C-RMI: An Efficient Alternate RMI Implementation

By

Jesse Eichar
B. Sc., University of Northern British Columbia

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Nigel Horspool, Supervisor (Department of Computer Science)

[redacted]

Dr. Hausi Mueller, Departmental Member (Department of Computer Science)

[redacted]

Dr. Jens Jahnke, Departmental Member (Department of Computer Science)

[redacted]

Dr. Kin Li, External Examiner (Department of Electrical & Computer Engineering)

Supervisor: Dr. Nigel Horspool

ABSTRACT

Many current software systems have a large distributed component. Because distributed systems are so important, methods and tools for developing efficient and safe distributed systems are of critical importance. The Java Remote Method Invocation (RMI) API provides excellent support for developing well-designed and safe distributed systems. Unfortunately, Java RMI is not as efficient as competing technologies such as Remote Procedure Call (RPC). This thesis presents C-RMI, an optimized Java RMI implementation. C-RMI maintains the strengths of Java RMI while improving performance. The first strategy is to move as much processing to compile-time as possible, leaving only the most critical routines to be performed at runtime. The second strategy is to optimize the main routines used by RMI, such as data marshalling and buffering. These strategies result in as much as an 88% performance increase.

Examiners:

[redacted]

Dr. Nigel Horspool, Supervisor (Department of Computer Science)

[redacted]

Dr. Hausi Mueller, Departmental Member (Department of Computer Science)

[redacted]

Dr. Jens Jahnke, Departmental Member (Department of Computer Science)

[redacted]

Dr. Kin Li, External Examiner (Department of Electrical & Computer Engineering)

Table of Contents

Table of Contents iii

List of Tables v

List of Figures vi

Chapter 1 - Introduction 1

Chapter 2 - Remote Procedure Calls 3

2.1 History and Architectures of Remote Procedure Calls 3

2.2 Data Marshalling 9

2.3 Summary of Related Java Technologies 18

2.3.1 Java Reflection 18

2.3.2 Java Input/Output Streams 19

2.3.3 Java Bytecode and Class Loading 20

2.4 Overview of Java RMI 21

Chapter 3 - Current RMI and Serialization Architecture 24

3.1 Overview of Java RMI Architecture 24

3.2 Overview of Java Serialization 30

3.3 Discussion of Java RMI 34

3.4 Other Improvements to Java RMI 39

3.4.1 Manta by Maassen, Nieuwpoort, Veldema, Bal and Plaat 39

3.4.2 KaRMI by Christian Nester 42

3.4.3 Object Caching RMI Implementation by Krishnaswamy, Walther, Bhola Bommaiah, Riley and Topol 44

Chapter 4 - C-RMI – A New Implementation of RMI 47

4.1 System Requirements 47

4.2 Performance Improvement Assessment 49

4.2.1 Compiled Serialization Routines 50

4.2.2 Fewer Serialization Codes 51

4.2.3 Improved Buffering 52

4.2.4 More Efficient Execution Flow 53

4.2.5 Compress Serialized Data for Rapid Data Transfer 54
4.3 C-RMI Design 55
4.3.1 Overview of C-RMI Architecture 55
4.3.2 C-RMI Serialization Support Classes 58
4.3.3 C-RMI Compiled Classes 59
4.3.4 C-RMI Interpretive Serialization 63
4.3.5 Thread Safety in C-RMI 66
4.4 Correctness and Limitations 67
Chapter 5 - Performance Evaluation 69
5.1 Experiment Specifications 69
5.2 Experimental Results 73
5.2.1 Heterogeneous Binary Tree 73
5.2.2 Partially Heterogeneous Binary Tree 76
5.2.3 Homogeneous Binary Tree 79
5.2.4 Doubly Linked List 82
5.3 Discussion 85
Chapter 6 - Conclusions 88
6.1 C-RMI - An Improved RMI Implementation 88
6.2 Directions for Future Research 89
Chapter 7 - Bibliography 92

List of Tables

Table 5-1 - Specification for data samples used by graphs.....72
Table 5-2 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a heterogeneous tree with 0 to 50000 nodes. 74
Table 5-3 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a heterogeneous tree with 0 to 500 nodes...76
Table 5-4 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a partially heterogeneous tree with 0 to 50000 nodes.....77
Table 5-5 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a partially heterogeneous tree with 0 to 500 nodes.....78
Table 5-6 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous tree with 0 to 50000 nodes. 80
Table 5-7 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous tree with 0 to 500 nodes. ..81
Table 5-8 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous doubly linked list with 0 to 5000 nodes.....83
Table 5-9 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous doubly linked list with 0 to 500 nodes.....85

List of Figures

Figure 2-1 - Flow of control and messages during the execution of RPC[11].....9
Figure 2-2 (a) little-endian representation (b) big-endian representation. Both are representations of the integer 5. 10
Figure 2-3 - (a) Definition of a struct that contains a struct.....12
Figure 2-4 – (a) An instance of the Map class and its object graph (b) The order the objects in the object graph would be serialized.....14
Figure 3-1 - Class Diagram and Relationships of UnicastRemoteObject.....27
Figure 3-2 - (a) Object Collaboration on Client (b) Object Collaboration on Server 28
Figure 3-3 - (a) A Simple Object Graph (b) Format of serialized data for the object graph in (a) 31
Figure 4-1 - Performance Comparison of C-RMI with Different Buffer Sizes 53
Figure 4-2 - C-RMI Architecture Class Diagram.....56
Figure 4-3 - Class Diagram of the C-RMI Serialization mechanism.....57
Figure 4-4 - C-RMI uses both compiled serialization and interpretive serialization to serialize an object that is a subclass of the expected class 65
Figure 5-1 - Data Structures Used as Parameters in Performance Tests (a) A homogeneous binary tree (b) A partially heterogeneous binary tree (c) A fully heterogeneous binary tree (d) A homogeneous doubly linked list.....70
Figure 5-2 - Performance of RMI implementations when the parameter is a heterogeneous tree with 0 to 50000 nodes in the tree. 74
Figure 5-3 - Performance of RMI implementations when the parameter is a heterogeneous tree with 0 to 500 nodes in the tree. 75
Figure 5-4 - Performance of RMI implementations when the parameter is a partially heterogeneous tree with 0 to 50000 nodes in the tree. 77
Figure 5-5 - Performance of RMI implementations when the parameter is a partially heterogeneous tree with 0 to 500 nodes in the tree. 78
Figure 5-6 - Performance of RMI implementations when the parameter is a homogeneous tree with 0 to 50000 nodes in the tree. 79

Figure 5-7 - RMI implementation performance when the parameter is a homogeneous tree of size 0 to 500 nodes. 81
Figure 5-8 - RMI implementation performance when the parameter is a homogeneous doubly linked list of size 0 to 5000 nodes. 83
Figure 5-9 - RMI implementation performance when the parameter is a homogeneous doubly linked list of size 0 to 500 nodes. 84
Figure 5-10 - Maximum parameter depths the different RMI implementations are capable of serializing 87

Chapter 1 - Introduction

Many current software systems have a large, and usually important, distributed component. Because distributed systems are so important, methods and tools for developing efficient and safe distributed systems are of critical importance. The Java Remote Method Invocation (RMI) API [14] provides excellent support for developing well-designed and safe distributed systems. Unfortunately, Java RMI is not as efficient as other competing technologies. This thesis presents an alternative RMI implementation called Compiled RMI or C-RMI. The purpose of C-RMI is to maintain the strengths of Java RMI while improving the performance as much as possible. The greatest speed improvements obtained by C-RMI were gained by using two simple but effective strategies. The first strategy is to perform as much processing during compile-time as possible, leaving only the most critical routines to be performed at runtime. The second strategy uses the fact that some of the components used by Java RMI are very general components, so the components can be used for other purposes as well. This code-sharing technique is often a good idea; unfortunately, it has a negative impact on the performance of Java RMI, which is highly undesirable. Instead of using the generic components used by Java RMI, C-RMI uses optimized components to maximize the performance of a remote method invocation.

This thesis is divided into six chapters. Chapter 2 provides the background for understanding the components and considerations of RMI. Chapter 3 explains the structure of Sun's Java RMI and discusses other improved RMI implementations. Chapter 4 provides a detailed explanation of C-RMI and Chapter 5 presents experimental data comparing the performance of C-RMI to the performance of Sun's Java RMI and

other implementations. Finally, Chapter 6 compares C-RMI to other RMI implementations, including Sun's Java RMI, and discusses conclusions based on the comparison. The final section in Chapter 6 makes some suggestions for improvements that could be made to C-RMI to further improve the performance.

