user data serialized) is the same, 256 *int* values using 1024 bytes. Any additional bytes are either needed for the serialization protocol, for type information, or to store the structure of the data itself (e.g., array sizes or object references). Next to the total bytes required, the table also shows a breakdown of these bytes into these three categories.

	Bytes	*Extra Information*			
Serialized Data	*Required*	*protocol*	*type*	*structure*	*total*
256 separate ints	1033	9	-	-	9
int[256]	1051	5	18	4	27
256 objects (1 field)	2580	260	1296	-	1556
object[256]	2605	261	1316	4	1581
object (256 fields)	2274	5	1245	-	1250
single linked list(256)	2591	260	1306	1	1567
double linked list(256)	3876	260	1315	1277	2852
binary tree(256)	2856	260	1315	257	1832

Table 2.1: Extra information required by different data structures (in bytes).

Writing 256 separate *int* values introduces little overhead. Only nine additional bytes are required for the serialization protocol. When serializing an *int[256]*, however, a type description of the array needs to be stored, adding 18 bytes to the stream. In addition to this, four bytes are used to store the size of the array. When we serialize 256 objects, each containing an *int* value, the number of additional bytes required to serialize the data increases sharply (even becoming larger than the payload). This increase is caused by the extra type information that needs to be saved. Although the full type description is only saved for the first object, the following objects still require a five-byte class handle to identify their class. Since the payload of the objects is only four bytes, the overhead of these handles is large. Serializing an array of these objects only introduces a small amount of extra type information. When we serialize a single object that contains 256 *int* fields, the amount of type information is only slightly reduced. Although this approach does not require a large number of type handles, it does increase the amount of type information needed for the first object, due to the many field descriptors required.

[6] Unless stated otherwise, each object has a single letter name and single letter field names

An interesting result is shown by the single linked list. Although this is a linked data structure, using references, hardly any extra bytes are required to store its structure. Because the list is serialized recursively, each reference field encountered refers to a previously unseen object that can be serialized immediately. Therefore no structure information (i.e., object handles) is required, making the single linked list more space efficient to serialize that an array of objects.

When a double linked list is serialized, however, extra structure information is needed. Each object contains a reference to a previous object that has already been serialized, thereby introducing a cycle in the graph. Each of these cycles (one for every object) requires a five-byte handle to be saved. As the table shows, the overhead of this structure information is considerable. Finally, if we use a balanced binary tree, no handles are needed. However, each of the leaf nodes in the tree contains two *null* references which also need to be saved. As a result, the tree requires more bytes to be stored than a single linked list or an object array.

Stream Protocol	
Stream Magic	4
Object	1 + *Type Description* + *Data*
Array	1 + *Type Description* + *Array Size* + *Data*
String	3 + *Text*

Structure Information	
Array Size	4
Object Reference (Handle)	5
Null Reference	1

Type Descriptions	
Object	16 + *Type Name* + *Field Descriptions*
Reference Array	19 + *Type Name*
Primitive Array	18
Type Reference (Handle)	5

Field Descriptions	
Reference Type	8 + *Field Name* + *Type as String*
Primitive Type	3 + *Field Name*

Table 2.2: Various sources of serialization overhead (in bytes)

The number of bytes required to store different types of information is shown in Table 2.2. Note that some of the entries in the table have a variable size. For example, the type description of an object requires 16 bytes, plus some additional space to store the name of the class (at least 1 byte per character), plus the space required to save a description for each of its fields. These field descriptions also have a variable size depending on the length of the field's name and type.

2.4.3 RMI runtime system

The RMI runtime system contains code related to the registry, communication, connection handling, and distributed garbage collection. Using Figure 2.11, we will now briefly describe these subjects.

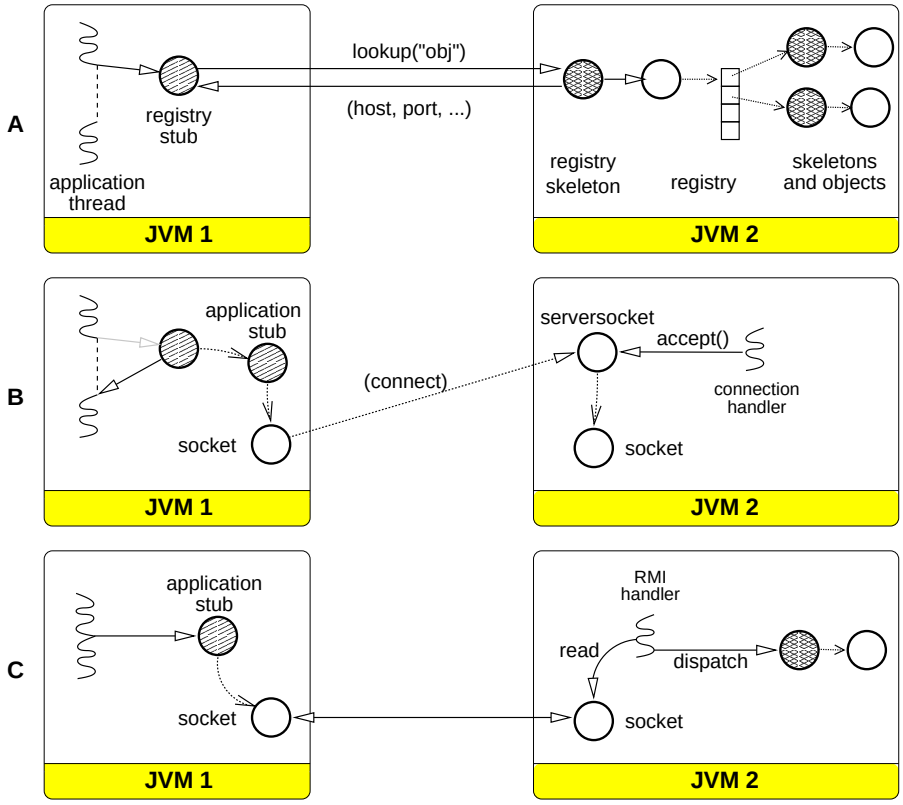

Figure 2.11: The registry and connection handling in RMI.

The RMI registry is used to keep track of the remote objects (and skeletons) exported by a JVM. It can be implemented using a regular remote object. Other JVMs can communicate with this registry object to look up any information they need to create remote references to the exported objects.

An example is shown in Figure 2.11(A), where JVM 2 exports two remote objects using a registry. An application thread on JVM 1 can use the *lookup* method of a local registry stub to find a remote object in the table of JVM 2. When the object is found, information returned on how it can be contacted and what type of stub is required (e.g.,

a port number and a class). When this information is received on JVM 1, it creates a stub of the appropriate type and sets up a communication channel between the stub and the skeleton. Figure 2.11(B) shows how this is implemented for the standard RMI implementation using TCP/IP sockets. The stub creates a new network *Socket* object and connects it to a *ServerSocket* on JVM 2, There, a *connection handler* thread is waiting for new connections. When the connect message from JVM 1 arrives, it is accepted by this connection handler thread, and a new socket communication channel between the JVMs is created. The connection handler thread is then promoted to be a *RMI thread.* It finds the skeleton object, and starts reading data from the socket connection (shown in Figure 2.11(C)). A new thread is created to serve as the connection handler.

A similar *thread replacement* scheme is used to handle incoming RMIs. Whenever an invocation is read from the socket, the RMI thread forwards it to the skeleton using the *dispatch* method. Before invoking *dispatch*, however, a new thread is created to replace the current RMI thread. This ensures that a blocking RMI will not deadlock the entire system.

The RMI runtime system also contains a *distributed garbage collector* (DGC). It is the responsibility of the DGC to prevent that remote objects are removed by the local garbage collector while they are still referenced by stubs on different JVMs. The DGC, like the registry, is implemented using a remote object. It contains a table of the remote objects located on the JVM. When a new stub is created for a remote object, the DGC on the server will be asked for a *lease*. This lease guarantees that the remote object will not be removed for a specified period of time. If a lease expires while the stub is still used, the DGC on the client machine must renew it to ensure that the stub remains usable. For this purpose, the client DGC contains a special thread which keeps track of the stubs, and periodically renews the leases by sending a message to the server JVM.

2.5 RMI performance

In this section, we will evaluate the communication performance of five different Java platforms that offer an RMI implementation. Four are based on a JIT compiler, and one uses the Manta platform. We will first describe our experimental setup and then use micro benchmarks to compare the performance of the different platforms. Using the RMI implementation based on the Manta compiler, we will then provide a more detailed analysis of the RMI performance and use this analysis to identify specific performance bottlenecks. Finally, we will use application kernels to evaluate the performance of RMI at an application level.

2.5.1 Experimental setup

The experiments described in this section are performed on the *DAS* cluster described in Section 1.2.1. For the comparison, we will use five different systems that offer RMI. The first four systems, Sun JDK 1.2, Sun JDK 1.4, IBM JDK 1.1.8 and IBM JDK 1.3.1 are based on a JIT compiler. The fifth system, that we will refer to as *Compiled Sun*, consists of a Sun JDK 1.1 RMI implementation compiled with the Manta compiler. The sequential speed of the code generated by the Manta compiler is roughly comparable to that of the IBM 1.1.8 JIT. Both perform much better than the Sun 1.2 JIT. Note that the Sun JDK 1.2, IBM JDK 1.1.8 and Manta where all developed around 1999. The Sun JDK 1.4 and IBM JDK 1.3.1 are more recent systems and provide a better performance for sequential code. The object serialization code offered by the Manta system is based on reflection, like the original implementation. To improve performance, however, a large part of the serialization in Manta is implemented in C. This allows Manta to exploit knowledge about the memory layout of objects. Manta can directly read and write the fields of an object, instead of using reflection calls. Manta does not require the use of (expensive) native calls to convert floats and doubles to bytes before writing them to the output stream. Instead, these values can directly be written into the stream buffer, converting them on-the-fly. Despite these optimizations, Manta uses the same serialization protocol as the other implementations.

All RMI implementations use socket-based communication. Three systems, Sun JDK 1.2, IBM JDK 1.1.8 and Manta also run over Myrinet. For this purpose, we use a socket interface on top of the Panda communication library that was explained in Section 1.2. We will refer to this socket interface as *PandaSockets*. PandaSockets is a re-implementation of Berkeley FastSockets [93] and supports zero-copy streams. Because a one-to-one mapping of socket messages and Panda messages can be made, the performance of FastSockets and Panda is close.

Efficiently interfacing the native PandaSockets library from Java is non-trivial. Most socket calls (e.g., *send*, *receive*, *accept*, *connect*) are blocking and suspend the calling *process* until the call has been completely serviced. However, since Java applications are multi-threaded, a blocking call should only suspend the calling *thread* and allow another runnable thread in the process to be scheduled. Otherwise deadlock may occur. To implement this behavior, all sockets are set to nonblocking mode. Threads must then use polling to determine when their socket call is ready. To prevent unnecessary polling or thread switches our implementation uses condition variables. A thread that would normally block enters the wait state instead. It is signaled when a poller thread notices that the socket is ready. The role of poller thread is taken by one of the blocked threads, because using a separate poller thread would always introduce thread switches. In this way, thread switches can often be prevented on the critical path of the RMI benchmarks.

2.5.2 Micro benchmarks

We have implemented several simple benchmark applications to evaluate the performance of the different Java platforms. These benchmarks typically consist of a single remote object that receives remote invocations from a different machine. To evaluate the impact of serialization on RMI performance, we have also created several serialization benchmarks. These measure the maximum (de-)serialization throughput that can be achieved on a single machine. Similarly, we have also created benchmarks that measure the performance of the Java Sockets implementation, which is used to implement RMI.

Serialization

Our first benchmarks measure the serialization performance on a single machine. The results are shown in Table 2.3. It shows the serialization (*write*) and deserialization (*read*) performance for the five different systems. The serialization performance is measured for three different primitive arrays, each containing 100 KBytes of data, and a balanced binary tree of 1023 objects, each containing 4 integer values (and two references).

	Sun JDK 1.2		*Sun JDK 1.4*		*IBM JDK 1.1.8*		*IBM JDK 1.3.1*		*Manta*	
benchmark	*write*	*read*	*write*	*read*	*write*	*read*	*write*	*read*	*write*	*read*
byte[]	75.5	20.6	120.5	26.3	108.0	19.8	115.5	33.2	∞	60.3
int[]	6.7	9.3	33.7	16.3	17.8	14.2	28.9	20.7	33.5	36.2
double[]	2.6	2.6	8.2	7.1	10.7	9.8	11.2	14.4	38.8	33.5
tree total	0.2	0.2	3.3	2.5	0.1	0.1	1.5	0.7	4.3	0.9
tree user	0.1	0.1	2.3	1.8	0.1	0.1	1.1	0.5	3.0	0.6

Table 2.3: Serialization throughput on the DAS (in Mbyte/s).

The serialization performance is measured using an *ObjectOutputStream* which writes the serialized data to a *NullOutputStream* (a stream which discards all data it receives). Using *writeObject*, an array or a tree is written into this stream 1000 times, resetting the stream after each write to circumvent the cycle detection. The total time required is measured, allowing us to calculate the average throughput. Each test is repeated 10 times (during a single run), to give the JIT compilers a chance to compile the bytecode. The table shows the best performance achieved during the 10 tests.

The deserialization benchmarks use a similar approach. The data is serialized once, using an *ObjectOutputStream*, and then buffered. Using an *ObjectInputStream*, the data is deserialized 1000 times (by repeatedly replaying the buffered data), allowing us to calculate the average deserialization throughput. The table shows the best performance over 10 tests.

To calculate the throughput for arrays, we only take the payload into account, since the overhead of protocol and type information is relatively low. However, as shown in Table 2.1, serializing a tree data structure requires a significant amount of extra

information to be saved on the stream. Therefore, Table 2.3 shows two numbers for trees, one for the total throughput, and one indicating the throughput of the payload (i.e., the 4 integers in each object).

As the table shows, the serialization performance varies for the different platforms. In general, deserialization is more expensive than serialization. This difference can be explained by the extra object allocation overhead during deserialization, and the cost of garbage collecting the deserialized objects. Deserialization also requires an extra copy. The bytes containing the serialized data are first copied from a buffer of the benchmark to a temporary buffer in the *ObjectInputStream*. Only then are they converted and stored in the array (or object).

The (de-)serialization of an *int* array is more expensive that the (de-) of a *byte* array. This is caused by the overhead to converting *int* values to bytes and vice-versa. The serialization of a *double* array is even more expensive. A *doubleToLongBits* method invocation is required to convert every *double* to a *long*, before it can be converted to bytes. The same applies for deserialization of a *double* array.

Note that serializing *byte* arrays is a special case. Since no conversion is required, the array can be passed directly to the underlying layer. Manta takes full advantage of this optimization, resulting in an "infinite" throughput (i.e., no serialization is required). The *ObjectOutputStream* implementations used in the JIT compilers are less efficient, resulting in a throughput of 108 and 116 MByte/s for the IBM JIT compilers and 75.5 and 121 MByte/s for the Sun JIT compilers.

This optimization cannot be applied when deserializing a *byte* array, since deserialization always requires the data to be copied into a newly allocated array or object. This reduces the throughput obtained when deserializing a *byte* array to 60.3 MByte/s for Manta, and 20 to 33 MByte/s for the JITs.

For trees, the serialization performance is significantly lower than for arrays. This shows that the cost of traversing the data structure and using the reflection mechanism to extract the data from the objects is expensive. The numbers for deserialization are even lower than the serialization numbers, due to the extra overhead of object allocation and garbage collection.[7]

Since RMI requires both serialization and deserialization of data, the maximum throughput that RMI can reach is the minimum value of the *write* and *read* columns. As the table shows, the maximum total serialization throughput reached for *byte* arrays is 60 MByte/s for Manta, and 20 to 33 MByte/s for the JIT compilers. For the other data types, the throughput is significantly lower.

When using Myrinet the maximum throughput offered by the Panda communication library is approximately 60 MByte/s. Therefore, this benchmark shows that the performance of serialization itself is already insufficient to fully utilize this throughput (except when *byte* arrays are used on Manta). As a result, an RMI implementation that uses this serialization and adds overhead of its own, will never be able to make full use of a high-performance network like Myrinet.

[7] The Sun JDK 1.4 has the highest tree deserialization performance due to its efficient garbage collector.

Null-RMI latency

Our next benchmark, the null-RMI latency test, measures the round-trip time of an RMI without any parameters or a return value. This value represents the minimal time required to do an RMI. Increasing the complexity of the RMI (e.g., by adding parameters or a return value) will result in a higher latency.

To interpret the RMI benchmark result, we also need to determine the maximum performance that is achieved by the communication library that is used to implement RMI. For that reason we also show latency numbers for Java benchmarks based on TCP communication, and for low-level (C) benchmarks using the kernel TCP/IP implementation on Fast Ethernet and PandaSockets on Myrinet.[8] For comparison, we also show the latency for a Panda RPC operation on Myrinet. These low-level benchmarks indicate the maximum performance which is available on each of the networks (using TCP), while the Java TCP benchmarks indicate the maximum performance available to the RMI implementations. By calculating the difference in latencies between the RMI and Java TCP benchmarks, we can estimate the overhead of the RMI implementation.

	Fast Ethernet		
System	*TCP*	*RMI*	*difference*
Sun JDK 1.2	215	1480	1265
Sun JDK 1.4	313	744	431
IBM JDK 1.1.8	250	720	470
IBM JDK 1.3.1	227	646	419
Compiled Sun	171	490	319
Kernel TCP/IP	171	-	-

	Myrinet		
System	*TCP*	*RMI*	*difference*
Sun JDK 1.2	67	1316	1249
IBM JDK 1.1.8	80	542	462
Compiled Sun	44	301	257
PandaSockets	40	-	-
Panda	31	-	-

Table 2.4: RMI Latency on Fast Ethernet and Myrinet, all numbers in μs

Table 2.4 shows that the lowest round-trip latency that we measured on the Fast Ethernet network (i.e., the Kernel TCP/IP result) is 171 μs. The null-RMI latency of Sun JDK 1.2, however, is almost nine times higher and takes 1480 μs per RMI. As the table shows, for the Sun JDK 1.2, the Java interface to TCP adds 44 μs, leaving 1265 μs of RMI overhead. The Sun JDK 1.4 effectively halves the RMI round trip time of its predecessor, 744 μs. In addition to its TCP round trip latency of 313 μs, the RMI implementation adds 431 μs. Both IBM JDK's are slightly faster.

[8] No data is available for the SUN JDK 1.4 and IBM JDK 1.3.1 running on Myrinet.

For the Compiled Sun system, the sequential speed of the code generated by the Manta compiler is comparable to that of the IBM 1.1.8 JIT. Manta, however, has a more efficient runtime system, with a user-space thread package and a much faster native interface. This results in a null-RMI latency of 490 μs. Unfortunately, this is still 2.9 times slower than the kernel TCP/IP latency.

The numbers in Table 2.4 show that on Fast Ethernet, 60 to 85% of the null-RMI benchmark latency is spent in the RMI implementations. This overhead becomes even greater when the Myrinet network is used. On Myrinet, 85 to 95% of the time is spend in the RMI implementations. Although the round trip latency of PandaSockets is only 40 μs and the latencies of the Java TCP benchmarks drop accordingly when they use it, the RMI overhead remains about the same. As a result, a null-RMI on Myrinet, using the Sun JDK, still takes 1316 μs, almost 33 times the latency offered by PandaSockets, and 42 times slower than a Panda round trip time. Although the IBM 1.1.8 JDK (542 μs) and Compiled Sun (301 μs) perform much better, they are still a factor of 17 and 10 slower than Panda.

RMI throughput

Our next benchmark measures the RMI throughput. We measure the round-trip time of an RMI with a 100 KByte array or a tree of 1023 objects as a parameter. We then calculate the number of bytes transferred per second. Note that only the size of the payload is considered in this calculation. Therefore, the throughput results in Table 2.5 represent the throughput that is available to the user. The actual throughput (as seen by the RMI runtime system) will be slightly higher.

	Fast Ethernet				
System	*TCP Socket*	*RMI byte[]*	*RMI int[]*	*RMI double[]*	*RMI Tree-1023*
Sun JDK 1.2	11.1	6.0	3.1	3.5	0.08
Sun JDK 1.4	9.7	5.3	4.9	3.4	0.6
IBM JDK 1.1.8	9.5	6.9	6.1	5.0	0.05
IBM JDK 1.3.1	10.3	5.9	5.4	5.2	0.4
Compiled Sun	11.2	9.1	7.2	7.2	0.5
Kernel TCP/IP	11.2	-	-	-	-
	Myrinet				
Sun JDK 1.2	58.5	7.3	3.8	3.7	0.08
IBM JDK 1.1.8	58.9	12.3	7.9	5.6	0.05
Compiled Sun	60.1	24.7	15.4	15.8	0.6
PandaSockets TCP	60.1	-	-	-	-
Panda RPC	60.5	-	-	-	-

Table 2.5: RMI Throughput on Fast Ethernet and Myrinet

Table 2.5 shows that the throughput achieved the kernel TCP/IP implementation on Fast Ethernet is 11.2 MByte/s. The Java TCP socket implementations of the Sun 1.2 JDK and Manta are able to fill this bandwidth. The performance of the other JDKs are slightly lower, 9.5 to 10.3 MByte/s. When we look at the RMI results, however, we see that none of the implementations are able to fully utilize the throughput offered by the network. As we have shown in Section 2.5.2, the serialization throughput is already limited, especially for the tree data structure. When the software overhead of RMI is added, the throughput is reduced even further.

On Myrinet, all three system used profit from the higher throughput offered by the network. The Compiled Sun system get the best results. Compared to the Fast Ethernet benchmarks the throughput is doubled. The only exception is the Tree-1023 benchmark, whose throughput is limited by the serialization performance, as shown in Table 2.3. However, even the highest throughput reached by Compiled Sun (24.7 MByte/s for an RMI with a *byte* array parameter) is significantly lower than the 60.1 MByte/s offered by PandaSockets.

Since the benchmarks indicate that the Compiled Sun system is the most efficient RMI implementation of the five, we use this system in the following section to represent RMI.

Breakdown

We will now present a breakdown of the time that Compiled Sun spends in remote method invocations. We use a benchmark that has zero to three empty objects (i.e., objects with no data fields) as parameters, while having no return value. The benchmarks are written in such a way that they do not trigger garbage collection. They are run using the Myrinet network. The results are shown in Table 2.6.

overhead	*Parameters*			
introduced by	*none*	*1 object*	*2 objects*	*3 objects*
Serialization	0	195	210	225
RMI	180	182	182	184
Communication	121	122	124	125
Method call	0	1	1	1
Total Latency	301	500	517	535

Table 2.6: Breakdown of Compiled Sun RMI on DAS using Myrinet (times in μs)

The measurements were done by inserting timing calls, using the Pentium Pro performance counters, which have a granularity of 5 nanoseconds. The serialization overhead includes the costs to serialize the arguments at the client side and deserialize them at the server side. The RMI overhead includes the time to initiate the RMI call at the client, handle the incoming call at the server, and process the reply (at the client). It excludes the time for (de)serialization and method invocation. The communication

overhead is the time from initiating the I/O transfer until receiving the reply, minus the time spent at the server side.

The simplest case is a null-RMI (an empty method without any parameters or return value), which takes 301 μs. Most of this time (180 μs) is spent in the RMI implementation, the rest (121 μs) is communication overhead. When object parameters are added to the call, the RMI and communication overhead stay almost the same, but a large amount of serialization overhead is added. The first object parameter adds 195 μs to the RMI. Any additional parameters add 15 μs per object. This difference is caused by the extra initialization of serialization related data structures, required when the first parameter is added.

2.5.3 Applications

The low-level benchmarks already indicate that the performance of RMI is not very good (especially the throughput for linked data structures). For parallel programming, however, a more relevant metric is the efficiency obtained with applications. To determine the impact of the RMI performance on application performance, we will use six parallel application kernels with different granularities. We briefly describe the application kernels and the input sizes used below, and then we discuss their performance using the Compiled Sun system. Each application program typically first creates the remote objects needed for interprocess communication and distributes the references to these objects among the machines. Therefore, the overhead of remote object creation, distributed garbage collection, and reference counting only occurs during initialization and has hardly any impact on application performance.

SOR Red/black Successive Overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid. The program distributes the grid row-wise among the machines. Each machine exchanges one row of the matrix with its neighbors at the beginning of each iteration. We used a 578×578 grid as input.

ASP The All-pairs Shortest Paths (ASP) program computes the shortest path between any two nodes of a given 1280-node graph. It uses a distance table that is distributed row-wise among the machines. At the beginning of each iteration, one machine needs to send a row of the matrix to all other machines. Since the Java RMI model lacks support for broadcasting, we expressed this communication pattern using a spanning tree. Each machine forwards the message along a binomial tree to two other machines, using RMIs and threads.

Radix is a histogram-based parallel sort program from the SPLASH-2 suite [111], ported to Java RMI. The program repeatedly performs a local sort phase (without communication) followed by a histogram merge phase which transfers histogram information. After this merge phase, the program moves some of the keys between machines, which also requires RMIs. The radix program performs a large number of RMIs. We used an array with 3,000,000 numbers as input.

FFT is a complex 1D Fast Fourier Transform program based on the SPLASH-2 code. The data is partitioned among the different machines. The communication pattern of FFT is a personalized all-to-all exchange, implemented using an RMI between every pair of machines. We used a data set with 2^{20} elements.

Water is another SPLASH application. This N-body simulation is parallelized by distributing the bodies (molecules) among the machines. Communication is primarily required to compute interactions with bodies assigned to remote machines. Our Java program uses message combining to obtain higher performance: each machine receives all bodies it needs from another machine using a single RMI. After each operation, updates are also sent using one RMI per destination machine. Since Water is an $O(N^2)$ algorithm and we optimized communication, the relative communication overhead is low. We used 1728 bodies for the measurements.

Barnes-Hut is an $O(N \log N)$ N-body simulation. Our Java program is based on the implementation of Blackston and Suel [12]. This code is optimized for distributed-memory architectures. Instead of finding out at runtime which bodies are needed to compute an interaction, as in the SPLASH-2 version of Barnes-Hut, this code precomputes where bodies are needed, and sends them in one collective communication phase before the actual computation starts. In this way, no stalls occur in the computation phase [12]. We used a problem with 30,000 bodies.

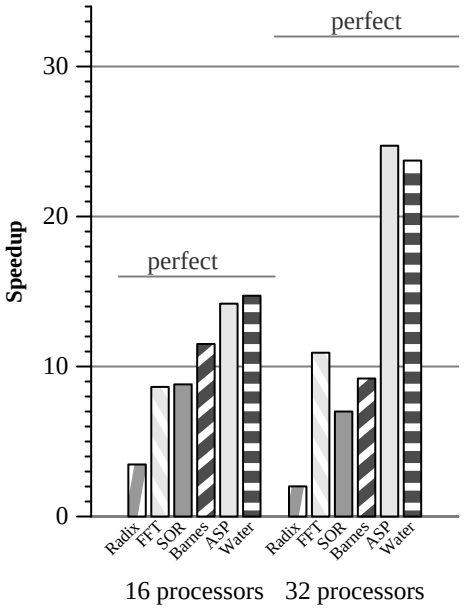

Figure 2.12: Speedup of applications on 16 and 32 machines using Compiled Sun.

Figure 2.12 shows the speedups for these six applications obtained by Compiled Sun on 16 and 32 machines. They are computed relative to the parallel program on a single CPU. As the figure shows, Compiled Sun performs well for only two applications, ASP and Water. FFT, SOR and Barnes have a reasonable speedup on 16 machines, but their performance hardly improves or even degrades on 32 machines. Radix does not achieve a substantial speedup.

To analyze the application performance further, we have done additional performance measurements. Table 2.7 shows the total number of messages sent (summed over all CPUs) and the amount of data transferred, using 16 or 32 CPUs. These numbers were measured at the Panda layer, so they include header data sent by the PandaSockets library and the RMI runtime system. Also, an RMI generates two Panda messages, a request and a reply.

	16 CPUs			32 CPUs		
Program	Time (s.)	# Messages	Data (MByte)	Time (s.)	# Messages	Data (MByte)
ASP	27.56	154248	100.23	15.83	319870	207.17
SOR	19.38	134765	84.62	24.39	285409	175.11
Radix	2.06	78674	64.87	3.56	183954	130.35
FFT	6.07	173962	157.37	4.80	204949	163.45
Water	26.78	16088	9.59	16.61	44984	20.05
Barnes-Hut	16.74	45439	25.20	20.91	171748	57.58

Table 2.7: Performance Data for Compiled Sun on 16 and 32 CPUs

As we can see, Water transfers the smallest amount of data of the six applications. On 32 CPUs, it only sends 2708 (44984/16.61) messages per second. These messages transfer 1.20 (20.05/16.61) MByte/s between the machines. This low communication overhead explains the reasonable performance of Water.

Although ASP communicates significantly more than Water (on average it sends 20207 messages and 13.1 MByte/s), it is based on asynchronous communication. The machines do not run *in lockstep*, but can perform their computations fairly independently of each other. As a result, ASP is not sensitive to a high latency. Also, the throughput required for ASP is not high enough to pose a problem for the Compiled Sun system. As a result, ASP obtains reasonable speedups of 14 and 25 on 16 and 32 machines, respectively.

The other four applications do run in lock step. After the machines have performed part of the computation, a communication phase is required to exchange the data that is needed for the next part of the computation. As a result, the time required to do the communication is on the *critical path* of the application. Any increase in latency or decrease in throughput is immediately reflected in the application performance. In Radix and FFT, a large amount of data is exchanged between machines. Their

performance is mostly limited by the throughput of Compiled Sun. SOR and Barnes-Hut communicate less data and are more sensitive to the latency.

2.6 Conclusion

In this chapter we have given a description of the RMI model, and have described a Sun JDK 1.1 based implementation. Due to its support for polymorphism, dynamic bytecode loading, and linked data structures, RMI is highly flexible and provides an expressive model. We believe that this makes RMI an attractive starting point for high-performance parallel programming in Java.

Using several micro benchmarks, we have evaluated the performance of the RMI and serialization implementations offered by five different Java platforms. Unfortunately, these benchmarks show that the (de-)serialization throughput is already limited. For arrays of integers and doubles, even a C based serialization implementation (as provided by Manta) is not efficient enough to use more than half the bandwidth offered by the Panda communication library running on Myrinet. For more linked data structures, such as binary trees, the performance is even worse, only reaching 1% of the bandwidth offered by Panda. The benchmarks also show that the RMI round-trip latency of the five Java platforms is significantly higher than their TCP round-trip latency, indicating that the RMI implementations are not very efficient. On Fast Ethernet, 60 to 85% of the round trip time is spend in the RMI implementation. On Myrinet, this is 85 to 95%.

Using six parallel application kernels, we have further evaluated the performance of one of the five RMI implementations, Compiled Sun. Although this implementation showed the highest performance in the micro benchmarks, the application level performance leaves much to be desired. Only two out of six applications have an acceptable speedup on 32 machines.

In conclusion, although RMI offers an attractive model for parallel programming in Java, it is hard to obtain an acceptable performance due to the complexities of its implementation. We have illustrated this by using three different Java platforms. A similar conclusion was reached in a recent Ph.D. thesis by Breg [15], that studies the performance of three parallel applications on five different RMI implementations. All implementations obtain very poor (or no) speedups on all applications due to the overhead of the RMI and serialization implementations.

In the next chapter we will discuss an alternative RMI implementation that was specifically designed for parallel programming, and compare its performance to the Compiled Sun implementation used in this chapter.

Chapter 3

Manta RMI

3.1 Introduction

In the previous chapter we have given a description of the RMI model, discussed an example implementation (based on the Sun JDK 1.1), and the performance of the RMI and serialization implementations offered by five different Java platforms. We have shown that for parallel programming, these RMI and serialization implementations have a large overhead, resulting in poor application speedups.

In this chapter we describe an alternative RMI implementation, called *Manta RMI*. Manta RMI was designed specifically for high-performance parallel programming, and reduces the overhead of both RMI and serialization. We will show that Manta RMI obtains a high performance (close to that of RPC systems), without sacrificing RMI's advanced features (e.g., polymorphism and flexibility).

Manta RMI has been designed to be *source level* compatible with the RMI standard (i.e., it uses the same syntax). Therefore, Manta RMI can be used to run any parallel application designed to use RMI. However, Manta RMI does *not* adhere to the RMI standard as described in [98]. This approach allows Manta RMI to use a different RMI protocol and an alternative, high-performance serialization implementation which is based on compile-time code generation instead of run-time reflection.

We have implemented Manta RMI in the Manta high-performance Java system by extending the compiler to generate RMI and serialization code, and by extending the runtime system with the necessary communication support. Instead of being implemented on top of TCP/IP, Manta RMI uses the Panda communication library that was described in Section 1.2.

Despite lacking conformance to the RMI standard, Manta RMI is able to interoperate with standard RMI implementations. A program running Manta RMI can communicate with a program running a standard RMI implementation, and the two programs can even exchange bytecodes (e.g., for polymorphism). To implement this

interoperability with other JVMs, the Manta system contains a standard RMI implementation and the standard serialization protocols in addition to its own protocols. To support dynamic class loading, the Manta runtime system contains a simple compiler that is able to compile bytecode and generate the necessary RMI and serialization code during run time.

The remainder of this chapter is structured as follows. The design and implementation of Manta RMI are discussed in Sections 3.2 and 3.3. In Sections 3.4 and 3.5, we give a detailed analysis of the communication performance of our system. For this analysis we will use the micro benchmarks and applications from Section 2.5, and compare the performance of Manta RMI to that of the *Compiled Sun* implementation used in that section. In Section 3.6 we look at related work. Section 3.7 presents conclusions.

3.2 Design of the Manta RMI system

The Manta RMI system is designed for distributed memory parallel processing on a *cluster computer*. A cluster typically consists of a large number of identical machines, interconnected using some high-performance network (e.g., Myrinet). Figure 3.1 shows an example of a Manta application running on a cluster of four machines.

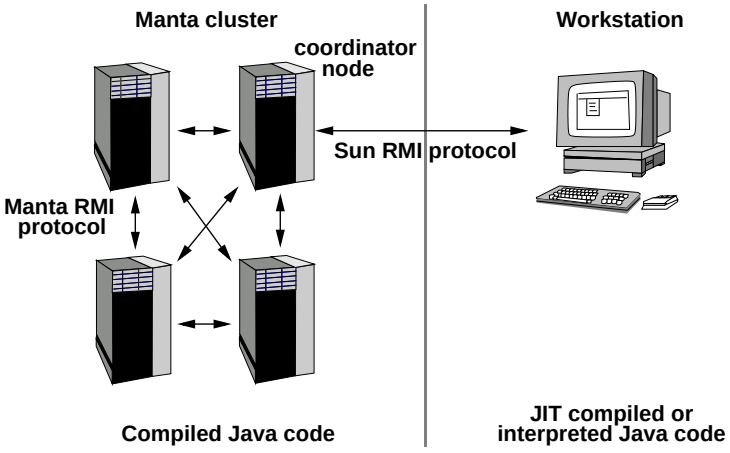

Figure 3.1: Example of a typical Manta application.

To use a cluster efficiently, it is important that the communication between the machines introduces as little overhead as possible. Therefore, the RMI implementation must be able to benefit from a high-performance network and introduce little software

overhead. Unfortunately, an RMI implementation as described in the previous chapter is not particularly suited for this. It is implemented on top of TCP/IP, which was not designed for parallel processing and does not allow a high-performance network like Myrinet to be used efficiently. We also showed that the software overhead introduced by serialization and the RMI runtime system is significant. Some of this overhead can be attributed to the fact that RMI is designed for distributed client-server applications. These applications run in a distributed and *heterogeneous* environment, where many different types of machines and operating systems are used. Such an environment requires the use of standardized network protocols like TCP/IP, and serialization protocols that make very few assumptions about the receiver. Because network latencies in the order of several milliseconds are typical in such an environment, obtaining a low communication overhead is less important.

Our Manta RMI system, however, was designed specifically to run parallel applications on a homogeneous cluster. In our design we assume that a parallel application consists of several processes that each run the same (Manta) program and communicate using RMI. We also assume that all performance-critical communication will take place between the machines of the cluster (i.e., communication with the 'outside world' does not require high-performance). As a result of these assumptions, it is not necessary for RMI communication within the cluster to use TCP/IP nor the standard RMI and serialization protocols. We can replace them by custom protocols that introduce less software overhead, and use advanced features offered by the communication network (e.g., zero-copy communication provided by Panda).

Like the standard RMI implementation, Manta RMI uses stubs generated by the compiler. Skeletons are not generated. Instead, the functionality of a skeleton is added directly to the remote objects itself. We will describe the generated Manta RMI code in more detail in Section 3.3.1. Manta RMI uses a high-performance serialization implementation that is based on compile-time code generation instead of runtime reflection. This approach reduces the serialization overhead significantly (see Section 3.3.2).

Although Manta RMI was primarily designed to run on a *homogeneous* cluster (where all machines are identical), it also has some support for *heterogeneous* clusters. Manta currently supports clusters consisting of a mix of little-endian and big-endian machines. This heterogeneity is handled by optimistically sending data in the native byte order of the sender, and, if necessary, having the receiver do the conversion. Differences in object layout are handled in a similar fashion. Heterogeneity is discussed further in Section 3.3.2.

For *interoperability*, Manta also supports communication with other JVMs using a slower and compatible RMI implementation (i.e., the *Compiled Sun* implementation). This allows a parallel Manta application running on a cluster to communicate with an 'external' JVM that can be used, for example, for visualization or steering purposes. Figure 3.1 shows an example where a graphical user interface (GUI) application is used to visualize the output of a parallel application running on the cluster. The parallel application itself is compiled with Manta and internally uses the Manta

RMI protocol for communication. The visualization software runs on a workstation and may use any JVM. The two applications can communicate using the standard RMI protocol.

Because two separate applications are used in this example, some exchange of bytecodes may be required. For example, the parallel Manta application may forward data to the GUI using some shared remote interface. The compiled Manta executable running on the cluster then needs a stub (implementing this interface) that refers to a remote object on the workstation. If this type of stub is not yet present in the compiled Manta executable, its bytecodes must be retrieved, compiled, and linked into the running Manta application. For this purpose, the Manta system contains a simple bytecode compiler.

Thus, the Manta RMI system logically consists of two parts: a *fast* RMI implementation that is used only for communication between Manta processes, and a *compatible* RMI implementation to interoperate with other, standard, JVMs. The structure of the Manta system is illustrated in Figure 3.2.

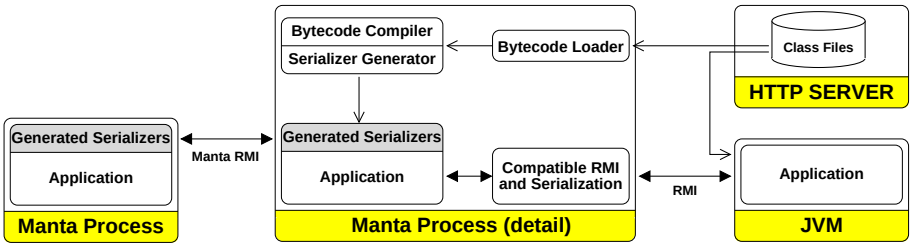

Figure 3.2: Structure of Manta.

The box in the middle describes the structure of a Manta process communicating with another Manta process (on the left), and with a non-Manta JVM (on the right). For communication with the other Manta process, the generated RMI and (de)serialization routines are used. Both are generated by Manta's native compiler. For communication with a JVM, Manta uses the compatible serialization and RMI implementations described in the previous chapter. The exchange of bytecodes between Manta and a JVM requires a shared file system or a HTTP server that contains the bytecodes for the classes of the application (shown as a separate process). The Manta process can then download bytecodes on demand, compile them, and link them into the running application. Manta's bytecode compiler also generates the necessary serialization codes.

3.3 Manta RMI implementation

A Manta RMI application can be divided into several layers, as shown in Figure 3.3. Although the layers shown in this figure are similar to that of a standard RMI application (shown in Figure 2.4), there are major differences.

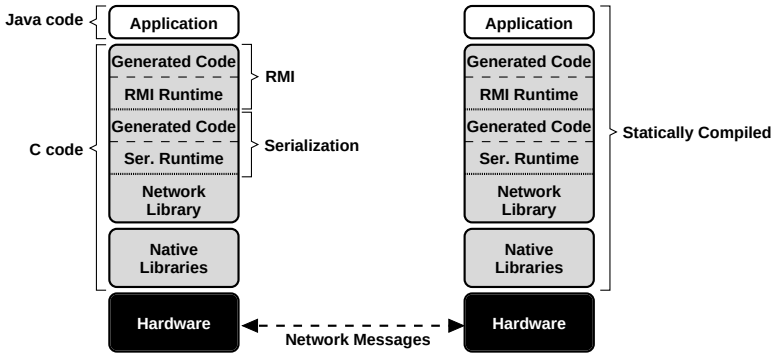

Figure 3.3: Layers of a Manta RMI application

All Manta application code is compiled statically (ahead-of-time) instead of dynamically (just-in-time). Java code is directly translated to native instructions that can be executed by the hardware. There is no need for a bytecode interpreter or JIT compiler, so there is no Java Virtual Machine layer present in Figure 3.3. Although there is a runtime bytecode compiler included in the Manta runtime system, it is only used infrequently. Most applications will be completely statically compiled.

Another difference is that all but one of the layers in a Manta application consist of native code. In Manta, the RMI implementation, serialization and network libraries are all implemented in C. Even the RMI code generated by the compiler is in C. This approach circumvents the limitations of the Java language when interfacing with the communication libraries (e.g., the lack of pointers), and improves the efficiency of the implementation.

The network library shown in Figure 3.3 consists of two parts. It contains a TCP/IP interface, which allows communication with an external JVM. For high-performance communication, Manta uses the Panda communication library instead.

To receive messages, the Manta RMI runtime system registers an *upcall handler* with the Panda library. Whenever Panda receives a message, it invokes this handler to forward the message to the Manta RMI runtime system. Depending on the complexity of the RMI call contained in the message, the Manta RMI runtime system can either create a new thread to handle the call, or handle it directly. This will be explained in more detail in the next section.

The final difference between the two figures is the serialization layer. In the origi-

nal implementation of Figure 2.4, serialization consists of a Java library that serializes objects using runtime reflection. In Manta, serialization is based on C code generated at compile time. This generated serialization code does not require runtime reflection, and can use zero-copy communication as offered by the Panda network library. Below, we will explain the Manta RMI and serialization implementation in detail.

3.3.1 Generated Manta RMI code

Like the RMI implementation described in the previous chapter, part of the Manta RMI implementation is based on generated code. When the example application shown in Figure 2.3 is compiled using the Manta compiler, a stub object is generated that looks similar to the one described in Section 2.4.1. However, Manta RMI stub objects are generated in C-code instead of Java. This C-based approach simplifies the interfacing of the generated code with the low-level communication library.

Stubs

Figure 3.4 shows pseudocode for the *print* function of the generated stub. When the Manta compiler generates RMI stubs, it applies several optimizations which we will describe while explaining this example. We will also compare the example to the generated stub shown in Figure 2.6.

```
marshal__print(class__PrinterStub *this, class__String *t) {
   MarshalStruct *m = allocMarshalStruct();
   ObjectTable *o = createObjectTable();
   writeHeader(m->outBuffer, this, CALL, MAY_BLOCK);
   writeObject(m->outBuffer, t, o);
   flushMessage(m->outBuffer);  /* Write data to the network. */

   fillMessage(m->inBuffer);    /* Receive reply. */
   int opcode = readInt(m->inBuffer);
   if (opcode == EXCEPTION) {
      class__Exception *ex = readObject(m->inBuffer, o);
      freeMarshalStruct(m);
      freeObjectTable(o);
      THROW_EXCEPTION(ex);
   } else {
      freeMarshalStruct(m);
      freeObjectTable(o);
   }
}
```

Figure 3.4: The generated marshal function for the *print* method (pseudocode).

The function starts by creating a *MarshalStruct*, a data structure that serves the same purpose as the *RemoteCall* object shown in Figure 2.6. It is used to store information during the lifetime of the RMI. To prevent a new *MarshalStruct* from being allocated for every RMI, the Manta RMI runtime system maintains a pool of these structures, allowing them to be reused.

Since the *print* method has an object parameter (the *String*), an *ObjectTable* is created that is used to detect cycles and duplicates during serialization. Whenever object serialization is invoked (the *writeObject* call), this table is passed as a parameter. As an optimization, the table will not be created when the RMI has no object parameters or object return value. This saves the cost of creating and initializing the table (although *ObjectTables* may be cached, they must alway be initialized before use).

Before the parameters are serialized, the header of the RMI message is created. Besides the usual information (such as opcodes, target and return address, etc.), the header also contains a special flag that indicates if the invoked method may block. This can occur when the remote method invokes *wait*, creates objects (thereby possibly triggering the garbage collector) or recursively invokes some other method that may block. Blocking RMI calls must always be handled in a separate thread to prevent the entire RMI runtime system from blocking. Therefore, the Manta compiler analyzes the remote methods during stub generation, and, for each method, makes a conservative estimation whether it is possible that the method may block. If so, the *MAY_BLOCK* flag is set in the header of the RMI message. This flag indicates to the Manta RMI runtime system receiving the message that the remote method must be run in a separate thread. If the *MAY_BLOCK* flag is not set, however, the method can be executed directly, thereby saving the cost of creating and switching to a new thread.

When the header is written and all parameters are serialized, the *flushMessage* function writes the message out to the network buffer. The *fillMessage* call then initiates reading the reply. After the result of the remote invocation has been deserialized, the *MarshalStruct* and *ObjectTable* are freed (i.e., returned to their caches). If necessary, a result or exception is returned.

Skeletons

On the receiving machine, the RMI is normally handled by skeleton objects. Skeleton objects do not introduce any extra functionality, they only serve as containers for the generated functions. However, they do use memory and produce garbage collection overhead. Therefore, they are not generated by the Manta compiler. Instead, the dispatch and unmarshal functions that handle the incoming RMI messages are added to the remote object itself. Like the functions of the stub objects, all 'skeleton' functions are generated in C. Figure 3.5 shows the pseudocode for the generated unmarshal function for the *print* method.

When a message is received by the runtime system, it unpacks the header and locates the target remote object (instead of the target skeleton). It can then invoke the appropriate unmarshal function to handle the remote invocation. In the example,

```
unmarshal__print(class__Printer *this, MarshalStruct *m) {
   class_Exception *exception = NULL;
   ObjectTable *o = createObjectTable();
   class__String *t = readObject(m->inBuffer, o);

   CALL_JAVA_FUNCTION(print, this, t, &exception);

   if (exception) {
      writeInt(m->outBuffer, EXCEPTION);
      writeObject(m->outBuffer, exception, o);
   } else {
      writeInt(m->outBuffer, RESULT_CALL);
   }

   /* Reply message is created, now write it to the network. */
   flushMessage(m->outBuffer);
   freeObjectTable(o);
}
```

Figure 3.5: The generated unmarshal function for the *print* method (pseudocode).

the *MAY_BLOCK* flag was set in the header, to indicate that the RMI method may block. Therefore, the unmarshal function (which invokes the RMI method) will run in a separate thread. If the flag was not set, the unmarshal function would be invoked directly by the runtime system, saving a thread switch.

To prevent the creation of a new thread for each blocking call, the Manta RMI runtime system maintains a *thread pool*. Threads can be extracted from this pool to handle potentially blocking RMIs. A new thread is only created when the thread pool is empty. When the RMI returns, the thread is returned to the pool. This approach saves the cost of creating a new thread, but, unfortunately, a thread switch is still required on the RMI critical path.

The unmarshal function creates an *ObjectTable* that is used to deserialize the *String* parameter from the message. Like in the marshaling function, this *ObjectTable* only needs to be created when the remote method has object parameters or an object return value. After deserializing the parameter, the Java method is invoked. The result value (possibly an exception or *void*) is returned to the caller using a network message.

3.3.2 Serialization

As we have shown in Chapter 2, the serialization of object method arguments is an important source of overhead in RMI implementations. The implementation of serialization is commonly written in Java and uses reflection to determine the state of each object during runtime. Also, the primitive fields of an object are always converted into

bytes and written to the network in network byte order (which may require an extra conversion step). Although Manta offers a more efficient implementation written in C, the throughput it obtained was still too low to effectively use the bandwidth offered by Panda.

To solve this problem, we have created a new serialization implementation, designed to minimize the runtime overhead. To avoid the overhead of reflection, our serialization implementation is based on compiler-generated serialization code. Only for classes that are not locally available, the serialization code is generated at runtime by the bytecode compiler. Even then, the overhead of this runtime serialization code generation is incurred only once, the first time a new class is used as an argument to some method invocation. For subsequent uses, the efficient generated serialization is available for reuse.

The overhead of copying the object content is avoided in Manta by using the scatter/gather interface of the Panda library. Also, by optimistically sending the data in host byte order instead of network byte order, unnecessary conversion can be avoided. We will now explain the Manta serialization implementation in more detail.

Compiler generated serialization functions

For every class that implements the *java.io.Serializable* interface, the Manta compiler generates two extra C-functions, the *serializer* and the *deserializer*. These functions are specialized to (de-)serialize the contents of objects of that single class. Since the compiler has complete information about the memory layout of the object, the generated serialization functions do not need any runtime reflection. Instead, they can directly read or write the object's fields.

To further speed up the (de-)serialization process, the compiler inserts pointers to the generated serialization functions into the method table of the class. To serialize a particular object, the serialization function pointer can be extracted from the object's method table, and invoked. On deserialization, the same procedure is applied. Using this approach, no runtime reflection is necessary to discover the contents of an object. Instead, only a single virtual function call is required. We will now show an example of the serialization code generated by the Manta compiler.

Figure 3.6 shows a *List* class and pseudocode for its generated serialization functions. The *List* class contains a primitive *int* field, a *Serializable* reference and a *List* reference pointing to the next node of the list. The compiler generates two functions, *serializeList* and *deserializeList*. Since their implementation is very similar we will only explain the serialization function in detail.

The *serializeList* function starts by writing the fields in the object to the network message. It does so by invoking the *write_fields* function which is implemented in Manta's runtime system. Besides the *MarshalStruct*, this function requires two parameters, a pointer that refers to the start of the data fields in the object (i.e., skipping the object headers), and an integer specifying the total *size* of the data fields. Note that all of the fields in the object are written, including any reference fields. These refer-

```
class List implements java.io.Serializable {
   int number;
   Serializable data;
   List next;
}

void serializeList(class__List *l, MarshalStruct *m, ObjectTable *o)
{
   write_fields(m, &(l->field_number), 12);
   if (l->field_data != NULL) write_object(l->field_data, m, o);
   if (l->field_next != NULL) write_object(l->field_next, m, o);
}

void deserializeList(class__List *l, MarshalStruct *m, ObjectTable *o)
{
   read_fields(m, &(l->field_number), 12);
   if (l->field_data != NULL) l->field_data = read_object(m, o);
   if (l->field_next != NULL) l->field_next = read_object(m, o);
}
```

Figure 3.6: The *List* class and its generated serialization functions (pseudocode).

ence fields may contain pointers to other objects. The *deserializeList* function uses the values of these fields (instead of special opcodes) to determine if another object must be read.

The *write_fields* function does not actually copy any data into a network message. Instead it stores the data's location and size in the *MarshalStruct*. The *write_fields* function illustrates the advantage of generating serialization functions in C. Unlike Java, C allows us to store a pointer that directly refers to the data fields in the object.

After the fields of the object have been written, the *write_object* function is invoked for each reference in the *List* object that is not *NULL*. Since it is not known at compile time what type of object the *data* field refers to, *write_object* determines how the *data* field should be handled. For example, if the field refers to a serializable object, the object's serializer will be invoked. If the field refers to a primitive array, the location of the array data is stored in the *MarshalStruct*. The *write_object* function also contains the necessary cycle-detection code.

Reducing the number of copies

In Manta serialization we prevent unnecessary copying of data by using the scatter/gather interface offered by the Panda communication library. Figures 3.7 and 3.8 show an example where a *double* array and an object containing two *int* fields are serialized and sent over the network. Figure 3.7 uses standard serialization, while Figure 3.8 uses Manta serialization.

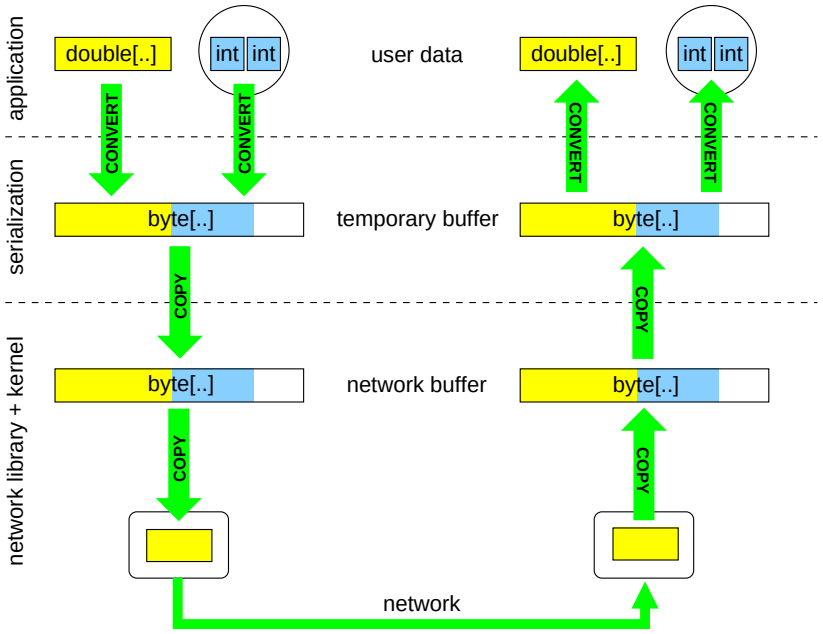

Figure 3.7: Copying behavior of communication using standard serialization.

The serialization standard (described in [97]), forces that all data in the array and object is converted into bytes. The conversion is usually done by using a byte array as a temporary buffer, as shown in Figure 3.7. Each *double* in the array and each *int* in the object is converted to bytes and stored in the buffer in network byte order. This temporary buffer is then passed to the network layer, which may copy the data again into one or more network messages. These messages are sent to another JVM, which, upon receiving them, copies the data into a byte array. During deserialization, these bytes will be converted back into *doubles* and *ints* which are stored in the received array and object.

Manta serialization, shown in Figure 3.8, uses a more efficient solution, based on Panda's scatter/gather interface. During the serialization process, no data is copied. Instead, a data structure called an *I/O vector* is created. This data structure stores pointers to (and the size of) all data that must be sent. When the construction of the I/O vector is complete, it is passed to the Panda communication library. Panda uses the pointers in the I/O vector to directly copy (or *gather*) the data from the array and object into network messages. When an advanced network architecture (like Myrinet) is used, these network messages may even be located on the network hardware itself. When the data is received on another machine, it is copied (or *scattered*) directly from

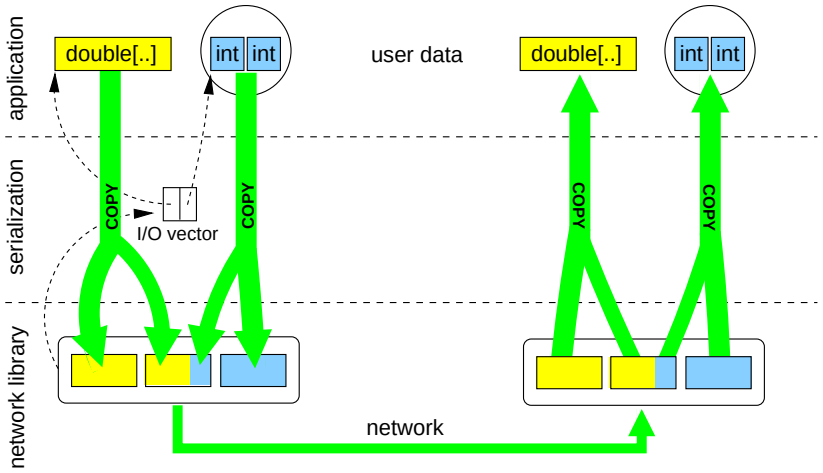

Figure 3.8: Copying behavior of communication using Manta serialization.

the network message (on the network card) into the destination array and object.

The approach used in Manta significantly reduces the number of copies required to send data. Where the standard serialization protocol requires at least five copies (possibly six), the Manta serialization protocol on Myrinet only copies the data twice, once into the sent network messages and once out of the received network messages. Since there are no extra copies made in host memory, such an approach is referred to as *zero-copy communication*.

Supporting heterogeneous systems

Manta serialization assumes that the same application is running on all participating machines of a homogeneous cluster. Data is copied directly from and to Java objects, without taking byte ordering or object layout into account. Although Manta was primarily designed to run on homogeneous clusters, it does have some support for heterogeneous clusters. Unlike normal serialization, which always converts the data to network byte order, Manta serialization optimistically sends data in the native byte order of the sender. If necessary, the receiver then does the conversion.

Figure 3.9 shows pseudocode for the heterogeneous deserialization function of the *List* class. After the object fields have been read from the network message, a special function *byteSwapList* (generated by the Manta compiler) is invoked which swaps the byte order of the fields. Note that this function is only invoked if the byte order of the sender is actually different. A similar approach is used to correct differences in the memory layout of the objects.

```
void byteSwapList(class__List *l) {
   SWAP4(1, &l->field_number);
}

void deserializeList(class__List *l, MarshalStruct *m, ObjectTable *o)
{
   read_fields(m, &(l->field_number), 12);
   if (m->swap) byteSwapList(l);
   if (l->field_data != NULL) l->field_data = read_object(m, o);
   if (l->field_next != NULL) l->field_next = read_object(m, o);
}
```

Figure 3.9: A heterogeneous deserialization function for *List* (pseudocode).

Overhead of type and structure information

Manta serialization is designed to reduce the processing overhead of serialization. However, it does not necessarily reduce the amount of data that is sent, as we will show below. Table 3.1 shows the number of bytes required to send 256 int values with Manta RMI using various approaches.

	Bytes	*Extra Information*			
Serialized Data	*Required*	*protocol*	*type*	*structure*	*total*
256 separate ints	1024	-	-	-	-
int[256]	1054	-	26	4	30
256 objects (1 field)	2071	-	1047	-	1047
object[256]	2101	-	1073	4	1077
object (256 fields)	1051	-	27	-	27
single linked list(256)	3095	-	1047	-	1047
double linked list(256)	5139	-	1047	1020	2067
binary tree(256)	4119	-	1047	-	1047

Table 3.1: Extra information required by different data structures (in bytes).

When we compare this table to Table 2.1 (which shows the values for the standard serialization implementation), we see that for the first five entries, Manta serialization requires fewer bytes. For the last three entries, however, Manta serialization uses more bytes. Because Manta also forwards the reference fields in the objects, four extra bytes are required for every *single linked list* object, while the *double linked list* and *binary tree* objects require eight extra bytes each.

As Table 3.1 shows, Manta serialization does not store any protocol information. This is not necessary, because Manta serialization is only used in combination with Manta RMI. The machine receiving an RMI knows what sort of the parameters must be deserialized (i.e., objects or primitive values). Therefore, storing extra protocol

information in the stream (e.g., to indicate the start of an object or a double value) is useless. This does make Manta serialization less flexible, however, as we will explain below.

Stream Protocol	
Object	*Type Description + Data*
Array	*Type Description + Array Size + Data*

Structure Information	
Array Size	4
Object Reference (Handle)	4
Null Reference	4

Type Descriptions	
Object or array	24 + *Type Name*
Type Reference (Handle)	4

Table 3.2: Various sources of serialization overhead (in bytes)

Table 3.2 shows the number of bytes required by Manta serialization to store different types of serialization information. Compared to Table 2.2, a number of entries are missing. Manta serialization treats *String*s as normal objects and uses the same type description for objects and arrays. It does not use any field descriptions or version number, since we assume that all machines use the same object implementation and layout.

Drawbacks of Manta's serialization

Manta serialization also has a number of drawbacks. It is less flexible than standard serialization because data is copied directly from a Java object into a network message, without first converting it into an independent format, or describing the fields of the objects. Therefore, versioning (as described in Section 2.4.2) is not supported. The sender and receiver of the object should agree in advance about the content and layout of the data in the object. Although this is hard to achieve in a distributed system, it is not a problem on a homogeneous cluster running a single parallel application.

Another effect of direct copying is that Manta serialization can not be used for anything other than network communication. The serialization standard was designed to allow reading from, or writing to, any stream of bytes. It can therefore be used for other applications, such as storing objects in a file. In contrast, Manta serialization can only be used for RMI and is designed to use the Panda communication library. However, Manta also provides a standard serialization implementation, which can be used for all other applications.

Manta serialization does not support the *writeObject* and *readObject* methods (see Section 2.4.2). As a result, custom serialization of objects is not possible. Manta does

support the *transient* keyword, however, which gives the programmer some control over the serialization process.

As we have shown above, Manta serialization does not necessarily transfer less data than standard serialization. Although this is not a major problem for a high-throughput networks like Myrinet, it can be a disadvantage for less efficient networks. However, this only occurs when a data structure consisting of a large number of objects is serialized. As we have shown in Table 2.3, the runtime overhead of serializing such a data structure is high, resulting in a low throughput. By reducing this runtime overhead, Manta serialization may still be able to increase the (user) throughput, even if it sends more data than the standard serialization.

3.3.3 The Manta RMI runtime system

The Manta RMI runtime system is similar in functionality to the runtime system described in Section 2.4.3. It offers a registry, connection handling and a distributed garbage collector. However, like the rest of Manta, the Manta RMI runtime system is implemented mostly in C. It was designed to minimize the RMI dispatch overhead, caused by copying, buffer management, thread switching, and indirect method calls. Most of the optimizations were already described in the previous section. By using the upcall mechanism offered by Panda, the Manta RMI runtime does not require a separate thread to handle incoming RMIs. A thread will only be created when an RMI potentially blocks. These threads, like many other data structures, are cached to prevent allocation overhead on the critical path.

Manta RMI and serialization optimize the sending of type descriptors for the parameters of an RMI call. When a serialized object is sent over the network, a description of its type is also sent to ensure that the receiver can correctly reconstruct the object. When several objects of the same type are sent, only the first needs a complete type description. For all subsequent objects the serialization protocol will refer back to this description. However, the standard RMI runtime completely resets the serialization streams after every call, to clear the cycle check information stored in the stream. Unfortunately, the type information is also reset. Therefore, the type descriptors can only be reused within a single RMI.

The Manta RMI runtime does not reset the serialization completely, but only clears the cycle check information. Therefore, subsequent RMIs in Manta can reuse existing type information. The first time Manta sends a class to a different machine, a type descriptor is sent along. The class is then given a type-id that is specific to the receiver. When more objects of this type are sent to the same destination machine, the type-id is reused. This type-id scheme works on a machine-to-machine basis, not just on separate connections. As a result, Manta even allows the type information to be shared between unrelated RMIs, not just between RMIs to the same object (i.e., using the same stub).

When the destination machine receives a type descriptor, it checks if it already

knows this class. If not, it tries to load the class' bytecode from the local disk or a HTTP server, and compiles it using the dynamic bytecode compiler.

Escape analysis

The Manta RMI runtime system and compiler also implement *escape analysis*. When objects are received as a parameter to an RMI, it is possible that their lifetime is limited to the RMI's execution. For example, the RMI may receive an array parameter, the data of which is immediately copied to a different array. This parameter array does not *escape* the RMI. It becomes garbage as soon as the RMI is completed. To prevent the garbage collection overhead of such objects, the Manta compiler determines whether the parameter object may escape (i.e., continue to be used after the RMI is finished). If not, the Manta runtime system returns such objects to the heap immediately after the RMI is finished.

3.4 Manta RMI performance

In this section, we compare the communication performance of Manta RMI against the *Compiled Sun* implementation described in Section 2.5. The experiments are run on the *DAS* cluster that is described in Section 1.2.1.

3.4.1 Micro benchmarks

We will now use the micro benchmarks that were introduced in the previous chapter to compare the serialization performance, latency and throughput obtained by Manta RMI and Compiled Sun.

Serialization

In the previous chapter we showed that the throughput of serialization was already lower than the throughput offered by the Myrinet network. As a result, the RMI throughput was severely limited. Table 3.3 shows the performance of Manta serialization. Unfortunately, Manta serialization is integrated into Manta RMI, making it hard to measure the serialization throughput separately. Therefore, we measure the throughput by inserting timing calls into the generated serialization code of a Manta RMI benchmark. The values in the *write* column of Table 3.3 shows the throughput obtained by the generated serializers, while the *read* column shows the throughput of the generated deserializers.

In Manta serialization, serializing an array consists of inserting a pointer into an I/O vector and storing a small amount of type and size information. Therefore, the overhead of serializing an array is constant (i.e., it is unrelated to the size of the array). As a result, the Manta serialization throughput for arrays is infinite.

	Manta Serialization	
benchmark	*write*	*read*
100 KByte byte[]	∞	55.0
100 KByte int[]	∞	55.0
100 KByte double[]	∞	54.9
1023 node tree total	20.0	7.6
1023 node tree payload	8.9	3.4

Table 3.3: Manta serialization throughput (MByte/s), on the DAS.

The deserialization on the receiving side includes allocating a destination array and copying the data from the Panda library to this array. Allocating a new array may also trigger the garbage collector, resulting in a significant delay before the deserialization can proceed. As a result, the deserialization throughput for arrays is approximately 55 MByte/s.

Serializing a tree is significantly more complex than serializing arrays. The entire tree must be traversed to insert all objects into the I/O vector, cycle detection must be performed, and type and reference information must be saved for every object. As a result, the total serialization throughput for tree data structures is only 20 MByte/s The throughput for the actual payload of the tree (i.e., the amount of user data transferred) is less than half this number, 8.9 MByte/s. When the tree is deserialized, object allocation, garbage collection and copying overhead are added, resulting in a throughput of 7.6 MByte/s (total) and 3.4 MByte/s (payload).

Null-RMI Latency

In Table 3.4 we show the null-RMI latency for the Manta RMI implementation on Fast Ethernet and Myrinet. For easy comparison we have added the latency numbers for three other RMI implementations (also shown in Table 2.4) and for Panda.

Null-RMI Latency	*Fast Ethernet*	*Myrinet*
Sun JDK 1.2	1480	1316
IBM JDK 1.1.8	720	542
Compiled Sun	490	301
Manta RMI	207	37
Panda	201	31

Table 3.4: RMI Latency on Fast Ethernet and Myrinet, all numbers in μs.

On Myrinet, Manta RMI obtains a null-RMI latency of 37 μs, only six microseconds higher than the latency offered by Panda. In comparison, the Sun JDK 1.2 (using just-in-time compilation) obtains a latency of 1316 μs, which is 35 times higher than Manta RMI. On Fast Ethernet, the difference between Panda and Manta is similar.

The most interesting comparison is between Manta RMI and *Compiled Sun* on Myrinet. Both these RMI implementations use the same compiler and runtime system, and they both communicate using the Panda library. Therefore, any performance difference is caused by the different RMI implementations. Also, *Compiled Sun* communicates using the socket model (PandaSockets on top of Panda), where Manta RMI uses Panda directly.

Table 3.4 shows that *Compiled Sun* obtains a null-RMI latency of 301 μs, which is 8 times slower than Manta RMI. Since serialization is not triggered in a null-RMI test (there are no parameters or return values), this difference is caused completely by the software overhead in the RMI and the network communication implementations. As Table 2.4 shows, the Java TCP layer of Compiled Sun obtains a round trip latency of 44 μs. Therefore, 279 μs can be attributed to the Compiled Sun RMI implementation, while the Manta RMI implementation introduces only 6 μs.

RMI Throughput

The throughput obtained by Manta RMI for several parameter types is shown in Table 3.5. Again, we have added the numbers for the various other RMI implementations (previously shown in Figure 2.5) and Panda.

	Fast Ethernet			
System	*byte[]*	*int[]*	*double[]*	*Tree 1023*
Sun JDK 1.2	6.0	3.1	3.5	0.08
IBM JDK 1.1.8	6.3	6.0	4.8	0.05
Compiled Sun	9.1	7.2	7.2	0.5
Manta RMI	10.7	10.7	10.7	1.7
Panda			11.2	

	Myrinet			
System	*byte[]*	*int[]*	*double[]*	*Tree 1023*
Sun JDK 1.2	7.3	3.8	3.7	0.08
IBM JDK 1.1.8	12.3	7.9	5.6	0.05
Compiled Sun	24.7	15.4	15.8	0.6
Manta RMI	52.5	52.2	52.2	2.6
Panda			60.5	

Table 3.5: RMI throughput on Fast Ethernet and Myrinet, in MByte/s.

Manta RMI achieves an array throughput of 52 MByte/s on Myrinet, much better than any of the other RMI implementations. In comparison, the throughput of *Compiled Sun* is 25 MByte/s for *byte* arrays, but drops to 16 MByte/s for *int* or *double* arrays. This decrease in throughput is caused by the conversion of the integers, floats, and doubles to bytes (and vice versa). As Manta RMI does not require any conversion, it can use the zero-copy communication offered by Panda. Therefore, Manta's

throughput for primitive arrays does not depend on the data type. When sending arrays, Manta RMI is able to fill 87% of the bandwidth offered by Panda. In comparison, *Compiled Sun* can only use 25% to 41% of Panda's bandwidth. These results clearly show the advantage of Manta's zero-copy approach to serialization.

The throughput of the binary tree benchmark is very low in comparison with the throughput achieved for arrays, only 2.6 MByte/s (payload). However, when we look at Table 3.3, we see that the payload throughput of the 1023-node tree is already limited to 3.4 MByte/s due to the overhead of deserialization. Table 3.5 also shows performance results on Fast Ethernet. Here, the differences are smaller, because the costs of network communication are higher.

Breakdown

We will now present a breakdown of the time that Manta spends in remote method invocations. The benchmark and measuring technique are described in more detail in Section 2.5.2. The Manta RMI results, shown in Table 3.6, can be compared directly to the results of *Compiled Sun*, shown in Table 2.6.

overhead	*Parameters*			
introduced by	*none*	*1 object*	*2 objects*	*3 objects*
Serialization	0	6	10	13
RMI	5	10	10	10
Communication	32	34	34	35
Method call	0	1	1	1
Total Latency	37	51	55	59

Table 3.6: Breakdown of Manta RMI on DAS using Myrinet (times are in μs).

A null-RMI on Myrinet takes about 37 μs with Manta. Only 5 μs are added to the round trip communication latency of Panda, which is 32 μs. Note that this communication latency is one microsecond higher than the Panda number shown in Figure 3.4. This is because a null-RMI does not send an empty Panda message. For a null-RMI 16 bytes are sent to the receiver, which returns a 10 byte reply. The Panda latency test sends empty messages instead.

When the RMI has a single object as parameter, the round trip latency increases to 51 μs. This large difference between passing zero or one object parameters can be explained as follows. First, as described in Section 3.3.1, the runtime system has to build a table to detect possible cycles and duplicates in the objects. This table can usually be extracted from a cache (saving memory allocation and initialization cost), but it must also be cleared and returned to the cache after use (adding a little overhead). The latency is increased further by the (de-)serialization of the object and the creation of a new parameter object on the server side. In total, this *Serialization* overhead adds

6 μs to the round trip latency. Note that since the cycle table is only created once, the cost of adding additional parameter objects is 3 to 4 μs rather than 6 μs.

Second, RMIs containing object parameters must be serviced by a dedicated thread because such an RMI may block by triggering garbage collection. This adds some 5 μs of thread-switching time to the *RMI Overhead* (regardless of the number of object parameters). Finally, adding parameter objects to the RMI requires more data to be sent, adding at least 2 μs to the *Communication* time.

When we compare this Manta RMI breakdown to the *Compiled Sun* breakdown shown in Table 2.6, we see that *Compiled Sun* is 8 times slower than Manta RMI (37 μs vs. 301 μs). Manta RMI improves the performance of all layers: compiler-generated serializers win by a factor 17 or more; the RMI overhead is 18 times lower; and the communication is 4 times faster.

3.4.2 Impact of specific performance optimizations

Manta RMI and Manta serialization perform various optimizations to improve their latency and throughput. Below, we analyze the impact of specific optimizations in more detail.

Using a scatter/gather interface

As explained in Section 1.2, the Panda library, on top of which Manta is built, provides a scatter/gather interface to minimize the number of memory copies needed. This optimization increases the throughput for Manta RMI. To assess the impact of this optimization we also measured the throughput obtained when the sender does not use the gather interface and must make an extra memory copy. In this case, the maximum throughput decreases from 53 to 44 MByte/s, because memory copies are expensive on a Pentium Pro [16]. This experiment thus clearly shows the importance of the scatter/gather interface. Unfortunately, dereferencing the scatter/gather vector involves extra processing, so the null-RMI latency of the current Manta RMI system is slightly higher than that for an earlier Panda version without the scatter/gather interface (34 versus 37 μs) [66].

Reducing byte swapping

As explained in Section 3.3.2, Manta serialization avoids byte order conversion by optimistically sending data in the byte order of the host. In standard serialization the sender always converts all data to network byte order. The receiver converts the format back to what it requires. In Manta, the receiver only does the conversion if necessary. So, if the sender and receiver have the same format, but this format is different from the standard RMI format, Sun RMI will do two byte-swap conversions while Manta will not do any byte swapping.

We measured the impact of this optimization by adding byte-swapping code to the sender side of Manta RMI. (This code is not present in the normal Manta system, since the sender never does byte swapping with Manta.) If byte swapping is performed by the sender and receiver, the throughput of Manta RMI for arrays of integers or floats decreases by almost a factor two. The maximum throughput obtained with byte swapping enabled is decreased from 53 to 30 MByte/s. This experiment clearly shows that unnecessary byte swapping adds a large overhead, which is partly due to the extra memory copies needed.

Escape analysis

As described in Section 3.3.3, the Manta RMI runtime system implements escape analysis. With this analysis, objects that are argument or result of an RMI but that do not escape from the method will be immediately returned to the heap. Without this optimization, such objects would be subject to garbage collection, reducing the RMI throughput. Without escape analysis the average throughput for Manta is reduced from 53 to 30 MByte/s. When we add escape analysis to the *Compiled Sun* implementation, the throughput for byte arrays is increased from 25 to 39 MByte/s. The other throughput numbers are hardly affected, however, because these cases also suffer from other forms of overhead, in particular byte swapping.

3.5 Application performance

The low-level benchmarks show that Manta obtains a substantially better latency and throughput than the *Compiled Sun* implementation. To determine the impact of the Manta RMI implementation on application performance, we have run the six parallel applications described in the previous chapter using Manta RMI. The application codes and input sizes are exactly the same as described in Section 2.5.3. To further evaluate the performance of Manta RMI, we have implemented four additional applications, which we describe below.

Figure 3.10 shows the speedups obtained by Manta RMI and *Compiled Sun* for the six applications. For both systems, the programs are compiled statically using the Manta compiler. The sequential execution times of the Manta RMI and *Compiled Sun* applications are very similar, since the applications are based on the same codes and both are compiled with the Manta compiler. However, some small variations do occur (e.g., due to differences in cache behavior). Therefore, we compute the speedups of all applications relative to the fastest of the two (parallel) versions running on a single machine. Therefore, a higher speedup always implies a shorter execution time.

Figure 3.10 shows that Manta RMI's higher communication performance results in substantially better application speedups. As described in Section 2.5.3, *Compiled Sun* performs well for only two applications, Water and ASP. In contrast, Manta RMI has a good performance on four out of five applications. Although Manta RMI is not

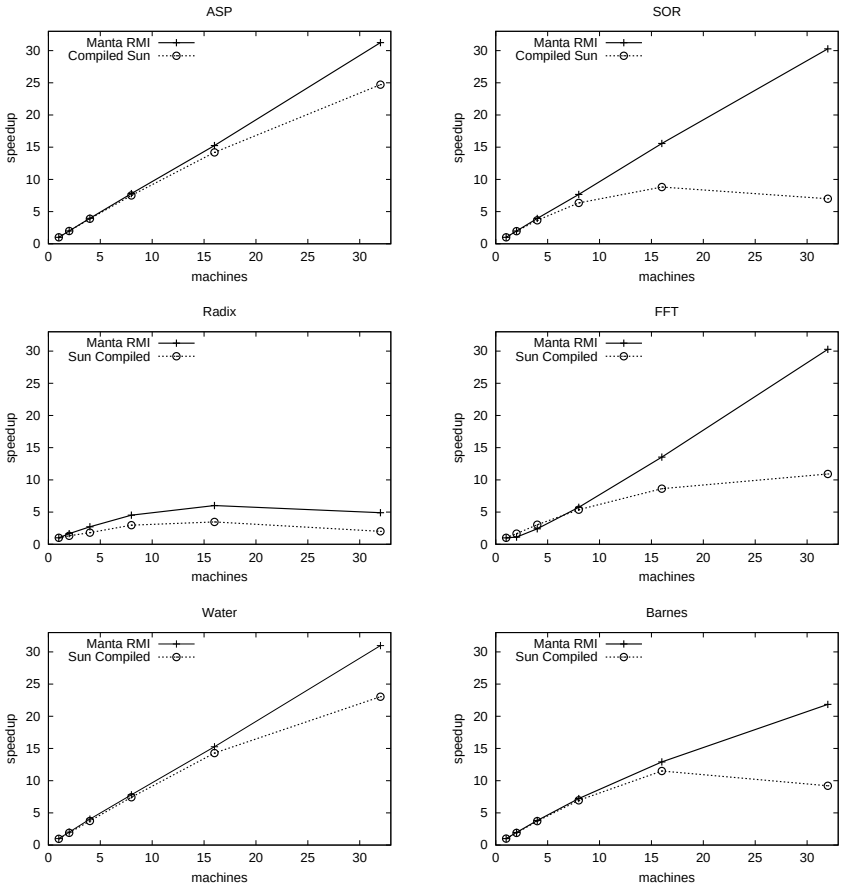

Figure 3.10: Application speedups.

able to obtain a good speedup with the Radix application, it still performs significantly better than *Compiled Sun* (maximum speedups of 6 and 3.5 respectively).

Table 3.7 gives performance data for the Manta RMI version of the applications, including the total number of messages sent (summed over all machines) and the amount of data transferred, using 16 or 32 machines. These numbers were measured at the Panda layer, so they include header data. Also, one Manta RMI generates two Panda messages, a request and a reply.

When we compare the numbers in Table 3.7 to their *Compiled Sun* counterparts, shown in Table 2.7, we see that the Compiled Sun sends far more messages for all applications than Manta RMI. The reason is that, as a result of the way serialization is

Program	16 CPUs Time (s.)	16 CPUs # Messages	16 CPUs Data (MByte)	32 CPUs Time (s.)	32 CPUs # Messages	32 CPUs Data (MByte)
ASP	25.64	38445	95.30	12.22	79453	196.96
SOR	10.96	44585	80.38	5.64	92139	166.11
Radix	1.19	9738	62.46	1.46	39418	124.68
FFT	3.87	8344	152.06	1.73	33080	157.14
Water	25.06	6023	8.44	12.54	23319	17.30
Barnes	14.90	18595	23.78	8.81	107170	52.26

Table 3.7: Performance Data for Manta on 16 and 32 CPUs.

implemented, Compiled Sun transfers large RMIs in chunks of 1 KByte. A seperate panda message is used for each chunk. In constrast, Manta RMI sends large RMIs as a single panda message. The volume of the data transferred by Manta RMI is somewhat lower than that for Compiled Sun, because Manta RMI sends fewer messages and thus fewer headers and does not send type descriptors for each class on every call.

Manta RMI obtains high efficiencies for all applications except Radix sort. Table 3.7 shows that Radix sends the largest number and volume of messages per second of all six applications. On 32 machines, almost 27,000 (39418/1.46) messages are sent per second, yielding a total rate of 85 (124.68/1.46) MByte/s. More detailed measurements show that, when running on 32 machines, each machine spends only 0.07 of the 1.49 seconds (i.e., 5% of the time) on computation. The other 1.42 seconds are spent in communication. This clearly shows that the computation to communication ratio in Radix is very low, thereby explaining its bad speedup.

When we compare the Radix speedup of Manta RMI to its Compiled Sun counterpart, however, we see that Manta RMI still performs 1.7 times better than Compiled Sun (on 16 machines). These results show that the low latency and high throughput offered by the Manta RMI implementation have a significant effect on application performance, even if the application itself is not very efficient.

3.5.1 Additional applications

To further evaluate the performance of Manta RMI, we have implemented four additional applications, which we will describe below. We will then discuss the speedup of these applications and the previous applications on 32 and 64 machines.

LEQ (Linear equation solver) is an iterative solver for linear systems of the form $Ax = b$. Each iteration refines a candidate solution vector x_i into a better solution x_{i+1}. This is repeated until the difference between x_{i+1} and x_i becomes smaller than a specified bound. The program is parallelized by partitioning a dense matrix containing the equation coefficients over the machines. In each iteration, each machine produces a part of the vector x_{i+1}, but needs all of vector x_i as its input. Therefore, all machines must exchange their partial solution vectors at the end of each iteration (using an op-

eration similar to MPI's *gather-to-all*). They must also decide if another iteration is necessary. To do this, each machine calculates the difference between their fragment of x_{i+1} and x_i. If the sum of these differences is smaller than some threshold, the application terminates. After each iteration, each machine forwards its partial solution and difference value to a central remote object. There, the partial solutions are combined into a new vector, and the sum of all the differences is determined. The results are forwarded to all machines using a spanning tree broadcast, similar to the one used in ASP. We used a 1000x1000 equation coefficient matrix.

ACP (the Arc Consistency program) can be used as a first step in solving Constraint Satisfaction Problems. The program takes as input a set of *n* variables in domain *m* and a set of binary constraints defined on some pairs of variables, that restrict the values these variables can take. The program eliminates impossible values from the domains by repeatedly applying the constraints, until no further restrictions are possible. The program is parallelized by dividing the variables statically among all machines. The solution is stored in a *n* by *m* matrix of booleans. Each boolean in this matrix describes a single (variable, value) pair. By setting a boolean to *false* the program restricts the values a variable can take. Every machine gets a copy of the matrix and restricts the values of its set of variables as much as possible. Because restricting a variable can have an effect on other variables, other machines are notified of these updates using a spanning tree broadcast. For termination detection, a single remote object is used. We use a problem size of 2000 variables. Each variable has 150 possible values.

TSP (the Traveling Salesperson Problem) computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. We use a branch-and-bound algorithm, which prunes a large part of the search space by ignoring partial routes that are already longer than the current best solution. The program is parallelized by distributing the search space over the different nodes. The program uses a centralized job queue to balance the load. Each job contains an initial path of a fixed number of cities; a node that executes the job computes the lengths of all possible continuations, pruning paths that are longer than the current best solution. To ensure that each machine has an up-to-date copy of the current best solution, updates of this value are forwarded to all machines (using a sequence of RMIs). The search space of TSP consists of 17 cities.