Chapter 2 - Remote Procedure Calls

This chapter provides a definition for the term "Remote Procedure Calls" and an explanation of how Remote Procedure Calls operate. The second section of this chapter covers the topic of *Parameter Marshalling*, which is of critical importance to Remote Procedure Calls. The last section briefly explains Java RMI and its relationship to Remote Procedure calls.

2.1 History and Architectures of Remote Procedure Calls

Given a client/server architecture, a client program may call a procedure to execute within the server program's process space. A procedure call made between two different processes, and possibly between two different computers, is called a *Remote Procedure Call* (RPC) [10], [15]. A remote procedure call, in practice, is not nearly this simple, but in principle an RPC is just that, a procedure call from a client program to a server program.

Direct communication architectures, such as Unix sockets, have been available since networks have been around and are supported by the operating system, so what makes RPC so special? To help answer this question, a standard technology called sockets is used for comparison against RPC technologies.

A socket is one end of a communication stream. It can be viewed as one end of a "pipe" connected to one program. A pipe is a unidirectional communication mechanism on Unix between processes. When a second socket is connected to the first, a complete pipe is formed. Data can be passed through the resulting pipe. The two processes must follow a protocol that specifies what data is sent so that useful communication can take

place. Sockets are very useful because they hide a great number of inner mechanisms involved with network communication. Program 2-1 and Program 2-2 illustrate a simple client/server architecture that uses sockets as the communication technology. In this example, the service that the server provides is simple multiplication. The client sends two bytes to the server using a socket. The server multiplies the two numbers and sends the result back to the user.

```
1  ServerSocket server = new ServerSocket(8888);
2  Socket socket = server.accept();
3  InputStream in = socket.getInputStream();
4  int param1 = in.read();
5  int param2 = in.read();
6  int product = param1*param2;
7  OutputStream out = socket.getOutputStream();
8  out.write(product);
9  socket.close();
```

Program 2-1 - Server Code Fragment using Sockets for Network Communication

```
1  Socket socket = new Socket(8888);
2  OutputStream out = socket.getOutputStream();
3  out.write(5);
4  out.write(6);
5  InputStream in = socket.getInputStream()
6  int product = in.read();
7  System.out.println("5*6 = "+product);
8  socket.close();
```

Program 2-2 - Client Code Fragment using Sockets for Network Communication

Program 2-3 and Program 2-4 show a system that has the same functionality as Program 2-1 and Program 2-2 but the communication technology used is Java's Remote

Method Invocation(RMI) instead of sockets. RMI is an object-oriented RPC implementation with some special features that set it apart from other RPC implementations. These features are explained in more detail in Section 2.4.

```
1  public int multiply( int param1, int param2 ){
2     return param1*param2;
3  }// end multiply
```

Program 2-3 - Server Code Fragment Using RMI for Network Communication

```
1  IServer server = (IServer) Naming.lookup("RemoteObjectID");
2  int product = server.multiply(5,6);
3  System.out.println("5*6 = ", product);
```

Program 2-4 - Client Code Fragment Using RMI for Network Communication

The benefits of Remote Procedure Calls over direct communication architectures, such as sockets, are many. Even in an incredibly simple system, like those illustrated in these examples, it is possible to see how much easier and safer it is to program using an RPC architecture. It is easier because there are fewer lines of codes and safer because there is less chance of errors being introduced into the program. One of the principal disadvantages of sockets and similar architectures is that they require the programmer to develop the communication routines for the clients and servers. With RPC, requests to a server are semantically and syntactically identical to calling a method contained in a local library. This is one of the most powerful features of the RPC architecture. Since the programmer does not need to switch programming modes and does not need to be concerned with the communication protocol, the risk of error will be reduced.

A second benefit that RPC holds over other architectures is built-in marshalling of parameters. The only data that may be sent through a Java socket connection are bytes. In Program 2-1 and Program 2-2 the numbers transmitted are bytes so no extra instructions are required. If integers, or more complex data types, were to be sent, they must be represented as a sequence of bytes; that representation may be sent to the server. Architectures like sockets require the programmer to provide the code necessary to encode and decode other data types as bytes. Encoding an instance of a data type as a byte representation is called data marshalling and decoding a byte representation is called unmarshalling. Programming the marshalling and unmarshalling routines is bug prone and can consume a great deal of time. In RPC architectures, marshalling and unmarshalling are part of the framework and, as a result, do not require the programmer to develop marshalling routines.

Two less obvious benefits of RPC over other lower level architectures are: built-in support for multiple clients and compile time type checking. In a low level architecture, like sockets, a programmer must design a way to manage multiple concurrent requests from clients and program the framework that will handle concurrent requests. RPC architectures remove this concern from the programmer because they have built-in support for handling multiple clients. The only precaution that the programmer must make is to ensure that the server method is thread-safe. Finally, because programming with RPC is syntactically identical to programming a centralized system, RPC procedure calls can be validated by a compiler in the same way a normal method can be.

The major last benefit of RPC over simpler architectures is that many implementations provide a *Naming* service of some type; clients can use the Naming services to locate servers that provide the needed service. Once again, if the programmer

uses sockets or similar architecture, the programmer must provide some mechanism to locate the services or specify the location of the servers, an inflexible solution. The first command in Program 2-4 is a request to the built-in Naming service in RMI. In RMI, the Naming service is called the RMI registry. Because a Naming service is typically a registry of some kind, the terms registry and Naming service are often used interchangeably.

Remote Procedure Call is not a new technology; the idea was first introduced in a thesis by B.J. Nelson in 1981 [10] and has since been used in many systems. CORBA implementations, DCOM and Sun all have RPC architectures. A new RPC specification, XML-RPC, already has many different implementations.

RPC systems typically have five major parts: a client, a server, a stub, a skeleton and a registry (Naming Service). The client is a program that calls remote procedures provided and executed by the server. The stub resides in the client process and handles parameter marshalling, unmarshalling return values and manages network communication for the client. The skeleton resides in the server process and performs similar duties as the stub: handling parameter unmarshalling, return marshalling and network communication. The stub and skeleton remove many burdens from the programmer. The last major part of a RPC system is the registry. The registry, or Naming service, maintains a list of services provided by servers. Clients can access a registry and obtain the address of the server and the remote method.

The first action that must be performed when starting a system that uses RPC is to start the registry. After the registry is started, the servers are started and all the services are registered with the registry. Finally the clients may run.

To understand how Remote Procedure Calls operate, the role of the stub and the skeleton must be well understood. As mentioned earlier, the skeleton and the stub play corresponding roles in the server and client processes. The skeleton and stub, in a nutshell, provide marshalling and network management, functionality that the programmer would have to program if using a socket-like technology. Examining Program 2-3 and Program 2-4 provides a better understanding of the roles of the stub and skeleton. The terminology used in this example is subtly different from standard RPC terminology because the example is programmed in Java and is explained using Java terminology as a result. Line 1 in Program 2-4, the client code, contacts the registry of remote objects; in Java, a reference to a stub is returned. Line 2 makes the remote procedure call. The method that is really called by the client is the method declared in the stub. The stub method accepts the parameters and marshals the parameters into a representation that can be sent through the network to the server. Finally, the stub establishes a network connection to the server, makes the request for the remote method to be executed, then sends the parameter data. On the server, a method declared in the skeleton is executed. The skeleton unmarshals the parameter data and calls the programmer's implementation of the remote method. When the remote method returns, the skeleton marshals the return value and sends the data back to the stub, which unmarshals and returns the data back to the client program. The control flow is summarized in Figure 2-1.