QR is a parallel implementation of QR factorization. In each iteration, one column, the *Householder vector H*, is broadcast to all machines, which update their columns using *H*. The current upper row and *H* are then deleted from the data set so that the size of *H* decreases by 1 in each iteration. The vector with maximum norm becomes the Householder vector for the next iteration. To determine which machine contains this vector, an operation similar to a reduce-to-all collective operation (as defined in the MPI standard [33]) is used. The machine containing the Householder vector then broadcasts it to all other machines using a spanning tree broadcast. We use a problem with a 2000x2000 matrix.

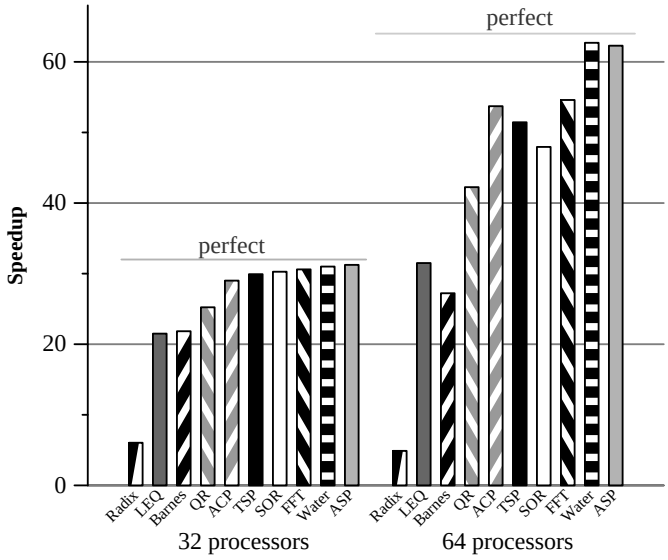

Figure 3.11: Speedup of applications on 32 and 64 machines using Manta RMI.

Figure 3.11 shows the performance of the applications. Nine out of the ten applications obtain a good speedup on 32 machines (Radix is the only exception). Of the four additional applications, TSP and ACP perform best with a speedup of 30 and 29 respectively. QR and LEQ have somewhat lower speedups of 25 and 22.

The communication behavior of TSP and ACP is very similar; both combine a centralized remote object with a simulated broadcast (a sequential send for TSP and a spanning tree for ACP). Also, for both applications, each machine can run its computation almost independently of the other machines (updates on the shared minimum in TSP and the boolean matrix in ACP can be processed asynchronously). Therefore, both applications obtain a good speedup on 32 machines, and also scale to 64 machines, where TSP obtains a speedup of 51, and ACP a speedup of 54.

In QR and LEQ, however, the communication is more synchronous and more complex. For both applications, all machines must participate in a collective operation (gather-to-all for LEQ and reduce-to-all for QR). As a result, all machines must run in lock step, and during the communication phase, machines are idle. This synchronous behavior results in lower speedups on 32 machines. When these applications are run on 64 machines, the overhead of communication becomes higher. More machines must participate in the collective operations, causing longer delays. Also, the amount of computation per machine is reduced by a factor two, thereby increasing the impact of the communication delays on the total run time. As a result, QR and LEQ obtain mediocre speedups of 42 and 32 on 64 machines.

Of the previous set of applications, ASP and Water have near perfect speedup on 64 machines. As we have explained in the previous chapter, this can be attributed to the small amount of communication done by Water, and the asynchronous nature of ASP. FFT also obtains a good speedup of 55. This application has the property that the amount of data communicated is constant (i.e., independent of the number of machines used). Although the number of messages per machine doubles whenever the number of machines is doubled, the size of each message is halved. Therefore, FFT obtains a good speedup, even though the number of messages grows significantly.

For Barnes-Hut, the speedup on 64 machines is only 27. As Table 3.7 shows, the number of messages sent by this application increases sharply when the number of machines is doubled. On average, when the total number of machines is doubled, the number of messages per machine is tripled, while the amount of data sent *per machine* remains constant. Therefore, every machine sends the same amount of data but uses three times the number of messages to do so. This affects scalability of Barnes-Hut resulting in a reduced speedup on 64 machines.

Finally, the Radix application still has hardly any speedup at all. On 32 machines, Radix already spent 95% of its time on communication. On 64 machines, when the amount of computation per machine is halved, this overhead grows even further. As a result, Radix obtains a speedup of 4.9 on 64 machines, even lower than the speedup of 6.0 on 32 machines.

3.5.2 Discussion

We have shown that the increased performance of Manta RMI has a positive effect on the speedup of the applications. For all six applications tested on both systems, Manta RMI obtains a speedup on 32 machines which is higher than that of Compiled Sun. Five out of the six applications actually have a good speedup when using Manta RMI, while only two applications obtain acceptable results with Compiled Sun.

However, after extending the set of applications to ten, we observed that some applications have problems with scaling up to 64 machines, especially if the application is synchronous in nature and uses complex (e.g., collective or broadcast) communication. On 64 machines, simulating such communication using multiple RMIs is no longer efficient enough to obtain good speedups.

This shows that there is a need to extend RMI with support for other forms of communication. For example, the Panda communication library used by Manta RMI also offers an efficient broadcast implementation that uses a spanning tree broadcast protocol implemented on the Myrinet network interfaces. If such a primitive could be used directly from Java, many applications would benefit. In the current (Java) spanning tree implementation, every time an RMI is forwarded, all parameters must be serialized on the sender and deserialized on the receiver, only to be serialized again when the receiver forwards the RMI to the next machine. When using a low-level (Myrinet) broadcast implementation, the data is only serialized once (by the sender)

and deserialized by every receiver. No intermediate serialization is required, significantly reducing the overhead.

Besides improving the performance, extending the RMI model will also be beneficial to the programmer. Implementing reduce-to-all, gather-to-all and spanning tree broadcast communication using RMI is quite complex and error prone. Providing the programmer with a simple but efficient alternative could significantly reduce the effort required for implementing parallel applications.

In the following chapters, we will describe how we have extended the RMI model with support for object replication and group communication, and show that these extentions improve the performance and reduce the complexity of the applications.

3.6 Related work

In this section we will describe related work. We will not only describe work that is directly related to our research on RMI and serialization, but we will also have a closer look at other Java-related research projects. Many of these projects can be found at the Java Grande Forum (*http://www.javagrande.org*).

Fast Communication Systems

Much research has been done since the 1980's on improving the performance of RPC protocols [45, 47, 91, 96, 99]. Several important ideas resulted from this research, including the use of compiler-generated (un)marshaling routines, avoiding thread-switching and layering overhead, and the need for efficient low-level communication mechanisms. Many of these ideas are used in today's communication protocols, including RMI implementations.

Except for the support for polymorphism, Manta's compiler-generated serialization is similar to Orca's serialization [7]. The optimization for nonblocking methods is similar to the single-threaded upcall model [58]. Small, nonblocking procedures are run in the interrupt handler to avoid expensive thread switches. Optimistic Active Messages is a related technique based on rollback at runtime [108].

Instead of kernel-level TCP/IP, Manta uses Panda on top of LFC, a highly efficient user-level communication substrate. Lessons learned from the implementation of other languages for cluster computing were found to be useful. These implementations are built around user-level communication primitives, such as Active Messages [29]. Examples are Concert [50], CRL [48], Orca [6], Split-C [25], and Jade [92]. Other projects on fast communication in extensible systems are SPIN [8], Exo-kernel [49], and Scout [72]. Several projects also study protected user-level network access from Java, often using VIA [21, 22, 110]. However, these systems do not support RMI.

Remote Method Invocation

There are several other papers which study the performance of RMI. In [83], for example, the performance of both serialization and RMI is analyzed. Their conclusion resembles our own. The serialization performance is limited due to the overhead of copying and conversion, while the RMI overhead is limited by an inefficient implementation. A similar analysis of serialization described in [70].

Several projects offer alternative RMI implementations. *KaRMI* [84] aims to be a drop-in replacement for both RMI and serialization. Their improved serialization implementation reduces copying overhead by improved buffering of data, and use a slim encoding of type information. They also provide an optimized RMI implementation, which supports non-TCP/IP communication.

The performance of Manta RMI is better than that of KaRMI. Their implementation has a null-RMI latency of 360 μs on a platform consisting of 350 MHz Pentium II PCs with Fast Ethernet (running Sun JDK 1.2), while Manta RMI has a null-RMI latency of 207 μs on 200 MHz Pentium Pro machines. On 500 MHz Digital Alpha machines, connected via Myrinet and running JDK 1.1.6, KaRMI obtains a null-RMI latency of 117 μs, while Manta RMI requires 37 μs when using Myrinet. Their benchmarks also indicate that they reach a maximum throughput of approximately 23 MByte/s when using Myrinet, well below the 52 MByte/s reached by Manta RMI.

These results clearly show the performance advantages of the approach used in Manta RMI. However, since KaRMI is implemented in pure Java, it does have the advantage that it can be used in combination with any JIT compiler.

NinjaRMI[1] is developed for the UC Berkeley Ninja project. NinjaRMI supports both reliable synchronous point-to-point communication (using TCP/IP) and unreliable, one-way or multicast communication (using UDP). We currently have no performance numbers for NinjaRMI. In [56], a similar UDP-based implementation of RMI is described, which provides reliable communication. However, performance numbers show hardly any improvements.

Asynchronous RMI [87] (ARMI) extends the RMI model with support for delayed result handling. Similar extentions are provided by *Reflective RMI* [101] (RRMI), which uses a reflection approach to invoke remote methods, instead of the remote reference (i.e., stubs and skeletons) approach. We will describe ARMI and RRMI in more detail in the related work section of Chapter 5.

Alternative Communication Models

In our approach, we optimize RMI to make it suitable for parallel programming. An alternative is to add support for an existing parallel programming model to Java

An example is *MPJ* [19, 39] (message passing for Java), an MPI-like message passing interface for Java, developed by the Java Grande Forum. This approach has

[1] http://www.cs.berkeley.edu/ mdw/proj/ninja/ninjarmi.html

the advantage that many programmers are familiar with MPI's programming model and that MPJ, like MPI, supports a richer set of communication styles than RMI, in particular collective communication.

MPJ can either be implemented purely in Java, or as a Java wrapper to an existing native library. Currently, no complete pure Java MPJ implementation exists yet. Several Java to MPI wrappers do exist, however [5, 37, 38]. In Chapter 5, we will use one of them, *mpiJava*, to evaluate the performance of our group communication extensions to RMI.

Unfortunately, using a Java wrapper to MPI results in some of the same problems that occurred in RMI and serialization. A high JNI[2] overhead, for example, causes the latency for calling MPI from Java to be much higher than calling MPI from C (346 versus 227 μs, measured on an SP2, see [37]). Also, when transferring objects, standard Java serialization is used, which does not perform well. However, most MPI applications transfer arrays, not objects. For this type of communication, MPI can be used directly, skipping Java serialization altogether.

Performance numbers presented in [38] show that benchmarks using a Java to MPI wrapper (and a JIT compiler) run approximately 2.5 times slower than their MPI/C or MPI/Fortran counterparts. However, when a native Java compiler (HPCJ [71]) is used instead of a JIT, this performance difference almost disappears. These results show that MPJ implementations are serious candidates for efficient parallel programming in Java, especially with the increasing performance of JIT compilers.

There is, however, one large disadvantage to MPJ. It is difficult to cleanly integrate MPI's message-passing style of communication into Java's object-oriented model. Also, MPI assumes a SPMD programming model that is quite different from Java's regular multi-threading model. In Chapter 5 we will introduce an alternative model for group communication in Java that cleanly integrates with the method-invocation style of communication used in RMI.

The *JavaParty* system [85] 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. It allows remote objects to be defined by adding a *remote* keyword to the class declaration, removes the need for elaborate exception catching of remote method invocations, and, most importantly, allows objects and threads to be created remotely. JavaParty is implemented on top of KaRMI.

IceT [40] uses message passing communication instead of RMI. It enables users to share JVMs across a network. A user can upload a class to another virtual machine using a PVM-like interface. By explicitly calling *send* and *receive* statements, work can be distributed among multiple JVMs

Satin [75–77] provides an alternative to the explicit communication of RMI and MPJ. In Satin, no explicit communication statements are required. Instead, the programmer uses a *divide-and-conquer* style to annotate potential parallelism in the application. The Satin system then automatically parallelizes the application using these

[2] The Java Native Interface is used to call C functions from Java applications.

annotations. The initial Satin implementation is based on the Manta compiler, and a pure Java version is currently being developed. The current Satin implementation achieves an excellent performance on both clusters and hierarchical wide-area systems. However, the divide-and-conquer model does limit the range of applications that can be expressed using Satin.

DSM Systems

Although RMI communication integrates cleanly into the Java object model, it is not completely transparent. The application programmer must explicitly define remote interfaces, bind remote objects using a registry, catch remote exceptions, and, more importantly, remote methods use call-by-value instead of call-by-reference semantics for passing parameters. Therefore, distributed shared memory (DSM) systems are an attractive alternative to using RMI. These systems provide a shared-memory programming model, while still executing on a distributed-memory system. They typically run multi-threaded Java applications, where both threads and data are automatically distributed across the available machines. It is then up to the DSM system that every thread is provided with the data that it needs.

Jackal is a software fine-grained DSM for Java based on the Manta compiler [103, 105, 106]. Jackal uses aggressive compile-time analysis to reduce the amount of communication required during run time. It uses a number of advanced optimizations, such as *object-graph aggregation* and *automatic computation migration*, to further increase the application performance. In [106], performance numbers are presented. There, the speedups of multi-threaded versions of TSP, ASP, SOR and Water running on the Jackal DSM are compared to the Manta RMI versions. The applications run on a maximum of 16 machines. The speedups show that for TSP and SOR, Jackal is able to achieve a performance similar to Manta RMI. For ASP and Water, however, Manta RMI easily outperforms Jackal.

Hyperion [2, 67] uses the same approach as Jackal. It translates Java applications to C, compiles them, and executes them on PM2, a distributed, multi-threaded runtime system that provides a DSM implementation.

A different approach is used in the *cJVM* system [3]. cJVM provides a parallel JVM implementation, capable of running on a cluster. Each machine in the cluster runs a cJVM process. This collection of processes appears as a single JVM to the application. When an object reference is forwarded from one cJVM process to another, a special proxy (similar to a remote interface) is created, which automatically forwards any method invocations to the original object. Internally, the cJVM processes use MPI for communication.

Kaffemik [1], also implements a parallel JVM. However, instead of generating proxies to share objects, Kaffemik allocates objects in a special address space which is shared between all machines. Similar approaches are used in *DOSA* [44] and *Java/DSM* [113], which are both implemented on top of TreadMarks [53], and in

JESSICA[61, 114]. In these systems, no explicit communication is necessary in the JVM implementation. All communication is handled by the underlying DSM system.

Language extentions

Some projects extend the Java language to make it more suitable for high-performance scientific computing. *Titanium* [112], for example, extends Java with features like immutable classes, fast multidimensional array access and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium to C. Titanium is built on the Split-C/Active Messages backend.

Spar/Java is a data and task parallel programming language for semiautomatic parallel programming [88]. It supports real multi-dimensional arrays, complex numbers, tuples, and parallelization constructs such as *foreach.* Spar compiles to C++.

JOMP extends Java with OpenMP like directives for parallel processing on shared memory multi-processor machines. These directives can be used to annotate parallelism in the application. JOMP uses a special Java-to-Java compiler, which rewrites the application, and exploits the parallelism by generating extra classes and inserting calls to the JOMP run time system. The rewritten application can then be compiled to bytecode an run on an ordinary JVM.

JIT and native compilers

Besides the Java implementations provided by Sun and IBM, several other JIT compilers and native Java compilers exist. Some systems, like *Jalapeño* [18], *SableVM* [34], *OpenJIT* [68], and *Jupiter* [28] are designed to be extensible, and are used as platforms for research into just-in-time compilation itself.

Other systems, like *NINJA* [71], are more domain specific. NINJA uses a technique called *semantic expansion* to obtain very high performance in a specific set of numerical applications. Using semantic expansion, complete Java classes or libraries are recognized by the compiler (or JIT), and replaced by a high-performance native implementation with the same semantics. This removes the need for any compilation or optimization, and has the advantage that no language extensions are necessary. When the same application is run on a different JVM (without semantic expansion), the ordinary bytecode implementation is used. Using this technique, NINJA has achieved a performance comparable to Fortran for some applications.

3.7 Conclusion

In this chapter, we have described an implementation of Java's RMI that is designed specifically for parallel programming on homogeneous cluster computers. This implementation, called Manta RMI, provides highly efficient communication. The Manta

RMI and serialization implementations are based on compile-time information to generate specialized serialization and marshaling routines. This approach, in combination with a fast communication library that supports zero-copy communication, significantly reduces the runtime overhead of serialization and RMI.

Communication with JVMs (running an RMI implementation that adheres to the standard [98]) is still possible, but slower. Using our system, all machines in a parallel system can communicate efficiently using Manta RMI, but they still can communicate and interoperate with machines running other Java implementations (e.g., for visualization).

To understand the performance implications of these optimizations, we compared the performance of Manta RMI with that of *Compiled Sun*, an RMI implementation based on Sun JDK 1.1, that is compiled with Manta's native compiler and achieves better latency and throughput than the other RMI implementations we tested. The performance comparison on a Myrinet-based Pentium Pro cluster shows that Manta RMI is substantially faster than this compiled Sun RMI implementation. On Myrinet, the null-RMI latency is improved by a factor of 8, from 301 μs (for *Compiled Sun*) down to 37 μs (for Manta RMI), only 6 μs slower than the Panda communication library used in the implementation.

Although latency and throughput benchmarks give useful insight into the performance of communication protocols, a more relevant factor for parallel programming is the impact on application performance. We therefore used a collection of ten parallel Java application to evaluate the performance of Manta RMI, and showed that Manta RMI obtains a significantly better speedup than Compiled Sun. We also show, however, that simulating complex collective or broadcast communication using the RMI model does not scale flawlessly to 64 machines.

Simulating complex collective or broadcast communication using multiple RMIs is no longer efficient enough to obtain good speedups. Therefore, in the following chapters, we will investigate how the RMI model can be extended to support more advanced forms of communication that make parallel programming in Java both more efficient and easier.

Chapter 4

Replicated Method Invocation

4.1 Introduction

In the previous chapters we have described the RMI model and we have shown how the RMI performance can be increased to a level suitable for high performance parallel programming. However, insufficient performance was not the only limitation of RMI. As we have shown in Section 2.2, the RMI model only supports *synchronous point-to-point* communication. Parallel applications often require more advanced models, such as *object replication* or *group communication*. These models simplify the writing of applications and provide an opportunity to further increase application performance. In this chapter, we will focus on extending Java with a simple and efficient object replication model that integrates cleanly with RMI. Group communication will be discussed further in Chapter 5.

Object replication is a well-known technique to improve the performance of parallel, object-based applications [6]. Several different forms of object replication have been proposed for Java. However, no scheme exists yet that transparently and efficiently supports replicated objects and that integrates cleanly with Java's primary communication mechanism, RMI. Some systems temporarily cache objects rather than trying to keep multiple copies of an object consistent [41, 56, 67, 104]. Some proposals have a programming model that is quite different from the object invocation model of RMI [102]. Also, performance results are often lacking or disappointing. The reason for these problems is the inherent difficulty in implementing object replication. In particular, it is hard to find a good programming abstraction that is easy to use, integrates well with RMI, and can be implemented efficiently.

In this chapter we introduce a compiler-based approach for object replication in Java that is designed to resemble RMI. Due to its similarity to RMI, we call our approach *Replicated Method Invocation* (RepMI). Our model does not allow arbitrarily complex object graphs to be replicated, but deliberately imposes restrictions to obtain

a clear programming model and high performance. Briefly, the RepMI model allows the programmer to define closed groups of objects, called *clouds*, that are replicated as a whole. A cloud has a single entry point, called the root object, on which its methods are invoked. RepMI uses compiler and runtime system support to determine which methods will only read (but not modify) the object cloud; such read-only methods are executed locally, without any communication. Methods that modify any data in the cloud are broadcast and applied to all replicas. A single broadcast message is used to update the entire cloud, independent of the number of objects it contains. The semantics of such replicated method invocations are similar to those of RMI.

We have implemented this scheme in the Manta system by extending the Manta compiler and runtime system. Using Manta, updating a simple object replicated on 64 Myrinet-connected machines takes 155 μs, only about four times the Manta RMI latency. To illustrate efficiency and ease of programming of replicated objects in Manta, we have also implemented five parallel Java applications that use replicated objects.

In this chapter, we provide an in-depth evaluation of our replication mechanism. The outline of the rest of this chapter is as follows. In Section 4.2, we introduce a new model, similar to RMI, that allows closed groups of objects to be replicated. In Section 4.3, we describe our implementation of this model as part of the Manta system and analyze the performance of this implementation on a Myrinet cluster using a micro benchmark. In Section 4.4, we evaluate the performance benefits of object replication in Java. In Section 4.5, we look at related work. Finally, in Section 4.6, we present our conclusions.

4.2 Model

The goal of our object replication mechanism is to make parallel Java applications more efficient, while reducing their implementation complexity.

As we have explained in Chapter 2, (parallel) applications that use RMI follow Java's object-oriented model in which client objects invoke methods on server objects in a location-transparent way. Each remote object is physically located at one machine. Although the RMI model hides object remoteness from the programmer, the actual object location has a strong impact on application performance. If a remote object is frequently accessed, the communication overhead can seriously degrade application performance. For some applications, this problem can be solved by replicating the remote object.

From the client's point of view, object replication is conceptually equivalent to the RMI model: methods are applied on a remote object. The difference is in the implementation. The replicated object may be physically located on multiple machines. The advantage of replication is that *read-only methods* (i.e., methods that do not modify the object's data) can be performed locally, without any communication. The disadvantage is that *write methods* (i.e., methods that do modify the object's data) become more complex and have to keep the state of object replicas consistent. For objects that

have a high read-write ratio (i.e., they are mostly read and hardly written), replication will reduce communication overhead.

Replication can be implemented in different ways, influencing both performance and the programming model. Many systems that use replication apply an *invalidation* scheme where the inconsistent replicas are removed (invalidated) after a write method. Experiences with the Orca language [6], however, show that for object-based languages an *update* protocol often is more efficient, especially if it is implemented using *function shipping*. With this strategy, each method invocation that modifies an object is applied to all replicas. For object-based systems, this strategy is often more efficient than invalidation schemes. Especially if a large amount of data is replicated (e.g., a big hash table), invalidation is unattractive, as each machine must then retrieve a new copy of the data on the next access. With function shipping, only the method and its parameters are forwarded, usually resulting in much smaller data transfers than with invalidation schemes or data shipping schemes, which send or broadcast entire objects.

Another advantage of using object replication with function shipping is that the model is similar to the RMI model. RMI provides a simpler form of function shipping that only forwards methods to a single (remote) object, instead of multiple (replicated) objects.

Because of these advantages, RepMI uses an update mechanism based on function shipping. The programming interface of RepMI (described in Section 4.2.1), is designed to resemble that of RMI. This minimizes the effort required to convert an RMI application to RepMI.

Figures 4.1 and 4.2 show an example of the RepMI model. In RepMI, references to a replicated object are called *replication references*. These references may only refer to a special interface type, a *replication interface*, that must be implemented by the object that is replicated. This is similar to the RMI model described in Section 2.2.

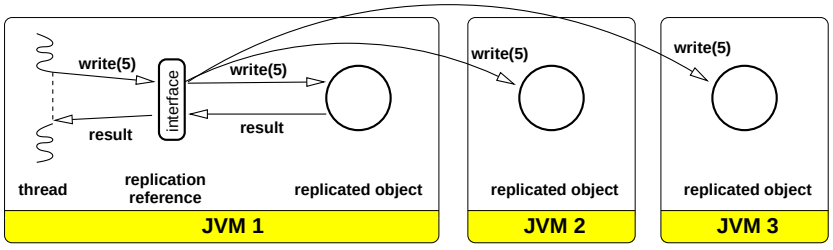

Figure 4.1: A write method is applied on an object replicated on three JVMs.

The examples show a thread that uses a replication reference to invoke methods on an object that is replicated on three machines. Such a method invocation can be

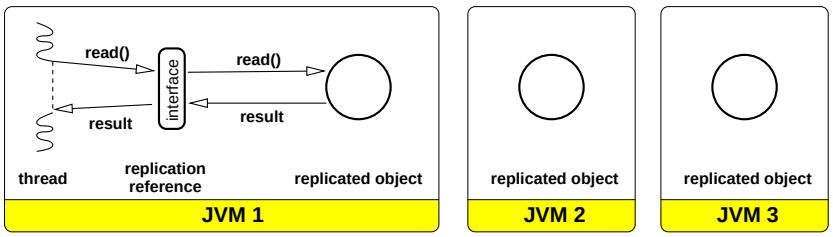

Figure 4.2: A read method is applied on an object replicated on three JVMs.

handled in two different ways. Write methods, which change the object, must be forwarded to all replicas (shown in Figure 4.1), while read methods, which do not change the object, are only applied on the local copy (shown in Figure 4.2). Since write methods are executed once per replica, multiple result values (or exceptions) may be produced. Because the replicas are kept consistent, we know that all the result values will be identical. Therefore, they can all be discarded except for the local result, which is returned to the invoking thread.

As with RMI, methods of a replicated object use *call-by-value* rather than *call-by-reference* semantics. A copy is made of all arguments and return values of the methods. Read methods also use call-by-value semantics, even though they do not require any network communication (we will explain this further in Section 4.2.2).

A difficult problem with object replication is that a method invoked on a given object can also access many other objects, by following references in the first object. A write method can thus access and update an arbitrarily complex graph of objects. Synchronizing multiple concurrent write methods on different (but possibly overlapping) object graphs is difficult and expensive. Also, if the function-shipping update strategy is applied naively to graphs of objects, broadcast communication would be needed for each object in the graph, resulting in high communication overhead. Languages like Orca, which are specially designed to support replication, avoid these problems by using an object model that forbids references between objects. Orca does support replicating linked data structures like lists and trees, but the entire data structure would be a *single* object.

A simple solution for Java would be to replicate only objects that do not contain references. Unfortunately, this would be far too restrictive since practically all data structures in Java consist of multiple objects. Our solution to this problem is to take an intermediate approach and replicate *closed groups of objects*, which we call *clouds*. A cloud is a programmer-defined collection of objects with a single entry point, that will be replicated and updated as a whole. An example is shown in Figure 4.3, where a cloud is used to replicate a tree data structure.

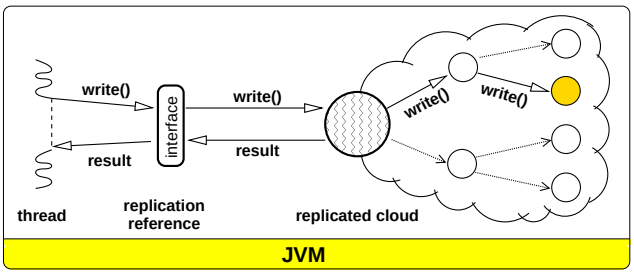

Figure 4.3: A replicated cloud.

The entry point of a cloud is called its *root*, and it is the only object that can be accessed by objects outside of the cloud. In addition, a cloud may contain an arbitrary graph of objects that are reachable from the root. These are called *node* objects. They may only be referenced from objects that are part of the same cloud.

In this model, the only way to manipulate (read or modify) the cloud is by invoking methods on the root object (through a replication interface). These invocations will be forwarded to the root object using call-by-value semantics. All other method invocations that occur inside the cloud will use normal call-by-reference semantics. When the cloud is replicated on multiple machines, the replicas can be kept consistent by broadcasting the write methods that are applied on the replication interface. The size of this broadcast message only depends on the parameters of the write method. It is independent of the number of objects in the cloud.

This model is general enough to express all common data structures like lists, graphs, hash tables, and so on. A number of restrictions are required on the behavior of the cloud objects to ensure that the replicas will remain consistent and to allow a simple and efficient implementation. These restrictions will be discussed in Section 4.2.2. As the Java object model has no notion of grouped objects (i.e., the clouds), we have defined a simple programming interface for RepMI to express this grouping mechanism. We will discuss this interface in the next section.

4.2.1 Programming interface and example

Object clouds are created at runtime. However, not every type of object may be part of a cloud. The application programmer can use two marker interfaces to mark cloud objects. Although this is not completely transparent, it is similar to the approach of RMI, where the marker interface *java.rmi.Remote* is used to identify remote objects.

To enable an object to be the root of a replicated cloud, it must implement the *manta.repmi.Root* interface. Objects that will be used as a node of a cloud must

implement *manta.repmi.Node*. The use of these interfaces allows a compiler[1] to recognize cloud objects and generate replication related code. This compiler must also enforce certain restrictions on the cloud objects, as we will explain in Section 4.2.2.

```
interface Stack extends manta.repmi.Root {
   public void push(int d) throws ReplicatedException;
   public int pop() throws ReplicatedException;
   public int top() throws ReplicatedException;
}

class StackRoot extends manta.repmi.CloudObject implements Stack {
   private StackNode top = null;

   public StackRoot() throws ReplicatedException { }

   public StackRoot(int val) throws ReplicatedException {
      push(val);
   }
   public synchronized void push(int val) throws ReplicatedException {
      top = new StackNode(val, top);
   }
   public synchronized int pop() throws ReplicatedException {
      StackNode temp = top;
      if (temp != null) top = top.prev;
      else ...  // throw exception.
      return temp.value;
   }
   public synchronized int top() throws ReplicatedException {
      if (top == null) ... // throw exception.
      return top.value;
   }
}

class StackNode extends manta.repmi.CloudObject
                implements manta.repmi.Node {

   StackNode prev;
   int value;

   public StackNode(int value, StackNode prev) {
      this.value = value;
      this.prev  = prev;
   }
}
```

Figure 4.4: A replicated stack (pseudocode).

[1] Like the Manta compiler or a special post processor such a the *rmic* RMI compiler.

To illustrate the use of the two marker interfaces, Figure 4.4 shows a simple example of a replicated stack. In the example, we first define a *Stack* interface that extends *manta.repmi.Root*. Therefore, any object implementing this *Stack* interface will be suitable for use as the root object of a cloud. Note that by defining methods in the *Stack* interface, we are actually defining the interface of a cloud: only the methods in *Stack* may be invoked on a cloud of which the root object implements *Stack*. Exactly the same approach is used in RMI to define which methods may be invoked on a remote object.

After the *Stack* interface has been defined, we can create a *StackRoot* class that will function as root object. This *StackRoot* uses a linear list of *StackNode* objects to store the actual data. The *StackNode* must implement the *manta.repmi.Node* interface, so they can become part of a cloud. Together with the root, these objects form a well-defined closed group.

When a replicated stack is created, the *push* method can be used to add *StackNode* objects to the cloud, while the *pop* method removes them. Both methods are write methods, since they change the contents of the cloud. The *top* method will only return the data on the top of the stack, without changing the cloud. Therefore, it is a read method.

Besides implementing the marker interfaces that identify them as root or node objects, the objects of the cloud must also extend *manta.repmi.CloudObject*. This object contains some basic functionality required by the RepMI runtime system. It also redefines some of the methods of *java.lang.Object* which would otherwise produce erroneous results in a replicated setting. This will be explained in more detail in the next section.

Figure 4.5 illustrates how the *Cloud.create* method can be used to create a new replicated stack (shown in the *Server* class). The function of this method is similar to that of *Naming.bind* of RMI. A new cloud is created with an object of the class *StackRoot* as its root. This cloud is then exported using the name *"stack1"*. Other machines can create a reference to this cloud by using the *Cloud.lookup* method (as shown in the *Client* class), or by receiving a reference from a different machine using a normal RMI call. When a new replicated cloud is created, the RepMI runtime system may immediately create replicas on all machines, or create them on demand whenever a reference to a cloud is first received. With both implementations, the RepMI runtime system must ensure that a reference to a cloud refers to the local replica.

The *Cloud.create* creates the root object using reflection. The *StackRoot* class used in the example is created using a *newInstance* call, which invokes the parameterless constructor of *StackRoot*. Root objects are also allowed to have a constructor with one or more parameters. These parameters must be encapsulated in an object array and passed as an additional parameter to *Cloud.create*, which in turn forwards copies of the parameters to the correct constructor.[2] An example of this is also shown in Figure 4.5, where a second root object is created using a constructor with an *int*

[2] The *java.lang.reflect* package contains all the required functionality to implement this.

```
import manta.repmi.Cloud;

class Server {
   public static void main(String [] args) {
      Stack s = (Stack) Cloud.create(StackRoot.class, "stack1");
      s.push(42);

      Object [] pm = { new Integer(5) };
      Stack s2 = (Stack) Cloud.create(StackRoot.class, "stack2", pm);
   }
}

class Client {
   public static void main(String [] args) {
      Stack s = (Stack) Cloud.lookup("stack1");
      int value = s.pop();
   }
}
```

Figure 4.5: Using a replicated stack (pseudocode).

value as a parameter.[3]

The *Cloud.create* method returns a replication reference which refers to the newly created root object. This replication reference serves the same purpose as a remote reference in RMI. It hides the actual location and implementation of the object from the user. Like remote references, replication references do not refer to the destination object directly, but refer to a stub object instead. This stub object contains the communication code necessary to implement RepMI.

From the programmer's point of view, the behavior of a replicated cloud is similar to that of RMI's remote objects. Clouds can be exported, looked up, and passed by reference via RMI, just like ordinary remote objects. Like in RMI, methods must be defined in a special interface before they can be invoked on a cloud. Like in RMI, these methods will use call-by-value semantics. Other than syntax, the only differences between RMI and RepMI will be the performance of the application and a number of restrictions that are imposed on the cloud objects. We will now discuss in detail how the clouds are kept consistent, explain the restrictions, and why they are necessary.

4.2.2 Replica Consistency

The RepMI model, like the RMI model, is not completely transparent. Restrictions apply on replicated objects due to the presence of multiple address spaces and the

[3] Note that Java's reflection API requires the constructor parameters to be forwarded in an object array. Therefore, to store the *int* parameter in such an array, it must first be encapsulated using an *Integer* object.

need to ensure the consistency of replicas. Some of these restrictions are embedded in the RepMI model and do not need to be enforced explicitly. Others need compiler support or a specific implementation of the RepMI runtime system to ensure that the replicated objects behave correctly. We will now discuss the restrictions in detail.

Preventing direct access

One of the most important steps towards keeping the replicas consistent is ensuring that the cloud can only be modified by applying a write method on a replication reference. Direct access to any of the fields or methods of cloud objects from outside of the cloud could immediately make the replicas inconsistent. Two examples are shown in Figure 4.6, where *thread 1* directly changes integer field *i* in the root object and *thread 2* invokes the method *write* on one of the node objects. Such direct changes to cloud objects will not be propagated to other replicas because they do not trigger the necessary communication code which is located in the replication reference. As a result, the replicas would become inconsistent.

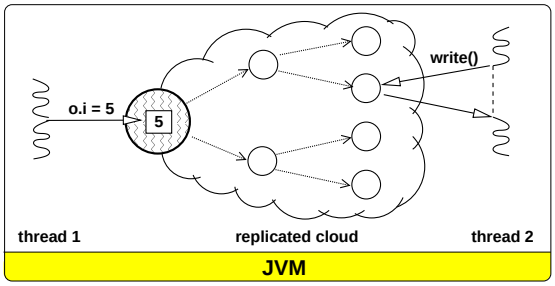

Figure 4.6: Incorrect use of cloud objects.

As the example shows, it is important to ensure that all accesses to the cloud are performed using the replication reference. Only then will the RepMI runtime system be able to forward the write operations to all replicas. Fortunately, the RepMI model is designed in such a way that it is relatively easy to enforce this restriction.

As we have described in the previous section, new clouds are created using a special library method, *Cloud.create*[4]. This method creates the root object, invokes the appropriate constructor and exports the Cloud. Instead of a direct reference to the root object it returns a replication reference. Since replication references are *interface* references, it is not possible to use them to directly access the fields of the root object. Only method invocations are allowed.

Node objects can be added to the cloud by explicitly creating them inside the cloud or by passing them as an argument to a method of the replication reference. Since the

[4]Using the *new* operation will not create a replicated cloud. Instead a normal object is created.

methods of the replication reference use call-by-value semantics, a copy will be made of all objects that are used as a parameter. This prevents the cloud from importing any direct references. For example, when a thread passes an object as an argument to a method invocation on a replicated tree data structure, it will always have a reference to this object. If the object would be inserted into the cloud in a call-by-reference manner, the thread would have a direct reference into the cloud (as shown by *thread 2* in Figure 4.6). Any use of this reference would make the cloud replicas inconsistent. By using call-by-value semantics, a new copy of the object is inserted into the cloud, to which no outside references exist.

For write methods, call-by-value semantics must be used because the method invocation may be forwarded to other machines. Therefore, using call-by-value semantics for replica consistency comes at no extra cost. Read methods, however, do become more expensive in RepMI. Although no communication is required for a read method (it is only applied on the local replica), call-by-value semantics are still used to prevent the cloud from exporting any references. For example, if one of the objects in a replicated tree data structure would be returned as a method result using call-by-reference, it would create a direct reference into the cloud. Instead, a copy is made of all returned objects. The returned reference will then refer to the copy, not to the original object which may still be part of the cloud.

Static fields and methods

Unfortunately, passing references as method arguments or return values is not the only way to import or export direct references to cloud objects. Static fields (or methods) are not part of any object instance, but can be seen as global variables (or global methods). They can be used anywhere in the Java code and do not require an object reference to access (a class type is used instead). Thus, a reference to a cloud object can simply be exported by storing it in a static field or passing it as a parameter to a static method. An example is shown in Figure 4.7, where *thread 1* is able to directly access a cloud object via a static reference field.

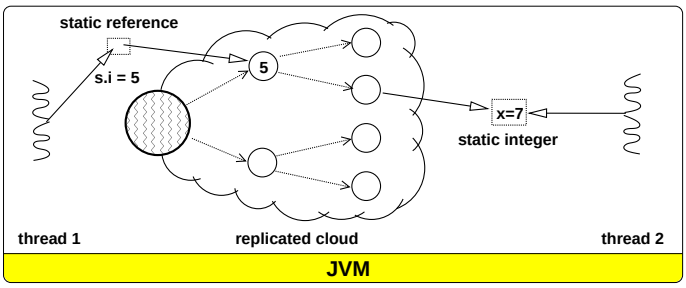

Figure 4.7: Incorrect use of static fields by cloud objects.

Another problem is introduced when cloud objects read data from static fields. Since the contents of static fields are not controlled by the RepMI runtime system (they are not part of any cloud object), there is no guarantee that their values cannot be changed by some external access. An example is shown in Figure 4.7, where *thread 2* modifies a static field that is also read by a cloud object. Because this static field may contain different values on different machines, using it may cause the replicas to become inconsistent. To prevent these problems, the use of static variables or methods is not allowed in cloud objects. This restriction must be enforced by a compiler.

Nondeterministic methods and exceptions

RepMI keeps the replicas consistent by forwarding the write methods. This approach assumes that the execution of the forwarded method will produce exactly the same result on all machines. Although this is generally true for most Java code, some methods may produce non-deterministic results.

One important example is the *hashCode* method of *java.lang.Object*. When an object is created, it is assigned a *hashcode* by the JVM, a single integer value which identifies the object and can be used to implement data structures such as hash tables. Unfortunately, the value of a hash code for a specific object depends on the implementation of the JVM and the behavior of the rest of the application. For example, a hash code may be determined by the object's memory location, but it could also be a sequence number that indicates that it is the Nth object that was allocated (the specification even allows the hash code to depend on the contents of the object). Thus, it may not be possible to deterministically determine the hash code value of a specific object, especially if multiple threads are present. Although the hash code of an object on one machine will normally remain constant throughout the object's lifetime, it is unlikely that the hash codes of two objects (replicas), created on different machines, will have the same value (especially since the JVM has no notion of object replication).

As a result, using the *hashCode* method to implemented a replicated data structure may result in replicas that are inconsistent. In a replicated hash table, for example, each replica may hash an object to a different location. Whether or not this inconsistency will cause problems depends on the application (e.g., a *lookup* operation may only indicate if the data is present, independent of its location).

Fortunately, this problem can be solved in RepMI by redefining the *hashCode* method in the *CloudObject*. By defining *hashCode* in such a way that it returns the same result on different JVMs, replicated data structures such as hash tables can be kept consistent.

Other nondeterministic methods may depend on I/O, the local time or start a thread which may be scheduled differently on different machines. By disallowing the use of threads and native or static methods inside clouds, RepMI prevents most of these problems. Unfortunately, it is only a partial solution. It is possible that the result of a method is influenced by the configuration of the local machine. For example, a write method that is forwarded to ten machines may produce nine identical results

and one *OutOfMemoryException.* There is very little that can be done in the RepMI model, compiler or runtime system to prevent this. The only option here is to detect the exception in the RepMI runtime system and abort the application.

Remote and replication references

As we described earlier, objects in a cloud may not have references to objects outside of the cloud. This restriction is enforced by using call-by-value semantics for methods invoked on the replication reference. When an object is passed as an argument to such a method, the object itself and all objects reachable by following its references will be copied into the cloud (and thus become part of it).