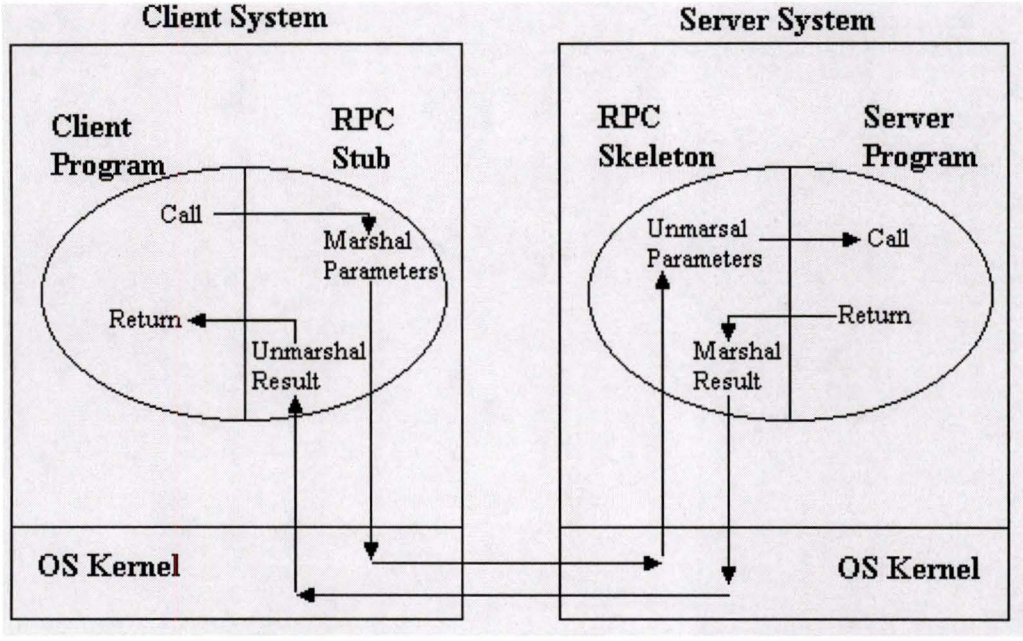

Figure 2-1 - Flow of control and messages during the execution of RPC[11]

Since each stub and skeleton is dependent on the signature of a corresponding remote procedure, a different stub and skeleton must be generated for each different remote procedure. If programmers were required to write the stubs and skeletons, many of the benefits of RPC would be lost. However, since stubs and skeletons tend to be simple, RPC architectures provide tools that generate stubs and skeletons. Typically, the generation tool takes the remote procedure's source code as input and outputs a stub and skeleton.

2.2 Data Marshalling

Data Marshalling is a very important part of an RPC architecture. How data is marshalled greatly affects the performance of a remote procedure call. As well as performance, other issues must be considered. These considerations make it difficult for

a programmer to develop a robust, flexible, distributed system using sockets or a similar architecture. One major issue that must be handled manifests itself in the simple operation of transmitting an integer between two different machines.

Different computers may have different representations for numbers, characters and other data items. One of the great benefits of distributed systems is the ability to use the processing power of multiple computers; even computers with different architectures can be used in a distributed system. Difficulties are introduced if distributed systems use heterogeneous computers. Two possible binary number formats are big-endian and little-endian formats.

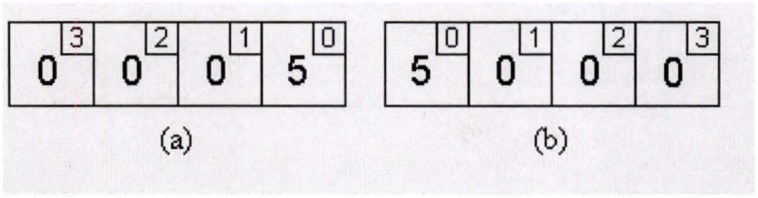

Figure 2-2 (a) little-endian representation (b) big-endian representation. Both are representations of the integer 5.

Figure 2-2 illustrates the difference between the two number formats. Both numbers are integers. On machines that use the little-endian number format, the most significant byte is stored in the highest address. The most significant byte on a big-endian machine is stored in the lowest address. Suppose that a server program, similar to Program 2-1, ran on a big-endian machine and a client program, Program 2-2, ran on a little-endian machine. If the integer 5 was sent, as in Figure 2-2(a), the server would incorrectly interpret the integer as 83,886,080 ($5x2^{24}$) because it runs on a big-endian machine. A possible solution to this problem is to send an additional byte that represents the format of the sending machine. The receiving program would check the byte and, if

necessary, translate the byte into the format understood by the local machine. This problem is not limited to integers. All data types (strings, floating point, etc.) can be represented differently on different systems. By taking care of inter-computer communication RPC architectures remove a great deal of work from the programmer.

Other technical issues are also involved with data marshalling; differing data formats are but one example. For more information on data marshalling see the book Distributed Operating Systems [15].

When remote procedure calls (RPC) were introduced, the dominant languages of the time were procedural languages. In procedural languages, the parameters that can be passed to a procedure are typically limited to primitive data types, data structures like arrays, C-style structs and pointers. A struct is a data structure that is used to encapsulate related data items such as those that form an address. Because all the data in a struct is in-line, marshalling a struct is simple. All the data fields of the struct are encoded one after another and sent to the server. Structs will be discussed in more detail when Java objects are discussed later in this section. A pointer variable is a more complex data type to marshal. A pointer is a reference to a memory location and would be invalid if the address value was marshalled and sent to another computer. In object-oriented languages, an object reference is essentially a pointer to an object. A pointer can be marshalled in the same manner as an object reference. Object references are discussed later in this section.

Since procedural languages were dominant, object-oriented languages have gained in popularity and RPC architectures have been modified so that objects can be used as parameters in remote procedure calls. An object encapsulates data similar to a struct, but can restrict access to the encapsulated data. An object also encapsulates functionality.

One important distinction that must be made between C-style structs and Java objects is shown in Figure 2-3. Figure 2-3(a) is a definition of two structs; the Record struct contains an Address struct. Figure 2-3(c) shows the definition of the equivalent classes. Figure 2-3(b) and (d) illustrate how memory allocation differs between structs and Java classes. The difference is subtle but important. Structs containing embedded structs are allocated together as one data type. In Java, each class is instantiated as a separate object and a class field is always instantiated as a reference to another object. In other words, structs may contain other structs but objects may only refer to other objects. An object reference is essentially a pointer that references the instantiated object. This distinction is important to marshalling because a reference does not always reference another object; that is, a reference may be null, meaning it does not reference an object. An embedded struct, on the other hand, always contains the embedded struct, therefore the embedded struct must always be marshalled.

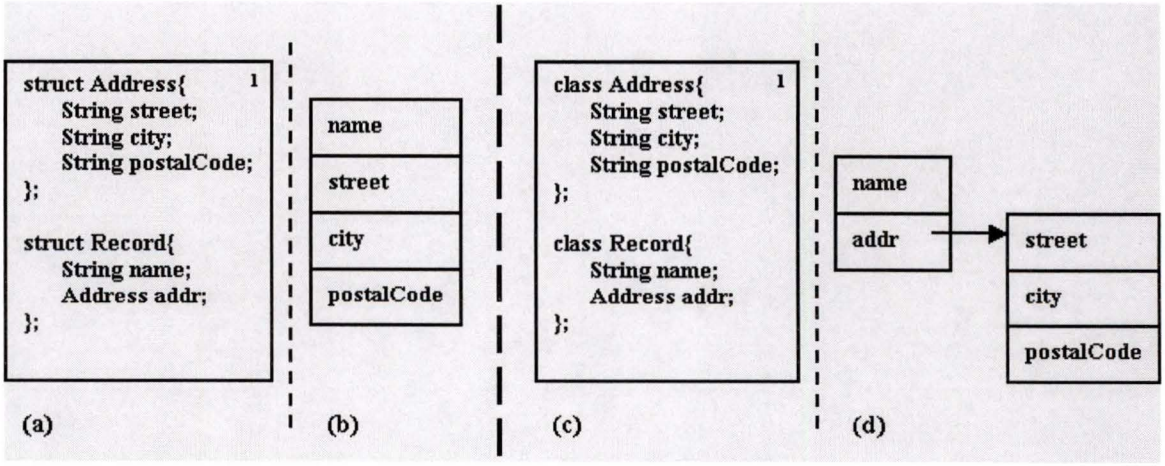

Figure 2-3 - (a) Definition of a struct that contains a struct

(b) An example of how the Record struct would be allocated in memory

(c) Definition of a class referencing a class

(d) Example of how an instance of the Record class would be represented in memory

Passing an object as a parameter is similar to passing a pointer. There are three options for passing an object as a parameter. An object can be passed by value, by reference or by copy-restore. If the object is passed by reference, the server must make an RPC call each time it needs to access the object's state. Remote procedure calls are slow operations, so objects are usually only passed by reference if the remote procedure makes only a few requests to the parameter object or if the parameter object is very large. Not all RPC architectures support remote references, forcing objects to be passed by value. To pass an object by value, all the object's fields must be marshalled and sent to the server. The server reconstitutes the object by creating a new instance of the object and setting its fields to the values that were sent. The important thing to note about pass-by-value semantics is that all changes made to the object by the remote method are lost. Passing an object using copy-restore semantics is similar to passing an object by value but try to retain the changes made by the remote method. The object is copied to the server in the same manner as if the object were passed by value. The difference occurs when the remote procedure is finished executing, when the object is marshalled on the server and sent back to the client. Copy-restore semantics are used to imitate pass-by-reference semantics but with improved performance. The disadvantage is that copy-restore semantics behave differently from pass-by-reference in some conditions. Consider the scenario where an object is passed using copy-restore and the object is modified on the client before the remote procedure call finishes execution. If the object is copied back to the client, the change made by the client is lost.

Marshalling objects is not a trivial matter for a number of reasons. One major complication in regards to marshalling objects stems from the fact that objects can

contain references to other objects. In object-oriented systems, parameters can be very complex arrangements of objects. In an RPC system with pass-by-value semantics or copy-restore semantics, the entire object graph must be marshalled and sent to the server computer. Because the object graph can contain cycles, the marshalling routines are complicated. The term *serialization* is commonly used to describe the process of marshalling all the objects in an object graph. Figure 2-4 is an example of an object graph.

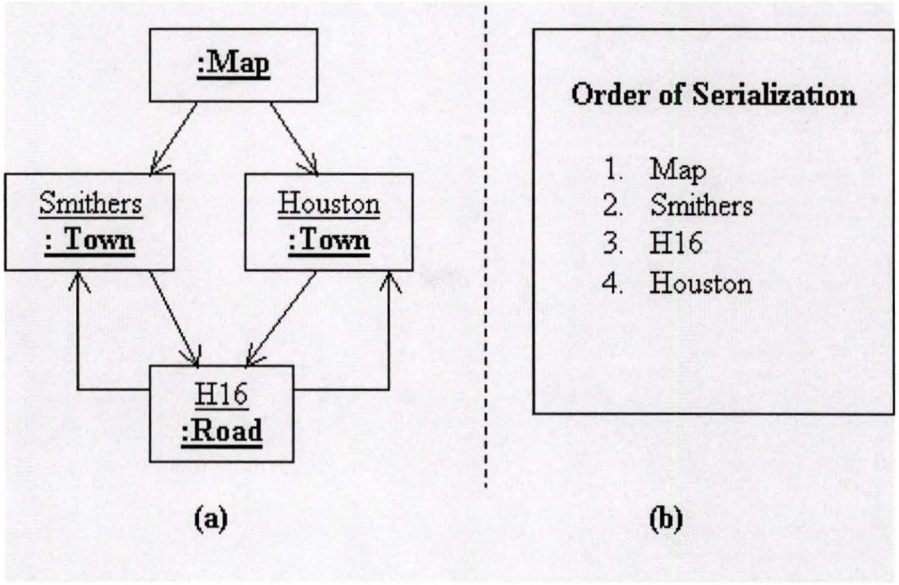

Figure 2-4 – (a) An instance of the Map class and its object graph (b) The order the objects in the object graph would be serialized

The purpose of serialization is to marshal all the objects in a given object graph exactly once and do so in constant time. The serialization process begins with the parameter object, the Map object in Figure 2-4. First, each field in the Map object that is either primitive or a null reference is encoded into a byte array. Second, if one of the object's fields is a reference to another object, that object must also be serialized. The Map object has two object references, a reference to the Town object Smithers and

another to the object Houston. If Map is serialized then both Houston and Smithers must also be serialized because they are part of the Map object's state.

The two Town objects are serialized the same way as the parameter object. First, all the primitive fields and null references are encoded in the byte array, then any objects that are referenced are serialized. The H16 Road object is referenced and must be serialized. The serialization process continues recursively until an object is reached that does not refer to any new objects. It is important to ensure that each object is only serialized once. In Figure 2-4, the Road object is referenced by both Town objects but should only be serialized once.

The decision whether the Smithers object is serialized before the Houston object is arbitrary; in this example, objects referred to by the leftmost arrow in Figure 2-4 are serialized first. After the Map object serialization begins, the Smithers object serialization begins, and then H16 object serialization begins. Keep in mind that serialization is a recursive process, even though the Map object serialization begins first, it is the last to finish.

Serialization of the H16 object begins with the primitives and null references being serialized. Since the leftmost arrow refers to the Smithers object, the Smithers object would be the next object to be serialized. However, because the Smithers object has previously been serialized, a reference indicating the Smithers object is serialized instead and then Houston object is serialized. The Houston object's primitive fields are serialized first, as always, then the object references are checked. Houston has an object reference to the H16 object; however, H16 has previously been serialized so a reference indicating the previously serialized H16 object is serialized. After the H16 reference is serialized, the Houston object serialization is complete and the recursion returns one

level, back to serializing H16. H16 has no other object references to serialize, so it is also complete and recursion returns one more level, back to the Smithers object. The Smithers object is complete, so serialization returns to the Map object. After Smithers is serialized, the Map object still needs to serialize the Houston reference. Since Houston has been serialized, only a reference is serialized and serialization of the Map object is complete.

There are two main strategies for object serialization: interpreted and compiled. **Compiled serialization** gets its name from the fact that the stub/skeleton compiler analyzes the classes that will be used in RPC calls, and the compiler generates routines specific to the classes that need to be serialized. In other words, compiled serialization performs most of the analysis work during compile time as opposed to runtime. **Interpreted serialization** is called interpretive because the objects to be serialized are analyzed during runtime, not compile time like in compiled serialization. Usually the RPC architecture has one general marshalling routine and one general unmarshalling routine. As an object is serialized, its fields are analyzed and sent to the server along with enough information for the server to decode the object information and instantiate a copy of the object.

The benefit of compiled serialization is speed; because class analysis is performed during compile time, less work is done during runtime. In addition, if both the stub and the skeleton are compiled at the same time, an efficient protocol can be used for sending object data between the client and server. Unfortunately, efficiency comes at the expense of inflexibility. Compiled serialization requires that each time a class is changed or fields added or deleted, the stub and skeleton must be recompiled and redistributed to all the clients. In addition to being restrictive in this manner, compiled serialization may break the object-oriented semantics of the RPC by not allowing an object that is a subclass of

the expected parameter class to be passed to the remote procedure. The difficulty occurs because the stub does not have the serialization routines for the subclass, only for an object of the expected class. The last major disadvantage is that many method invocations tend to have similar instructions for marshalling and unmarshalling so that the client applications are larger than if one general routine was used for all marshalling and one generic routine was used for all unmarshalling. A second impact that having more serialization code has is during high load periods the CPU cache is constantly loading in new instruction sets resulting in many more disk accesses than in interpreted serialization.

Interpreted serialization is nearly the mirror image of compiled serialization in terms of benefits. Its primary benefit is its flexibility and its main disadvantage is its slower speed. The analysis of the parameter class is performed during runtime instead of compile time and as a result must be performed each time that the object is to be serialized, instead of just once during compilation. The greatest benefit that interpreted serialization can claim is that it is extremely flexible. Since the object to be serialized is analyzed during runtime, a subclass can easily be substituted, maintaining object-oriented semantics. Not only can a parameter be overloaded with a subclass, but developers can modify any of the classes without having to recompile the skeleton and stubs. Unlike compiled serialization, which requires different serialization routines for each object or each remote procedure, depending on the implementation, interpreted serialization only requires one serialization routine. This results in smaller applications.

Since both strategies have desirable elements, attempts have been made to combine the two strategies. An example of a hybrid strategy was presented by Paul Menage [8]. The idea is to compile serialization routines for each remote procedure only once during runtime and reuse the code generated from that point on. The strategy also

provides interpretive routines for flexibility. The speed of this hybrid strategy is similar to that of compiled serialization because specialized routines are compiled for each remote procedure. At the same time, the strategy is as flexible as interpreted serialization because of the interpretive routines provided with the architecture that can be used if a parameter is overloaded with a subclass.

2.3 Summary of Related Java Technologies

This thesis assumes a rudimentary understanding of certain Java technologies. This section summarizes the technologies used by Java RMI, specifically the Java Reflection mechanism, Java Streams and Java's bytecode and class loading paradigm.

2.3.1 Java Reflection

Java Reflection allows runtime analysis and manipulation of classes. It provides routines for analyzing and manipulating a class in almost any manner. For example, the reflection mechanism can be used to obtain a list of constructors for a class, determine the parameter types required for each constructor, and create an object using one of the constructors. The reflection mechanism also has methods to get a list of the methods declared in a class and provides routines to invoke the methods. The reflection mechanism is used heavily by Java's serialization technology.

The reflection package has a class for each component of a class, for instance, the Class class, the Method class, the Field class, the Array class, and the Modifier class. Each component of a class has a corresponding reflection class that provides routines for accessing and manipulating information about the component. For example, the Method

class provides methods for invoking methods, and the Field class provides methods for setting Fields.