Remote objects and clouds, however, are never passed by value, even if the method uses call-by-value semantics. For example, if a remote reference is forwarded to all replicas of a cloud, the remote object will not be replicated. Instead, all cloud replicas will share a reference to *the same* remote object. This can lead to the *nested invocation problem* [69], illustrated in Figure 4.8.

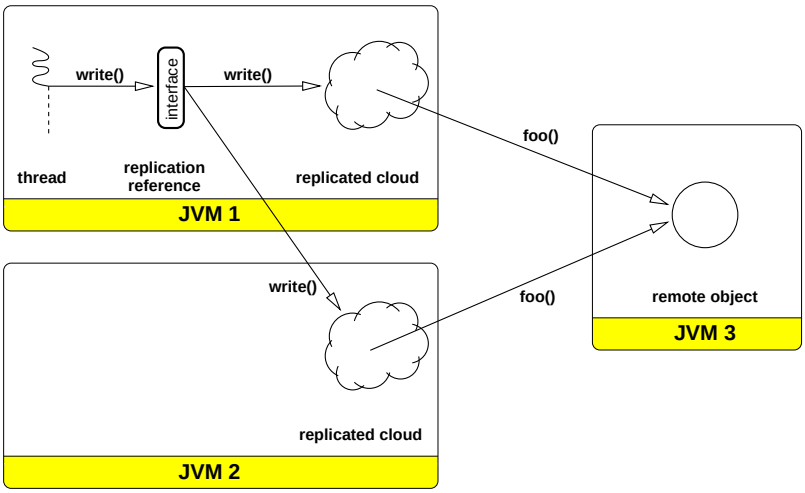

Figure 4.8: The nested method invocation problem

In this example, a thread on JVM 1 invokes a write method on a replication reference. This invocation is forwarded to both replicas of the cloud (on JVM 1 and 2). When the method is applied on the replicas, it leads to a *nested invocation*, *foo*, on the remote object. Because there are two replicas of the cloud, *foo* will be invoked twice. In general, this leads to erroneous program behavior because the result now depends on the actual number of replicas. A similar problem occurs with replication

references. If one cloud contains a reference to another cloud, an invocation on the first may lead to multiple invocations on the second.

To prevent this nested invocation problem, cloud objects are not allowed to contain or use any remote or replication references. This can partly be checked using compiler analysis (none of the fields, local variables or method arguments in a cloud object may have an interface type that extends *java.rmi.Remote* or *manta.repmi.Root*). However, due to Java's support for multiple inheritance for interfaces, some runtime support is also necessary. For example, an interface B may extend both *java.rmi.Remote* and another interface A. As a result, a method expecting a normal interface reference parameter of type A may actually receive a remote reference parameter of type B at runtime. Since is impossible to detect this at compile time, some runtime checks are needed to ensure that no remote references are inserted into a cloud.

Using inheritance in RepMI

In our API, two marker interfaces, the *root* interface (*manta.repmi.Root*) and the *node* interface (*manta.repmi.Node*), can be used to mark an object as either the root or a node of a cloud. An object can implement such an interface directly or through inheritance. For example, the root interface is normally extended by some other interface that defines which methods may be applied on a cloud. This interface, which has *manta.repmi.Root* as an ancestor, will then be implemented by the class of a root object. The node interface, however, is usually directly implemented by node object classes (since it is not used to define any methods). Root and node classes can be extended by other types, which will then also become root and node classes. All classes that implement the root or node interface must also extend the class *manta.repmi.CloudObject* and vice-versa. Thus, all cloud objects will always have *CloudObject* as an ancestor and can always be identified as being either a root or node object.

An object is not allowed to implement both root and node interfaces at the same time, because that would make it more difficult to cleanly separate different clouds from each other. For the same reason, an interface is not allowed to extend both a root and a node interface. However, it is allowed for a class to implement multiple node interfaces or multiple root interfaces. Similarly, an interface is allowed to extend multiple root or multiple node interfaces. Since remote references are not allowed inside clouds, root and node classes are not allowed to also implement a remote interface.

In RepMI, we want to be able to determine *at compile time* that all cloud objects behave according to the rules. Therefore, the compiler restricts the type of object or interface references that can be used in cloud objects (as fields, local variables or method arguments). Only object references of a class type that has *manta.repmi.CloudObject* as an ancestor may be used. All interface references must have *manta.repmi.Node* as an ancestor. This ensures that any object that implements such an interface will also extend *manta.repmi.CloudObject*, and is thus checked by the compiler. Primitive types and *java.lang.String* may always be used in cloud objects. Arrays may also be used, provided that they contain data of one of the types mentioned above.

Read/write analysis

The advantage of object replication over RMI is that methods which only read objects can be performed locally, without any communication. Only write methods cause communication across the set of replicas. To distinguish between read and write methods, the compiler has to analyze the method implementations. It checks if the method contains assignments to object fields, if there are calls to the *notify* or *notifyAll* synchronization primitives, and if there are calls to other write methods. If so, the method is classified as a write method, otherwise it is considered to be a read method.

Unfortunately, this analysis cannot be performed completely at compile time. When a read method performs a method invocation itself (a *nested* invocation), it is hard to determine at compile time which method will be invoked. Due to Java's support for inheritance, the actual method that is invoked depends on the runtime type of the object. Since a read-only method of one class may be overridden by a write method in a subclass (or vice versa), it may not be known until runtime whether such a nested method invocation reads or writes a cloud. Still, it is important to execute each method in the correct mode (read or write). If a read-only method would be executed as if it were a write method, it would be broadcast, resulting in a high communication overhead. Even worse, if a write method would accidentally be executed in read mode, erroneous program behavior would occur. Due to this problem, the final check to distinguish between read and write methods must be performed at run time.

This can be implemented by inserting extra code into the methods of cloud objects (or by generating method wrappers) in which the current execution mode is checked. When a read-only method is first invoked on a cloud, it starts by registering (e.g., in the RepMI runtime system) that it is running in *read* mode. This execution mode is always checked before a write method is executed, allowing the write method to be aborted and restarted in write mode.

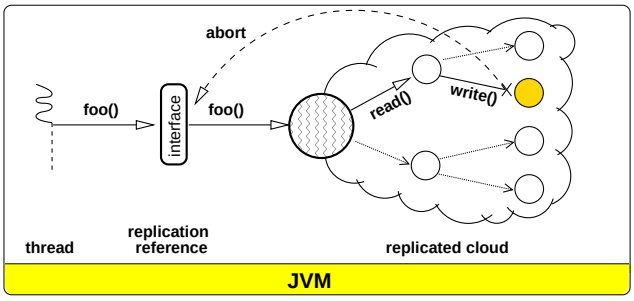

Figure 4.9: The aborting write methods when in read mode.

An example is shown in Figure 4.9, where the invocation of a read-only method on the root object leads to the invocation of a write method on a node object. This

write method is aborted before any data in the node object has changed. It can then be restarted as a write method.

The restart of this method can be performed safely, because so far only read operations have been executed, and the state of the objects in the cloud has not changed. To abort the current invocation, a special exception can be thrown. This exception must be caught in the RepMI runtime system or in the generated replication code. As Figure 4.9 shows, it is possible to restart nested invocations by using this exception mechanism. No extra nesting information has to be kept, other than the information that is already present for normal exception handling.

The read/write analysis in the compiler may be implemented conservatively by always classifying methods that contain assignments to object fields as write methods, even if the assignments may only be executed conditionally. However, it is also possible to insert execution mode checks into the different conditional blocks in the method, so that the method is only executed as a write method when it is absolutely necessary. This non-conservative approach allows the RepMI runtime system to do a runtime analysis of the behavior of methods invoked on the root object. For example, if a method is a potential write method (e.g., it contains an assignment in a conditional block), the RepMI runtime system may decide to execute the method in read mode, because the method's history shows that it is hardly ever aborted (i.e., the conditional block with assignment is never executed).

A compile time optimization can be done when all the classes used in a replicated data structure are marked as *final*. This keyword implies that it is not possible to extend these classes with a subclass. Therefore, methods can never be redefined, allowing the compiler to do a complete read/write analysis of all methods used in the data structure. If a conservative read/write classification is used, the results can be propagated all the way back to the methods in the root object. It is then no longer necessary to insert extra checks into the method of the node objects, thereby reducing the overhead.

Even if the data structures are not marked as *final*, this optimization can still be partly applied. If the compiler can determine that no nested invocations can take place in a read method, it is not necessary to include the checks in the generated replication code for the method, thus reducing the overhead.

Communication, synchronization and thread scheduling

Up until now, we have mostly concentrated on how the behavior of cloud objects must be restricted to ensure the consistency of the replicas. This consistency, however, also requires a correct implementation of the RepMI runtime system. Above, we have already shown that the support of the runtime system (and generated code) is required to implement read/write analysis. Two other aspects of the runtime system that have an impact on replica consistency are write method forwarding and synchronization, which we will now explain in more detail.

RepMI uses a function shipping approach to keep the replicas of a cloud consistent. Method invocations are forwarded to all machines containing a replica, and then

applied to the local root object. To keep the replicas consistent, it is important that all replicas apply these method invocations in exactly the same order.

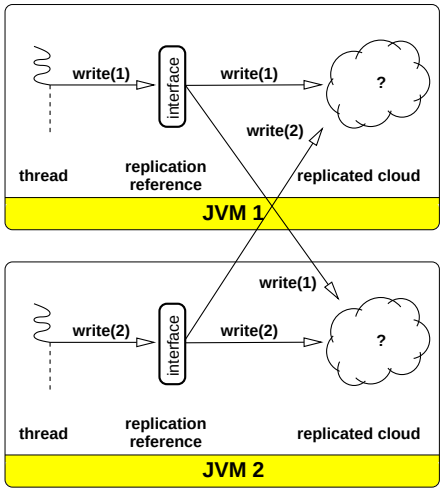

Figure 4.10: Concurrent write methods.

An example is shown in Figure 4.10, where a write method is simultaneously invoked on two replication references that refer to the same cloud. If these write methods are naively forwarded to both replicas, there is no guarantee they will be received in the same order. For example, it is likely that the replica on JVM 1 will first receive the *write(1)* invocation, while the replica on JVM 2 first receives the *write(2)* invocation. Thus, by applying the method invocations in the order in which they are received, the replicas could become inconsistent.

To solve this problem, the method invocations are forwarded using *totally-ordered* broadcast communication [30]. This form of communication guarantees that all messages are delivered in the same order to all receivers. Thus, in the example of Figure 4.10, both replicas receive either *write(1)* followed by *write(2)* or they both receive *write(2)* followed by *write(1)*. As a result, the methods can be applied to both replicas in the same order.

An important thing to note here is that using reliable, totally-ordered broadcast communication does still not guarantee that the replicas are kept consistent. It is important that the runtime system ensures that the ordering imposed on the methods by the communication is preserved throughout the method's lifetimes. For example, if the RepMI runtime system would start a new thread to handle every incoming write method, the total execution order introduced by the communication would immediately be lost again, since these threads may run in any order.

Another option is to use a single thread, that consecutively processes all write methods. When it receives a message, it unpacks the method, and applies it on the root object. Only after this method has run to completion, will the thread be ready for the next message. Unfortunately, this single-threaded scheme cannot be used for executing write methods that may block by calling *wait*. In this case, no other write methods will be able to run, including the one intended to wake up the blocked method.

```
class Bin extends ... implements ... {
   boolean filled = false;
   int value;

   public synchronized int get() throws ... {
      while (!filled) wait();
      filled = false;
      notifyAll();
      return value;
   }
   public synchronized void put(int i) throws ... {
      while (filled) wait();
      value = i;
      filled = true;
      notifyAll();
   }
}
```

Figure 4.11: Pseudocode for a replicated *Bin* object

This problem is illustrated in Figure 4.11, which presents a *Bin* object, a simple bounded buffer with a single data slot. The *get* method will block until a value has been written into the bin, then it empties the bin, and wakes up other, waiting, methods. The *put* method will block until the bin is empty, it will fill the bin, and then wake up waiting methods. Both *put* and *get* are write methods (they change *filled* and call *notifyAll*), and are therefore broadcast to all replicas. If a *get* calls *wait* because the *Bin* object is empty, the thread serving the write methods would block and the *put* that was intended to wake up the *get* would never be executed.

This problem can be solved by using a solution similar to the *Weaver* abstraction introduced in [95]. Whenever a thread handling a write methods blocks, a new thread is created to handle the next write method. Although this ensures that the RepMI runtime system will never deadlock, it re-introduces the previous problem. When the blocked threads wake up, there are multiple threads executing write methods, which may run in any order. Therefore, the RepMI runtime system must ensure that there is always exactly one such thread running, and, if multiple runnable threads exist, that the same thread (i.e., handling the same write method) is selected on all machines. Only then will the total execution order for write methods be guaranteed.

Unfortunately, selecting the same thread to run on all machines is a non-trivial task. A Java application does not have any control over the order in which the threads are scheduled. Also, the *wait, notify* and *notifyAll* methods that are used to implement the synchronization can not be redefined (they are *final*). We therefore need help from the compiler, which must rewrite these calls to use replication-aware alternatives that can be implemented in *CloudObject*. The following algorithm can then be used to ensure that the method invocations are handled in exactly the same way on all machines.

When the RepMI runtime system receives a method invocation, it is given a sequence number and a new thread is created to handle its invocation. This thread is not started immediately, but instead it is marked as being *runnable* and inserted into a queue. The RepMI runtime system will then select the runnable thread with the lowest sequence number from the queue, and allow only this thread to run. The thread will either run until the method is completed (after which the thread is destroyed and a new one selected), or until the method blocks using a call to a replication-aware alternative to *wait*. This alternative *wait* method will change the status of the thread from runnable to *blocked* and signal the RepMI runtime system that it must select another thread from the queue to run. The RepMI runtime system will then again select the runnable thread with the lowest sequence number. A method may use a replication-aware alternative to *notify* and *notifyAll* to unblock other threads. These threads will not be allowed to run immediately, instead their status will be changed from blocked to runnable. The alternative to the *notifyAll* method will change the state of all threads waiting for some condition, while the alternative *notify* only changes the state of the waiting thread with the lowest sequence number. The notified threads will eventually be selected to run once they become the runnable thread with the lowest sequence number.

The approach described here will ensure that there is always exactly one thread running a write method invocation and that all machines will schedule all threads in exactly the same order. Note that this is not necessarily the most efficient approach. Starting a new thread for every method invocation will introduce a significant amount of overhead. This can be prevented by reusing threads as much as possible. For example, if a method runs until completion and there are no runnable threads in the queue, the current thread can be used to handle the next method invocation.

Only the execution of write methods is handled by the RepMI runtime system using the approach described above. Since read methods are executed on a single machine, no ordering is required. Read methods are directly executed by the application threads. This makes it possible that a write method is running concurrently with one or more read methods, all accessing the same replica. Therefore, just like in RMI or multi-threaded Java applications, RepMI requires that the methods used in the replicated cloud are properly synchronized.

Consistency model

When the write method invocations are correctly handled by the RepMI runtime system and the methods in the replicated cloud are correctly synchronized, RepMI will provide the programmer with a model that is *sequentially consistent*. Sequential consistency was first introduced by Lamport [57] and states:

The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Although for RepMI we must replace the words *operations* by *methods* and *processors* by *threads*, the model still fits into this definition. Methods that are invoked on a replicated cloud are always *synchronous*. After the invocation, the invoking thread must wait for a result before it can continue with the next method. This satisfies the second half of the definition. By using the totally ordered broadcast communication and replication-aware thread scheduling, we convert concurrent write operations into a sequence of write operations. This sequence is the same on all replicas, thereby satisfying the first half of the definition.

Using multiple clouds and mixing RepMI with RMI

Our description of consistency in RepMI so far has focused on the problems relating to a single replicated cloud. However, in RepMI, it is perfectly legal to use multiple independent replicated clouds. It is also allowed to mix RepMI clouds with remote objects of RMI. Both may lead to consistency problems, which must be solved by the RepMI runtime system.

We have already shown that the write methods on a single replicated cloud must be ordered. We will now show, however, that *all* write methods must be ordered, independent of their target. An example is shown in Figure 4.12 where two independent threads on two different machines make use of two replicated clouds. The thread on JVM 2 first writes a value into cloud 1, followed by writing a value into cloud 2. At the same time, the thread on JVM 1 first reads from cloud 1, then from cloud 2. As the example shows, if the write methods on clouds 1 and 2 are not ordered, it may be possible for the write method on cloud 2 to overtake the write method on cloud 1. This problem can occur when the network messages are delivered in the wrong order or when the RepMI runtime system scheduled the write operations on cloud 1 and 2 independently of each other. As a result, the thread on JVM 1 will read old data from cloud 1 and new data from cloud 2. This clearly violates the second constraint of the sequential consistency definition.

To solve this problem, the totally ordered broadcast communication must be used to order *all* write methods invoked on *all* replicated clouds. It is not enough to only order the broadcasts for each cloud separately. The same applies to the thread schedul-

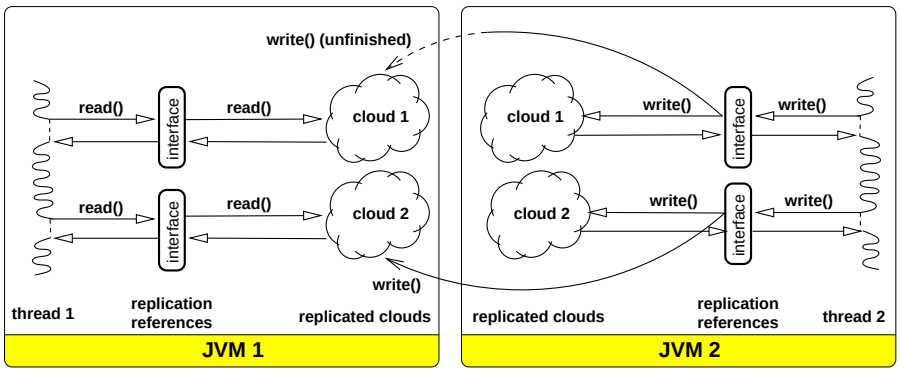

Figure 4.12: Incorrect message ordering when using multiple clouds.

ing in the RepMI runtime system. All threads executing write operations, regardless of the target cloud must run in the same order on all machines. Also, there may only be one such thread running on a machine at any moment in time.

A similar problem is introduced when a combination of RepMI and RMI is used. Figure 4.13 shows an example, where a thread on JVM 1 invokes a write method on a cloud, followed by a method on a remote object on JVM 2. As the example shows, the remote method running on JVM 2 may use the data of the local cloud replica. Unfortunately, it is possible that the RMI message overtakes the message that contains the RepMI method. Thus, when the remote method reads data from the replica, it may not be updated yet, even though the RMI was invoked *after* the RepMI write method.

This problem can be solved by ensuring that the remote method on JVM 2 is not executed before the write method of JVM 1 has been handled. The solution for the previous example required a total order on all write methods. The sequence numbers of these method can therefore be used as a “global clock”. When the RMI is forwarded from JVM 1 to JVM 2, the RMI runtime system must include a number that indicates the “time” on JVM 1. Execution of the remote method on JVM 2 must be stalled until the same “time” has been reached. The same ordering must be done for the result value of the RMI. Obviously, this does require a change in the RMI runtime system. This problem was described in detail in [30].

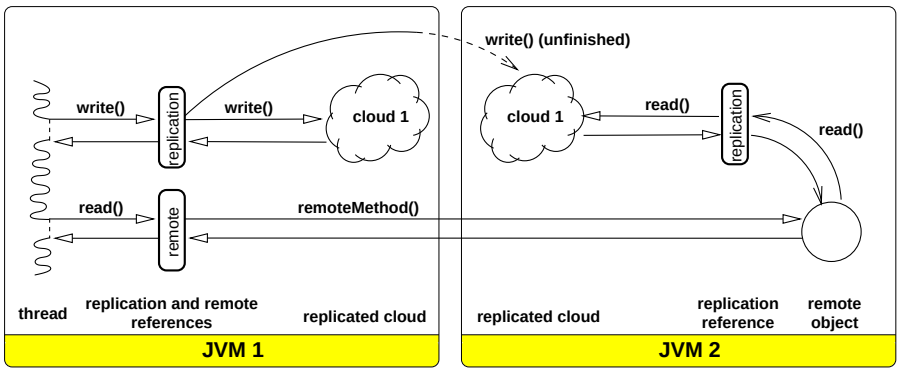

Figure 4.13: Incorrect message ordering when mixing RepMI with RMI.

Summary

To summarize, our model deliberately forbids references between different clouds and between clouds and remote objects. Every object inside the cloud extends *CloudObject*, and extends an interface that uniquely identifies it as being either a root or a node object. A cloud will always contain one root object. It may contain any number of node objects. Method invocations on the cloud will be forwarded to the root object using call-by-value semantics. If the method is a write method, it will be forwarded to all replicas, while read-only methods are only forwarded to the local replica. To ensure that all cloud objects behave correctly, the compiler will check that cloud objects only use the following types:

- primitive
- *java.lang.String*
- any object type that extends *manta.repmi.CloudObject*
- any interface type that extends *manta.repmi.Node*
- (possibly multidimensional) arrays of any of these types

Cloud objects may not contain or use:

- static fields or methods

- native methods
- threads

As a result of these restrictions, a cloud is a closed group of objects that can be replicated efficiently by broadcasting methods that change the cloud. By using reliable, totally-ordered broadcast communication and replication-aware thread scheduling, the RepMI runtime system ensures that the replicas remain consistent.

By increasing the complexity of the runtime system, it would be possible to lift some of the restrictions on cloud objects. For example, references between clouds and remote objects can be allowed if the system can detect that a remote method is invoked from inside a cloud. Only one of the remote invocations would then have to be executed, provided that the result is passed to all replicas of the cloud. A similar solution could be used to implement references between clouds. However, we found that our current replication model was flexible enough for the applications that we investigated. Therefore, we have not tried to lift any of these restrictions.

4.3 Implementation

We have implemented a prototype for the RepMI model using the Manta platform. This implementation consists of compiler-generated code and support in Manta's runtime system. We will now briefly describe this implementation.

We have adapted Manta's compiler to recognize classes implementing the *manta.repmi.Root* and *manta.repmi.Node* interfaces. It checks the restrictions on these classes, analyzes the methods to distinguish between read and write methods, and generates stubs and method wrappers.

The function of RepMI stubs is similar to that of RMI stubs. They implement the interface of the root object and serve as a replication reference. It is the stub's responsibility to forward write methods to all replicas. Like in RMI, Manta does not use any skeletons for RepMI. Instead, the code for receiving write method invocations is inserted directly into the root object.

The compiler generates method wrappers for all methods of root and node objects. These wrappers contain communication code (for write methods), code to copy parameters and return values (for read methods), and code to check the execution mode and perform write method restarts when necessary (for both read and write methods). After execution mode checking and copying, read methods are invoked on the local replica from within the method wrapper.

To forward a write method, Manta broadcasts a call header and the parameters to all replicas. The broadcast mechanism we use is part of the underlying Panda layer [6], which handles all communication between machines. Panda's broadcast is totally ordered and reliable, so all machines receive all messages and receive them in the same order.

The RepMI runtime system of Manta then handles the messages as described in Section 4.2.2. Only one write method is executed at a time, and when multiple blocking write methods are used, the threads are scheduled in the same order on all machines. Manta reduces the thread creation overhead by only creating new threads when absolutely necessary.

For transferring parameter objects, Manta serialization is used. As described in the previous chapter, this serialization code is generated by the Manta compiler and is highly efficient.

Whenever an application creates a new cloud, a unique identifier (i.e., the combination of object pointer and machine number) is assigned to it. The new cloud is immediately created on all participating machines by forwarding the *Cloud.create* call using Panda's totally ordered broadcast mechanism. This ensures that clouds are always created and initialized on all machines before any write operation attempts to modify them. Although the replicates of a cloud are immediately established on all machines, the application views them as being replicated on demand. Only the thread that created the cloud immediately receives a reference to it. By forwarding this reference (e.g., using an RMI) or by using *Cloud.lookup* calls, threads on other machines can also obtain a reference to it.

An optimization of this scheme would be to only replicate a cloud on those machines that actually contain a reference to it, and remove a replica when the last reference on a machine is removed. This would avoid memory and processing overhead caused by unused replicas. As a drawback, elaborate replica management would have to be implemented. Our current implementation assumes that the number of clouds used in an application will be small and simply replicates each cloud on all machines.

4.4 Performance evaluation

To evaluate the performance of Manta's RepMI implementation, we use the same experimentation platform as in the previous chapters, the *DAS*. Manta's runtime system has access to the network in user space via the Panda communication library described in Section 1.2

As in previous chapters, we will use both micro benchmarks and applications in our performance evaluation. To evalute whether RepMI also reduces the complexity of writing parallel applications, we will also compare the code sizes of the different versions of the applications.

4.4.1 Mirco benchmarks

We start our performance evalution of RepMI by comparing the overhead of RepMI read and write methods to similar method invocations on normal objects, to RMIs to a remote object on the same machine, and to RMIs to a remote object on a different machine. The results are shown in Table 4.1. All tests run on a single machine, except

the test which performs an RMI to a remote object (which requires two machines). The *write* method increases an *int* counter in the object. The *read* method returns the value of this counter.

invocation	*write*	*read*
normal	0.10	0.08
RMI, local	16	19
RMI, remote	45	46
RepMI	24	0.22

Table 4.1: Time required for method invocation (in μs).

As Table 4.1 shows, a normal method invocation takes about 0.1 μs. The numbers for RMI are much higher, even when the remote object is located at the same machine (RMI, local). With an object on the same machine, the RMIs take 16 and 19 μs, respectively.[5] RMI calls to objects on a remote machine take 45 and 46 μs, respectively. With replicated objects, read methods are performed locally and take 0.22 μs, independently of the number of replicas. A write method on a replicated object takes at least 24 μs, 8 μs more than a local RMI.

The RepMI numbers in Table 4.1 are minimal values. The overhead of a write method increases with the number of replicas, as we will explain in more detail below. The overhead for a read method depends on the complexity of the method. The Manta compiler analyzes the methods during compile time, and tries to generate the most efficient method wrappers. For example, the code for handling nested write method invocations in a read method (see Section 4.2.2) does not have to be generated if there are no invocations present in the method. Similarly, copying method parameters or result values is only necessary for reference types (i.e., objects or arrays). Primitive values (e.g., *ints* or *doubles*) can be directly forwarded or returned. In Table 4.2 we show a breakdown of the different sources of overhead in RepMI read methods.

source	*time*
Local read method	0.076
RepMI wrapper	0.148
Parameter copying	0.088
Result copying	0.064
Abort and restart support	0.278

Table 4.2: Execution cost of RepMI read method (in μs).

The table shows that executing the actual Java method (the *read* method described above) requires 76 ns. The generated RepMI wrapper adds 148 ns. This overhead can

[5] The read RMI is more expensive because it returns a value that has to be serialized.

be partly attributed to the extra indirection (the extra function call to the generated wrapper). The wrapper also contains a call to the RepMI runtime system to check if there are any queued write methods that must be handled first. Copying a single non-primitive parameter takes at least 88 ns, while 64 ns is needed to copy a non-primitive result. These two numbers show the minimal overhead introduced by the copying routines (i.e., the overhead when the references are *null*). When actual objects must be copied, the overhead will be significantly higher. Finally, adding support for restarting nested write method invocations introduces another 278 ns. This table shows that the minimal time required for a RepMI read method ranges from 0.22 μs for a simple method (without any copying or restart support) to 0.65 μs for a complex method (with copying and restart support).

We will now have a closer look at the performance of write operations in RepMI. As a benchmark we use a replicated cloud containing a single object. This cloud is replicated on one or more machines. We then measure the time required to forward a write operation to all replicas. The results are shown in Table 4.3.

When a write method is invoked on a replicated cloud, the thread invoking the method blocks only until the write method has been applied to the local replica and a result is returned. All other copies are updated asynchronously. As a result, the broadcast message may not even have arrived yet on all machines when the thread continues. This allows several write method invocations to be pipelined. Although this pipelining effect may be beneficial to some applications, synchronous applications (e.g., QR or LEQ) do no usually benefit.

Therefore, the RepMI column in Table 4.3 shows two results. The first number denotes the *latency*, i.e., the time required for a write method to reach each of the replicas. The second number denotes the *gap* time, i.e., the minimal time between two (pipelined) write method invocations by the same thread. The latency gives an indication of the overhead that RepMI causes in synchronous applications, while the gap time is a measure for the overhead in asynchronous applications.

The gap time can easily be determined by applying a large number of write method invocations on a cloud, waiting until all invocations have been applied to all replicas (e.g., by using a *barrier* operation after the last invocation), and then calculating the average time per invocation.

Determining the latency, however, requires elaborate measurement procedures [26, 79]. To measure the time from invoking a write method until the invocation reaches a certain replica, we perform the following test.[6] An object is replicated on several machines. A thread on a selected machine reads the local time t_s and performs an RMI call on a remote object on machine 0. This RMI method invokes a write method on the replicated object and returns. On every machine, this write method retrieves the local time t_e and stores it in the replica. After the RMI has returned, the selected machine reads the stored time from the local replica, takes the difference

[6] This test breaks one of the restrictions described in Section 4.2.2 since it reads the local time. Therefore, some compiler checks must be disabled to compile this test.

$t_e - t_s$, and subtracts 1/2 of the time required for an RMI call. The result is the latency required to reach the replica on this machine. This test is repeated for each replica. The maximum of all values measured in this way is the broadcast latency.

	RepMI			*Panda*			*RMI naive push*	*RMI naive pull*	*RMI binomial tree*		
machines	*lat*	/	*gap*	*lat*	/	*gap*	*lat*	*lat*	*lat*	/	*gap*
1	31	/	31	-	/	4	-	-	-	/	-
2	60	/	60	45.0	/	28	56	54	58	/	58
4	65	/	63	51.9	/	33	160	72	131	/	116
8	78	/	63	64.3	/	33	372	154	285	/	204
16	93	/	62	75.5	/	34	800	320	489	/	314
32	111	/	63	90.7	/	35	1674	638	757	/	467
64	155	/	65	104.5	/	37	3336	1384	1076	/	622

Table 4.3: Broadcast latency and gap times on a Myrinet cluster (in μs).

As Table 4.3 shows, invoking a write method on a single replica takes 31 μs. This value is higher than the 24 μs shown in Table 4.1 because the write method used in this benchmark is significantly more complex. Not only must it retrieve the local time, but it must also signal a read method that a write has been received (this requires a lock to be taken). To update two replicas, 60 μs are required, approximately the same amount of time needed to do one local and one remote RMI. However, as more replicas are added, the communication cost only increases slowly. Updating four replicas instead of two adds just 5 μs. To update 64 replicas, 155 μs is required, roughly the time needed to do four RMIs. If we look at the gap time, we see that on two or more machines it remains stable at approximately 63 μs. This shows that the cost of a write method on the invoking machine is independent of the number replicas.

For comparison, the table also shows a *Panda* column which contains the latencies and gap times for the *totally-ordered* broadcast primitive offered by the Panda library. This primitive is used to implement RepMI. Panda uses the low-level broadcast offered by LFC, which uses an efficient spanning-tree protocol running on the Myrinet network interfaces. Panda extends the LFC broadcast with message ordering and also delivers the message on the sending machine.

As the table shows, RepMI adds an average overhead of 22 μs to the Panda broadcast latency. RepMI's gap time is 29 μs higher than that of Panda. This overhead is caused by the way RepMI handles local write operations. Although the Panda broadcast operation may return before the message is delivered to all machines, the RepMI runtime system will block the sending application thread until the message is received locally. A runtime system thread then applies the write invocation to the local replica and returns the result value to the waiting application thread. Only after this result has been delivered will the application thread be able to continue. This local message handling, which includes two thread switches, adds an overhead of 29 μs to Panda's gap time. The overhead added to Panda's broadcast latency is slightly lower because the

latency time is measured inside the write operation itself. As a result, the time needed to receive the message, to switch to the runtime system thread, and to invoke the Java method are included in this measurement. However, the time required to switch back to the application thread is not included. The gap time measurement does include this overhead.

To illustrate the performance gain that RepMI offers over RMI, we have added the results of three RMI broadcast simulations to the table. These simulated broadcasts vary in implementation complexity and performance. Note that these implementations do not use an RMI to forward data to the local machine. Therefore, there are no latency numbers for broadcasting to a single (i.e., the same) machine.

The *RMI naive push* implementation simulates a broadcast by forwarding data using a series of RMIs, one to each machine. In this naive implementation, every additional machine adds an extra RMI round trip latency. As a result, the total time required to reach all machines grows linearly, to 3336 μs on 64 machines. The *RMI naive pull* implementation uses the opposite approach. Data is stored in a single remote object on the sending machine. All other machines retrieve this data by performing an RMI to this object. Although this implementation allows a moderate amount of communication overlap (i.e., several RMIs are on the network simultaneously), the amount of contention on the remote object grows with the number of machines. This results in a latency of 1384 μs to reach 64 machines. The *RMI binomial tree* is an optimized RMI broadcast simulation. It uses a binomial-tree algorithm to forward the data, allowing communication to be performed in parallel in different branches of the tree. Not only does this reduce the latency of the simulated broadcast, but it allows a moderate amount of pipelining to occur. On 64 machines, this implementation has the lowest latency, 1076 μs. The gap time of 622 μs shows that the overhead on the sending machine is significantly lower than that of the other RMI broadcast simulations. However, it is still a factor of ten higher than RepMI's gap time.

We have also performed throughput measurements on RepMI. By measuring the time it takes to apply a write method with a 100 KByte *byte* array as a parameter, 1000 times, we can determine the effective throughput. The results are shown in Table 4.4, like in the latency test, we have also added numbers for similar benchmarks running on Panda and for the three broadcast simulations using RMI.

For Panda, we have implemented two different versions of the benchmark, *Static Panda* and *Dynamic Panda*. The first benchmark is optimized to show the full performance of Panda communication. No memory allocation or thread switches are done during the measurements. The second benchmark is designed to simulate the behavior of Java applications more closely. In this benchmark, the memory required to receive the messages is allocated dynamically, when the message arrives. To simulate Java's garbage collection, a new block of memory is allocated for every message. All allocated blocks are freed simultaneously when the total amount of memory exceeds a certain size. In the second benchmark, the messages are also handled by a separate communication thread instead of the application thread. Because the *Dynamic*

Panda benchmark includes both memory allocation and thread switching overhead, it gives and indication of the maximum performance which can be obtained by the Java benchmarks.

machines	*RepMI*	*Static Panda*	*Dynamic Panda*	*RMI naive push*	*RMI naive pull*	*RMI binomial tree*
1	31.5	333	60.6	48.1	44.4	49.8
2	19.0	56.6	22.8	20.1	21.6	24.5
4	18.7	39.0	21.5	8.2	14.0	12.7
8	17.9	30.7	20.1	3.9	7.5	7.5
16	17.9	22.0	17.9	1.9	3.8	5.0
32	16.4	22.4	17.0	0.9	2.0	3.4
64	11.1	13.3	11.5	0.5	1.0	2.5

Table 4.4: Broadcast throughput on a Myrinet cluster (in MByte/s).

When only one machine is used, Panda does not transfer any data. Instead, the data is directly copied from the send to the receive buffer. The performance of such an operation is mainly determined by the performance of the cache. The *Static Panda* benchmark, which uses a single pre-allocated buffer, can take full advantage of the cache performance. This results in a high throughput of 333 MByte/s. In the *Dynamic Panda* benchmark, however, the data is always copied to a freshly allocated piece of memory, resulting in cache misses during copying. As a result, the throughput is limited by the memory bandwidth, 61 MByte/s.

Like Panda, the RMI broadcast simulations do not use any communication to deliver the message to the sending machine, but use *System.arraycopy* to copy the data to a newly created array. The extra overhead of array creation reduces the throughput of the RMI implementations to 45-50 MByte/s.

RepMI also suffers from this array creation overhead. In addition, RepMI uses synchronous communication on the sending machine. The sending application thread is not allowed to continue until the message is handled locally, and a result is returned. As a result, the RepMI throughput on one machine is about 50% of that obtained by the *Dynamic Panda* benchmark, 31 MByte/s.

On two machines, the throughput achieved by *Dynamic Panda*, RepMI and RMI are comparable, approximately 20 Mbyte/s. Only the *Static Panda* version achieves throughput which is higher, 57 MByte/s. This shows that the extra overhead of RepMI on the sending machine hardly effects the performance when two machines are used. On more than two machines, the relative overhead of RepMI decreases even further. The RepMI throughput on 64 machines is only slightly lower than the *Dynamic Panda* throughput at 11 MByte/s. The *Static Panda* performance on 64 machines is not much higher, 13 MByte/s.

On more than two machines, the RMI broadcast implementations do not perform well. Their throughput drops rapidly to a maximum of 3 MByte/s for the binomial tree

implementation. This low throughput can be attributed to an inefficient implementation (in the naive RMI versions) and high communication and serialization overhead (in all versions). While the RepMI version serializes the data once, the RMI versions must serialize the data once *for every destination*. Also, the RMI versions must use synchronous remote invocations, while the low-level broadcast implementation can use more efficient asynchronous messages.

These numbers clearly show the benefit of using a communication model that allows the use of an efficient low-level broadcast implementation. On 64 machines, the throughput of RepMI is close to the performance offered by Panda, while the RMI implementations clearly suffer from communication and serialization overhead.

4.4.2 Applications

In this section we will use five application kernels to evaluate our RepMI implementation further. These kernels, ASP, TSP, ACP, QR and LEQ, were also used to evaluate the RMI performance in the previous two chapters. They are described in more detail in Sections 2.5.3 and 3.5.1.

For all applications, we use the Manta RMI results presented earlier. These applications are optimized to improve their performance. Some, for example, use a spanning tree protocol to simulate a broadcast. We will refer to these implementations as the *optimized* RMI versions. For each application, we have also implemented a new *naive* RMI version, where data is shared using remote objects without taking locality into account. The *replicated* versions of the applications were created by adapting these *naive* versions so that they replicate their shared objects. This approach allows us to compare both the speedup and code complexity of the three versions. The results are shown in Figure 4.14. All speedup values are computed relative to the speed of the version running the fastest on a single machine. As a result, the speedups shown for the optimized RMI versions may differ slightly from the speedups shown in the previous chapter. We will now present performance numbers for each of the applications and compare the implementation of each of the versions.

TSP keeps track of the best solution found so far, and uses this information to prune part of the search space. Our implementations of TSP store the current solution in an object of class *Minimum*. We have implemented three different versions of this *Minimum* class. The TSP graph of Figure 4.14 shows the performance. All speedups are computed relative to the optimized RMI version on one machine, which runs for 460 seconds.

The naive RMI version uses a single remote object shared by all machines. This is a straightforward implementation, which, unfortunately, does not perform well, since an expensive RMI is needed whenever a machine wants to read the latest minimum value. As a result, the naive RMI version has no speedup at all.

The optimized RMI version was also used in the previous chapters. In this version, each machine contains its own *Minimum* object. Each of these objects contains a

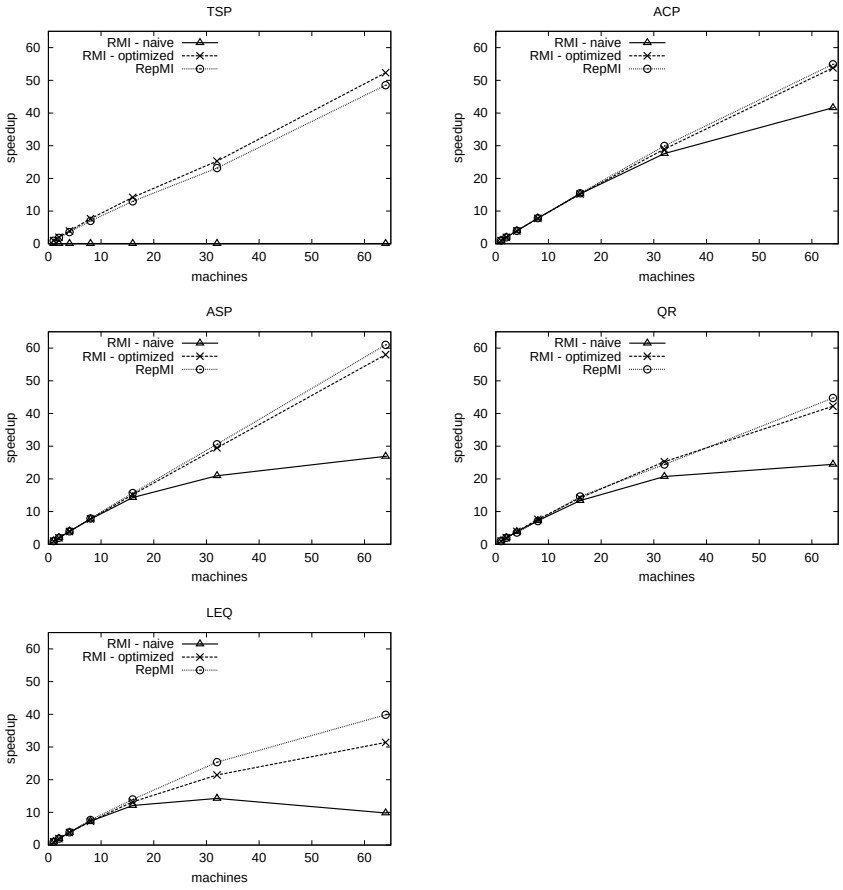

Figure 4.14: Application speedups.

vector of references to all other minimum objects. Whenever a new solution is stored in a *Minimum* object, it is forwarded to all others by sequentially invoking an RMI on each of them. To read the latest minimum value, a normal method invocation on the local *Minimum* object is used. As we have already shown in the previous chapter, the optimized RMI version achieves a speedup of 52 on 64 machines.