2.3.2 Java Input/Output Streams

Java uses data streams for input and output. A data stream is similar to a pipe. With an output stream a program writes to a "sink," the sink may be a socket, a file or some other construct that accepts data, such as a display. Input streams read from data sources. For example, an input stream could read from a socket, a file, or a keyboard.

There are two types of data streams, character streams and byte streams. Character streams use characters as the basic unit of output and input, while byte streams use bytes as the basic data element. All data streams inherit from the four basic input and output streams: InputStream, OutputStream, Reader and Writer. For example, the FileOutputStream is a stream that writes bytes to files and the FileReader reads characters from a file.

A second very important part of Java's input and output architecture is a collection of classes that derive from the filter stream classes. Filter streams are similar to a decorator in the decorator pattern. The purpose of a filter stream is to add functionality to a data stream. A filter stream wraps around a data input stream and "filters" the data being written to or read from the stream. ObjectInputStream is an example of a filter stream; it automatically serializes an object before writing to the stream. Without the ObjectInputStream, the programmer would have to serialize an object into bytes before writing to an input stream.

Another useful set of filter streams is BufferInputStream and BufferOutputStream. A normal stream's usual behaviour is to write or read one character or byte at a time. Buffer streams modify the behaviour of the streams so that requests are queued and multiple reads and writes are performed in one burst, usually improving the performance of a program.

RMI uses object streams wrapped around a socket stream for network communication. Object streams are buffered streams resulting in fewer network accesses.

2.3.3 Java Bytecode and Class Loading

The last Java technology covered in this section is Java bytecode and class loading. Bytecode enables Java to be platform independent. Instead of compiling a class definition into a file containing platform dependent instructions, Java compiles its classes into bytecode. At runtime, an interpreter, the Java Virtual Machine, translates the bytecode into instructions the platform can understand. Because bytecode is independent of the platform it runs on, it can be copied to another platform and run there without recompiling the class.

Normally bytecodes are loaded from the same machine that the Java Virtual Machine runs on; however, this is not a requirement. The Java classloader can be extended to allow classes to be loaded from across a network. This is possible because the classes are compiled into bytecode. RMI uses such a classloader when it receives a parameter of a type the server does not hold locally.

2.4 Overview of Java RMI

The Java programming language has an RPC architecture as one of the core components of the language. The architecture is called "Remote Method Invocation" or "RMI." This section explains the basic mechanisms and components of RMI, Chapter 3 provides a more detailed analysis of RMI.

When discussing Java, and most object-oriented technology, slightly different terminology is used for discussing execution flow and describing the structure of programs. The terminology now switches to a more object-oriented terminology.

Java RMI is built using two of Java's other core technologies: Java's sockets and Java's serialization technology. RMI uses sockets as the underlying communication framework. Program 2-1 and Program 2-2 illustrate how a simple client and a simple server can be programmed using sockets. RMI uses sockets in a similar, if more sophisticated, manner for communication. As with most RPC architectures, the underlying network communication is not seen and how sockets are used is not a concern of the programmer.

The two main components of Java serialization are the ObjectInputStream and the ObjectOutputStream. These objects provide routines that can serialize or deserialize any Java object or primitive. Java uses these two object streams for parameter marshalling. It is important to note that the object streams are very general and have uses for other applications besides RMI. An example of such a use is saving an object's state to disk for later use.

The Java serialization technology uses the interpreted serialization strategy based on reflection. Using reflection, the ObjectOutputStream queries the object being serialized about its fields and superclasses. Because object streams are very general, the

state of the object is encoded along with more information about the types of the fields and the types of the object. Chapter 3 examines serialization in more detail.

The ObjectInputStream class contains few surprises. It reads in an array of bytes, the output of ObjectOutputStream, and using that information instantiates an exact copy of the serialized object. All the information about the object and the object's fields are encoded in the byte array so the ObjectInputStream can recreate the entire object graph from the information in the byte array and does not need to use reflection to determine the types of the object's fields.

As mentioned earlier RMI also has a remote object registry. When a server is started, the server checks whether an object registry is running and starts one if necessary. Once it has located a registry, the server registers all of its remote objects in the registry. Locating and creating a registry, as with registering remote objects, is not automatically taken care of by RMI. These tasks are the programmer's responsibility but are simple commands; one line for each remote object to be registered and only three or four to locate or create a registry. Once the objects are registered, clients access the registry for a reference to the remote object. Using the remote reference the client can request services from the remote object.

The semantics of a remote method invocation vary only slightly from the semantics of a normal method invocation. The semantics for passing a primitive parameter are the same for both local method invocation and remote method invocation. Primitive parameters use pass-by-value semantics; any changes made to the primitive values affect only the value within the scope of the method invoked. In local method invocation, objects are always passed by reference, so changes to the parameter object are persistent beyond the scope of the method. For remote method invocation, objects can be

passed using pass-by-value semantics or pass-by-reference semantics. If an object is passed by value, it is serialized and sent to the server and all changes to the object are ignored; the same semantics are used for primitive parameters.

In RMI, a parameter that is passed by reference must be a remote object. For an object to be a remote object, the object must meet one basic criterion: the object, or one of its superclasses, must implement the Remote interface. Care must be taken when passing an object as a parameter because the decision of how the object is passed to the remote method is dependent on whether the object is a remote object. If the parameter happens to be a remote object then it is passed by reference, it cannot be passed by value. An object that is not a remote object cannot be passed by reference. Unexpected side effects can occur when an object is passed by reference when the programmer expects the object to be passed by value.

If a parameter is a remote object, it is passed by reference and the server must make remote calls to the object to access its state or to request its services. Since the parameter is not copied to the server, all the changes to the parameter during execution of the remote method are persistent, unlike parameters that are passed by value.

The reason that pass-by-value semantics are normally used in RMI is for increased efficiency. Remote calls are very slow compared to local calls. If a remote method makes multiple accesses to an object, it is much more efficient to copy the entire object to the server. Perhaps to be more consistent, copy-restore semantics would have been more useful.

Chapter 3 - Current RMI and Serialization Architecture

This chapter is divided into four main sections. The first section provides an in-depth explanation of Java's RMI version 1.3.1. The second section discusses Java's serialization mechanism in detail. In the third section, Java's RMI is criticized and the fourth and final section discusses various optimizations to Java RMI that have been published.

3.1 Overview of Java RMI Architecture

By definition, an object is a remote object if the object implements at least one remote interface. A Remote interface is an interface that declares all of a remote object's remote methods, that is, all the methods that can be invoked remotely by a client program. Each remote interface must also extend the interface *Remote,* which has no methods. Program 3-1 shows the declaration of two remote interfaces. A final requirement for a remote interface is that all methods declared in a remote interface must be declared as throwing a remote exception.

```
1  public interface IRemote extends Remote{
2     public void execute( Parameter p1 ) throws RemoteException;
3  } // end interface
4  public interface ManagerRemote extends Remote{
5     public void manageObject( Parameter p1 ) throws  RemoteException;
6  } //end interface
```

Program 3-1 - Two Remote Interfaces

Program 3-2 is a remote object that implements the two remote interfaces defined in Program 3-1. The ServerSideRemoteObject in Program 3-2 extends the UnicastRemoteObject, providing remote object functionality. The UnicastRemoteObject class is discussed in more detail in Section 3.1.

```
1  public class ServerSideRemoteObject extends UnicastRemoteObject
2            implements IRemote, ManagerRemote{
3      public void execute( Parameter p1 ) throws RemoteException{
4            // method implementation
5      } // end execute()
6      public void manageObject( Parameter p1 )
7                                throws RemoteException{
8            // method implementation
9      } //end manageObject()
10 } // end class
```

Program 3-2 - A Remote object that implements two remote interfaces

A remote object does not need to implement two remote interfaces; one is enough. Remote objects sometimes implement different interfaces for clients that require different services from the object. For example, an administrator might require functionality that can modify how the object is setup, while other clients only require the normal functionality of the object.

After compiling the ServerSideRemoteObject class, the RMI stub compiler, called rmic, generates the stub class from the resulting bytecode. In the same manner as most other RPC architectures, the stub in RMI acts like a proxy for the remote object. The stub implements all the remote interfaces that the remote object implements so that the stub can act as a proxy. The remote method implementations provided by the stub are very

simple. The implementations wrap all the parameters into an object array and send them, along with a code indicating which method to execute, to the server.

```
1  try{
2      IRemote remote = Registry.lookup("RemoteObjectID");
3      remote.execute( p1 );
4  }catch( RemoteException re ){}
```

Program 3-3 - Code Fragment for client making a Remote Method Invocation

Program 3-3 is an example of a program fragment that would be part of a client. In this example the client accesses the registry for a copy of the remote object's stub, then invokes the execute method. The registry is a remote object existing on the same server as the remote objects registered with it. The Registry class is a utility class with a number of static methods for binding, unbinding and locating remote objects. The Registry class's static lookup method, invoked by the client, locates the object registry, on the server, and returns a stub for the remote object. When the client has received the stub the execute() method is invoked on the stub and the stub starts the process of invoking the remote method execute().

According to the definition of a remote object, an object needs only to implement a remote interface to be a remote object. However, for the object to function correctly, two other requirements must be satisfied. The first additional requirement for a remote object is that the object extends the abstract class, RemoteObject. The RemoteObject class provides the java.lang.Object behavior for remote objects. The Object methods hashcode(), equals() and toString() are overridden by RemoteObject so that the semantics for a remote object remain consistent with the semantics of a local object. The methods

must be overridden because the method for determining whether two references refer to the same object, equals(), is drastically different if the object in question is remote or local. Methods hashcode() and toString() also require different implementations for a remote object.

RemoteObject also defines two abstract methods: getStub() and getRef(). The RMI framework uses the two methods when "exporting" the object. Exporting an object is the process of preparing an object to accept remote requests. When an object is exported, a stub object is instantiated along with a remote reference and a network endpoint is created. Remote reference objects are discussed in more detail in Section 3.1.

The UnicastRemoteObject is a descendant of the RemoteObject class and is the class that a remote object should directly inherit. The RemoteObject class might provide support for remote object semantics, but the UnicastRemoteObject class essentially *is* RMI. Figure 3-1 shows the class diagram and relationships of the UnicastRemoteObject class.

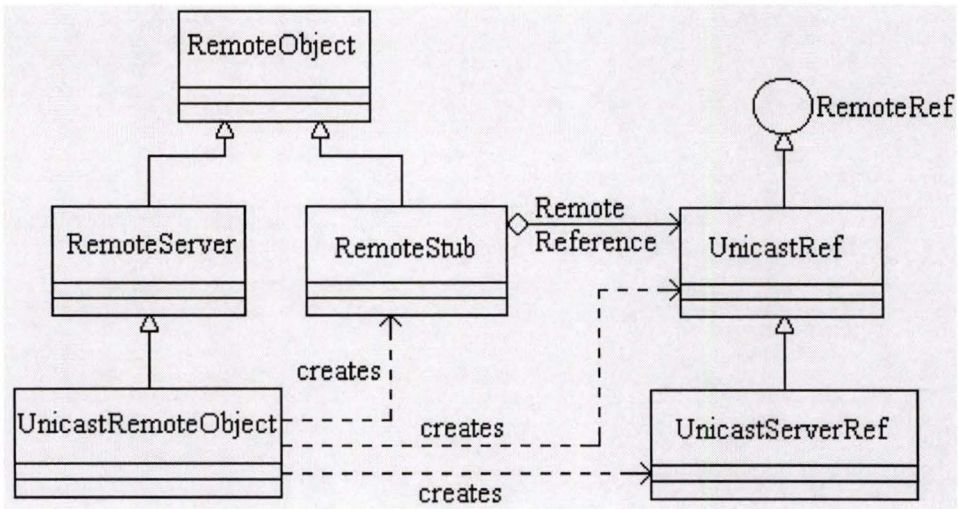

Figure 3-1 - Class Diagram and Relationships of UnicastRemoteObject

The RemoteObject class ensures that object semantics are correct for remote objects and the RemoteServer class is responsible for creating the network endpoint, one end of a network link. By extending the UnicastRemoteObject class, an object obtains the functionality of a remote object. The UnicastRemoteObject class exports the remote object and creates the UnicastRef and UnicastServerRef. As shown in Figure 3-1, the UnicastRef is part of the Stub object. The UnicastRef object manages remote method invocation on the client. The UnicastServerRef exists on the server and handles remote method invocation on the server side.

Figure 3-2 shows the execution flow of one remote method invocation. The bold arrows indicate where execution flow changes between client and server. Execution starts at message 1. Message 6 is the first message on the server, control returns to the client at message 13.

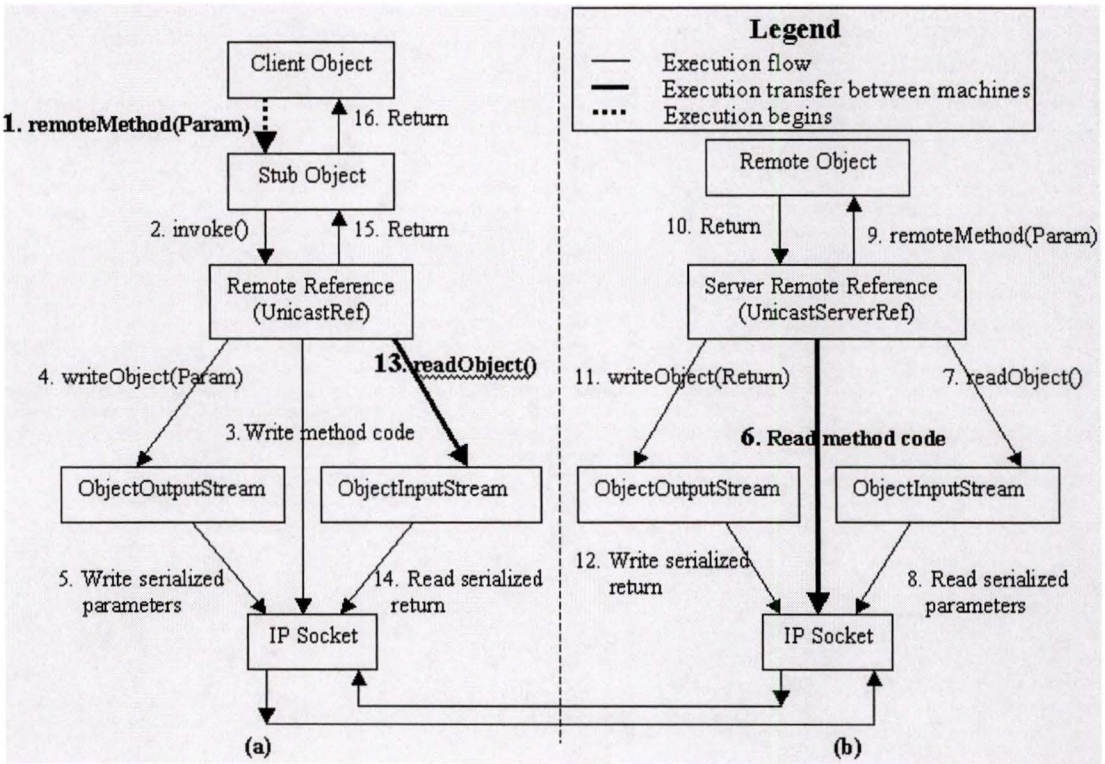

Figure 3-2 - (a) Object Collaboration on Client (b) Object Collaboration on Server

In the same way as an RPC, the client object invokes the method with the same signature as the remote method in the stub object (message 1). The second message from the client to the remote reference, the UnicastRef in Sun's implementation, is a request to invoke a remote method. Message 3 is the request for a remote method to be invoked. A method code is sent from the client to the server. The server side of the remote reference uses the method code to determine which method of which object is being requested, message 6. Once the method request has been made, the remote reference needs to serialize and send the parameters. Message 4 shows the remote reference writing the parameter object to the ObjectOutputStream, which serializes the object and writes the serialized data to the socket, message 5.

One of the goals of Sun's RMI implementation seems to be to minimize the size of the stub class. When a stub's method is called, the stub forwards all the parameters to the remote reference in an object array. If the parameter is a primitive, a wrapper object for the primitive is instantiated and added to the object array. Then, during parameter marshalling, all of the objects in the object array are analyzed using reflection to determine how they should be serialized. If an object is a primitive wrapper class, such as an Integer object, the primitive value is obtained from the object and serialized using a type specific serialization method. This decision has a negative impact on performance. If primitives were serialized in the stub, where the type is known, then it wouldn't be necessary to create the wrapper objects or use reflection to determine the object types.

Recall from Chapter 2, that when data is written to a socket, the socket sends the data to an associated socket. The data can then be read from the second, receiving socket. The lines connecting the client socket to the server socket represent this flow of data.

On the server, after the method code has been read the method parameters must be read and deserialized. Message 7 is the request to the ObjectInputStream to read the parameter data from the socket, message 8, and deserialized the parameter object. Once the server-side remote reference has the parameters the requested remote method is invoked, message 9. At message 10, the remote method finishes execution and returns.