The replicated version of TSP is almost identical to the naive RMI version. The only difference is that the *Minimum* class is marked as being a replicated object instead of a remote object. The replicated *Minimum* class is shown in Figure 4.15. A more detailed analysis of the code sizes of the applications will be given in Section 4.4.3.

```
interface i_Minimum extends manta.repmi.Root {
   public void set(int min) throws ...;
   public int get() throws ...;
}

class Minimum extends manta.repmi.CloudObject implements i_Minimum {
   private int minimum = Integer.MAX_VALUE;

   public void set(int min) throws ... {
      if (min < minimum) minimum = min;
   }
   public int get() throws ... {
      return minimum;
   }
}
```

Figure 4.15: RepMI implementation of the *Minimum* class.

The performance of the replicated implementation of TSP is almost identical to the optimized RMI version. Because the *Minimum* object is replicated on all machines, all *set* method invocations are automatically forwarded, while each *get* method invocation is handled locally. Figure 4.14 shows that the replicated TSP version obtains a speedup of 48 on 64 machines. The performance difference between the optimized and replicated versions originates in the overhead of reading the minimum value from the replicated object. As a result, the replicated version is approximately 10% slower than the optimized version, independent of the number of machines used.

In **ACP** a matrix of boolean values is shared between the machines. The ACP graph in Figure 4.14 shows the results for the three versions. All speedup values are computed relative to the naive version on one machine, which runs for 533 seconds.

The naive RMI implementation of ACP uses a single remote object to store the shared boolean matrix. Each machine retrieves a copy of the data when it needs it, and sends all the updates to the matrix using a single RMI. This approach limits the amount of communication, resulting in a reasonable speedup of 42 on 64 machines.

In the optimized RMI, implementation each machine contains a private copy of the boolean matrix. A spanning tree broadcast is used to forward updates to all other machines. An extra thread is used on every machine to apply these updates to the local copy of the boolean matrix. Because the local copy of the matrix is now always up-to-date, it saves the time needed in the naive RMI version to make a copy. This improves the speedup on 64 machines to 53.

The replicated version uses a replicated object to store the shared boolean matrix. This object is almost identical to the remote object used in the naive RMI version, except that it is replicated. This allows the object to use the efficient low-level broadcast, improving the speedup even further to 55 on 64 machines.

In **ASP**, one machine broadcasts a row of data at the beginning of each iteration. In the naive RMI version, we use a single remote object to store this row of data, allowing all machines to read it. When a machine requests a row which has not been produced yet, the RMI will block until the data is available. Because each machine has to fetch each row for itself, each row has to be sent across the network multiple times, causing high overhead on the machine that owns the row. For instance, if 64 machines are used, each row is sent 63 times.

The ASP graph in Figure 4.14 shows the results for a 2000x2000 distance matrix. Note that this is a larger data set than we used the previous chapters. All speedup values are computed relative to the naive version on one machine, which runs for 1067 seconds. The naive RMI version performs well up to 8 machines. On more machines, the overhead for sending the rows becomes prohibitive, limiting the speedup to 27 on 64 machines. As we already described in the previous chapters, the optimized RMI version of ASP uses a spanning tree to simulate the broadcast of a row, resulting in a much better speedup of 58 on 64 machines.

The replicated ASP implementation uses a single, replicated *Broadcast* object, shown in Figure 4.16. Whenever a machine stores a row into this object, it is forwarded to all replicas using the efficient broadcast protocol provided by Panda. Each machine can then retrieve this row locally. As with TSP, the replicated version of ASP is as simple as the naive version. Figure 4.14 shows that it performs even better than the manually optimized RMI version, achieving a speedup of 61 on 64 machines. This is due to the low-level broadcast which offers a higher throughput than the broadcast simulation using RMI. By using a low-level implementation, parameter objects only have to be serialized once per broadcast, rather than multiple times (as is required when using an RMI broadcast tree).

Despite the good performance of the replicated version of ASP, this application also illustrates a problem with using RepMI to broadcast data in asynchronous applications. As Figure 4.16 shows, data is stored in the replicated *Broadcast* object, but never removed. The *retrieve* method returns a copy of the stored data, but it can not remove it, even when it is read by all machines and is no longer needed. Any code in the *retrieve* method which alters the state of the object (e.g., registering the number of times the data is read) would immediately turn it into a *write* method. It can then no longer be executed locally, but must be broadcast instead, thereby causing significant overhead. This shows that the consistency model of RepMI is too strict for some applications. Fortunately, the amount of data broadcast in ASP is small enough to fit into memory, even if unused data is not removed. However, for applications with larger data sets, this may become a problem.

Communication in **QR** is more complex than in the previous applications. All machines participate in a collective reduce-to-all operation, followed by a broadcast performed by a single machine. The QR graph in Figure 4.14 shows the results for a 2000x2000 matrix. The speedups are computed relative to the replicated version on one machine, which runs for 2635 seconds.

```
class i_Broadcast extends manta.repmi.Root {
   public int [] retrieve(int i) throws  ...;
   public void store(int i, int [] row) throws ...;
}

class Broadcast extends CloudObject implements i_Broadcast {
   private int[][] tab;
   private int size;

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

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

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

Figure 4.16: Replicated implementation of the *Broadcast* class.

In the naive RMI version, both the broadcast and reduce-to-all operations are implemented by a single remote object. The broadcast implementation is similar to the one used in the naive version of ASP. For the reduce-to-all operation, each machine submits a value to the central using an RMI. The RMI blocks until all values are submitted, after which the result of the reduction is returned to all machines. The naive implementations of both broadcast and reduce-to-all have an impact on the performance of QR, producing a poor speedup of 24 on 64 machines.

In the optimized RMI version, as with ASP, the broadcast objects are replaced by a spanning tree protocol, while the replicated version uses a replicated object. Unfortunately, a complex collective operation as reduce-to-all can not be expressed efficiently using replicated objects. Therefore both the optimized RMI replicated versions of QR use the same *binomial-tree algorithm* to implement this operation. In this reduce-to-all algorithm, each machine contains a leaf node of a binomial tree. A value is submitted

to this leaf node and sent towards the root of the tree. A reduce between two values is done at every intermediate level. This causes a single value to arrive in the root of the tree, which can then be sent back down to the machines.

Replacing the broadcast and reduce objects improves the speedup of QR on 64 machines to 42 in the optimized version, and to 45 in the replicated version. As with ASP, this difference is caused by the low-level broadcast mechanism used by the replication system.

Like QR, **LEQ** uses complex collective communication operations. Not only does it require a reduce-to-all operation for termination detection, but it also uses a gather-to-all operation to reconstruct the data after every iteration. This communication pattern causes the machines to synchronize at each iteration. The LEQ graph in Figure 4.14 shows the LEQ results for a 1000x1000 matrix. All speedup values are computed relative to the replicated version on one machine, which runs for 1610 seconds.

In the naive RMI version, both the gather-to-all and reduce-to-all operations are implemented by a single remote object. After an iteration, each machine sends its part of the solution vector and the value to be reduced to this object and waits for the result of the reduce-to-all. The remote object then assembles the entire solution vector and reduces all the values to a single result. If required, all machines retrieve a copy of the solution vector to use in the next iteration. The limited performance of both the reduce and gather implementation results in a speedup of 8 on 64 machines.

In the optimized RMI version, the vector fragments and values to be reduced are broadcast using a spanning tree, similar to the one introduced in ASP. Each machine can then locally assemble the vector and reduce the values. Unlike in previous applications, where one machine was broadcasting, in LEQ all machines are required to broadcast data. This requires a large number of RMIs to complete the communication, causing more overhead than in the previous programs. For example, on 64 machines, 4032 RMIs are needed per iteration, while ASP only needs 63 RMIs per iteration. Due to this overhead, the speedup of the optimized RMI version is only 31 on 64 machines.

As with the other applications the replicated version is similar to the naive RMI code, but uses a replicated object. The replicated object can profit from the efficient low-level broadcast used by the replication system. Instead of 4032 RMIs, only 64 broadcast messages are required to complete the communication. This results in a reasonable speedup of 40 on 64 machines.

4.4.3 Source Code Sizes

We have shown the performance of five applications, each implemented in three different ways. Although the naive versions of the applications usually have a poor performance compared to the other versions, they also have the simplest implementation. Shared data are identified by the programmer and encapsulated using a remote object without taking locality or RMI overhead into account. When we do take these

issues into account, as is done in the optimized versions, the code size of the implementation can increase significantly. Table 4.5 shows the code size of the applications relative to the naive version. These numbers are produced by removing the comments and whitespace from the program source, and then counting the number of characters required for the entire program. The weighed average number is calculated by adding the sizes of all naive implementations, all optimized implementations, and all replicated implementations, and comparing those to each other.

	version		
application	*naive*	*optimized*	*replicated*
TSP	100	106	100
ACP	100	114	100
ASP	100	168	98
QR	100	120	100
LEQ	100	135	97
weighed average	100	121	100

Table 4.5: Code sizes of the applications relative to naive version, in %.

The table shows that the optimized version of an application is always bigger than the naive version. The increase in size varies from 6% with TSP to 68% with ASP. On average the optimized implementation is 21% bigger that the naive implementation. The table also shows the advantage of our replication system. Although the replicated versions of the application usually have the best performance, their size is comparable to the naive versions. Of the five applications, three show no significant difference in code size, while the other two applications are even slightly smaller. On average, the implementation using replication has the same size as the naive implementation.

4.4.4 Discussion

In the five applications, replication is used for three different purposes: sharing data, broadcasting, and collective communication. Table 4.6 shows the communication patterns of the different applications.

TSP and ACP use replication to share data that are read very frequently, but written infrequently and at irregular intervals. RepMI is perfectly suited to handle this type of application, because it provides a simple way of expressing shared data. As Table 4.5 shows, the replicated versions of these applications show no increase in code size compared to the naive implementations.

ASP and QR use replication to implement a broadcast. By writing data into a replicated object it is effectively broadcast to all machines, which can then read the data locally. These applications mainly use replication to take advantage of the efficient low-level broadcast provided by the replication system. As shown in Table 4.5, implementing a broadcast using replication is much simpler than simulating a broad-

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

Table 4.6: Communication patterns of the applications.

cast using RMI. However, as we have explained above, replication is not necessarily the best model to express a broadcast. For example, because of the asynchronous nature of ASP, data can be stored in a replicated object, but never removed. Due to the strict consistency model of RepMI, it is not possible to implement the required administration code in the method which reads the data, without turning it into a write operation. Therefore, the broadcast object used in ASP stores all data it has ever received. In synchronous applications, like QR, it is simpler to implement a broadcast using RepMI, because it is implicitly known when the data may be removed. Note that QR does not use replication to implement the reduce-to-all communication. Instead, a more efficient binomial-tree algorithm is used.

LEQ uses a replicated object to simultaneously perform a gather-to-all and reduce-to-all operation. Although the replicated version of LEQ has a reasonable performance and is even slightly smaller (3%) than the naive version, replication is not be the most optimal way to express collective communication. Replication can only be used to express broadcast communication, while efficient reduce-to-all or gather-to-all implementations are often based on a combination of broadcast and binomial-tree or many-to-one communication.

In conclusion, the applications show that RepMI is an efficient mechanism to express shared data in Java, providing both high performance and a simple programming model. For all applications, RepMI shows a performance that is similar to, or better than the optimized RMI versions, while the code size is almost as small as the naive RMI versions. However, for applications that require broadcasting or collective communication instead of shared data, the consistency model of RepMI can be too strict. Also, RepMI is not necessarily the best model to efficiently express complex collective communication such as reduce-to-all or gather-to-all operations. In the next chapter, we will investigate how the RMI model can be extended to support such complex group communication.

4.5 Related work

In this section we will describe related work. The focus will be on work related to object replication. High-performance communication, parallel programming in Java

and other Java-related research projects have already been described in 3.6.

In RepMI, we use a function shipping approach to object replication that was inspired by the Orca system [6, 30]. The Manta platform, which we used to implement RepMI, is based on the same communication system as Orca (Panda). However, there are many important differences between Orca and RepMI. The Orca language was designed specifically to allow object replication. In particular, its object model is very simple. Orca is actually an *object-based* language instead of an *object-oriented* language. Large parts of the applications are written using a *procedural* programming style. Only shared data is expressed using objects. Orca does not allow references between objects and only supports method invocations on *single* objects. Orca programs read and write one object at a time, much like Distributed Shared Memory (DSM) programs read and write single memory locations one at a time. Synchronization in Orca is much more restrictive than in Java. It only allows methods to block initially, but not halfway during their invocation. Because Orca does not support polymorphism, the read/write analysis on methods can be performed entirely at compile time.

In contrast, Java uses a more advanced object-oriented programming model that was not designed with object replication in mind. Therefore, implementing replicated objects in Java is much harder. Unlike Orca, Java allows the use of references between objects. In RepMI, we therefore introduced a clustering concept to allow efficient replication of object graphs (clouds). We also described the restrictions that must be imposed on these replicated cloud objects to ensure that all replicas remain consistent.

In Orca, method invocations on objects are not allowed to block. Instead, special *guard* statements can be used to check for synchronization conditions before the method is really started. After the method starts, it will always run to completion. In constrast, RepMI allows method invocations on replicated objects to use the normal Java synchronization primitives. As a result, a method may block at any point during its execution. It may even block several times. By using special implementations of Java's *wait*, *notify*, and *notifyAll* primitives in combination with replication-aware thread scheduling, RepMI ensures that all replicas remain consistent. Since RepMI also supports polymorphism, the read/write analysis of methods can not entirely be performed at compile time. Instead, RepMI uses a combination of compile-time analysis and run-time checks. The runtime system aborts any methods that are incorrectly executed in read mode, and restarts them as write operations. This implementation also allows a more dynamic approach, where the runtime system decides whether a method is a read or a write method based on historic information.

An alternative to replication is to use a Distributed Shared Memory (DSM) system. These systems provide a shared-memory programming model, while still executing on a distributed-memory system. This allows object to be shared between threads running on different machines. It is up to the DSM system to ensure that these objects are kept consistent. Several Java-based DSM systems are described in Section 3.6.

Unlike Java-based DSM systems, RepMI is not completely transparent. This can be seen as a disadvantage, because some effort is required from the programmer to

mark objects as being replicated and cluster these objects into a cloud. However, the approach used in RepMI also has very clear advantages. In RepMI, it is completely clear which objects must be replicated. In a DSM, the system must figure out dynamically, during run time, how the objects must be distributed over the available machines and which objects may be replicated (if replication is supported at all). Most systems only support *caching*, where a machine can get a temporary read-only copy of an object. These copies are invalidated (deleted) whenever the object is changed, forcing machines to retrieve a new copy. This may lead to significant communication overhead when large objects or data structures are changed. In contrast, RepMI allows graphs of objects to be replicated as a whole and uses function shipping to apply the changes to all replicas. Therefore, forwarding a single method invocation is enough to change all objects in a data structure. This eliminates the need for further communication and reduces the communication overhead.

Some DSM systems try to reduce the communication overhead by applying advanced optimizations or use specialized programming models. *Jackal* [105, 106], for example, uses *object-graph aggregation* and *automatic computation migration*, to reduce the communication overhead. Using these optimizations, the system tries to reconstruct the object graphs during run time and uses function shipping to prevent unnecessary caching of objects. Although these optimizations do improve the performance of the Jackal system, application measurements show that its performance is, at best, equal to optimized RMI implementations, while we have shown that RepMI applications often outperform their optimized RMI counterparts.

The VJava [60] system offers caching using a scheme called ObjectViews. With ObjectViews, threads can have different *views* of a shared object. The system can determine at compile time if it is safe to access the object concurrently through two different views. This information is used to reduce the number of invalidation messages sent. Although this approach reduces the overhead of object caching, it also gives up the biggest advantage of DSM systems, transparency. The different views of an object have to be defined by the programmer using a special extension to the Java language.

The Javanaise system [41] uses groups of objects (called *clusters*) in a way similar to RepMI, but relies on object caching instead of replication. Machines can fetch read-only copies of a cluster from a centralized server. Those copies will be invalidated when a machine requests write permission on the cluster, causing considerable overhead with updating large clusters. In the clustering mechanism of Javanaise, a *cluster* object (corresponding to RepMI's *root* object) serves as the entry point to the cluster. Programmers have to annotate methods as read or write operations, a task performed automatically in RepMI. Finally, Javanaise has no notion of *node* objects and any serializable object can be part of a cluster. As a result, it is up to the programmer to ensure replica consistency.

Object replication is not only used to improve performance, but also for *fault-tolerance*. In systems like OGS [31, 32] and AQuA [89], for example, fault-tolerance

is added to CORBA [81] by replicating server objects on several machines. This ensures that a service is always available, even if one of the machines crashes. It also allows server objects to be shut down for maintenance or to be replaced by a new implementation without having to stop the service. Also, for security, a request can be forwarded to multiple replicas. Their replies can then be compared to ensure that it is valid.

It is hard to compare these systems to RepMI, because they have very different objectives. RepMI purely is designed to run on relatively small (and reliable) cluster computers, and therefore focuses on performance. Fault tolerance will become an issue if RepMI is used on a large scale distributed system. However, this is currently not the focus of our work.

4.6 Conclusions

In this chapter, we have presented RepMI, an efficient approach to object replication in Java. RepMI was designed to both improve the performance and reduce the complexity of parallel Java applications. We have kept the RepMI programming model close to that of RMI, to ensure that existing parallel RMI applications can easily be converted to use object replication.

In RepMI, we deliberately use a restricted model to obtain a clear programming model and allow a high performance implementation. Instead of separately replicating the objects of an arbitrarily complex graph, RepMI allows the creation of closed groups of objects (*clouds*), that are replicated as a whole. Each cloud has a single object as its entry point (the *root*), which is the only object on which methods may be invoked from outside of a cloud. Like in RMI, the method invocations on the root object of a cloud use call-by-value semantics.

Using a combination of compiler analysis and runtime system support, it can be determined which methods will only read the cloud and which methods may also modify it. A single broadcast message can then be used to forward method invocations that modify the cloud to all replicas. Method invocations that only read the cloud can directly be applied to the local replica and do not require any communication at all.

We have explained the restrictions that must be imposed on objects in a replicated cloud to ensure that all replicas remain consistent, and how these restrictions can be enforced during compile time or run time. We have also described the run-time support required for RepMI, such as the use of totally-ordered broadcast to forward write methods and the need for replication-aware thread scheduling.

Using the Manta platform as a basis, we have created a RepMI implementation, of which we evaluate the performance. Micro benchmarks show that simple read operations on a replicated cloud take about 0.22 μs, only twice the time required for a normal method invocation. Write operations take 155 μs to update 64 replicas, roughly the time needed to do four RMIs. Because RepMI allows multiple write operations to be pipelined, a new write operation can be performed every 63 μs, independent

of the number of replicas. These numbers show that, compared to RMI, RepMI can significantly reduce the communication overhead, especially for shared data with a high read/write ratio.

We further evaluated our system with five application kernels and showed that RepMI allows a simple and straight-forward implementation of shared-object applications, and often outperforms manually optimized versions based on RMI. However, we also showed that although RepMI is perfectly suited to express shared data, it is not necessarily the best model to express collective or group communication (even though the applications which need these forms of communication performed better using RepMI than using RMI). In the next chapter, we will look more closely at how we can extend the RMI model further to support complex group communication.

Chapter 5

Group Method Invocation

5.1 Introduction

In the previous chapters we have shown how Java RMI can be implemented efficiently and how Java can be extended with an object replication model, RepMI, that is both efficient and simple to use. Although an efficient RMI in combination with object replication greatly increases the suitability of Java for parallel programming, one piece of the puzzle is still missing, *group communication*.[1] RMI can only be used to express point-to-point communication, and object replication (or RepMI) is only suitable for expressing shared data. Although alternative forms of communication can be simulated using RMI or RepMI, this is often cumbersome and inefficient. For example, complex communication such as personalized broadcast can be simulated using either multiple RMI calls or a replicated object. The first approach will be cumbersome (i.e., it requires complex communication code), while the second approach will not provide optimal efficiency (i.e., the data is broadcast to all machines, while each machine only needs a subset). Also, for many applications, the consistency model of replicated objects is too strict.

What is needed for many parallel applications is the ability to combine several objects (distributed over several JVMs) into a *group*, and to communicate with this group as a whole, without the strict consistency requirements of RepMI. It must be possible to express complex communication, like personalized broadcast and data reduction, using a simple application programming interface (API). Like in RMI and RepMI, the complexity of the actual communication implementation must be hidden inside the runtime system and the compiler-generated code.

One approach to introduce group communication is to extend Java with support

[1] Please note that we use the term *group communication* to refer to arbitrary forms of communication with a group of objects; in the operating systems community, the term is often used to denote *multicast*, which we regard as just one specific form of group communication.

for collective communication through an external library such as MPI [19, 37, 38]. This increases expressiveness at the cost of adding a separate model based on message passing, which does not integrate well with the method-invocation based object model. Also, MPI was designed to deal with static groups of processes rather than with objects and threads. MPI mainly uses *collective* communication. A communication operation (e.g., a broadcast) must be explicitly invoked by all participating processes. This requires processes to execute in lock-step, a model which is ill-suited for object-oriented programs (which are often multi-threaded).

In this chapter, we introduce a more elegant approach to integrate flexible group communication into Java. Our goal is to express group communication using method invocations, just as RMI is used to express point-to-point communication, and RepMI is used to express shared data. We will generalize the RMI model in such a way that it can express communication with a group of objects. We call this extended model *Group Method Invocation* (GMI). The key idea of GMI is to extend the way in which a method invocation and its result value are handled. For example, in RMI a method invocation will always be forwarded to a single (remote) object. The thread that invoked the method must always wait for a result to be returned before it can continue. In GMI, a method invocation can either be forwarded to a single object or to a group of objects. The parameters to this invocation can be either be forwarded without modification, or personalized for each destination (using a user defined method). Any result values (including exceptions) can be returned normally, discarded, forwarded to a separate object, or, when multiple result values are produced, combined into a single result. All schemes for handling invocations can be combined with all schemes for handling results, giving a fully orthogonal design.

Due to this orthogonal approach, GMI is both simple and highly expressive. GMI can be used to simulate MPI-style collective communication, where all members of a group collectively invoke the same communication primitive. Unlike MPI, however, GMI also allows an individual thread to invoke a method on a group of objects, without requiring active involvement from other threads (e.g., to read information from a group of objects). GMI can also express many communication patterns that have thus far been regarded as unrelated, special primitives. In particular, futures and voting can easily be expressed using GMI.

In the following sections, we will show that GMI is an expressive, efficient, and easy-to-use framework for parallel programming that supports a rich variety of communication patterns, while cleanly integrating with the method-invocation based communication models of RMI and RepMI. The remainder of this chapter is structured as follows. Section 5.2 describes the GMI model. In Section 5.3, we describe the API and show examples of GMI code. Section 5.4 describes the implementation of GMI using the Manta system. In Section 5.5, we use GMI to implement a variety of applications and benchmarks, including parts of the MPJ benchmark suite from the Java Grande Forum [17], and provide a detailed performance evaluation of both low-level and application performance of GMI. We will also compare the performance of GMI

applications to applications using RMI, RepMI, and mpiJava [5] (a Java binding to the MPI libraries). Finally, Section 5.6 presents related work and Section 5.7 concludes.

5.2 The GMI Model

In this section, we describe the GMI model, and how it is used to implement group communication. We will first show how we have generalized the RMI model (described in detail in Section 2.2) to support groups of objects, and then describe how communication patterns within these groups can be used to implement many types of group communication.

5.2.1 Generalizing the RMI model

In GMI, we generalize the RMI model in three ways. First, in RMI, a remote reference can only be used to refer to a single remote object. In contrast, a reference in GMI can be used to refer to a group of objects. These *group objects* may be distributed across a number of machines. Figure 5.1 shows an example of such a *group reference*. On JVM 1, a single group reference is used to refer to a group of two objects. These objects are located on JVMs 2 and 3. Note that, like remote references in RMI, group references have an interface type. This interface is used as a *marker interface*, that is recognised by a compiler. This compiler can then generate a stub object that contains the necessary communication code.

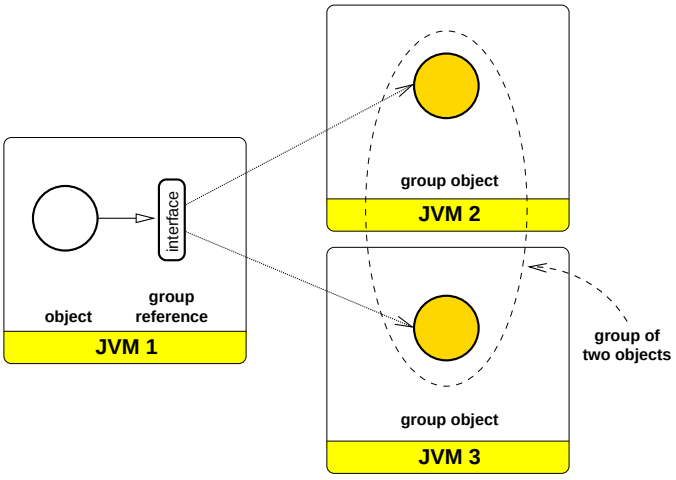

Figure 5.1: A group reference referring to two objects.

Second, the RMI model only supports synchronous point-to-point communication. GMI extends this limited model by adding several ways of forwarding methods and handling the replies. For example, GMI allows the programmer to express broadcast and asynchronous communication. This generalization requires the compiler to generate different types of communication code for each method.

Third, using a simple API, GMI allows the programmer to configure a group reference *at run time* to use specific forwarding and result-handling schemes. Each method in the group reference can be configured separately, and different forwarding and result-handling schemes can be combined, giving a rich variety of communication mechanisms. By configuring the group references at run time, the communication behavior of the application can easily be adapted to changing requirements. Some forwarding and result handling schemes require application-specific code to be provided by the programmer in order to work correctly. We will describe the available forwarding and reply-handling schemes in detail in Sections 5.2.3 and 5.2.4.

5.2.2 Object Groups

In GMI, groups of objects can be created at runtime. A group consists of several objects, possibly on different machines. For proper semantics of group operations, groups have a fixed size and become immutable upon creation [74]. The objects in the group are ordered and can be identified by their *ranks*. This rank and the group size are available to the objects in the group, allowing them to adapt their behavior according to their rank in the group.

Each group of objects has a certain type, the *group interface*. This interface serves a similar function as a remote interface in RMI and the replication interface in RepMI. It defines which methods may be invoked on the group and allows the compiler to generate the necessary code (e.g., stub and skeleton objects). Like remote and replicated interfaces, a group interface uses call-by-value semantics. The group interface must be implemented by all objects in the group. This ensures that the methods in the group interface can be invoked on every group object. The objects in a group may have different types, as long as they implement the same group interface.

Group interfaces are defined just like remote or replication interfaces. The programmer defines an interface that contains a number of methods and extends a 'special' interface that triggers the necessary code generation in the compiler. This will be explained in more detail in Section 5.3.

5.2.3 Forwarding Schemes

GMI supports several different ways of forwarding methods to the objects in a group. Depending on the needs of the application, each method in a group interface can be configured separately to use one of these forwarding schemes. GMI currently offers the following forwarding schemes:

- *single invocation*
  The invocation is forwarded to a single (possibly remote) object of the group, identified via its rank.

- *group invocation*
  The invocation is forwarded to every object in the group.

- *personalized invocation*
  The invocation is forwarded to every object in the group, while the parameters are *personalized* for each destination using a user-defined method.

- *combined invocation*
  Using different group references, several threads collectively invoke the same method on a group. These invocations are combined into a single invocation using a user-defined method. This invocation is then forwarded to the group using one of the previous forwarding schemes.

The first scheme, *single invocation*, is similar to a normal RMI; method invocations are forwarded to a single object. Unlike RMI, however, GMI allows this destination to be reconfigured at run time. The programmer can specify to which group object the method invocations must be forwarded when a method is configured to use this scheme.

The second scheme, *group invocation* can be used to express a broadcast. The same method invocation (with identical parameters) is forwarded to every object in a group.

The third scheme, *personalized invocation* is suitable for distributing data (or computations) over a group. Before the invocation is forwarded to each of the group objects, a user-defined *personalization* method is invoked. This method serves as a filter for the parameters to the group method. It gets the parameters and the size of the group as input, and produces a personalized set of parameters for each destination. A *personalized* version of the method invocation is then sent to each group object.

The fourth scheme, *combined invocation*, allows multiple threads (possibly on different JVMs) to collectively invoke a method on a group. Each thread separately invokes the same method on a group reference. These method invocations are then combined into a single invocation using an *invocation combiner* method. An invocation combiner acts as a filter that takes the parameters of all invocations and combines them into a single set of parameters. A new invocation, using this new set of parameters, is then applied to the group. To forward this new invocation, one of the three previous invocation schemes must be selected.

Figure 5.2 shows an example, where two threads collectively invoke a method on a group of three objects. These two invocations are first combined into a single invocation, which is then forwarded to every object in the group using a personalized

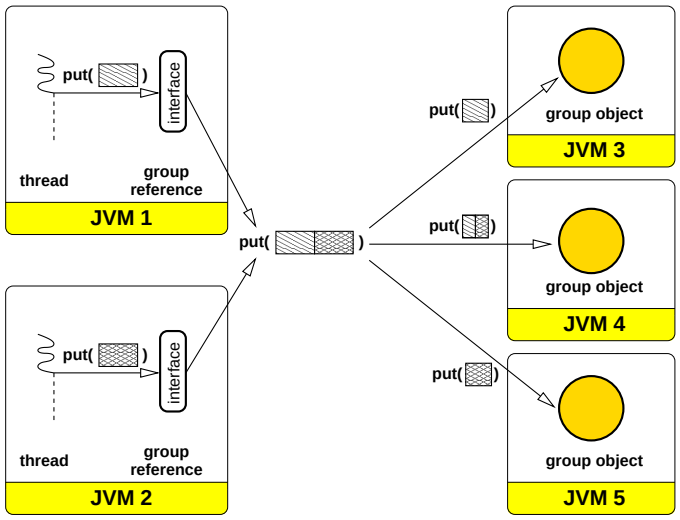

Figure 5.2: An example of invocation combining and personalization.

invocation.[2] This is illustrated by the 'parameter' of the methods, which is depicted as a box filled with a certain pattern. The two boxes of the original invocations are first combined into a single one, and then split again into three separate boxes (each containing part of the parameters of the original invocation).

The example of Figure 5.2 illustrates another feature of the GMI model. In GMI, there is no relation between the number of objects in a group and the number of threads that invoke methods on these objects. Any thread that acquires a group reference may invoke methods on a group. Similarly, there is no relation between the location of the group objects and the location of the threads. Both the threads and the group objects may be distributed across several JVMs. As the example shows, it is possible for a thread to be located at a JVM that does not contain any group objects.

This model is much more general than the *Single Program Multiple Data* (SPMD) model that is traditionally used in communication libraries such as MPI. Thus, GMI applications can use communication schemes that are difficult to express using MPI. It is also possible to implement MPI-style collective communication operations using GMI. For example, the communication shown in Figure 5.2 resembles MPI's *all-to-all* operation. Section 5.3.2 shows more examples of how MPI-style collective operations can be simulated with GMI.

[2] In this example we use a combined invocation followed by a personalized invocation. It is also possible to use a single or group invocation after combining.

In GMI, each method of a group reference can be configured separately to use a specific forwarding scheme. Thus, the group reference can be configured in such a way that each of its methods behaves differently. This can be implemented by having the compiler generate a stub object to act as the group reference. This stub contains specialized communication code for each method to support every forwarding scheme. The programmer can then select at run time which compiler-generated communication code is used to forward a specific method. Although this approach increases the code size of the stub (compared to the RMI stubs), it allows the compiler to use compile-time information to generate optimized communication code for each method.[3]

When a method is configured to use one of the first three forwarding schemes, the configuration can be performed locally (i.e., on a single group reference). The fourth forwarding scheme (combined invocation) requires a *collective* configuration. Because multiple group references are involved in producing a single invocation, the programmer must specify as a part of the configuration how many invocations are to be combined, which method is used to do the combination, and how the resulting invocation will be forwarded. To prevent that multiple concurrent combine invocations on the same group interfere with each other, they must also be given a unique identifier. After a method of a group reference is configured to use a combined invocation, the GMI runtime system can use the unique identifier to locate the other group references that wish to participate. These group references may be located on different JVMs, so communication between the different instances of the GMI runtime system may be necessary. The runtime systems will wait until the correct number of group references has joined and check if they agree on the configuration parameters (e.g., the number of invocations that are combined, the invocation combiner method, etc.). If all parameters match, the method is configured correctly and can be used to do a combined invocation.

Configuring a method to use combined invocation can be quite expensive (it requires communication between the JVMs). However, once configured, a method can be used several times without requiring reconfiguration. Therefore, the configuration cost are amortized over multiple invocations.

5.2.4 Result-Handling Schemes

RMI only supports synchronous method invocations. When a thread invokes a remote method, it must always wait for a result before it can continue. For some parallel applications, this is too restrictive. The GMI model therefore does not require that a result is always returned to the invoker. Instead, it offers a variety of result-handling schemes that can be used to express asynchronous communication, futures, and other primitives. The code implementing these result handling schemes can be generated

[3]The Manta compiler, for example, generates some 32 Kbyte of GMI and serialization code for a group interface with a single method that uses an object parameter. In comparison, 8 Kbyte is generated for a similar remote interface.

by the compiler and located either in the group objects themselves, or in a specialized skeleton object. GMI offers the following result handling schemes:

- *discard results*
  No results are returned at all (this includes exceptions).
- *return one result*
  A single result is returned. If the method is invoked on a single object, the result can be returned directly. Otherwise, if the method is applied to multiple objects, one of the results (preselected via a rank) is returned, and all others are discarded.
- *forward results*
  All results are returned, but they are forwarded to a user-defined *handler* object, rather than being returned to the original invoker.
- *combine results*
  Combine all results into a single one, using a user-defined *combine* method. This combined result is returned to the invoker.
- *personalize result*
  A personalized result is returned to each of the threads participating in the invocation (useful when a combined invocation is used). A user-defined *result-personalizer* object is used to personalize the results.

The first scheme, *discard results*, allows a method to be invoked asynchronously. The invocation will return immediately (without waiting for a reply). If the method itself returns a result, a default value of the correct type (e.g., '0.0' or a *null* reference) will be returned instead. Any result values that are produced (including exceptions) will be directly discarded by the group object (or skeleton).

The second result handling scheme, *return one result*, is the default way of handling results in unicast invocations like RMI. The group reference forwards the method invocation and blocks until the result is returned. If multiple results are produced, the user selects *in advance* one of the group objects by its rank. Only that group object will return a result, the rest will be discarded. This has the advantage that only a single result has to be returned, possibly avoiding communication overhead.

The third scheme, *forward results*, allows all results (including exceptions) to be forwarded to a separate *handler* object. Like in the *discard results* scheme, the method returns immediately, returning a default result value if necessary. However, the group objects will now return their results normally. When such a result arrives at the group reference (i.e., the stub) it will be forwarded to a user-defined handler object, where it can be retrieved later. This mechanism can be used to implement futures (where the result value is stored until it is explicitly retrieved), voting (where all results are collected and the most frequently occurring one is returned), selection (of one result) and combining (of several results).

The fourth scheme, *combine results*, is useful when multiple results are produced. The group reference forwards the method invocation and then blocks. All the group objects will return a result. A user-defined combine method is then used, which acts as a filter that takes multiple return values (including exceptions) as input, and produces a single result as output. This result is returned to the invoking thread.

Different approaches can be used to combine the results. One option is to simply return all results to the machine that contains the invoking thread and then combine them using a single invocation of the user-defined combine method. This implementation is useful for combine methods that *gather* the results. Results are gathered when the combine method takes all results that are produced, and then combines them into a single, larger result that is returned (e.g., a number of small arrays that are combined into a single, large array).

A combine method can also be used to *reduce* the results. Results are reduced when all results are combined into a single value that is, in general, the same size or complexity as one of the original results. Typical examples of reduce operations are determining the minimum or maximum value of all results or calculating the sum of all results.

Reduce-style combine operations have the advantage that they can be executed in parallel. Instead of returning all results to a single machine, the group objects or skeletons can communicate amongst each other to combine all results into a single value. If this communication is arranged in a tree structure, large parts of the combine operations are executed in parallel. This does require the combine method to be defined differently. Instead of combining all results at once, the results are combined *pairwise*. It must also be commutative (i.e., the correct value is produced regardless of the order in which the results are combined).

An example is shown in Figure 5.3. There, a method invocation, *get()* is broadcast to a group of four objects (on different JVMs). Each of the group objects contains a *double* value, which is returned by the *get()* method. A combine method will be used to compute the sum of all these returned values. This sum is returned as the final result. After the *get()* method has been applied, the group objects (or generated skeletons) communicate amongst each other to determine the final result. The results of JVMs 2 and 3 can be combined in parallel with the results of JVMs 4 and 5. JVM 4 then forwards its (combined) result to JVM 2, which adds it to its local (combined) result and returns the final value to the invoker.

The implementation of GMI can determine if a combine method requires a gather-style or a reduce-style communication by looking at the type of combine method provided by the user. Reduce style combine methods require two result values as parameters, while gather-style combine methods use an array that contains all result values. Using this information, the correct communication style can be selected during the configuration of a method. Because the combined value is returned as the result value of the method, the return type of the combine methods must be of the same type as the results that they combine.

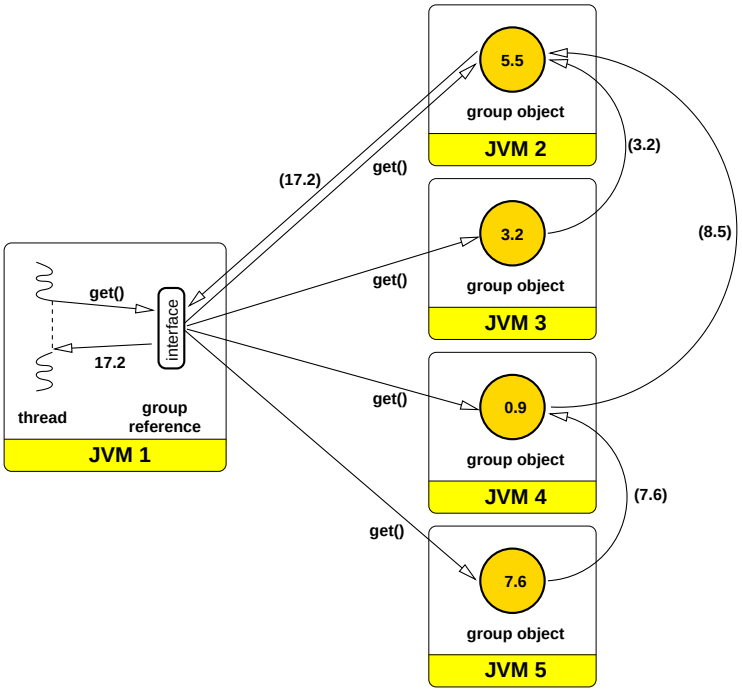

Figure 5.3: An example of result combining

The combine method must also be able to handle any exceptions that were returned instead of regular result values. These exceptions are provided to the combine method as extra parameters. The combine method itself may also produce an exception. When a gather-style combine method is used, this exception will be returned to the invoking thread. For a reduce-style combine, the exception may also be forwarded to another group object (if it was an intermediate result). In Section 5.3, we will illustrate how combine methods can be defined by the programmer.

The final result-handling scheme, *personalize result*, allows a *personalized* result value to be returned to each of the invoking threads. This is mainly useful when a combined invocation is used. A user-defined *result personalization* method must be provided during the configuration of the method. The result personalization method takes one result value and splits it into a number of personalized result values that will be returned to each of the group interfaces. One of the other result-handling schemes must be used to produce the result value that will be personalized. When the forward results scheme is used, the personalization method may be invoked repeatedly (for each of the results). This communication is similar to the *all-to-all* operation of MPI.

5.2.5 Combining Forwarding and Reply-Handling Schemes

We have presented four different schemes to forward a method invocation and five schemes for handling the results. GMI allows these schemes to be combined orthogonally, resulting in a wide variety of useful communication patterns. Table 5.1 shows several combinations.

communication	*invocation*	*result*
RMI	single	return one
future	single	forward
asynchronous RMI	single	discard
broadcast	group	discard
reduce or gather	group	combine
scatter	personalized	discard
scatter-gather	personalized	combine
reduce	combined + single	discard
gather	combined + single	discard
reduce-to-all	combined + group	discard
gather-to-all	combined + group	discard
all-to-all	combined + personalized	discard
scatter	combined + single	return one + personalize
all-to-all	combined + group	combine + personalizer

Table 5.1: Combinations of forwarding and reply handling

As the table shows, GMI is able to express RMI-style communication. Besides the normal, synchronous RMI, GMI also supports two asynchronous flavors (*future* and *asynchronous RMI*).

In the table, *scatter, gather(-to-all), reduce(-to-all)* and *all-to-all* refer to the functionality of the respective collective operations from MPI. However, there is a major difference between the GMI communication and their MPI counterparts. In MPI, *all* members of a group must *collectively* invoke the communication operation. In GMI, this is only required with combined invocations. Normal invocations allow a single thread to invoke a method on a group of objects without active involvement from other threads. For example, when combining a personalized invocation with result combining (shown as *scatter-gather* in the table), a single thread forwards a personalized invocation to the entire group and receives the combined result.

The table also shows that different GMI configurations may result in similar communication patterns. For example, *gather*-style communication can be expressed using a group invocation and a combined result, or a combined-single invocation with no result. This illustrates another major difference between the communication models of GMI and MPI. While MPI uses a message-passing model, GMI is based on a method-invocation model. Therefore, in GMI, communication always consists of an invocation being forwarded and (optionally) a result being returned. Consequently, several MPI-style operations can be simulated by GMI either by using the invocation handling, or by using the result handling. While the first generally requires the use of

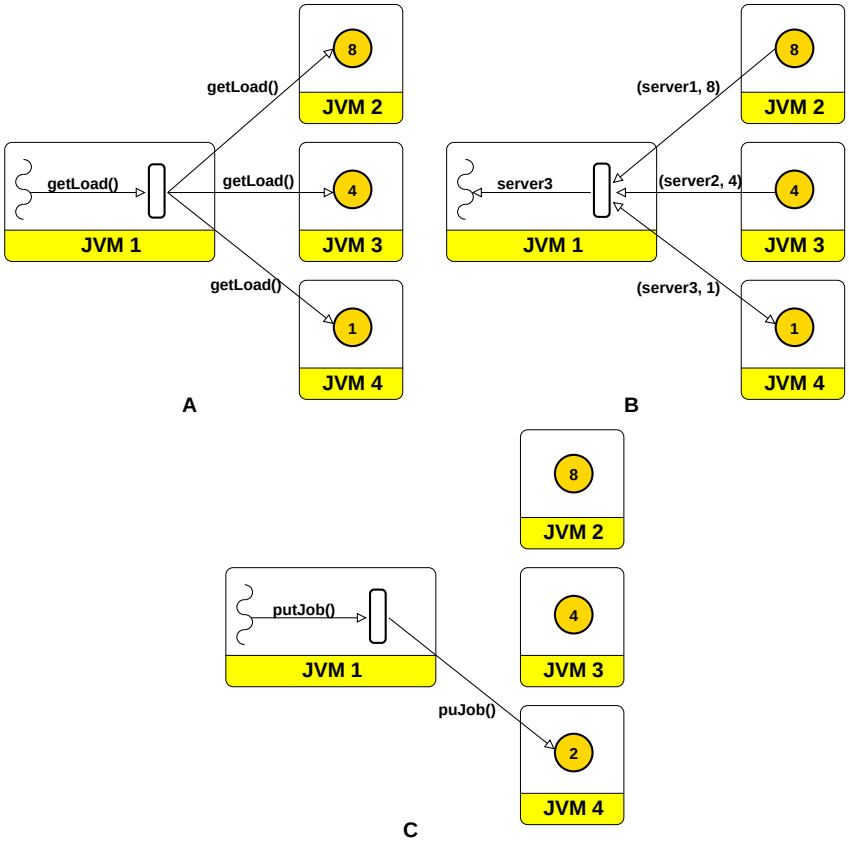

Figure 5.4: Load balancing using GMI

a combined invocation (where multiple threads must actively participate), the second can also be done by a single thread (something which is hard to express using MPI). We will now illustrate such single-thread group operations by showing how GMI can be used to implement a load balancing mechanism.

In Figure 5.4, a thread on JVM 1 produces jobs for three servers running on JVMs 2, 3, and 4. Every server contains an object that monitors the load of the machine. Together these object form a group. Whenever a new job is produced, the thread invokes the *getLoad* method on each of the objects using a group invocation (A). Each object executes this method and returns an estimate of the local load. The thread then selects the least loaded server using a combine operation (B), and sends that server a job (C).

This application is difficult to express using MPI-style collective communication, because it is not known in advance when the load balancing information will be requested. Such asynchronous events are easily expressed with group invocations, but not with collective communication. When using collective communication, the servers would either frequently have to poll for incoming *getLoad* messages, or periodically broadcast their latest load information, both of which would degrade performance.

5.2.6 Invocation Ordering, Consistency and Synchronization

The purpose of GMI is to free the programmer from having to implement the complex communication code required for group communication, thereby allowing the focus to be on the application itself (i.e., *what* is implemented using the communication, not *how* the communication implemented). Unlike the RepMI model, the GMI model does not guarantee any sort of consistency between the group objects. There are no restrictions on the code that can be executed by a group method. Group methods are allowed to use static variables, native methods, threads, etc. GMI, like RMI, is only concerned with handling the method invocations and result values. What happens after the invocation has been delivered and before the result is returned, is up to the application. However, the method invocation and result handling schemes offered by GMI have several properties that may influence how the application is implemented. We will now briefly describe these properties.

Like in RMI, invocations in GMI are *reliable*. When a method is invoked, it will always be executed on the target objects(s). Similarly, method results will always be delivered (unless the *discard results* scheme is used). Note that it is relatively easy to extend GMI with *unreliable* method invocations.

GMI allows multiple method invocations to run *concurrently* on a group object. When a method invocation arrives at a group object, conceptually a new thread is started to handle this invocation (for efficiency, this may be implemented differently). Several of these threads may run concurrently. GMI does not give any guarantees about the order in which these threads are scheduled.

In GMI, multiple asynchronous method invocations (i.e., discarding their results) that are forwarded by the same group reference are *not* guaranteed to run in FIFO (*First In First Out*) order. Even if the network messages containing the method invocations are delivered in FIFO order, each method invocation is executed by a separate thread. These threads may be scheduled in a non-FIFO order. Also, using an asynchronous method invocation is conceptually the same as starting a thread. The method invocations may use normal Java synchronization primitives to block or signal other threads. A number of method invocations originating from the same group reference may even need to interact by sharing data or using *wait*/*notify* constructs. Guaranteeing FIFO-ness for these invocations would not only require the GMI runtime system to implement its own thread scheduling and deadlock prevention (as is implemented in the RepMI runtime system), but may also restrict its performance and usefulness.

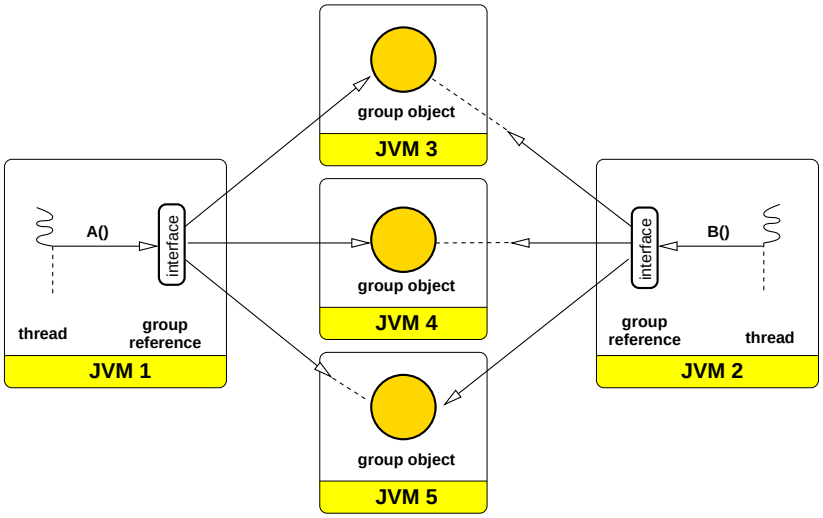

Figure 5.5: Simultaneous broadcasts to a group are not ordered.

When several method invocations are broadcast simultaneously on the same group of objects, GMI does not guarantee that each group object receives or executes these invocations in the same order. An example is shown in Figure 5.5, where two group invocations, *A* and *B*, are applied on a group of three objects. The two objects on JVMs 3 and 4 receive these invocations in the order *A()*, *B()*, while the third object, on JVM 5, receives them in the order *B()*, *A()*. In which order the methods are actually executed depends on how the threads are scheduled on each JVM. GMI does not provide a totally-ordered broadcast because there are many applications that do not require this. For example, in an application like QR factorization, it is known in advance that only one machine at a time will do a broadcast. In applications like ASP, each method invocation that is broadcast updates a different part of a data structure. The order in which the method invocations are received is not important since they cannot interfere with each other. Some applications, like TSP, method invocations are broadcast to update a piece of shared data. These methods are *commutative*, they will produce the same result regardless of the order in which they are invoked. Using a totally-ordered broadcast in these types of applications will only result in extra communication overhead. We have therefore chosen not to use a totally-ordered broadcast in GMI. For applications that require a totally-ordered broadcast to ensure the consistency of group objects, the RepMI programming model should be used instead.

Because several invocations on a group of objects may run concurrently, the GMI implementation must be able to handle several result combining operations simultane-

ously. For example, both the group invocations shown in Figure 5.5 may use reduce-style result combining. A naive implementation of this result combining could easily produce a deadlock or mix up the result messages of the two operations. Therefore, the group objects (or their skeletons) must be able to handle the communication of multiple reduction operations simultaneously.

To use a combined method invocation, the programmer must configure the method in advance to specify which group references will participate in the invocation. It is important that each of these group references produces an invocation at approximately the same time, because method combining is a *collective* operation. Each participating group reference will block until all invocations are combined. Only then will the combined invocation be forwarded.

To summarize, the focus of the GMI model is on handling method invocations and result values. Multiple method invocations may run concurrently on one group object. Invocations that are broadcast may be received in different orders on different objects. To prevent problems, the programmer must ensure that the group objects are properly synchronized and are able to handle invocations that arrive 'out-of-order'. These precautions are similar to those required when writing RMI or multi-threaded applications. The programmer must also ensure that combined method invocations are invoked 'collectively'.

5.3 API

In this section, we will describe the API that we have designed for the GMI model. Just like RMI, GMI uses a marker interface and runtime support (e.g., library classes), but no language extensions. We have defined a Java package *group* that contains the necessary classes and interfaces for using groups of objects. We will first describe the classes in this *group* package, and then show some examples of how the API can be used in applications.

5.3.1 The *Group* package

Figure 5.6 shows part of the classes in the group package. The *GroupInterface* is a marker interface similar to *java.rmi.Remote*. It does not define any methods, but serves as a signal to the compiler. The compiler must recognize any interfaces that extend this type as being a group interface and then generate the required stub and skeleton classes.

The *GroupMember* class has a similar role as *java.rmi.UnicastRemoteObject* in RMI. It must always be extended by group objects, since it contains some basic functionality required by the GMI runtime system. When created, a group object must invoke the constructor of *GroupMember* to ensure that the object will be correctly initialized.

```
package group;

public interface GroupInterface { // empty }

public class GroupMember {
   public GroupMember()
   public final int getRank()
   public final int getSize()
   public void groupInit()
}

public final class Group {
   public static void create(String name, Class type, int size)
   public static void join(String name, GroupMember member)
   public static void join(String name, GroupMember member, int rank)
   public static GroupInterface lookup(String name)
   public static GroupMethod findMethod(GroupInterface i, String desc)
}

public final class GroupMethod {
   public void configure(InvocationScheme inv, ReplyScheme rep)
}

public class InvocationScheme {
   // base class for invocation schemes
}

public class ReplyScheme {
   // base class for reply schemes
}
```

Figure 5.6: The API of GMI.

The *Group* class contains static methods that allow the programmer to create and manipulate groups. Using the *create* method, a new group can be created. This group must be given a *name*, a *type*, and a *size*. The *type* must be a class representing a group interface. After a group has been created, the *join* method can be used to add objects to the group. Only objects that implement the correct interface (i.e., the *type* of the group) may be added. An optional *rank* parameter may be passed to *join* to indicate the desired rank of the object within the group.[4] When no *rank* parameter is specified, the GMI runtime system will assign a rank to each object. The join method will block until the group has reached the correct size.[5] The group is then ready for use.

[4] This can be used to ensure that multiple groups are ranked consistently.

[5] GMI does currently not support changing group sizes.

When the group is complete, the *getSize* and *getRank* methods of the *GroupMember* class can be used to determine the size of the group and the rank of an object. Also, the *groupInit* method will be invoked automatically once the group has been completed. Subclasses of *GroupMember* (i.e., the group objects) can redefine this method and use it to implement any initialization that depends on the rank of the object or size of the group.

The *lookup* method can be used to create new group references. Its objective is similar to *java.rmi.Naming.lookup*. It returns a generic group reference which must be cast to the correct type before use (i.e., the *type* of the group it refers to). Any JVM is allowed to create a group reference. To do so, it does not need to contain any group objects itself.

To configure the invocation and reply-handling schemes for a method of a group reference, *findMethod* can be used. As parameters, it requires a group reference and a string description of the method that must be found. A *GroupMethod* object will be returned as a result. This object represents a certain method of a specific group reference. It contains a single method *configure* that can be used to set the invocation and reply handling schemes of the method it represents. The *configure* method requires two objects as parameters, an *InvocationScheme* and a *ReplyScheme*. These two classes can not be used directly, but serve as a basis for several classes shown in Figures 5.7 and 5.8.

Figure 5.7 shows the API of the classes that represent the four different invocation handling schemes offered by GMI. Each class contains a constructor that receives the necessary parameters for that type of invocation handing scheme.

For example, a *PersonalizedInvocation* requires a *Personalizer* as a parameter. A *Personalizer* is a so-called *function object*. Because the Java language has no support for *function pointers*, simple objects, containing one or more methods with a well-known name, are used instead. The *Personalizer* object contains a method that is capable of producing a personalized version of the method parameters for every group object. *Personalizer* objects will be described in more detail below.

The *CombinedInvocation* has particularly complex constructors. They require the programmer to provide a unique *identifier* for the combined invocation, the *number* of group references that will participate in the call, the *rank* of the local group reference, a function object capable of combining the parameters of the different invocations (*combiner*) and an invocation scheme that must be used to forward the combined invocation (*inv*). The two constructors accept different types of function objects to combine the invocations. This will be explained in more detail below. The constructors will throw a *ConfigurationException* if any of the parameters are not correct.

In Figure 5.8, the API is shown of the classes that represent the reply handling schemes supported by GMI. Again, each constructor receives the necessary parameters for that form of reply handling. For example, the *ForwardReply* constructor requires a *Forwarder* object. Any replies produced by the method invocation will be forwarded to this object instead of being returned to the invoker. The *CombineReply*

```
public final class SingleInvocation extends InvocationScheme {
   public SingleInvocation(int destination)
                          throws ConfigurationException
}

public final class GroupInvocation extends InvocationScheme {
   public GroupInvocation() throws ConfigurationException
}

public final class PersonalizedInvocation extends InvocationScheme {
   public PersonalizedInvocation(Personalizer p)
                                 throws ConfigurationException
}

public final class CombinedInvocation extends InvocationScheme {
   public CombinedInvocation(String identifier, int rank, int number,
                             FlatInvocationCombiner combiner,
                             InvocationScheme inv)
                             throws ConfigurationException
   public CombinedInvocation(String identifier, int rank, int number,
                             BinomialInvocationCombiner combiner,
                             InvocationScheme inv)
                             throws ConfigurationException
}
```

Figure 5.7: Objects that represent the invocation handling schemes.

class has two constructors. The first one receives a *BinomialCombiner* as a parameter, that is able to combine the result values reduce-style. The second constructor requires a *FlatCombiner* object as parameter, which is able to combine the result values in gather-style.

The *PersonalizedReply* class can be used in combination with a combined method invocation. The *ReplyScheme* parameter indicates how the group object(s) must return the result(s). When one (possibly combined) result is returned, the *ReplyPersonalizer* object is then used to personalize this result for each of the participating group references. If multiple results are forwarded, the *ReplyPersonalizer* object will be repeatedly used to personalize each of these results.

Many of the *InvocationScheme* and *ReplyScheme* classes require the use of function objects. Figure 5.9 shows the pseudocode for the classes of the four reply handling function objects that are used in GMI. These classes are not intended to be used directly, but must be extended by subclasses which behave according to the needs of the application. Each class shown in Figure 5.9 contains methods to handle every possible type of result value.

For example, the *Forwarder* class contains several *forward* methods, one with a boolean *result* value, one with a *byte* result value, one with an *Object* result value, etc.

```
public final class DiscardReply extends ReplyScheme {
   public DiscardReply() throws ConfigurationException
}

public final class ReturnReply extends ReplyScheme {
   public ReturnReply(int source) throws ConfigurationException
}

public final class ForwardReply extends ReplyScheme
   public ForwardReply(Forwarder fw) throws ConfigurationException
}

public final class CombineReply extends ReplyScheme {
   public CombineReply(BinomialCombiner b) throws ConfigurationExc...
   public CombineReply(FlatCombiner f) throws ConfigurationException
}

public final class PersonalizedReply extends ReplyScheme {
   public PersonalizedReply(ReplyPersonalizer rp, ReplyScheme prev)
                            throws ConfigurationException
}
```

Figure 5.8: Objects that represent the reply handling schemes.

(for brevity, not all methods are shown). None of these methods actually do anything other than throwing an exception. Instead, the programmer is expected to create a class that extends *Forwarder* and redefines one or more of the *forward* methods to behave according to the needs of the application. An object of this new class may then be used as a parameter to a *ForwardReply* constructor. After a method has been invoked, all result values will be forwarded to this object by invoking the *forward* method with the right *result* type. The *rank* of the group object that produced the result, and the *size* of the group are also forwarded.

The same approach is used in the other function objects. In the *FlatCombiner* class, the methods have arrays as parameters that will be used to return all result values at once. A single (combined) value is returned as the result of the method. The *BinomialCombiner* class contains methods that can be used to do a pairwise combine of the results. The *ReplyPersonalizer* class contains methods that take a single result value as input and produce an array of result values as output (one personalized value for each destination in the array entries). *BinomialCombiner* and *ReplyPersonalizer* objects may be forwarded to other JVMs and are therefore serializable. In contrast, the *Forwarder* and *FlatCombiner* objects will always remain on the JVM that they were created on.

The classes of the function objects required for invocation handling are shown in Figure 5.10. A *Personalizer* is used to split the parameters of a single method invoca-

```
public class Forwarder() {
   public void forward(int rank, int size) // void result
   public void forward(int rank, int size, boolean result)
   public void forward(int rank, int size, byte result)
   public void forward(int rank, int size, Object result)
   public void forward(int rank, int size, Exception result)
   ...
}

public class FlatCombiner {
   public boolean combine(Exception [] ex) // void result
   public boolean combine(boolean [] results, Exception [] ex)
   public byte combine(byte [] results, Exception [] ex)
   public Object combine(Object [] results, Exception [] ex)
   ...
}

public class BinomialCombiner implements java.io.Serializable {
   public boolean combine(int rank1, Exception e1,
                          int rank2, Exception e2, int size)
   public boolean combine(int rank1, boolean res1, Exception e1,
                          int rank2, boolean res2, Exception e2,
                          int size)
   public byte combine(int rank1, byte result1, Exception e1,
                       int rank2, byte result2, Exception e2,
                       int size)
   public Object combine(int rank1, Object result1, Exception e1,
                         int rank2, Object result2, Exception e2,
                         int size)
   ...
}

public class ReplyPersonalizer implements java.io.Serializable {
   public void personalize(boolean in, boolean [] out)
   public void personalize(byte in, byte [] out)
   public void personalize(Object in, Object [] out)
   public void personalize(Exception in, Exception [] out)
   ...
}
```

Figure 5.9: Pseudocode for the reply handling function objects.

```
public class Personalizer {
   public void personalize(ParameterVector in, ParameterVector [] out)
}

public class FlatInvocationCombiner {
   public void combine(ParameterVector [] in, ParameterVector out)
}

public class BinomialInvocationCombiner {
   public void combine(ParameterVector in1, ParameterVector in2,
                       ParameterVector out)
}

public class ParameterVector {
   public boolean readBoolean(int p)
   public byte    readByte(int p)
   public Object  readObject(int p)
   ...

   public void write(int p, boolean value)
   public void write(int p, byte value)
   public void write(int p, Object value)
   ...
   public void writeSubArray(int p, int off, int size, boolean [] val)
   public void writeSubArray(int p, int off, int size, byte [] val)
   public void writeSubArray(int p, int off, int size, Object [] val)
   ...
}
```

Figure 5.10: Pseudocode for the invocation handling function objects.

tion into a several, personalized, sets of parameters. The *InvocationCombiners* do the opposite: they combine the parameters of several invocations of the same method into one set of parameters. As with result combining, there are two ways to combine invocations: *reduce-style*, implemented by the *BinomialInvocationCombiner*, and *gather-style*, implemented by the *FlatInvocationCombiner*. The personalization and invocation combining classes contain a single method, *personalize* and *combine*, respectively. They can not be used directly, but must be extended by application-specific implementations.

Because the methods that are personalized (or combined) may have any combination of parameters, it is not feasible to statically define all possible *personalize* or *combine* methods (we did use this approach in the result handling function objects). Instead, we use special *ParameterVector* objects, which are designed to encapsulate the parameters of a group method. For example, when the *personalize* method is invoked, a *ParameterVector* object, *in*, is provided that contains the parameters of the

method invocation that must be personalized. Using methods like *readBoolean* and *readObject*, the parameters can be extracted from this *ParameterVector* object. The integer parameter *p* specifies which parameter must be returned. An array of *ParameterVector* objects, *out*, is also provided to the *personalize* method. Each of these objects represents the parameters for one of the personalized method invocations. The personalized parameters can then be stored in these objects by applying *write* methods. The integer *p* specifies into which parameter the *value* must be stored. The implementation of *ParameterVector* must check if all parameters are written correctly before they are forwarded to a group object. As an optimization, the compiler can generate subclasses of *ParameterVector* that are specialized to handle the parameters of a certain group method.

Since personalized invocations are often used to distribute array data across a group, a *writeSubArray* method is also provided, which allows this operation to be expressed efficiently (without copying the data). In the next, section we will give several examples that show how the API can be used in applications.

5.3.2 Examples

We now show five examples that illustrate how GMI can be used in applications. Using the API described above, creating a group object is similar to creating an RMI object. A *group interface* (extending the interface *GroupInterface*) must be created, which defines the methods that are common to the group objects. To be part of a group, the object must implement this group interface and extend the *GroupMember* class. An example is shown in Figure 5.11.

The group interface *Bin* contains two methods, *put* and *get.* The group object, *BinImpl*, implements these methods. The *put* method stores the array parameter in the object while *get* retrieves the array. Both *put* and *get* use the normal Java synchronization primitives to ensure that no data is lost.

The *Server* class contains the main server program, which is run on several machines. One server creates a new group called *BinGroup.*[6] All servers then create a new *BinImpl* object and add it to the group. The object group is now ready for use. Using the *lookup* method, a *Client* application can create a new group reference. As will be shown in the following examples, each of the methods in this group reference can be configured separately to use a specific combination of forwarding and reply handling schemes. Multiple group references, using different configurations for the same method, can also be used.

It is possible to merge the *Server* and *Client* code into a single application. It that case, an *SPMD-style* application is created where every machine contains both a group object and a thread that participates in the computation.

[6]We assume that some infrastructure is present to allow multiple Java programs running on different machines to participate in a parallel computation. E.g., every machine is capable to contact every other participating machine, and the machines are ordered by a ranking scheme.

```
import group.*;

interface Bin extends GroupInterface {
   public void put(double [] v);
   public double [] get();
}

class BinImpl extends GroupMember implements Bin {
   private double [] data = null;
   public synchronized void put(double [] v) {
      while (data != null) wait();
      data = v;
      notifyAll();
   }
   public synchronized double [] get() {
      while (data == null) wait();
      double [] temp = data;
      data = null;
      notifyAll();
      return temp;
   }
}

class Server {
   public static void main(String [] args) {
      if (server rank is 0) {
        // number of servers is N
        Group.create("BinGroup", Bin, N);
      }
      Group.join("BinGroup", new BinImpl());
   }
}

class Client {
   public static void main(String [] args) {
      Bin group = (Bin) Group.lookup("BinGroup");
      doWork(group); // do something useful here
   }
}
```

Figure 5.11: A simple GMI application (pseudocode).

Figure 5.12 shows the different application styles that can be expressed using GMI. The first example shows a *client-server style* application, where a thread on the client (JVM 1) invokes methods on a group of two objects on the servers (JVMs 2 and 3). In this example, the servers are *passive*. They will only execute the methods that are invoked on the group objects. No other code is running on the servers.

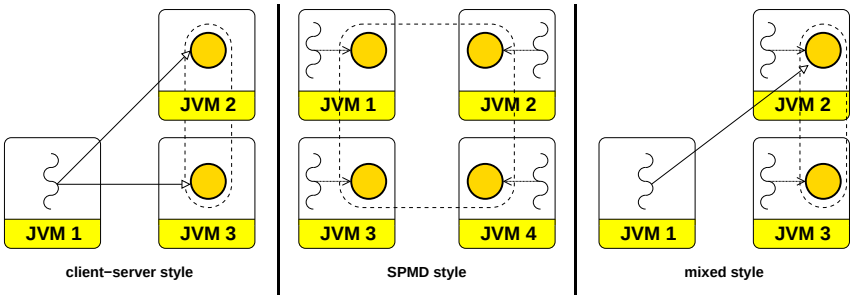

Figure 5.12: Different application styles supported by GMI.

The second example shows an *SPMD style* application, where every machine contains a group object and a thread that is executing application code. The threads can communicate by applying methods on the group objects.

The third example shows a *mixed style*, where a combination of client-server and SPMD is used. In this application style, an 'external' JVM communicates with an SPMD style computation. This approach is useful for visualization or steering purposes.

We will now extend the example of Figure 5.11 to illustrate how several important forms of group communication can be expressed using GMI. We will give different implementations of the *doWork* method, which is called from the *main* method of the *Client* class of Figure 5.11.

Group Invocation

Figure 5.13 illustrates a group invocation (i.e., broadcast) in GMI. Before the *put* method can be configured, a *GroupMethod* object is created using the *findMethod* call. Note that a description of the *put* method is passed to *findMethod* as an ordinary string.

```
public static void doWork(Bin group) {
   // configure the method
   GroupMethod m = Group.findMethod(group, "void put(double [])");
   m.configure(new GroupInvocation(), new DiscardReply());

   // invoke the method
   group.put(new double[100]);
}
```

Figure 5.13: A group invocation.

Using the *GroupMethod* object, *put* is configured to use an asynchronous broadcast. This can be expressed by invoking the *configure* method of the *GroupMethod* object that represents *put*, and passing a *GroupInvocation* and *DiscardReply* as parameters.

After the configuration, the *put* method can be invoked. This invocation will then be forwarded to all objects in the group. Any result produced will be discarded. This example clearly illustrates the ease of expressing group communication in GMI. Note that the group method may be invoked many times, it only has to be configured once.

Personalized Group Invocation

Personalized group invocations are frequently used in parallel programs. Figure 5.14 shows an example of how such an invocation can be expressed in GMI. The *put* method is configured to use a *PersonalizedInvocation* and therefore requires a *Personalizer* object. The definition of this object, called *Split* is also shown in Figure 5.14.

```
public static void doWork(Bin group) {
   GroupMethod m = Group.findMethod(group, "void put(double [])");
   Personalizer p = new Split();
   m.configure(new PersonalizedInvocation(p), new DiscardReply());

   group.put(new double[100]);
}

class Split extends Personalizer {
   void personalize(ParameterVector in, ParameterVector [] out) {
      double [] param = (double []) in.readObject(0);
      int size = (param.length / out.length);
      for (int i=0;i<param.length;i++)
         out[i].writeSubArray(0, i*size, size, param);
   }
}
```

Figure 5.14: A personalized invocation.

The *Split* class contains a single method, *personalize*, which redefines the method of the *Personalizer* class which it extends. This *personalize* method first extracts the array parameter from the *ParameterVector* called *in*. It then calculates how this array must be divided over the group objects.[7] Finally, each piece of the array is stored in a different output *ParameterVector* using the *writeSubArray* method.

When the *put* method now is invoked, the *personalize* method of the *Split* object will be invoked once, using a *ParameterVector* to pass the *double[100]* parameter.

[7] This calculation assumes that the array can be divided evenly.

Several *ParameterVector* objects will be produced as output (one for each group object). Their contents will be forwarded as parameters of method invocations to the different objects in the group.

Result Combining

In the next example, shown in Figure 5.15, gather-style result combining is used to retrieve all the arrays stored in the objects of the *BinGroup* and combine them into a single array. In the *doWork* method, the *get* method is configured to use a group invocation. Thus, when the *get* method is invoked, it is forwarded to all group objects. Each group object produces a *double* array as a result value. These arrays must be combined using a *Gather* object.

```
public static void doWork(Bin group) {
   GroupMethod m = Group.findMethod(group, "double [] get()");
   FlatCombiner f = new Gather();
   m.configure(new GroupInvocation(), new CombineReply(f));

   double [] result = group.get();
}
class Gather extends FlatCombiner {
   public Object combine(Object [] results, Exception [] ex) {
      double [] result = new double[results.length*SIZE];
      for (int i=0;i<results.length;i++) {
         if (ex[i] != null) // throw exception
         System.arraycopy((double [])results[i], 0,
                                       result, i*SIZE, SIZE);
      }
      return result;
    }
}
```

Figure 5.15: Combining a result.

This *Gather* class is shown in Figure 5.15. By extending the *FlatCombiner* class, it indicates that it implements a gather-style combine operation. Its *combine* method receives all the returned arrays (or exceptions) as parameters. The data in these arrays is then copied into a single array, which is returned. This new array will be used as the result of the *get* method. In this simple example, the *combine* method assumes that all arrays have an equal size (i.e., *SIZE*), which is known in advance. More complicated combine operations with variable sized data are also possible.

Result Forwarding

The use of asynchronous communication is often essential for adequate performance of parallel applications. One example is the use of *futures*, which allows the invocation of a method and the reception of its return value to be handled separately. Figure 5.16 shows how this can be implemented in GMI using the *forward result* scheme.

```
public static void doWork(Bin group) {
   GroupMethod m = Group.findMethod(group, "double [] get()");
   Forwarder f = new Future()
   m.configure(new SingleInvocation(0), new ForwardReply(f));

   group.get();
   // .. do some useful computations here
   double [] result = f.getResult();
}

class Future extends Forwarder {
   private double [] array = null;
   public synchronized void forward(int rank, int size, Object res) {
      array = (double []) res;
      notifyAll();
   }
   public synchronized double [] getResult() throws Exception {
      while (array == null) wait();
      return array;
   }
}
```

Figure 5.16: Forwarding a result.

The *get* method is configured in such a way that invocations are forwarded to a single object in the group. The return value is forwarded to an external object, a *Future*. As Figure 5.16 shows, the *Future* class extends the *Forwarder* class offered by GMI. It redefines the method that handles object results. The method stores its result parameter in the *Future* object and notifies any waiting threads that a result has been received.

When the *get* method is invoked, it will be forwarded to the first object of the group (i.e., the object with rank *0*), and immediately return a default value (i.e., *null*). This allows the *doWork* method to do some useful computations while waiting for the result of *get*. When the *doWork* method is ready, it can invoke *getResult* to retrieve the value returned by *get*. The *getResult* method will block if no value is available yet. Also, the *getResult* method is not defined in the *Forwarder* class. *Forwarder* only specifies how the method results are delivered (i.e., by using the *forward* methods). How these results are stored or retrieved is up to the application.

Invocation Combining

Our final example illustrates the use of *invocation combining*. With invocation combining, several threads simultaneously invoke the same method. These invocations are then combined into a single one. Figure 5.17 shows an example of invocation combining.

```
public static void doWork(Bin group) {
   GroupMethod m = Group.findMethod(group, "double [] get()");
   CombinedInvocation c = new CombinedInvocation("gather-to-all",
                                                 rank,
                                                 size,
                                                 new Combiner(),
                                                 new GroupInvocation());
   m.configure(ci, new CombineReply(new Gather())); // will block
   double [] result = group.get(); // will block
}

class Combiner extends FlatInvocationCombiner {
   void combine(ParameterVector [] in, ParameterVector out) {
       // empty
   }
}
```

Figure 5.17: Invocation combining.

The *get* method is configured to be a combined invocation that requires the participation of *size* group references (i.e., threads). The string *"gather-to-all"* is used as the unique identifier for the operation. This identifier is used by the GMI runtime system to locate all the participating group references. The *Combiner* class is used to combine the parameters of the invocations. Since the *get* method has no parameters this is an empty method.

When the *configure* method is invoked, it will block until enough group references have joined the combined invocation (i.e., *size*-1 other group references must also *configure* their *get* method in the same way). After the *get* method is invoked, the group references will communicate to combine the sixteen invocations into one. After the invocation has been combined, a *GroupInvocation* is used to forward the new invocation to all objects in the group.

The replies are combined using the *Gather* object that was described in the result combining example. Because no *ReplyPersonalizer* is used, the same (combined) result will be returned to all participating group references. Thus, the example of Figure 5.17 implements an operation that is similar to MPI's *gather-to-all* operation. In the next section, we will describe in more detail how the collective operations of MPI can be simulated using GMI.

Implementing MPI-style Collective Operations

The examples shown above are all based on a *client-server* style GMI application. Many parallel algorithms exist, however, that use an (MPI-oriented) SPMD style programming model and collective communication. Fortunately, these types of applications can be implemented using GMI relatively straightforwardly, as we will show in the following examples.

```
class SPMD {
   public static void main(String [] args) {
      if (server rank is 0) {
         // number of servers is N
         Group.create("SPMDGroup", Bin, N);
      }
      BinImpl local = new BinImpl();
      Group.join("SPMDGroup", local);
      Bin group = (Bin) Group.lookup("SPMDGroup");
      doWork(group, local);
   }
}
```

Figure 5.18: An SPMD-style group application

Figure 5.18 shows an example of an SPMD-style application. The *SPMD* class contains the application code, which is a combination of the client and server examples shown in Figure 5.11. The application uses two different references to the *BinImpl* object. A group reference, *group*, which is created using the *lookup* method, and a normal reference, *local*, which directly refers to the group object. Both of these references are needed to simulate the collective operations of MPI, so they are both passed to the *doWork* method.

Figure 5.19 shows how a collective broadcast operation can be simulated in GMI. On each JVM, the *doWork* method starts by configuring the *put* method to use a group invocation without a result. The *broadcast* method can then be invoked, supplying the group and local reference as parameters. Only one JVM is allowed to supply the data to broadcast. All others must use *null* as the *data* parameter.

The JVM that supplies the data then invokes *put* on the group reference to forward the data to all objects in the group. Every JVM then invokes the *get* method on the local reference to retrieve the data stored by *put.* Note that *get* will block until the data has arrived. The array is then returned on all JVMs.

The implementation of a collective *all-to-all exchange* is shown in Figure 5.20. On every JVM, the *doWork* method configures *put* to use a combined personalized invocation, without a result. Every JVM then creates a *data* array, which contains one element for each object in the group. After this data array is used in some computation, the JVMs want to exchange their data.

```
public static void doWork(Bin group, BinImpl local) {
   GroupMethod m = Group.findMethod(group, "void put(double [])");
   m.configure(new GroupInvocation(), new DiscardResult());

   double [] data = null;
   if (local.getRank() == 0) {
      data = broadcast(group, local, new double[100]);
   } else {
      data = broadcast(group, local, null);
   }
   // use data here
}

static double [] broadcast(Bin group, BinImpl local, double [] data) {
   if (data != null) {
      group.put(data);
   }
   return local.get();
}
```

Figure 5.19: Simulated collective broadcast in GMI.

In *exchange*, each JVM invokes the *put* method, supplying its own data as a parameter. All invocations of *put* are then combined into a single one using an *ArrayCombiner* object. The *combine* method of this object takes all array parameters and consecutively writes their first elements into a larger result array. It then writes all second elements, etc. The large result array is then returned.

This result array will not be directly forwarded to the group objects. Instead it is personalized again using the *Split* class that we described in the *personalized invocation* example. Thus, each group object will receive a *put* invocation with an array parameter that is a fragment of the *result* array returned by the *ArrayCombiner*. Because this *ArrayCombiner* has 'mixed' the elements of the arrays it received as input, the array parameter of each *put* contains a collection of elements that were sent by different JVMs. Finally, each JVM invokes the *get* method on its local group object, which blocks until the personalized *put* method has been received. The result of the *get* is returned as the result of the *exchange* operation.

As these examples show, the collective operations of MPI and its SPMD programming model can be simulated rather straightforwardly using GMI. Other collective operations, like *reduce-to-all* and *gather-to-all* can be simulated using similar constructs. Operations like *reduce-to-one* and *gather-to-one* use the reverse approach. Each JVM stores its data in the local group object by using a *put* invocation on the local reference. One of the JVMs then applies a result-combining group invocation of *get* to the group reference. This invocation will retrieve the data from all group objects and combine it into a single value, which is then returned to the JVM.