Once the remote method has finished execution, the server-side remote reference writes the return value to the ObjectOutputStream (message 11), which serializes the return value and writes the serialized data to the socket, message 12.

Once execution is returned to the client, the process is simple, the return value is read and deserialized, messages 13 and 14, then the value is returned to the stub, which returns the value to the client object, messages 15 and 16.

3.2 Overview of Java Serialization

In Chapter 2, it was noted that parameter marshalling has a large impact on the efficiency of a remote call. In this section, Java's serialization mechanism is examined in detail.

Java Serialization is strictly an interpretive method of serialization and is based on Java Reflection. The ObjectOutputStream and the ObjectInputStream are the two classes used by programs to serialize objects. Chapter 2 also discussed Java streams; object streams are examples of filter streams, meaning that they add functionality to an underlying data stream. The ObjectOutputStream class serializes objects and primitives and writes the serialized data to the underlying data stream. The ObjectInputStream class reads serialized data from the underlying data stream and reconstitutes copies of the

serialized objects. The process of deserializing an object is the exact inverse of serializing an object, so only the serialization process is covered here.

The object diagram in Figure 3-3(a) shows one element of hashtable. The element contains references to two objects of a hypothetical type called Generic. The generic type has one field, which is a reference to another Generic. The key object has a reference to the value object. However, the value value's reference is null.

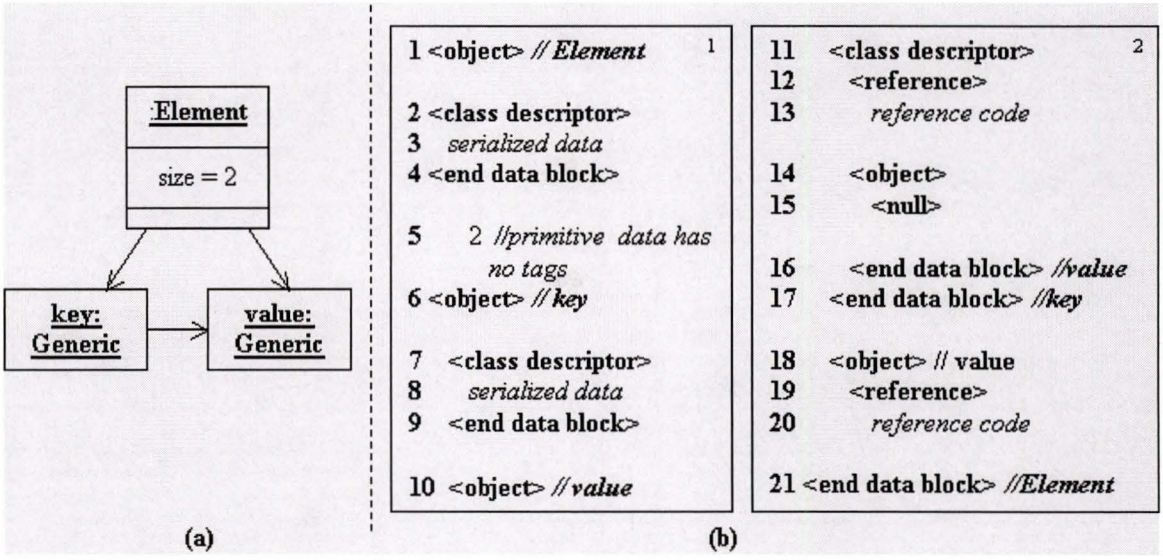

Figure 3-3 - (a) A Simple Object Graph (b) Format of serialized data for the object graph in (a)

Figure 3-3(b) shows how the serialized data would be formatted if the element object were to be serialized. Figure 3-3(b) shows the order of the object serialization, the codes used in serialization and the order of class descriptors serialization. Class descriptors are discussed later in this section.

The tags, such as <object>, are not strings as shown in Figure 3-3(b), each tag is a one-byte code that indicates what type of serialized data follows the code. The <object>

code indicates that the following data is the serialized data of an object. The <end data block> code is a delimiter, usually showing the end of the serialized data for an object.

The first code in Figure 3-3(b), <object>, indicates that an object is being serialized. The object is the root node, Element, in the object graph in Figure 3-3(a). The serialized object has three main parts; the first part is a class descriptor, indicated by the <class descriptor> code and deliminated by the <end block data> code. The second part of a serialized object is the serialized primitive data. The primitive data does not have a code to indicate the start or end of the data. In Figure 3-3(b), the primitive data for Object element is the field size and is written after the class descriptor. The last part is the serialized object graph.

The element object in Figure 3-3 has references to two objects a key and a value. The <object> code at line 6 starts serializing the first object referenced by the Element object. The <object> at line 18 is where the value reference would be serialized if it had not been serialized earlier. However, the key contains a reference to the value object as well, so the value object is serialized as part of the serialization of the key, line 10. As a result, the value is not serialized at line 18. Instead, a reference is made to the previously serialized object. The code, <reference>, is used to indicate that the object referenced by the Element object has already been serialized. After the <reference> code, an integer is serialized. The integer represents one of the previously serialized objects. Notice that at line 11, when the class descriptor of the value object is to be serialized a <reference> code is used. This is because the class descriptor for objects of type Generic was serialized when the key object was serialized, line 7.

Serialization codes, the tags shown in Figure 3-3 (b), are used for a few reasons. One reason is that if the bytecode is not available on the machine that is deserializing the

object, the data can still be obtained. The codes are also used to validate that the object has been serialized correctly. The last major purpose of the codes is to determine how to deserialize the following data. For example, if a class descriptor code is received, the ObjectInputStream class knows to deserialize a class descriptor and not an object.

In addition to adding serialization codes to the serialized data, Java RMI also includes class descriptors. Notice that after each object tag in Figure 3-3(b) there is a class descriptor tag, followed by the serialized class descriptor. The purpose of a class descriptor is similar to the purpose of the Class class in Java Reflection. The Class class provides routines for making queries about a class's attributes, such as what methods the class declares. A class descriptor has a similar purpose for serialization but only provides routines for making queries about a class's fields. The reason that class descriptors are used instead of the object class is for efficiency.

When a class descriptor is created, all the information required by the class descriptor is obtained using reflection on the associated class. The data that a class descriptor covers is the data about the class's fields and superclasses. This initialization stage is performed so that the native methods used by Java reflection are not used more than once.

Native methods are used for two main reasons: efficiency and because the function cannot be performed using Java. The current hotspot compilers are quite fast, particularly because they compile certain parts of code into machine language. For a native method to be more efficient, the function being performed must be relatively time consuming because there is a certain amount of overhead required to invoke a native method and the speed gains made using the native method must be great enough to compensate for the invocation overhead. The second reason native methods are used are

to perform functions that are not possible using Java. Java Reflection requires native methods to access internal information about a class. Almost all of the Java Reflection methods are native because they cannot be implemented using Java. In this case, efficiency is not why the methods are native. Because Java Reflection is not efficient, the class descriptor classes are used to reduce the inefficiency as much as possible.

Class descriptors are serialized for the same reasons that serialization tags are serialized; so that serialized data can be retrieved even when the bytecode for the class has been lost and so deserialization can verify that an object was properly serialized. Each time an object of a new type is serialized, an associated class descriptor is also serialized, lines 2 and 7 in Figure 3-3(b).

Together, the serialized class descriptors and the serialization codes significantly increase the amount of serialized data. For many applications, such as storing objects on disk, the increased amount of serialized data is necessary; for RMI, it is not. The need to deserialize an object without the bytecode will never occur because the bytecode can always be loaded from the client. Because of the additional data, each remote invocation takes a significantly longer time because more data must be sent to the server.

3.3 Discussion of Java RMI

The primary design feature one notices when studying Sun's Java RMI implementation is flexibility [5]. A second design feature is reusability[5]. Both of these features imply that performance may be sacrificed. One manner in which performance is sacrificed for flexibility is the manner that Java Streams are used. The stream hierarchy used by RMI is as follows: a SocketOutputStream wrapped by the

ConnectionOutputStream class, which is a filter stream. ConnectionOutputStream extends MarshalOutputStream and MarshalOutputStream extends ObjectOutputStream. The benefit of using a hierarchy of streams this way is flexibility; new functionality can be easily added by extending ConnectionOutputStream or new network protocols added by replacing SocketOutputStream, which uses the TCP/IP protocol, with another socket output stream which implements an alternate protocol. This structure also promotes reusability of streams. For example, the socket streams can be used by any application to access a network through the sockets. However, streams being associated in this manner are somewhat inefficient. The basis of the inefficiency is caused by the use of the filter streams. Because a filter stream is being used, each method call to the stream is invoked at least twice, once in the filter stream and a second time in the data stream. If the filter stream were incorporated into the data stream, fewer method calls would need to be made. The inflexibility introduced by using this method would most likely not be worth the speed increase.