```
public static void doWork(Bin group, BinImpl local) {
   GroupMethod m = Group.findMethod(group, "void put(double [])");
   PersonalizedInvocation pi = new PersonalizedInvocation(new Split());

   CombinedInvocation ci = new CombinedInvocation("exchange",
                                                  somerank,
                                                  local.getSize(),
                                                  new ArrayCombiner(),
                                                  pi);
   m.configure(ci, new DiscardReply());

   double [] data = new data[local.getSize()];
   // use data here

   data = exchange(group, local, data);
   // use data here
}


static double [] exchange(Bin group, BinImpl local, double [] data) {
   group.put(data);
   return local.get();
}


class ArrayCombiner extends FlatInvocationCombiner {
   void combine(ParameterVector [] in, ParameterVector out) {
      double[][] temp = new double[in.length][];
      for (int i=0;i<in.length;i++)
          temp[i] = (double [])in[i].readObject();

      double [] result = new double[in.length*in.length];
      for (int i=0;i<in.length;i++) {
         for (int j=0;j<in.length;j++) {
            result[i*in.length+j] = temp[j][i];
         }
      }
      out.write(0, result);
   }
}
```

Figure 5.20: Simulated collective all-to-all exchange in GMI.

5.4 Implementation

In this section we will describe the implementation of GMI. As in the previous chapters, we have chosen to use the Manta platform as a basis for our implementation. Below, we will describe how we have extended Manta to support the group model.

5.4.1 Compiler

We have extended the Manta compiler to recognize the *GroupInterface* and generate the communication code required for GMI. Java systems that are based on bytecode instead of a native compiler would need a preprocessor similar to *rmic* used for RMI.

When the compiler encounters an interface that extends *GroupInterface*, it generates a *stub* object similar to the ones used in RMI and RepMI. The GMI stubs re-use the highly efficient serialization code that we described in Chapter 3. They contain the communication code that is used to forward the method invocations. When a group reference is used in the application, this is actually a reference to a GMI stub.

GMI stubs, unlike RMI stubs, contain several different types of communication code. For each method in the group interface, communication code is generated to support each invocation forwarding scheme. Every stub also contains a table of information about the group methods it implements. There is an entry in this table for every group method, containing information about the forwarding scheme, reply-handling scheme, and the function objects that are used. The information in this table can be changed by using the *GroupMethod.configure* method.

When a compiler-generated group method is invoked on the stub, it checks the table to see which forwarding scheme and reply-handling scheme must be used for this method. It then invokes the appropriate compiler-generated communication code. Because we are using the Manta system, the communication code in the stubs can directly invoke Panda communication routines, which in turn use efficient network-level primitives. For example, to do a group invocation, the stub can use the efficient broadcast offered by Panda. To implement combined invocations, the stubs are able to communicate with each other. For reduce-style combining of method parameters, an efficient binomial tree algorithm is used, similar to the one used to implement the *reduce* operation of MPI. For gather-style invocation combining, asynchronous point-to-point messages are used.

The Manta compiler also generates *skeleton* code. As with RMI, Manta does not generate a separate skeleton object. Instead, the skeleton code is inserted directly into the group object itself. In RMI, the purpose of the skeleton code is to handle incoming method invocations (i.e., apply the methods on the remote object), and to return the invocation result to the stubs. The skeleton code of GMI, like its stub code, is more complex. Not only does GMI have a number of different ways of handling the results, but the skeletons (or group object) must also be able to communicate amongst each other to combine several result values into a single one. This result combining is

implemented in a similar way as the invocation combining. Like the stub code, the skeleton code used in Manta can directly access the Panda communication library.

5.4.2 Runtime System

For the Manta system, we have implemented GMI's API completely in Java. Creating and joining groups is implemented using RMI. A central RMI object at a well-known location functions as a group registry. When a new group is created, its name is registered in this group registry, allowing other machines to access this group. When an object tries to join a group, a request is sent to this central registry (using RMI), containing information about this object. The registry waits until all join requests have been received to complete the group, and then returns all group information (size, type, location of the objects, rank to object mapping etc.) to every group member.

The configuration of combined invocations is implemented in a similar way. The stubs that participate in a combined invocation can locate each other at configuration time by using the unique name of the operation and the centralized registry. When all stubs have signed up, the registry returns the necessary information to all of them.

The Manta runtime system (written in C) also contains a small portion of GMI-related code. This is mainly concerned with handling the incoming messages and delivering them at the destination stub or skeleton.

The Java implementation of GMI's API is approximately 2000 lines of code. The C implementation of the GMI-related part of the Manta runtime system is somewhat smaller, approximately 1500 lines. The compiler-related code, required for generating stubs and skeletons is significantly larger, some 6000 lines of code. Although much of this code (especially the compiler) is fairly straightforward, this gives some indication of the complexity hidden from the programmer by GMI.

5.5 Performance Results

We will now evaluate the performance of the GMI implementation in Manta. This evaluation will consist of three parts.

We will first discuss the results of several micro benchmarks that we have implemented using GMI. We use these micro benchmarks to measure the cost of several frequently-used GMI primitives. To ensure that GMI's flexible model does not cause too much overhead, we compare its efficiency to alternative implementations in mpiJava [5] (version 1.2) and MPI C. The mpiJava library is a Java language binding to a native MPI library (in our case MPICH), which provides highly-efficient collective communication. To ensure that the sequential speed of the Java code is the same for GMI and mpiJava, we have compiled the mpiJava library and benchmarks with the Manta compiler. Performance differences can therefore be attributed to the different communication libraries. This approach allows us to compare performance of GMI to a less expressive, but highly optimized, approach that just invokes MPI routines from

Java. The MPI C benchmarks use the same MPI implementation (MPICH), but use C instead of Java code.

Next, we compare the application level performance of GMI and mpiJava. For this purpose, we have ported two kernels and one application from the Java Grande Forum MPJ benchmark suite [17] to GMI. We also ported three application kernels used in the previous chapters to mpiJava. Together, the applications cover a wide range of group communication primitives, including broadcast, reduce-to-all and gather-to-all. We will then compare the speedups achieved by the GMI and mpiJava versions. For each application, we report speedups relative to the fastest of the two versions on one machine so that higher speedups always translate to lower execution times.

Finally, we will compare the performance of several GMI applications to versions that use RepMI and RMI. These applications were also used in the previous chapters. This allows us to evaluate the performance benefits of GMI compared to the previous communication models.

Like in the previous chapters, all performance measurements are done on the DAS system, [8] a cluster of 200 MHz Pentium Pro processors with 128 MByte of main memory. All boards are connected by a 1.2 Gbit/sec Myrinet network. The system runs RedHat Linux 6.2 (kernel version 2.2.16). In our micro benchmarks and applications, we run a single executable on every machine involved in the computation.

After our performance evaluation, we will evaluate the complexity of writing parallel application with GMI by comparing the code sizes of GMI, RMI, RepMI and mpiJava applications.

5.5.1 Micro benchmarks

In this section we discuss the performance of four basic group operations, *broadcast*, *personalized broadcast*, *gather-to-all* and *reduce-to-all*, implemented by using GMI, mpiJava and MPI C. The purpose of this comparison it to ensure that the flexible, group communication offered by GMI does not have an unacceptably high overhead in comparison to the efficient collective communication offered by MPI.

All benchmarks are written in SPMD-style. Therefore, for GMI, each participating machine contains an object that is part of a group. For the first two benchmarks only one machine will invoke methods on this group. The other machines do not invoke any group methods. They only initialize their local group object and handle the invocations that they receive. In the other two benchmarks, a combined invocation is used, which requires all machines to invoke a method on the group. For every benchmark, all machines perform a normal method invocation on their local group object after each group invocation to retrieve the data. The latency of each of the operations is shown in Figures 5.21 to 5.24

The latency of the broadcast operations is shown in Figure 5.21. To prevent pipelining effects in the broadcast operation, we have implemented benchmarks sim-

[8] http://www.cs.vu.nl/das/

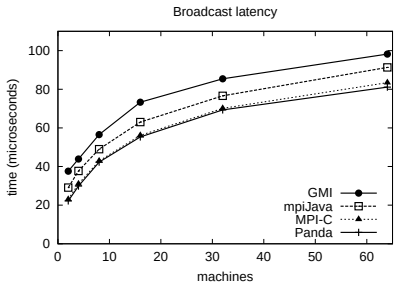

Figure 5.21: Latency of broadcast operation.

ilar to those described in Section 4.4. Therefore, the results in the figure indicate the time required for the broadcast to reach all machines. In addition, this figure also shows the performance of the low-level Panda broadcast, which is used to implement the other broadcast operations. Since no message ordering is required for GMI, this benchmark uses an *unordered* broadcast. This is more efficient than the *totally-ordered* broadcast, which was used in the previous chapter to implement the RepMI model.

As the figure shows, GMI group invocation adds a constant overhead of some 16 μs to the Panda broadcast latency. This overhead can be attributed to the way in which method invocations are received in GMI. When a message is received, the GMI runtime system selects a thread from a pool to handle the method invocation. As a result, a thread switch must be performed before the method invocation can be handled. Because the GMI benchmark is written in SPMD style, the application thread invokes a normal method on the local group object to retrieve the result of the group method invocation. Therefore, two thread switches are required to handle each GMI method, one to switch to the pool thread which handles the GMI method, and one to switch back to the application thread, which is waiting for the result. Since the two methods use *wait/notify* constructs to signal each other they are also *synchronized* (i.e., a lock must be acquired before they may be invoked).

In MPI, broadcast messages are both sent and received using explicit *broadcast* statements. Therefore, no thread switches or locking is required to handle the messages. As a result, the overhead that MPI C adds to Panda is small, approximately 1 μs. The mpiJava library, which is implemented on top of MPI C, adds more overhead to Panda, 8 μs. This shows that the overhead of using GMI's flexible group communication to simulate an SPMD-style broadcast operation is small. The broadcast latency is only 8 μs higher than that of mpiJava.

In Table 5.2, we compare the throughput obtained by GMI to that of MPI C and mpiJava. The MPI benchmarks do not perform any communication when running on a single machine. Therefore no throughputs are shown.

machines	*GMI*	*mpiJava*	*MPI C*	*Static Panda*	*Dynamic Panda*	*Dynamic mpiJava*
1	60.7	-	-	334	61.1	-
2	30.1	56.4	56.6	56.8	31.3	30.0
4	27.8	43.2	43.2	43.2	30.0	28.6
8	25.7	21.0	21.4	21.4	28.0	24.2
16	23.8	22.2	22.4	22.5	24.6	20.6
32	18.3	22.5	22.6	22.6	22.2	18.6
64	11.3	13.3	13.3	13.3	13.2	12.5

Table 5.2: Broadcast throughput (in MByte/s).

In this benchmark, there is a significant difference between the behavior of the GMI benchmark and the MPI and mpiJava versions. In MPI and mpiJava, the broadcast message is received using an explicit receive statement, which provides a pre-allocated buffer as a destination for the message. In GMI, like in RMI and in RepMI, it is not possible to pre-allocate any buffers because a method-invocation model is used. This model does not allow explicit receive statements. Also, due to polymorphism, the exact type and size of the received objects are not known during compile time. Therefore, the objects can only be created dynamically, when a message is deserialized. This problem also occurs in mpiJava when objects are transferred. When using primitive arrays, however, mpiJava uses the more efficient explicit receive model.

As a result, the GMI throughput benchmark (like its RMI and RepMI counterparts) suffers from memory allocation overhead, which is not present in the MPI-based benchmarks. Due to this overhead, the GMI throughput on two machines is approximately 46% lower than that of the MPI benchmarks. When the number of machines increases, however, the overhead decreases. On 64 machines, the throughput of GMI is 15% lower than that of MPI.

For comparison, we have also added the results of two Panda and an extra mpiJava benchmark. The *Static Panda* benchmark uses a pre-allocated buffer to receive the broadcast data. Therefore, it behaves similarly to the MPI benchmarks. The *Dynamic Panda* benchmark dynamically allocates the memory required to receive a message, and therefore behaves more like the GMI benchmark. Similarly, the *Dynamic mpiJava* benchmark dynamically allocates a new array for every broadcast. Although this is not usually done in MPI-style applications, it does simulate the memory allocation and copying behavior of the GMI benchmark.

As the table shows, both the MPI C and mpiJava benchmark have a performance that is very similar to the *Static Panda* benchmark, while the throughput of the GMI benchmark is similar to that of the *Dynamic mpiJava* and *Dynamic Panda* benchmarks. This clearly illustrates that the reduced throughput of GMI may be attributed nearly entirely to the overhead of memory allocation.[9]

[9]The *Static Panda* benchmark shows a reduced throughput on 8 and 16 machines due to a flow-control problem in the broadcast implementation. In the *Dynamic Panda* benchmark this problem does not occur

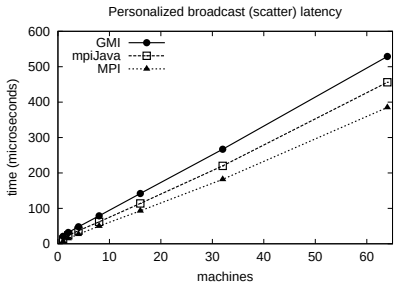

Figure 5.22: Latency of personalized broadcast (scatter) operation.

Figure 5.22 shows the latency of the personalized broadcast operation (or *scatter* in MPI terminology). Both GMI and MPI implement this operation by separately forwarding a part of the data to every machine (using consecutive send operations). As a result, the latency of the personalized broadcast operation increases linearly with the number of machines.

The MPI C implementation has the lowest latency, ranging from 7 μs on a single machine to 385 μs on 64 machine. The mpiJava version is slightly slower, 12 μs on a single machine to 456 μs on 64 machines. When we look more closely at the results, we see that mpiJava adds a fixed overhead to MPI C of approximately 5 μs and adds an extra 1 μs for every destination machine.

The personalized broadcast operation of GMI is more general that MPI's scatter operation. Instead of distributing a single array over the machines, as MPI does, GMI allows the parameters of a method to be personalized in a user-defined way. However, because of this flexibility, GMI's personalized broadcast is more expensive than a scatter operation. Its latency ranges from 21 μs on one machine, to 529 μs on 64 machines.

In Figure 5.23 the latency of the reduce-to-all operation is shown. The MPI C version again has the lowest latency, followed closely by the mpiJava version. The GMI version is slightly slower than mpiJava. However, the differences are small. The mpiJava version is on average only 10 μs slower than the MPI C version, while the GMI version is 23 μs slower than mpiJava.

The difference between GMI and mpiJava can be attributed to the differences in the operations themselves. In mpiJava, an *Allreduce* operation uses a binomial-tree algorithm to combine data from every machine into one piece of data, which is then returned to all machine as the result of the operation. In GMI, however, the reduce operation is only the first step. It is used to combine several method invocations into a single one, which is then applied to all objects in a group. A second (local) method

because the memory allocation overhead slows down the sender.

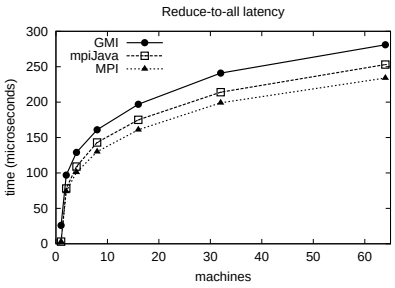

Figure 5.23: Latency of reduce-to-all operation.

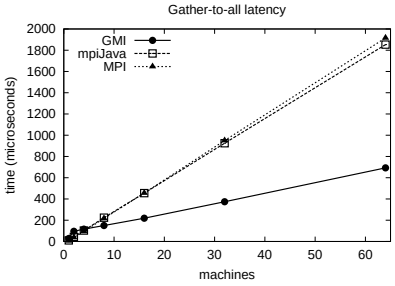

Figure 5.24: Latency of gather-to-all operation.

invocation is then needed to retrieve the data from the local group object. As the graph shows, this approach is slightly more expensive.

Finally, Figure 5.24 shows the latency of the gather-to-all operations. The figure clearly shows that the *Allgather* operation of MPI C (which is also used by mpiJava) uses a different algorithm than GMI. In GMI, all machines forward their method invocation to one machine, which combines them into a single invocation. This invocation is then broadcast and applied to all group objects.

The MPI implementation uses a ring algorithm, which clearly results in a higher latency. On 64 machines, GMI obtains a gather-to-all latency of 693 μs, for MPI C and mpiJava this latency is 1916 and 1853 μs, respectively.

Table 5.3 shows the throughput obtained by the different gather-to-all implementations. As the table shows, MPI and mpiJava achieve a throughput which is approximately twice that of GMI. Because the GMI gather-to-all operation is implemented using a broadcast, the throughput is already limited to the broadcast throughput shown in Figure 5.2. Unfortunately, the gather-to-all operation increases this overhead further. All machines first send their data to a single machine, which combines all data and

machines	*GMI*	*mpiJava*	*MPI C*
2	19.0	52.2	55.6
4	15.9	37.0	38.5
8	14.7	31.1	31.4
16	13.2	27.2	27.7
32	12.0	21.8	22.8
64	9.7	16.2	17.7

Table 5.3: Throughput of the various gather-to-all implementations (in MByte/s).

broadcasts the result. This introduces additional communication and copying overhead.

GMI does not directly support gather-to-all or reduce-to-all operations. GMI offers a more general combined invocation model instead, which allows several method invocations (each containing part of the data in their parameters) to be combined into a single invocation. Depending on the implementation of the used-defined combine method, a binomial-tree or flat communication scheme is selected, which resemble reduce-to-all and gather-to-all operations. This approach can easily be extended to support other communication schemes, such as the gather-to-all algorithm used in MPI. However, using a different algorithm will not prevent the memory allocation overhead of deserialization. Therefore, the throughput of serialization based communication, as used in GMI, will always be lower than that of explicit receipt communication, as used in MPI.

5.5.2 Applications

In this section we will study the performance of GMI on an application level. We will do this in two ways. First, using six applications, we will compare the performance of GMI to implementations using mpiJava. Second, by using GMI to implement some of the applications described in the previous chapters, we can evaluate if using GMI offers a performance gain compared to using RepMI or Manta's efficient RMI.

GMI vs. mpiJava

To compare GMI's application performance to that of mpiJava, we use six different applications. The first three applications are taken from sections 2 and 3 of the Java Grande benchmark suite [17]. These applications come in different problem sizes, but we only show results for the largest problem size. The last three applications were taken from our own application set and ported to mpiJava. Figure 5.25 shows the speedups achieved with the applications, relative to the fastest version on a single machine.

We will now briefly describe the results and give a description of the applications that we have not introduced previously.

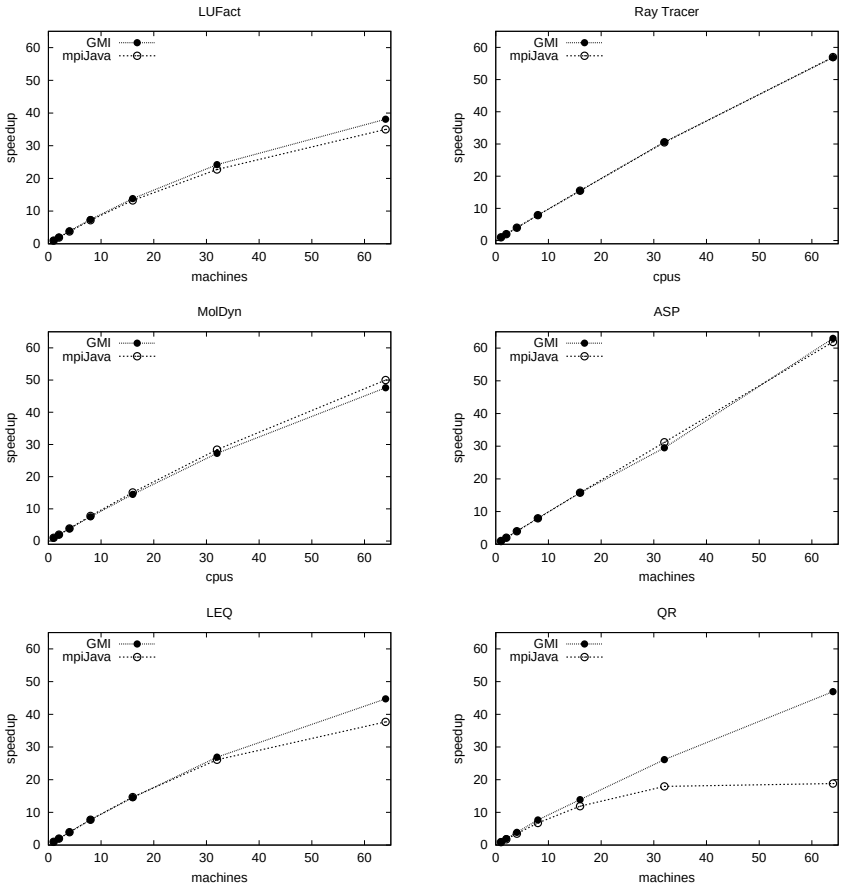

Figure 5.25: Speedups of GMI and mpiJava applications.

The **LUFact** application performs a parallel LU factorization, followed by a sequential triangular solve. The machines communicate by broadcasting integers and arrays of doubles. In the GMI version, these two broadcasts are expressed by forwarding a single *put* method with two parameters to a group of objects. In mpiJava, the integer is stored in a spare entry of a double array to prevent an extra broadcast. On 64 machines, the GMI version has a slightly better speedup than the mpiJava version (38 compared to 35). The difference in speedup is caused by the communication used to gather the result on a single machine at the end of the computation. Because the mpiJava version of LUFact uses *send* operations of type *OBJECT*, the data is serialized (using standard serialization) before it is forwarded. This results in a reduced

throughput. The GMI version uses the more efficient serialization generated by the Manta compiler.

Ray Tracer renders a scene of 64 spheres. Each machine renders part of the scene which is simultaneously generated on all nodes. Each node calculates a checksum over its part of the scene. A reduce operation is used to combine these checksums into a single value. The machines send the rendered pixels to machine 0 by individual messages. The speedups achieved by mpiJava and by GMI are almost identical.

MolDyn is an N-body simulation that models argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. For each iteration, the mpiJava version uses six reduce-to-all summation operations to update the atoms. In GMI, each machine stores its atom data in an object. These objects are then combined using a single combined invocation on the group. The objects, which serve as parameters to this invocation, are combined into one by using a reduce-style invocation combiner. The resulting object is forwarded to all objects in the group, where they are stored until they are retrieved locally. The speedup of MolDyn for GMI and mpiJava are almost identical, 47 for GMI and 50 for mpiJava (on 64 machines). The mpiJava version is faster because of its highly optimized summation operations for (arrays of) primitive types. Unlike GMI, mpiJava implements these operations completely in the library, it does not have to invoke user-defined methods to perform the operations. Neither does it require the use of serialization to communicate. The more general approach of GMI results in a slightly lower speedup.

ASP is described in Section 2.5.3. It communicates by having one machine broadcasting a single array of data to all others at the beginning of every iteration. Both the GMI and mpiJava version obtain an excellent speedup, because the amount of communication needed for each iteration (broadcast a row) is small compared to the computation time (updating multiple rows of the distance matrix).

LEQ is described in Section 3.5.1. The program partitions a dense 1000×1000 matrix containing the equation coefficients over the processors. In each iteration, each processor produces a part of the candidate solution vector x_{i+1}. Using a gather-to-all operation, these partial solution vectors are combined at the end of each iteration. The processors use a reduce-to-all operation to decide if another iteration is necessary, Both versions have a similar performance up to 32 machines, where both achieve a speedup of approximately 25. However, on 64 machines GMI achieves the best speedup, 45, while the mpiJava version only reaches 38.

QR is described in detail in Section 3.5.1. In this application, a reduce-to-all operation is used at the beginning of each to decide which machine must broadcast data. GMI obtains much better speedups for QR than mpiJava (48 compared to 20, on 64 processors), because mpiJava suffers from serialization overhead caused by the *allreduce* operation that uses objects as data. Instead of forwarding the data directly to the MPI library, as is done with arrays, mpiJava must now first serialize the data using the standard serialization mechanism. In contrast, GMI, is optimized to handle objects and uses the highly-efficient serialization code generated by the Manta compiler.

GMI vs. RMI and RepMI

We will now compare the performance of GMI to that of RepMI and RMI. For this purpose will use six application kernels that were also used in the previous chapters. Figure 5.26 shows the speedups achieved with the applications, relative to the fastest version on a single machine.

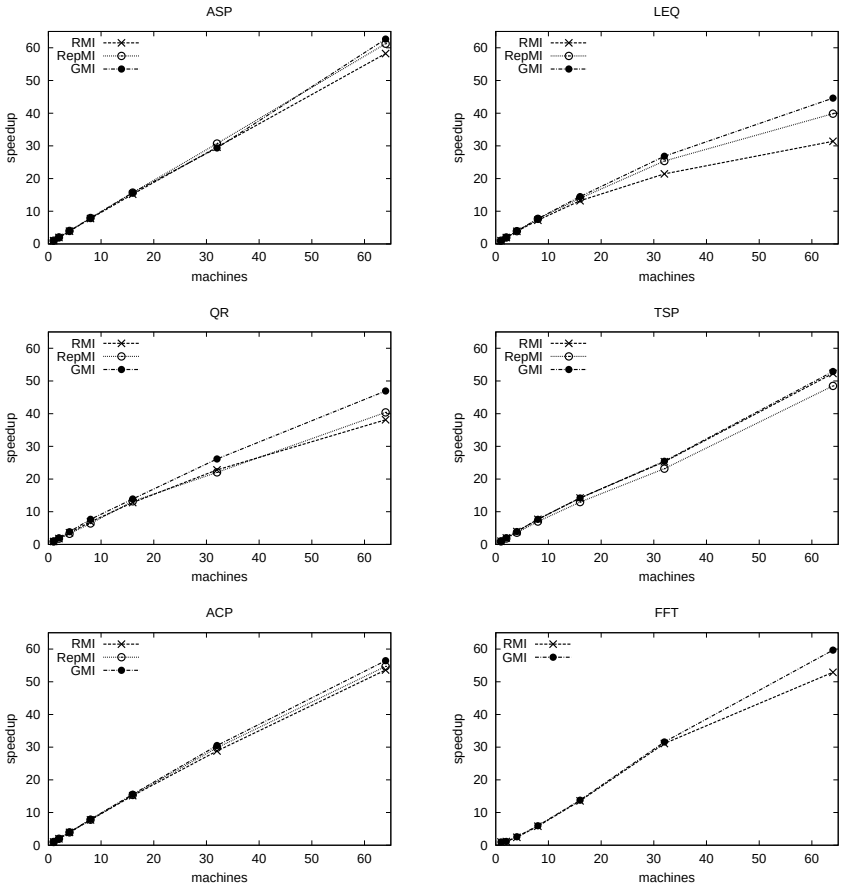

Figure 5.26: Speedups of RMI, RepMI and GMI applications.

ASP communicates by having one machine broadcasting a single array of data to all others at the beginning of every iteration. In GMI, this is implemented by using a group invocation which discards the results. RepMI used a single replicated object, while RMI used a broadcast simulation. Both the RMI and RepMI versions already

obtained an excellent speedup. The RepMI version was slightly faster due to its use of an efficient low-level broadcast implementation. The GMI version performs slightly better because it uses an unordered broadcast instead of the totally-ordered broadcast used in RepMI. It also discards the result values of the method that was broadcast, while RepMI always returns the local result.

LEQ uses a reduce-to-all and gather-to-all operation in each iteration. The RMI version implements this using a combination of a central remote object (to reduce and gather the data) and a broadcast simulation (to return the results to all machines). In the RepMI version, a single replicated object is used to implement a combined reduce-to-all and gather-to-all operation. Every machine writes its data into the replicated object. Each of these write operations is then broadcast to all replicas. When a replica has received all data, the result of the reduce-to-all and gather-to-all can be read locally.

The GMI version uses a more efficient approach. Using a single combined group invocation, all method invocations are gathered on a single machine, where they are combined into a single invocation. This invocation is then broadcast to all group objects. Each machine can then read the data from its local group object. Compared to the RepMI version, the GMI version reduces the number of broadcast messages to one. As a result, the GMI version obtains the highest speedup, 45 on 64 machines, while the RepMI and RMI versions obtain a speedup of 40 and 31, respectively.

In the **QR** application a reduce-to-all operation is used to decide which machine must perform a broadcast. Since a reduce-to-all can not be implemented efficiently using replication, both the RepMI and RMI versions use remote objects and a binary-tree algorithm to implement the reduce-to-all operation. For the broadcast operation, RepMI uses a single replicated object, while RMI uses a spanning tree protocol.

Using GMI, both operations can be expressed efficiently. For the reduce-to-all, an asynchronous combined group invocation is used, where each machine submits a method invocation. These invocations are combined into a single invocation using an efficient binomial-tree algorithm. The resulting invocation is then forwarded to all machines. The results of these invocations are discarded. For the broadcast, an asynchronous group invocation is used.

Because GMI provides an efficient implementation for both the reduce-to-all and broadcast operation, it obtains the highest speedup of the three versions, 47 on 64 machines. The RepMI and RMI version obtain speedups of 40 and 37.

In **TSP** (see Section 2.5.3) a global minimum value must be kept up to date on all machines. In the RMI version, this is implemented by storing this value in a remote object on every machine. Whenever the value is updated, a remote invocation is applied to each of the objects. In RepMI, this application could be expressed naturally, by storing the global minimum in a replicated object. The GMI version is similar to RMI. Every machine contains a group object which stores the minimum value. However, to update this value, GMI can use a single group invocation. On 64 machines, the three versions of TSP have very similar speedups, 53 for GMI, 52 for RMI and 48 for RepMI, respectively.

The **ACP** application (see Section 3.5.1) is similar to TSP, but shares a boolean matrix between machines instead of a single minimum value. Similar solutions are used to implement the communication of ACP. The three versions have almost identical speedups, 56 for GMI, 55 for RepMI and 54 for RMI (on 64 machines).

In **FFT** (see Section 2.5.3) data is exchanged between all processors. The RMI version implements this using separate RMIs between every pair of machines. GMI uses a similar approach, but uses asynchronous invocation (which discards the results) to minimize the communication latency. FFT can not be expressed efficiently using replication, so no RepMI version exists. The two versions have almost identical performance up to 32 machines. On 64 machines, the GMI version performs slightly better with a speedup of 59, compared to 53 for RMI.

5.5.3 Discussion

In this section, we have evaluated the performance of our GMI implementation using both micro benchmarks and application kernels, and compared its performance to that of mpiJava, MPI C, RMI, and RepMI.

The micro benchmarks have shown that the latencies in GMI and mpiJava of frequently used operations, are comparable. Although the latencies of GMI are somewhat higher, the differences are small. When we compare the throughput of the two systems, however, we see that mpiJava obtains much better results, especially on small numbers of machines. The lower throughput of GMI is not caused by the GMI model or implementation, but by a more general problem. In MPI and mpiJava, explicit receive statements are used which provide a pre-allocated buffer to receive the message in. This approach can only be used, however, if it is known in advance how large the received message is going to be. Although this is usually known in the array based communication of MPI, it is generally not the case in the object based communication of Java. When objects are transferred between machines, the exact type (and size) of the objects can often not be determined during compile time (due to inheritance). It is also natural in Java to transfer graphs of objects, making it even harder to determine in advance how large a message is going to be. As a result, it is not possible to pre-allocate the objects or object graphs that are going to be received in a message. However, allocating them dynamically, when the message is deserialized, causes memory-allocation overhead, which reduces the throughput.

This is a general problem which is not only present in GMI, but also in RepMI and RMI. Even mpiJava suffers from this overhead when objects are transferred instead of arrays. When we look at application performance, however, we see that the GMI and mpiJava versions have similar speedups. This indicates that, at least for the application used here, the lower throughput of GMI has little influence on application performance.

We have also compared the performance of GMI to RepMI and RMI using six applications. In all six applications, the GMI version obtained a better speedup than

the other versions. In TSP the performance of GMI and RMI was almost identical. The two applications that use complex reduce-to-all and gather-to-all operations (LEQ and QR) show the biggest improvement in speedup. This shows that these operations can be expressed more efficiently in GMI than using RMI or RepMI.

5.5.4 Source Code Sizes

As in the previous chapter, we will give a brief evaluation of the source code complexity of GMI applications compared to their RMI, RepMI and mpiJava counterparts. For this comparison we will use the same approach as described in Section 4.4.3. We remove all comments and whitespace from the program source, and then count the number of characters required for the entire program. The results are shown in Tables 5.4 and 5.5.

application	*naive RMI*	*optimized RMI*	*RepMI*	*GMI*
TSP	100	106	100	100
ACP	100	114	100	100
ASP	100	168	98	83
QR	100	120	100	101
LEQ	100	135	97	89
weighed average	100	121	100	98

Table 5.4: Code sizes of the applications relative to the naive version, in %.

For the applications shown in Table 5.4 we show the code size relative to the *naive RMI* version from in the previous chapter. In this naive RMI version, shared data is encapsulated by a remote object without taking locality or RMI overhead into account. This approach results in small but inefficient applications. The *optimized RMI* versions use more complex communication algorithms that do take locality and RMI overhead into account. As a result, these versions are more efficient, but also larger.[10]

As Table 5.4 shows, the GMI versions of the applications have the same size or are smaller than the other versions. The only exception is QR, where the GMI version is slightly larger (1 %) that the naive RMI and RepMI versions. In general, the GMI versions are smaller, because GMI allows complex communication to be expressed with little effort. The optimized RMI version of ASP, for example, requires the implementation of a complex spanning tree broadcast algorithm. In contrast, the GMI version can directly use the efficient broadcast offered by Panda. As a result, the optimized RMI version of ASP is 68 % larger than the naive RMI version.

The GMI version of ASP is smaller than the naive RMI and RepMI versions, because GMI is able to express a *barrier* operation by using a combined invocation.

[10] We use the optimized RMI versions in the performance analysis shown above.

The other versions use an RMI-based barrier implementation instead, which increases their code size. Similarly, the GMI version of LEQ is smaller because GMI can express *gather-to-all* operations more efficiently.

application	*GMI*	*mpiJava*
ASP	100	51
LEQ	100	75
QR	100	86
LUFact	100	86
RayTrace	100	90
MolDyn	100	96
weighed average	100	85

Table 5.5: Code sizes of the applications relative to the GMI version, in %.

In Table 5.5 we compare the size of the GMI and mpiJava versions of the applications. The table shows that the mpiJava versions are, on average, 15 % smaller. Unlike mpiJava, GMI does not offer a complete communication library. Instead it only offers a communication infrastructure that allows the programmer to easily create application-specific communication. For example, mpiJava offers a *broadcast* operation that can be used directly. GMI does not offer a broadcast primitive, but instead allows the programmer to create a group of objects and define how each method invocation must be forwarded to this group. As a result, communication using GMI is more flexible that mpiJava communication, but also requires more source code to configure.

The code size difference is largest in simple applications, like ASP, where the mpiJava primitives can be used directly. For more complex applications the size difference is smaller because they contain more computation related code (thereby reducing the relative size of the communication related code). These applications may also require more complex communication which cannot be directly expressed by the mpiJava primitives. The mpiJava version of the MolDyn application, for example, requires six consecutive *reduce-to-all* operations. In the GMI version, this can be expressed using a single method invocation. As a result, the size difference between the mpiJava and GMI versions of MolDyn is only 4 %.

5.6 Related Work

As we have already described in Sections 3.6 and 4.5, there are many research projects that investigate parallel programming in Java. Although it is generally recognized that efficient communication mechanisms are a vital building block for high-performance Java [38, 39, 54, 84], most projects focus on the RMI point-to-point communication performance. We have shown in Chapter 3, however, that an efficient RMI is a good

basis for writing high-performance parallel applications, but it is not sufficient. Many parallel applications require more complex communication patterns that are hard to express efficiently using the synchronous point-to-point communication of RMI. Several projects exist that extend RMI with support for asynchronous communication.

Asynchronous RMI [87] (ARMI) extends the RMI model with support for delayed result handling. ARMI is compatible with the RMI standard. By using a special stub compiler (*armic*), the RMI stubs are replaced by stubs that support delayed result handling. The server-side skeleton objects are not changed. When an application thread invokes a method on an ARMI stub, it receives a *receipt* which can be used to collect the result later. Performance measurements performed by the authors indicate that when running a matrix multiplication on two machines, ARMI achieves a speedup of 30% over RMI implementation. No further performance data is available.

Active RMI [51] uses an approach similar to ARMI. It also allows objects to be created on remote machines and extends server objects with the ability to actively select the order in which they service incoming invocations. No performance data is available for Active RMI.

Reflective RMI [101] (RRMI), uses a reflection based approach to invoke remote methods, instead of the regular stub/skeleton based approach. This allows any object to be used remotely, not just objects for which stubs and skeletons have been created during compile time. RRMI requires the programmer to create *method descriptors* at run time, which contain all information necessary to invoke a method (e.g., method name, parameter values etc.). A remote method invocation can then be performed using the *invoke* and *invokeAsynchronous* call of a remote reference. These calls take a method descriptor as a parameter. The *invokeAsynchronous* call returns a handle that can later be used to retrieve the reply. It is not possible to discard the result. Very little performance information is available about RRMI. The authors only show that their approach has a slightly higher latency than RMI.

The *Ajents* [46] library uses a similar approach. Like in RRMI, any object in Ajents can be used remotely. Ajents does not require the creation of method descriptors. Instead, all the necessary parameters can be provided directly to the *Ajents.rmi* and *Ajents.armi* calls, which can be used to express an RMI. The *Ajents.armi* returns a *Future* object that can be used to retrieve the result. Ajents also supports remote creation of objects and object migration. In Ajents, an RMI is approximately 1.8 times faster than the a standard RMI implementation and they achieve a speedup of 7.6 when running a matrix multiply application on 8 machines.

Although the reflection-based approaches do increase the flexibility of remote invocations, they also increase the complexity of the applications by providing a different interface for calls to local and remote objects. In contrast, GMI only uses a reflection-like mechanism to configure the methods of a group reference. For method invocations, GMI uses the normal Java notation.

All of the platforms described above use a future-like approach to support asynchronous RMIs, where some special value is returned that allows the result to be

retrieved later. This is similar to the *forward result* scheme offered by GMI. However, GMI also allows the result value to be discarded at the server side. This has the advantage that no unnecessary messages are sent.

The systems described above only extend RMI with support for asynchronous invocations. Other systems, like *NinjaRMI*[11], and the one described in [56], also extend Java RMI with support for multicast invocations. However, NinjaRMI only offers unreliable multicast, while the system described in [56] shows very little performance improvement compared to using regular RMI calls.

CORBA

The Common Object Request Broker Architecture (CORBA) [81] is similar in functionality to RMI. CORBA uses a special interface definition language (IDL) to define remote interfaces. As with RMI, these interfaces are used to generate stubs and skeletons. Unlike RMI, however, CORBA is language independent; stubs and skeletons can be generated in any programming language and are able to inter operate regardless of their implementation language. This feature is important in commercial environments; it allows independent development of client and server software. For parallel programming, however, it is a disadvantage. Due to its language independence, CORBA does not integrate cleanly into any programming language. For example, parameters to remote invocations can only be of a type defined in the IDL language. Types native to the implementation language cannot be used. As a result, the communication mechanism in CORBA is essentially RPC. It does not support polymorphism in its parameter passing mechanism, a feature often used in Java programs (RMI does support this).

Besides synchronous two-way communication, CORBA also offers asynchronous one-way communication and *deferred synchronous two-way* communication, where the result of the invocation can be collected at a later time (this is similar to the *future* result handling offered by GMI). No group communication is provided. However, CORBA does allow *interceptor* functions to be inserted into a limited number of hooks in the runtime system. Interceptors allow the standard method invocation mechanism to be modified. They are applied to all methods that pass through that interception point. As a result, interceptors must be very general and able to handle any method invocation. Therefore, their usefulness for implementing complex group operations is limited. The functionality of GMI depends heavily on an interceptor-like scheme, which uses function objects to modify the behavior of invoked methods. The difference is that GMI uses modification functions that are specific to a single method of a single stub. This makes GMI's function objects more flexible and easier to implement.

Smart proxies [109] change the behavior of an application by extending the stubs generated by the IDL compiler. Unfortunately, implementing smart proxies is quite complex, because the programmer has to implement all communication code. For the complex group communication patterns as defined with GMI, this approach is

[11] http://www.cs.berkeley.edu/ mdw/proj/ninja/ninjarmi.html

hardly feasible. With GMI, the stub is completely compiler generated (including all communication code). The programmer only needs to configure the stub at runtime, using function objects if necessary.

Several systems exist that try to make CORBA more suitable for parallel programming. PARDIS [52] extends CORBA with the notion of an SPMD object, which consists of computing threads that each perform a part of the computation required for an invocation. Using an addition to the IDL, the programmer can specify the data distribution among the computation threads. In PARDIS, CORBA is only used to provide a convenient interface to an already parallel program. The computation threads in the SPMD object use some other communication package (e.g., MPI) to communicate with each other. As a result, PARDIS programs must be written using a mix of communication libraries (e.g., CORBA and MPI), making the implementation more complicated. Similar proposals to extend CORBA can be found in [80, 90].

Alternative models

An alternative to using a method invocation based communication model is to use a communication model outside Java's object model, like MPI-style message passing. There are several projects that use this approach [5, 19, 27, 36, 74]. The advantage of using MPI-style message passing is that many programmers are familiar with MPI and that many parallel applications exist that are based on MPI communication. MPI supports a rich set of communication styles, in particular collective communication. We will briefly describe three of these projects, *MPJ* [19], *mpiJava* [5], and *CCJ* [74].

MPJ was developed by the Java Grande Forum, and aims to provide an MPI-like message passing interface to Java. MPJ can either be implemented purely in Java, or as a Java wrapper to an existing native library. A pure-Java implementation has the advantage that it is highly portable and is able to run on any JVM without modification. Unfortunately, no complete pure Java implementation of MPJ is available yet.[12]

MpiJava is an example of a Java binding to an existing MPI library. Efficient MPI implementations are available on most platforms, and the micro benchmarks of Section 5.5 show that the overhead of adding a Java binding is small. Therefore, mpiJava can be expected to run efficiently on most platforms.

A disadvantage of using MPI-style message passing is that the MPI model does not integrate cleanly into the Java object model. While communication between Java objects (even in sequential programs) is expressed using method invocations, mpiJava and MPJ are based on explicit communication statements. In addition, MPI-style communication primitives are primarily designed for transmitting arrays of primitive data types (e.g., doubles), not for handling the complex object data structures often used in Java (e.g., lists and graphs). For collective communication, MPI uses an SPMD programming model based on groups of processes. This model requires all processes to run in lock-step.

[12] Information on MPJ can be found on http://www.dsg.port.ac.uk/projects/research/mpj/index.html.

The CCJ library tries to reduce these disadvantages by implementing MPI-like collective operations using an interface that fits better into Java's object-oriented model. However, CCJ is still conceptually close to MPI. In CCJ, communication is based on passing objects, not just arrays. Although CCJ still uses an SPMD programming model, it is based on groups of threads rather than groups of processes. CCJ implements only a limited set of collective operations and is implemented using RMI.

Because CCJ is implemented using RMI, it is unable to exploit efficient low-level communication primitives. This significantly reduces its performance. Performance measurements of CCJ using Manta RMI and the DAS cluster show that GMI outperforms CCJ, both using micro benchmarks and application kernels. For example, the QR application using CCJ achieves a speedup of 41 on 64 machines, compared to 47 for GMI. LEQ only achieves a speedup of 16 when using CCJ, compared to 45 for GMI.

Although inspired by the collective communication offered by MPI and CCJ, GMI provides a better integration of group communication into Java's object model. In GMI, communication is expressed using method invocations on a group of objects, instead of explicit communication statements. These GMI methods can be used to transfer any (serializable) data type by simply passing them as method parameters. In contrast, mpiJava and MPJ always require the data to be stored in an array, while CCJ requires it to be encapsulated by an object. While mpiJava, MPJ and CCJ are primarily designed for SPMD-style applications, the GMI model is general enough to support SPMD-style applications, client-server style applications, and even a mix of both.

Communication in *JavaNOW* [100] uses Linda-like tuple spaces [35]. Named data items can be inserted into such a tuple space, where they remain until some process chooses to receive them. JavaNOW extends this model by implementing several MPI-like collective operations on these data items. Data items can be broadcast or scattered to different tuple spaces, or gathered from different tuple spaces into one destination space. No performance data is available on JavaNOW.

The *ProActive* [4] system is based on the concept of *Active Objects*, which seems to be similar to the *Clouds* introduced in the previous chapter. An active object consists of a graph of objects with one designated *root*. This root is the only object capable of receiving remote invocations. ProActive allows groups of identical (active) objects to be created and distributed over a number of machines. Invocations on such a group will be broadcast to all group members. A *Future* object is returned that collects the result values of all invocations, allowing the invoking thread to continue immediately. When a group of objects is passed as a parameter to a method invocation on a different group, it is also possible to distribute these parameter objects in a round-robin fashion over the destination group instead of forwarding the entire group to every destination. ProActive is implemented using RMI.

Although the ProActive system clearly resembles GMI, there are several major differences. The most obvious difference is that ProActive does not allow the pro-

grammer to configure how the method invocations and result values are handled. Invocations are always broadcast to all objects in a group and result values are always gathered into a *Future* object. Due to this limited communication model, ProActive is not able to express *reduce*, *reduce-to-all* or *gather-to-all* style communication. In contrast, GMI allows each method to be configured to use a wide range of forwarding and result handling schemes. Although ProActive does allow a *scatter* like forwarding of methods, it can only scatter one group of objects onto another. GMI supports a personalized invocation with any type and number of parameters. In ProActive, all objects in a group must have the same type (although the use of inheritance is allowed). In GMI, any type of object may be part of a group, as long as all objects implement the same group interface. Finally, because ProActive is implemented using RMI, it can not exploit efficient low-level communication primitives, something which GMI was specifically designed to do.

Data-parallel languages emphasize the use of (distributed) data structures. Several examples of object-oriented data-parallel languages exists, such as Data-Parallel Orca [43], pC++ [14], Illinois Concert C++ [23], C** [59], CHAOS++ [20] and Taco [78]. These languages allow the definition of special data structures that are automatically distributed over the available machines (possibly using some distribution defined by the programmer). Parallel computations can then be expressed by applying functions to these distributed data structures. These functions can then be applied, in parallel, to all the elements of the data structure. Some data parallel languages (e.g., CHAOS++, C** and Taco) also allow scatter, gather or reduce style operations to be applied to the distributed data structures.

Data-parallel languages provide a higher level model to the programmer than GMI. While data-parallel languages completely hide all communication from the programmer, GMI does the opposite. In GMI, we intentionally expose the forwarding and reply handling schemes to allow the programmer to select the schemes that are appropriate for the application. Data-parallel languages are best suited for *regular* problems, where the same operations are performed on all data items. GMI can also express *irregular* problems, where each data item requires different operations (e.g., the TSP application used in Section 5.5).

5.7 Conclusions

In this chapter, we have introduced a new model for group communication that generalizes RMI by supporting different schemes for forwarding method invocations and handling their results. The different forwarding and reply handling schemes can be combined. This allows a rich variety of useful communication primitives to be expressed in a way that cleanly integrates into Java RMI. The resulting model, Group Method Invocation (GMI), is highly expressive, easy-to-use, and efficient.

GMI supports invoking a method on single or multiple objects, optionally personalizing invocations for each destination or combining several method invocations into

a single one. It also supports various ways of handling the method result values, such as gathering, combining, returning, and forwarding. In GMI, every group method may be configured separately to use a combination of an invocation forwarding and reply handling scheme. GMI's expressiveness is illustrated by the fact that it can be used to express many existing communication primitives, such as RMI, asynchronous RMI, futures, and voting. It can be both used to express MPI-style collective operations, where all threads collectively call the same operation, as well as RMI-style group operations where a single thread invokes a method on multiple objects.

The communication model of GMI integrates cleanly into Java's object-oriented programming model. Existing Java bindings to external collective communication libraries, such as MPI, do not integrate cleanly into Java. They are designed for SPMD-style parallelism and use explicit communication statements. In contrast, GMI, like RMI, allows communication to be expressed using method invocations on (groups of) objects. No explicit send or receive statements are necessary. Nevertheless, GMI can be used to write SPMD-style parallel applications, where all threads run in lockstep. However, GMI is also capable of expressing client-server style applications, which are more asynchronous in nature. It can even be used to mix the two models.

Despite the expressiveness of GMI, the API is small. It contains methods for creating groups, creating group references, and configuring group methods. By extending a number of simple objects, the programmer is able to define application specific ways to personalize or combine method invocations and to gather, combine or forward results.

GMI was designed for high performance, in particular to exploit efficient low-level communication primitives (such as a broadcast). Using several micro benchmarks, we have evaluated the performance of GMI and compared it to that of mpiJava and MPI C. This comparison has shown that the latency of several collective operations using GMI and mpiJava are comparable. For example, on a Myrinet-based cluster, our GMI implementation can invoke a method on a group of 64 objects in about 98 μs. A mpiJava broadcast operation on 64 machines takes 91 μs to complete.

The micro benchmarks also show that the throughput of GMI is not as high as that of mpiJava. This problem is not limited to GMI, but is a general problem with serialization-based communication. However, performance measurements using six different applications show that this does not necessarily have a negative effect on application speedup. The GMI and mpiJava versions of the applications have a very similar performance.

We have also compared the application level performance of our GMI implementation to that of RepMI and RMI. These measurement show that GMI obtains the highest speedup for all six applications. Especially the applications that use complex collective operations, such as *reduce-to-all* or *gather-to-all*, benefit from the use of GMI.

In conclusion, this chapter describes an efficient, easy-to-use and expressive mechanism for adding group communication to Java.

Chapter 6

Conclusions

In this thesis, we have investigated how communication in Java can be made suitable for high-performance parallel programming on homogeneous cluster computers. Our goal was to design a platform that provides highly efficient communication, preferably using communication models that integrate cleanly into Java and are easy to use. For this reason, we have taken the existing RMI model as a starting point in our work.

We have given a description of the RMI model and analyzed the performance of several RMI implementations to evaluate their suitability for high-performance parallel programming. This analysis showed that these existing RMI implementations are not efficient enough to fully utilize a high-performance network (such as Myrinet). This resulted in poor application speedups.

To solve this problem, we designed and implemented *Manta RMI*, an alternative, high-performance RMI implementation that is specifically optimized for parallel programming on a homogeneous cluster computer. Manta RMI introduces little overhead, and can therefore fully benefit from performance offered by a high-performance network. We showed that using Manta RMI significantly increases the speedup of parallel applications.

However, it was also clear that some applications cannot be expressed efficiently using the limited model of RMI (i.e., synchronous point-to-point communication). For example, using RMI to express shared data or group communication is often cumbersome and inefficient, especially on large numbers of machines. For these applications, we have introduced two new communication models: Replicated Method Invocation (RepMI) and Group Method Invocation (GMI).

RepMI is specifically designed to express shared data using replicated objects. Adding object replication to Java is a non-trivial task, however. Unlike other languages (like Orca [6]) the Java language was not designed to support object replication. Java's support for arbitrary graphs of objects, polymorphism, and the ability of methods to block at any point make the design and implementation of RepMI significantly more complex. The RepMI model that we have presented in this thesis solves these

problems using a combination of compiler analysis, generated code, runtime system support, and by deliberately imposing restrictions on the data that may be replicated.

We have introduced the concept of *clouds*: closed groups of objects that are replicated as a whole. Using clouds, complex data structures can be replicated efficiently. We have shown that several restrictions must be applied to objects in a cloud to ensure that all replicas remain consistent. Using replication aware thread scheduling, we correctly implement arbitrarily blocking methods in replicated clouds. To determine what methods may be executed locally and what methods have to be forwarded to all replicas, read/write analysis is required. However, due to Java's support for polymorphism, it is not possible to completely perform read/write analysis at compile time. We have solved this problem by using compiler generated code that is able to perform the final read/write analysis at run time. Using several application kernels, we have shown that RepMI outperforms manually optimized RMI applications, while simultaneously reducing the application complexity.

To allow applications to efficiently express group and collective communication, we have introduced the GMI model. In GMI, a (distributed) group of objects can be addressed using a single *group reference*. GMI allows applications to dynamically select different ways of forwarding the method invocations to the group. Applications can also specify how the method result must be handled. Several different forwarding and reply handling schemes are available and can be combined. This makes GMI a highly expressive model.

In GMI we have also introduced the notion of a *combined method invocation*. In a combined method invocation, several threads collectively invoke the same method on a group reference. These method invocations are then combined into a single invocation (using a user-defined method) that is then forwarded to the group. The notion of a combined method invocation allows a clean integration of collective communication into the Java object model.

GMI's expressiveness is illustrated by the fact that it can be used to express many existing communication primitives, such as RMI, asynchronous RMI, broadcast, scatter, reduce, gather, futures, voting, etc. It can be used to express MPI-style collective operations, where all threads collectively call the same operation, as well as RMI-style group operations where a single thread invokes a method on multiple objects.

Performance measurements using a Manta-based GMI implementation show that GMI allows complex group operations to be implemented efficiently. Although GMI is not as efficient as mpiJava (a Java binding to an external MPI library), the differences are usually small. The application level performance of GMI and mpiJava are almost identical.

We have also compared the application level performance of GMI to RepMI and Manta RMI. For applications, the GMI version obtains a better speedup than the other versions. The applications that use complex reduce-to-all and gather-to-all operations show the biggest improvement in speedup. These are exactly the type of operations that GMI was designed to support.

The combination of these three models, RMI, RepMI and GMI, provides a platform that integrates cleanly into the Java language, is highly expressive, and can be implemented efficiently. It is therefore a suitable platform for high-performance parallel programming in Java.

Bibliography

[1] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik - A Distributed JVM Featuring a Single Address Space Architecture. In *Proc. of the USENIX JVM Research and Technology Symposium Work-in-progress Session*, Monterey, California, April 2001.

[2] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling Multithreaded Java Bytecode for Distributed Execution. In *Proc. of Euro-Par 2000*, Lecture Notes in Computer Science 1900, pages 1039–1052, München, Germany, August 2000. Springer.

[3] Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proc. of the International Conference on Parallel Processing (ICPP'99)*, pages 4–11, Fukushima, Japan, September 1999.

[4] L. Baduel, F. Baude, and D. Caromel. Efficient, Flexible and Typed Group Communications for Java. In *Proc. of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, Washington, November 2002. ACM Press.

[5] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and S. Lim. mpiJava: An Object-Oriented Java Interface to MPI. In *Proc. of the International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, Lecture Notes in Computer Science 1586, pages 748–762, San Juan, Puerto Rico, April 1999. Springer.

[6] H.E. Bal, R.A.F. Bhoedjang, R.F.H 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.

[7] H.E. Bal, R.A.F. Bhoedjang, R.F.H. 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.

[8] 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 *Proc. of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, Copper Mountain, CO, 1995.

[9] R.A.F. Bhoedjang. *Communication Architectures for Parallel-Programming Systems*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, June 2000.

[10] R.A.F. Bhoedjang and K.G. Langendoen. Friendly and Efficient Message Handling. In *Proc. of the 29th Annual Hawaii International Conference of System Sciences (HICSS-29)*, pages 121–130, January 1996.

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

[12] D. Blackston and T. Suel. Highly Portable and Efficient Implementations of Parallel Adaptive N-Body Methods. In *Proc. of the 1997 ACM/IEEE Conference on Supercomputing (SC'97)*, San Jose, CA, November 1997.

[13] 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, January 1995.

[14] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[15] F. Breg. *Java for High Performance Computing*. PhD thesis, Leiden University, November 2001.

[16] 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 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 214–224, Seattle, WA, June 1997.

[17] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, and R.A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12:375–388, May 2000.

[18] M.G. Burke, J-D. Choi, S. Fink, D. Grove, M.Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proc. of the ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.

[19] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, September 2000.

[20] C. Chang, A. Sussman, and J. Saltz. CHAOS++. In G.V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Scientific and Engineering Computation Series, chapter 4, pages 131–174. MIT Press, 1996.

[21] C-C. Chang and T. von Eicken. A Software Architecture for Zero-Copy RPC in Java. Technical Report 98-1708, Cornell University, September 1998.

[22] C-C. Chang and T. von Eicken. Interfacing Java with the Virtual Interface Architecture. In *Proc. of the ACM 1999 Java Grande Conference*, pages 51–57, San Francisco, CA, June 1999.

[23] A. Chien and J.T. Dolby. ICC++. In G.V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Scientific and Engineering Computation Series, chapter 9, pages 343–382. MIT Press, 1996.

[24] D.D. Clark. The Structuring of Systems Using Upcalls. In *Proc. of the 10th Symp. on Operating Systems Principles*, pages 171–180, Orcas Island, WA, December 1985.

[25] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of the 1993 ACM/IEEE Conference on Supercomputing (SC'93)*, pages 262–273, Portland, Oregon, November 1993.

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

[27] K. Dincer. Ubiquitous Message Passing Interface implementation in Java: JMPI. In *Proc. of the 13th International Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing*. IEEE, 1998.

[28] P. Doyle and T. Abdelrahman. Jupiter: A Modular and Extensible JVM. In *Proc. of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing (ICS'01)*, pages 37–48. ACM Press, June 2001.

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

[30] A. Fekete, M.F. Kaashoek, and N.A. Lynch. Implementing Sequentially Consistent Shared Objects Using Broadcast and Point-to-Point Communication. In *Proc. of the International Conference on Distributed Computing Systems*, pages 439–449, 1995.

[31] P. Felber and R. Guerraoui. Programming with Object Groups in CORBA. *IEEE Concurrency*, 8(1):48–48, January-March 2000.

[32] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA Objects. *Advances in Distributed Systems*, pages 254–276, 2000.

[33] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3-4), 1994.

[34] E.M. Gagnon and L.J. Hendren. SableVM:A Research Framework for the Efficient Execution of Java Bytecode. In *Proc. of the USENIX JVM Research and Technology Symposium*, pages 27–39, Monterey, California, April 2001.

[35] D. Gelernter. Generative Communication in Linda. *Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[36] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, September-November 1998.

[37] V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Proc. of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*, Portland, OR, November 1999.

[38] V. Getov and M. Philippsen. Java Communications for Large-Scale Parallel Computing. In *Proc. of the 3rd International Conference on Large Scale Scientific Computing*, Lecture Notes in Computer Science 2179, pages 33–35, Sozopol, Bulgaria, June 2001. Springer.

[39] V. Getov, G. von Laszewski, M. Philippsen, and I. Foster. Multiparadigm Communications in Java for Grid Computing. *Communications of the ACM*, 44:118–125, October 2001.

[40] P.A. Gray and V.S. Sunderam. IceT: Distributed Computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1167, November 1997.

[41] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *Proc. of Middleware'98*, pages 339–354, The Lake District, England, September 1998. Springer.

[42] M. Haines and K.G. Langendoen. Platform-Independent Runtime Optimizations using OpenThreads. In *Proc. of the 11th International Parallel Processing Symposium (IPPS'97)*, pages 460–466, Geneva, Switzerland, April 1997.

[43] S. Ben Hassen, H.E. Bal, and C. Jacobs. A Task and Data Parallel Programming Language based on Shared Objects. *ACM. Trans. on Programming Languages and Systems*, 20(6):1131–1170, November 1998.

[44] Y.C. Hu, W. Yu, A.L. Cox, D.S. Wallach, and W. Zwaenepoel. Runtime Support for Distributed Sharing in Strongly Typed Languages. Technical report, Rice University, 1999.

[45] N.C. Hutchinson, L.L. Peterson, M.B. Abbott, and S. O'Malley. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proc. of the 12th ACM Symp. on Operating System Principles*, pages 91–101, Litchfield Park, AZ, December 1989.

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

[47] D.B. Johnson and W. Zwaenepoel. The Peregrine High-Performance RPC System. Technical Report TR91-151, Rice University, March 1991.

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

[49] 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 *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, 1997.

[50] V. Karamcheti and A.A. Chien. Concert - Efficient Runtime Support for Concurrent Object-Oriented. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC'93)*, pages 598–607, Portland, Oregon, November 1993.

[51] M. Karaorman and J. Bruno. Active-RMI: Active Remote Method Invocation System for Distributed Computing using Active Java Objects. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '98)*, pages 414–427, Santa Barbara, California, August 1998.

[52] K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-Level Parallel Distributed Computation. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC'97)*, pages 1–14, San Jose, November 1997.

[53] 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.

[54] T. Kielmann, P. Hatcher, L. Bougé, and H.E. Bal. Enabling Java for High-Performance Computing. *Communications of the ACM*, 44:110–117, October 2001.

[55] A. Krall and R. Grafl. CACAO: A 64 bit JavaVM Just-in-Time Compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, November 1997.

[56] 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.

[57] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

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

[59] J.R. Larus, B. Richards, and G. Viswanathan. C**. In G.V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, Scientific and Engineering Computation Series, chapter 8, pages 297–342. MIT Press, 1996.

[60] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *Proc. of the 1999 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 447–460, Denver, CO, October 1999.

[61] M.J.M. Ma, C-L. Wang, and F.C.M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[62] J. Maassen, T. Kielmann, and H.E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13:681–712, July-August 2001.

[63] J. Maassen, T. Kielmann, and H.E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *In proceedings of LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 1–6, Washington DC, March 2002.

[64] J. Maassen and R.V. van Nieuwpoort. *Fast Parallel Java*. Master's thesis, Vrije Universiteit Amsterdam, September 1998.

[65] J. Maassen, R.V. van Nieuwpoort, R.S. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R.F.H Hofman. Efficient Java RMI for Parallel Programming. *ACM. Trans. on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

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

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

[68] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT A Reflective Java JIT Compiler. In *Proc. of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, BC, 1998. ACM Press.

[69] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Building Reliable Client-Server Software Using Actively Replicated Objects. In *Proc. of International Conference on Technology of Object Oriented Languages and Systems (TOOLS '95)*, Versailles, France, March 1995. Prentice Hall.

[70] M. Migliardi and V. Sunderam. Networking Performance for Distributed Objects in Java. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2001.

[71] J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, P. Wu, and G. Almasi. The NINJA project. *Communications of the ACM*, 44(10):102–109, October 2001.

[72] 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, Seattle, Washington, October 1996. ACM Press.

[73] 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'97)*, Portland, OR, June 1997.

[74] A. Nelisse, J. Maassen, T. Kielmann, and H.E. Bal. CCJ: Object-based Message Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15:341–369, March-April 2003.

[75] R.V. van Nieuwpoort. *Efficient Java-Centric Grid-Computing*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, September 2003. (To be published).

[76] R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Euro-Par 2000 Parallel Processing*, Lecture Notes in Computer Science 1900, pages 690–699, Munich, Germany, August 2000. Springer.

[77] R.V. van Nieuwpoort, T. Kielmann, and H.E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, Snowbird, UT, June 2001.

[78] J. Nolte, M. Sato, and Y. Ishikawa. Template Based Structured Collections. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 483–491, Cancun, Mexico, 2000.

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

[80] Object Managment Group. *CORBA,Parallel Processing RFP*. June 1999. http://www.omg.org.

[81] Object Managment Group. *The Common Object Request Broker: Architecture and Specification, 2.3 ed.* June 1999. http://www.omg.org.

[82] M. Paleczny, C. Vick, and C. Click. The Java Hotspot Server Compiler. In *Proc. of the USENIX JVM Research and Technology Symposium*, pages 27–39, Monterey, California, April 2001.

[83] M. Philippsen. Bandwidth, Latency, and other Problems of RMI and Serialization, May 1998.

[84] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[85] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[86] 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 *Proc. of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, OR, 1997.

[87] R.R. Raje, J.I. William, and M. Boyles. Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[88] K. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[89] Y. Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[90] C. René and T. Priol. MPI Code Encapsulation using Parallel CORBA Object. In *Proc. of the IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 3–10, Redondo Beach, CA, August 1999.

[91] R. van Renesse, J.M. van Staveren, and A.S. Tanenbaum. Performance of the Amoeba Distributed Operating System. *Software — Practice and Experience*, 19:223–234, March 1989.

[92] 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.

[93] S.H. Rodrigues, T.E. Anderson, and D.E. Culler. High-Performance Local Communication With Fast Sockets. In *Proc. of the USENIX Annual Technical Conference '97*, Anaheim, CA, January 1997.

[94] T. Rühl. *Collective Computation in Object-based Parallel Programming Languages*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, November 2000.

[95] T. Rühl and H.E. Bal. Synchronizing Operations on Multiple Objects. In *Proc. of the 2nd Workshop on Runtime Systems for Parallel Programming*, Orlando, FL, March 1998.

[96] M.D. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Trans. on Computer Systems*, 8(1):1–17, February 1990.

[97] Sun Microsystems. *Object Serialization Specification*, JDK 1.1 FCS, Online at http://java.sun.com edition.

[98] Sun Microsystems. *Java Remote Method Invocation Specification*, JDK 1.1 FCS, Online at http://java.sun.com edition, February 1997.

[99] C.A. Thekkath and H.M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Trans. on Computer Systems*, 11(2):179–203, May 1993.

[100] G.K. Thiruvathukal, P.M. Dickens, and S. Bhatti. Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience*, 12:1093–1116, September 2000.

[101] G.K. Thiruvathukal, L.S. Thomas, and A. T. Korczynski. Reflective Remote Method Invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, September - November 1998.

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

[103] R.S. Veldema. *Compiler and Runtime Optimizations for Fine Grained Distributed Shared Memory Systems*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2003. (To be published).

[104] R.S. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed Shared Memory Management for Java. In *Proc. of the 6th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2000)*, pages 256–264, Lommel, Belgium, June 2000.

[105] R.S. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, and H.E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Proc. of the ACM 2001 Java Grande Conference*, June 2001.

[106] R.S. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, C.J.H. Jacobs, and H.E. Bal. Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems. In *Proc. of PPoPP-2001 Symposium on Principles and Practice of Parallel Programming*, pages 83–93, June 2001.

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

[108] 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. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 217–226, Santa Barbara, CA, July 1995.

[109] N. Wang, K. Parameswaran, and D. C. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, San Antonio, 2001.

[110] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, 12(7):519–538, May 2000.

[111] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[112] 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 *Proc. of the ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, 1998.

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

[114] W. Zhu, C-L. Wang, and F.C.M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.

Samenvatting

De laatste jaren wordt de programmeertaal Java in toenemende mate gebruikt voor het schrijven van rekenintensieve parallelle programma's. In dit soort programma's worden tegelijkertijd meerdere computers gebruikt om gezamenlijk een complexe berekening uit te voeren. Java is een moderne, objectgeoriënteerde programmeertaal met automatisch geheugen beheer en uitgebreide ondersteuning voor parallellisme, communicatie, beveiliging, en heterogeniteit (d.w.z. dat Java programma's op de meeste computers werken, ongeacht welke hardware of besturingssysteem er gebruikt wordt). Daarnaast worden Java programmeeromgevingen standaard geleverd met een uitgebreide software bibliotheek. Dit alles zorgt ervoor dat Java een aantrekkelijke programmeertaal is voor het ontwikkelen van grote (eventueel parallelle) programma's.

Standaard biedt de Java programmeeromgeving twee manieren aan om parallelle programma's te maken. Voor *shared-memory* machines (computers met meerdere processoren die samen één geheugen delen), kunnen programma's geschreven worden die gebruik maken van meerdere threads. Threads zijn (semi-) onafhankelijke programma onderdelen die tegelijkertijd kunnen worden uitgevoerd. Daarnaast bied Java het *Remote Method Invocation* (RMI) mechanisme aan. Met deze *methode aanroep op afstand* kan een Java programma op de ene computer methoden aanroepen van een Java programma op een andere computer. Alle parameters en resultaten van een RMI worden automatisch van de ene naar de andere computer verstuurd. Met behulp van RMI is het dus mogelijk programma's te schrijven die meerdere onafhankelijke computers laten samenwerken. In tegenstelling tot het thread model is het niet nodig dat deze computers een gezamenlijk geheugen hebben. Een netwerkverbinding is voldoende. Dit soort systemen worden ook wel aangeduid als *distributed-memory* machines.

Op zich is het op *methode aanroep* gebaseerde communicatie model van RMI niet nieuw. Het oudere Remote Procedure Call (RPC) principe biedt een vergelijkbaar model. RMI kan dan ook gezien worden als een objectgeoriënteerde versie van RPC. Als gevolg van deze objectgeoriënteerdheid heeft RMI diverse eigenschappen die niet in RPC te vinden zijn. Het opvallendste verschil is dat RMI ondersteuning biedt voor polymorphisme.

In objectgeoriënteerde talen zoals Java is het mogelijk dat een actuele parameter van een methode aanroep een subtype is van de formele parameter. Dit betekent dat het

van tevoren niet helemaal zeker is wat voor type objecten er als parameter doorgegeven gaan worden.

Deze eigenschap heeft verstrekkende gevolgen voor de communicatie code die in RMI gebruikt wordt om de parameters tussen computers te transporteren. In tegenstelling tot RPC, waar van de voren precies bekend is welke type gegevens er verstuurd of ontvangen moeten worden en hoe groot deze gegevens zijn, kan dit bij RMI pas op het laatste moment bepaald worden. Pas tijdens het inpakken van de parameters bij de verzender (de zgn. *serializatie*) en het uitpakken van het parameters bij de ontvanger (de zgn. *deserializatie*) wordt er bekend wat er precies verstuurd moet worden. Het kan zelfs voorkomen dat de ontvanger een type object ontvangt die hij nog nooit eerder gezien heeft. In zo'n geval moet eerst de programma code van het desbetreffende type object opgezocht en ingeladen worden voordat de RMI parameter kan worden uitgepakt (dit heet *dynamic class loading*). Deze objectgeoriënteerde eigenschappen maken de RMI implementatie aanzienlijk complexer dan die van RPC.

Het op methode aanroep gebaseerde communicatie model van RMI lijkt sprekend op de communicatie die plaats vindt tussen normale objecten van een (niet parallel) Java programma. Ook daar communiceren de objecten door middel van het aanroepen van elkaars methoden. Als gevolg hiervan past RMI uitstekend in het programmeer model van Java. Wij zijn daarom van mening dat het RMI model een goed uitgangspunt vormt voor het ontwikkelen van programmeer modellen die het schrijven van rekenintensieve parallelle programma's in Java kunnen vereenvoudigen.

Helaas leverden de oorspronkelijke Java implementaties inferieure prestaties, zowel voor sequentiële Java code als voor de communicatie snelheid van RMI. Beiden vormen een ernstige belemmering voor het toepassen van Java voor rekenintensieve parallelle programma's op distributed-memory machines.

Gelukkig is er in de afgelopen jaren veel onderzoek gedaan naar het verbeteren van de prestaties van de verschillende Java implementaties. Door de oorspronkelijke interpretatie technieken te vervangen door just-in-time ("net op tijd") compilers, traditionele compilers, en gespecialiseerde hardware, is er veel vooruitgang geboekt. Als gevolg hiervan doet de snelheid van sequentiële Java code tegenwoordig nauwelijks meer onder voor de snelheid van traditioneel gecompileerde talen zoals C, C++ en Fortran.

De prestaties van de verschillende RMI implementaties hebben niet zo'n stormachtige ontwikkeling doorgemaakt. Hoewel de prestaties wel verbeterd zijn, zijn ze nog niet goed genoeg voor parallelle programma's. Het probleem is dat RMI oorspronkelijk niet bedoeld is voor parallelle programma's. In plaats daarvan is RMI ontworpen voor zogenaamde *client-server* programma's die gebruikt worden in een gedistribueerde en heterogene omgeving. Hierbij kan bijvoorbeeld gedacht worden aan toepassingen zoals internet winkels en telebankieren. Bij dit soort toepassingen is het van belang dat de programma's goed werken op uiteenlopende combinaties van hardware en besturingssystemen, dat ze veilig zijn in het gebruik, en dat er eventueel meerdere versies van het programma in omloop kunnen zijn. De absolute commu-

nicatie snelheid is in dit soort programma's van ondergeschikt belang, ook omdat de communicatie vaak over langere afstanden (b.v. het internet) of tragere lijnen (b.v. modems) plaatsvindt. Communicatie tijden in de orde van enkele milliseconden zijn dan heel normaal.

Voor parallelle programma's worden daarentegen meestal *sterk verbonden homogene cluster computers* gebruikt. Dit zijn systemen die bestaan uit een (groot) aantal identieke computers die onderling verbonden zijn door een zeer snel netwerk. Bij dit soort systemen is de ondersteuning van heterogeniteit, beveiliging en verschillende versies nauwelijks van belang. In plaats daarvan draait alles om communicatie- en rekensnelheid. Bestaande RMI implementaties zijn dus niet efficiënt genoeg voor het maken van parallelle programma's op dit soort systemen.

Parallelle programma's maken ook vaak gebruik van complexe communicatie vormen waar meer dan twee computers bij betrokken zijn. Voorbeelden hiervan zijn *broadcast* (zenden naar iedereen), *scatter* (het verdelen van gegevens over alle computers), en *gather* (het verzamelen van gegevens die over meerdere computers verspreid staan). Helaas biedt het huidige RMI model alleen ondersteuning voor communicatie tussen twee computers. Met behulp van meerdere RMI's kunnen deze complexere vormen van communicatie weliswaar uitgedrukt worden, maar dit is over het algemeen omslachtig en minder efficiënt.

In dit proefschrift, met de titel *Methode Aanroep Gebaseerde Communicatie Modellen voor Parallel Programmeren in Java* laten we zien dat het mogelijk is een RMI implementatie te creëren die efficiënt genoeg is om gebruikt te worden voor parallelle programma's op homogene cluster computers. Daarna gebruiken we onze RMI implementatie als een basis voor de ontwikkeling van twee nieuwe modellen, die geschikt zijn om complexere vormen van communicatie uit te drukken.

De bovenstaande introductie wordt ook gegeven in hoofdstuk 1. In dit hoofdstuk beschrijven we ook *Manta*, het Java platform dat als basis dient voor ons onderzoek. In tegenstelling tot veel andere Java platformen maakt Manta gebruik van een traditionele compiler in plaats van een just-in-time compiler. Ten slotte geven we een beschrijving van de *Distributed ASCI Supercomputer* (DAS), de homogene cluster computer die we gebruiken voor onze experimenten. De verschillende machines van de DAS zijn met elkaar verbonden door een snel Myrinet netwerk.

In hoofdstuk 2 geven we een uitgebreide beschrijving van het RMI model en de RMI syntax. Ook beschrijven we een RMI implementatie en laten we zien hoe serialization werkt. Serialization wordt gebruikt om de parameters en resultaten van RMI's te transporteren. Met behulp van een aantal testprogramma's vergelijken we de prestaties van de serialization en RMI implementaties van 5 verschillende Java platformen, Sun JDK 1.2 en 1.4, IBM JDK 1.1.8 en 1.3.1 en *Compiled Sun*. Compiled Sun is een RMI implementatie gebaseerd op Sun JDK 1.1, die gecompileerd is met de Manta compiler. Ook gebruikt Compiled Sun de serialization implementatie van Manta. Deze is volledig geïmplementeerd in C, in plaats van Java.

De resultaten van de testprogramma's laten zien dat geen van de vijf implemen-

taties in staat is het snelle Myrinet netwerk van de DAS efficiënt te gebruiken. Zelfs de snelste van de vijf implementaties, Compiled Sun, gebruikt in het gunstigste geval slechts 41% van de beschikbare netwerk bandbreedte. Ook duurt het uitvoeren van een RMI naar een andere machine ten minste 301 microseconden, bijna 10 maal zo lang als de minimale tijd die nodig is om een netwerk bericht te versturen.

Aparte testprogramma's laten zien dat de bandbreedte van serializatie al beperkt is. Zelfs de op C gebaseerde serializatie implementatie van Manta is bij het versturen van arrays van integer of double getallen niet snel genoeg om meer dan de helft van de beschikbare bandbreedte van het Myrinet netwerk te benutten. Voor complexere datastructuren, zoals binaire bomen, word slechts 1% van de beschikbare bandbreedte benut.

Vervolgens gebruiken we zes parallelle programma's om de prestaties van de Compiled Sun implementatie verder te evalueren. Hoewel deze implementatie de beste resultaten liet zien bij de eerdere testprogramma's, zijn slechts twee van de zes programma's in staat een acceptabel prestatie niveau te halen op 32 machines. De conclusie van hoofdstuk 2 luidt dan ook dat RMI een aantrekkelijk model biedt voor parallel programmeren in Java, maar dat het moeilijk is acceptabele prestaties te halen vanwege de ingewikkelde implementatie.

In hoofdstuk 3 geven we een uitgebreide beschrijving van *Manta RMI*, een RMI en serializatie implementatie die specifiek ontworpen is voor parallel programmeren op homogene cluster computers. Bij deze implementatie genereert de compiler speciale RMI en serializatie routines, die, in combinatie met een snelle communicatie bibliotheek, zorgen voor een significante verbetering van de RMI prestaties. In het gunstigste geval is Manta RMI in staat 87% van de (Myrinet) netwerk bandbreedte te gebruiken. De minimale tijd die nodig is voor een RMI naar een andere machine is bij Manta RMI slechts 37 microseconden, 8 maal zo snel als Compiled Sun RMI, en slechts 6 microseconden langzamer dan de minimale tijd die nodig is om een netwerk bericht te versturen. Ook op applicatie niveau presteert Manta RMI aanzienlijk beter. Vijf van de zes programma's (die ook in hoofdstuk 2 gebruikt werden) halen op 32 machines met Manta RMI een goede tot uitstekende prestatie.

Helaas laten uitgebreidere metingen ook zien dat de prestaties van sommige programma's op 64 machines niet optimaal zijn. Dit komt voornamelijk voor bij programma's die gebruik maken van complexere communicatie vormen waarbij meer dan twee machines betrokken zijn. Voor dit soort programma's vormt RMI een beperking, omdat het alleen ondersteuning bied voor communicatie tussen twee computers. Complexere communicatie vormen kunnen weliswaar met RMI gesimuleerd worden, maar dit is omslachtig en minder efficiënt, zoals de applicatie metingen laten zien. In hoofdstuk 4 en 5 gebruiken we daarom onze RMI implementatie als een basis voor de ontwikkeling van twee alternatieve modellen die ondersteuning bieden voor complexere vormen van communicatie.

In hoofdstuk 4 introduceren we *Replicated Method Invocation* (RepMI). Dit is een nieuwe manier om object replicatie toe te passen in Java. Omdat het programmeer

model van RepMI veel overeenkomsten vertoont met het RMI model, is het redelijk eenvoudig om bestaande parallelle RMI programma's te vertalen naar RepMI. Door opzettelijk enkele beperkingen in het RepMI model aan te brengen ontstaat er een duidelijk programmeer model dat efficiënt geïmplementeerd kan worden.

Met behulp van RepMI kunnen gesloten grafen van objecten (zgn. *clouds*) gerepliceerd worden. Elke cloud heeft slechts één object, de *root*, dat als toegang dient tot de graaf. De root is het enige object in de graaf waarop objecten van buiten de graaf methoden op aan mogen roepen.

We laten zien dat het door een combinatie van compiler analyse en run-time ondersteuning mogelijk is om te bepalen welke methodes de gerepliceerde graaf veranderen (zgn. schrijf operaties), en welke slechts waarden uit de graaf lezen (zgn. lees operaties). Door de schrijf operaties met behulp van een totaal geordende broadcast operatie naar alle replica's te sturen kunnen deze consistent gehouden worden. De lees operaties kunnen rechtstreeks op de locale replica uitgevoerd worden, zonder dat er enige communicatie nodig is. In het hoofdstuk geven we ook een uitgebreide beschrijving van de beperkingen die door het RepMI model opgelegd worden, en de voorwaarden waar de RepMI implementatie aan moet voldoen om er zeker van te zijn dat alle replica's van een graaf consistent blijven.

Vervolgens gebruiken we een aantal kleine testprogramma's en vijf parallelle programma's om de efficiëntie van onze RepMI implementatie in Manta te beoordelen. Uit deze tests blijkt dat RepMI uitstekend geschikt is voor het uitdrukken van gedeelde data en broadcast communicatie. Het veranderen van 64 objecten (op verschillende machines) bijvoorbeeld, duurt met RepMI slechts 155 microseconden, ongeveer even lang als vier RMI's. Ook laten we zien dat RepMI programma's even goed of zelfs beter presteren dan ingewikkelde, handmatig geoptimaliseerde RMI programma's, terwijl ze qua implementatie complexiteit vergelijkbaar zijn met de meest eenvoudige RMI versie die mogelijk is.

In hoofdstuk 5 wordt *Group Method Invocation* (GMI) beschreven. Het GMI model kan het beste gezien worden als een generalisatie van het RMI model. Terwijl het RMI model gebaseerd is op communicatie met één object, gaat GMI uit van communicatie met een groep van objecten. Daarnaast ondersteunt GMI een aantal verschillende manieren voor het aanroepen van methoden en het retourneren van hun resultaat. Zo is het in GMI mogelijk een methode aanroep uit te voeren op één object of van meerdere objecten, waarbij de parameters van de aanroep eventueel verdeeld kunnen worden over alle objecten. Ook is het mogelijk meerdere aanroepen te combineren tot één enkele voordat deze verstuurd wordt. De resultaten van een GMI aanroep kunnen worden weggegooid, geretourneerd, doorgestuurd, of gecombineerd worden tot één enkel resultaat. Ook is het mogelijk een resultaat op te splitsen in meerdere resultaten.

Omdat de verschillende manieren van methode aanroep en resultaat afhandeling voor elke methode afzonderlijk en tijdens run-time gecombineerd kunnen worden, biedt GMI een zeer flexibel en expressief model. Vele bestaande communicatie primi-

tieven, zoals RMI, asynchrone RMI, futures, broadcast, scatter, gather en reduce, kunnen met GMI eenvoudig uitgedrukt worden op een manier die netjes integreert met het objectgeoriënteerde programmeer model van Java. Hiermee onderscheidt GMI zich van bestaande oplossingen zoals het gebruik van een extern *Message Passing Interface* (MPI) pakket.

Hoewel MPI pakketten over het algemeen efficiënte communicatie primitieven aanbieden, passen de daar gebruikte *message-passing* communicatie en *Single Program Multiple Data* programmeermodellen niet goed in Java. Het *mpiJava* pakket is een voorbeeld van zo'n oplossing. Met behulp van enkele kleine testprogramma's en zes parallelle programma's vergelijken we de prestaties van een Manta GMI implementatie met mpiJava. Hoewel mpiJava in de testprogramma's meestal iets beter presteert, is het verschil op applicatie niveau over het algemeen erg klein.

Hierna vergelijken we met behulp van zes programma's de prestaties van GMI, RMI en RepMI. In alle programma's levert GMI de beste prestaties. Vooral de programma's waar complexere communicatie primitieven gebruikt worden, zoals reduce-to-all en gather-to-all, hebben profijt bij het gebruik van GMI.

In hoofdstuk 6 besluiten we dit proefschrift. Uit het gepresenteerde werk blijkt dat het mogelijk is RMI en serializatie implementaties te maken die efficiënt genoeg zijn voor het schrijven van parallelle programma's voor een homogene cluster computer. Ook hebben we twee alternatieve modellen geïntroduceerd, RepMI en GMI, en hebben we laten zien dat door het gebruik van deze modellen parallelle programma's efficiënter worden en eenvoudiger te programmeren zijn.

Curriculum Vitae

Personal data

Name:	Jason Maassen
Date of birth:	March 19th, 1973, Vlissingen, The Netherlands
Nationality:	Dutch
Current address:	Vrije Universiteit Faculty of Sciences Department of Computer Science De Boelelaan 1081a 1081 HV Amsterdam The Netherlands jason@cs.vu.nl

Education

1986-1991	HAVO Openbare Scholengemeenschap Huygenwaard Heerhugowaard, The Netherlands
1991-1993	VWO Openbare Scholengemeenschap Huygenwaard Heerhugowaard, The Netherlands
1993-1998	Master's degree (cum laude) in Computer Science Vrije Universiteit Amsterdam, The Netherlands
1998-2002	Ph.D student Vrije Universiteit Amsterdam, The Netherlands

Professional Experience

1997-1998 Teaching assistant for the courses
Introduction to Programming and *Data-structures*
Vrije Universiteit
Amsterdam, The Netherlands

1999-2000 Teaching assistant for the course
Computer Organization and Operating Systems
Vrije Universiteit
Amsterdam, The Netherlands

2001-2003 Teaching assistant presenting the lecture
C for Java Programmers
Vrije Universiteit
Amsterdam, The Netherlands

2002-2005 Researcher
Vrije Universiteit
Amsterdam, The Netherlands

Publications

Journal Publications

- Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area Parallel Programming Using the Remote Method Invocation Model, In *Concurrency Practice and Experience*, volume 12, issue 8, pages 643-666, July 2000.

- Jason Maassen, Thilo Kielmann and Henri E. Bal. Parallel Application Experience with Replicated Method Invocation. In *Concurrency and Computation: Practice and Experience*, volume 13, issue 8-9, pages 681-712, July-August 2001.

- Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient RMI for Parallel Programming. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 23, number 6, pages 747-775, November 2001.

- Arnold Nelisse, Jason Maassen, Thilo Kielmann, and Henri E. Bal. CCJ: Object-based Message Passing and Collective Communication in Java. In *Concurrency and Computation: Practice and Experience*, volume 15, issue 3-5, pages 341-369, March-April 2003.

Conference Publications

- Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In Proceedings of the *Seventh ACM SIGPLAN Symposium on Priciples and Practice of Parallel Programming (PPoPP'99)*, pages 173-182, Atlanta, Georgia, May 4-6, 1999.

- Henri E. Bal, Aske Plaat, Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, and Ronald Veldema. Parallel Computing on Wide-Area Clusters: the

Albatross Project, In Proceedings of the *Extreme Linux Workshop*, pages 20-24, Monterey, California, June 8-10, 1999.

- Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java, In Proceedings of the *ACM 1999 Java Grande Conference*, pages 8-14, San Francisco, California, June 12-14, 1999.

- Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. The Albatross Project: Parallel Application Support for Computational Grids. In Proceeding of the *1st European GRID Forum Workshop*, pages 341-348, Poznan, Poland, April 12-13, 2000.

- Jason Maassen, Thilo Kielmann, Henri E. Bal. Efficient Replicated Method Invocation in Java. In Proceedings of the *ACM 2000 Java Grande Conference*, pages 88-96, San Francisco, California June 3-4, 2000.

- Arnold Nelisse, Thilo Kielmann, Henri E. Bal, and Jason Maassen. Object-based Collective Communication in Java. In Proceedings of the *Joint ACM Java Grande-ISCOPE 2001 Conference*, pages 11-20, Palo Alto, California, June 2-4, 2001.

- Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In Proceedings of the *LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 1-6, Washington DC, March 22-23, 2002.

- Rob van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, Henri E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In Proceedings of the *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18-27, November 3-5, 2002, Seattle, Washington, USA.

Technical Reports and National Conferences

- Ronald Veldema, Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, and Aske Plaat. Efficient Remote Method Invocation, *Technical Report IR-450*, Vrije Universiteit Amsterdam, September, 1998.

- Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java, In Proceedings of the *Fifth Annual Conference of the Advanced School for Computing and Imaging, ASCI'99*, pages 338-347, Heijen, The Netherlands, June 15-17, 1999.

- Jason Maassen, Thilo Kielmann, Henri E. Bal. Efficient Replicated Method Invocation in Java. In Proceedings of the *Sixth Annual Conference of the Advanced School for Computing and Imaging, ASCI'00*, pages 169-176, Lommel, Belgium, June 14-16, 2000.