A second alternative to a filter stream is to extend the data stream and add the functionality to the subclass. This alternative is more efficient because not all methods would require two method calls, but it is less efficient than the first alternative. If the write method were to be overridden in the subclass to add functionality, the implementation of the super class must still be invoked; so two method calls must be made in this case. Not all method calls would require a second method invocation, which is the case when using the filter stream. When a filter stream is used, all methods in the data stream must also be implemented in the filter stream. The method implementations in the filter stream must at least invoke the corresponding method in the data stream, resulting in a minimum of two method calls for every operation. If a data stream is

extended then only those methods that need to be modified must be overridden, so fewer method calls would be required. A second benefit of the inheritance method is that, when a method in a superclass is invoked it does not use dynamic linking like normal method calls and as a result is more efficient than normal method calls, which are used by a filter stream invoking a method in a data stream. The disadvantage is again a loss of flexibility, although this solution is more flexible than when the functionality of the filter stream is combined with the functionality of the data stream into one class.

Another way to improve the performance of Java RMI is to use an alternate communication protocol. The TCP/IP protocol is not as efficient as other protocols. For example, if a version of the UDP protocol were used as an alternative, some performance improvements could be obtained without sacrificing flexibility. If the UDP protocol were to be used, I would recommend that the protocol be modified so that packets are not lost. While TCP/IP is inefficient when compared to other high performance protocols, such as UDP or Myrinet, TCP/IP is very common, more common than most other high-performance protocols.

One of the major sacrifices of performance for flexibility can be found in the serialization mechanism used by RMI. This mechanism is both designed for flexibility and reusability. While these design decisions are practical, they seriously impact the performance of RMI. It could be practical to develop a serialization mechanism that is optimized for RMI serialization. Certain aspects of RMI can be exploited for a much more efficient method of serialization than that used by Sun's Java RMI. ObjectOutputStream and ObjectInputStream are very general and make absolutely no assumptions about the object being serialized. In RMI, however, the object type is known

and, in the current implementation, is totally ignored. There are many ways to optimize an RMI specific serialization mechanism.

For RMI, the serialization process does not need to be purely interpretive as it is implemented in the ObjectOutputStream and the ObjectInputStream classes. Developing a hybrid compiled/interpreted method would provide much better performance and still maintain all the flexibility currently provided by the ObjectOutputStream and ObjectInputStream classes.

One of the most glaring implementation flaws present in the serialization used by RMI is primitive parameter handling. In the current implementation, primitive parameters are wrapped in a wrapper object so they can be passed to the UnicastRef's invoke() method. Then they are analyzed using reflection to determine what type they are so that they can be serialized. This implementation creates an object for each primitive parameter that does not need to be created; later runtime analysis is performed to determine the type of the parameter, even though the type is known at compile time. Since primitive types cannot be extended, there is no way that the parameter could contain a value of a type other than the primitive type defined in the method definition. No flexibility would be lost if serialization of primitives was made the responsibility of the stub. The server side implementation would not even need to be modified to support this change.

Another optimization that could be made would be to compile more information about the parameters, for example class descriptors, like those used by ObjectOutputStream could be compiled so that no reflection is required to generate the class definitions. However, given a data structure that contains many objects of one class, this optimization would offer little performance improvement over the current

implementation because reflection is used once to create a class description and for all following serialization operations of objects of that type, the class description is used for efficient serialization. This optimization is likely to provide significant improvements for remote calls where there are few objects of the same type.

All the suggestion made so far provide significantly improved performance only if the network is very fast, for example, communication between virtual machines on the same computer. If the network is very slow, improving the performance of an RMI call on the server and client will not result in a large overall performance improvement because the network communication is the primary performance bottleneck. To improve the performance on systems using slower networks, the amount of data sent to the server and back to the client must be reduced or a more efficient communication call must be used. There are two ways that the amount of data can be reduced. The first method is the simplest. If the amount of data that must be sent is large, a fast compression scheme can be used to compress the data and transmit the compressed data. Using compression to reduce the amount of serialized data would also be beneficial to the current serialization mechanism. If the data is compressed, the time taken to perform the compression and decompression must be less than the time gained for this method to be useful. Because compressing large data sets is more likely result in a bigger gain, a minimum data size, where the data is compressed, can be set.

The second method to reduce the amount of data can only be used for RMI because the data types of the objects to be serialized are known. When serializing an object, ObjectOutputStream adds a relatively large amount of data describing the object being serialized. Because the types are known a large fraction of the data is redundant, and, for RMI serialization, can be omitted. The omission of descriptive data in RMI

would reduce the amount of serialized data, reducing the time required to transmit the data. This method would work best for object graphs that are mostly made up of a heterogeneous set of objects because the first time that the ObjectOutputStream serializes an object, the associated class descriptor is also output. Later, when an object of the same type is serialized, the class descriptor is not serialized, just a reference number indicating the previously serialized instance.

3.4 Other Improvements to Java RMI

There have been a number of other improvements made to Sun's Java RMI implementations. This section discusses the merits and drawbacks of the other modifications made to Sun's Java RMI. The improvements discussed are the Manta system by Maassen, Nieuwpoort, Veldema, Bal and Plaat [3], [4], [5], [6], [7], [9], [16] the KaRMI system by Christian Nester [11] and an RMI implementation that uses object caching to improve performance, by Krishnaswamy, Walther, Bhola Bommaiah, Riley, Topol and Ahamad [2]. Another improvement, not discussed in detail, is the Asynchronous Remote Method Invocation (ARMI) implementation [13]. The intention of ARMI is to extend RMI to include asynchronous calls so that certain procedures, such as matrix vector multiplication, will perform quicker. ARMI does not improve general RMI performance, however, so it is not a good alternative to C-RMI.

3.4.1 Manta by Maassen, Nieuwpoort, Veldema, Bal and Plaat

The Manta system is an RMI implementation originally started in 1998 [7] and has been an ongoing project since. The Manta system is designed for parallel computing,

which requires drastically improved performance over Sun's RMI implementation. As a result, the design focus is on performance rather than flexibility and maintainability, which is the focus of Sun's Java implementation. The intention of Manta is to maintain the same polymorphic behavior as Java RMI but with greatly improved performance. To obtain improved performance the Manta system has stopped using IP sockets for low-level communication, stopped using Java streams for input and output, changed most of the RMI program code into C [16] and implemented a drastically different serialization mechanism [6]. In addition to these core changes Manta has more recently added other more peripheral changes such as object replication [3] and Collective Communication in Java (CCJ), a communication library that adds Message Passing Interface (MPI)-like operations to Java [9]. However, neither of these changes are discussed in this paper because an object replication strategy is discussed in Section 3.4.2 and CCJ only improves performance for a very focused group of programs.

In Manta's implementation, virtually all of RMI has been reprogrammed in C. The remote references, the serialization and communication routines are programmed in C instead of Java. By programming RMI in C instead of Java, all of Manta is compiled into native code instead of bytecode. Since Manta is all native code, it executes faster because it does not need to be interpreted like bytecode. However, being all native, much of the portability and maintainability of Sun's Java RMI is lost. Naturally, since Manta is programmed in C instead of Java, different ports must be made for different platforms. Because the Manta is not portable like Java, any revision of the system must be performed for each different port, decreasing the maintainability of the system.

Instead of using sockets for low-level communication, Manta uses the Panda communication library [Bal, et al. 1998]. The Panda communication library is more

optimized for high throughput and parallel processing than sockets. When used on a fast Ethernet network, Panda uses a reliable version of the UDP protocol. Using Panda instead of TCP/IP and sockets, Manta improves the performance of transporting RMI data. To further decrease the time taken to input and output data, Manta reads and writes directly to and from Panda buffers instead of performing input and output using Java streams.

The greatest change Manta makes to the serialization mechanism is to change all the serialization routines into native code. Because serialization may require a great deal of processing, changing the process into native code can greatly decrease the processing time. To further improve serialization performance, Manta changes from a purely interpretive serialization strategy to a mainly compiled serialization strategy. The RMIC compiler used by Manta outputs, in addition to the stub and skeleton, native routines for serializing the parameters of the remote methods. However, so that a subclass can be substituted as a parameter, the strategy must also be interpretive. To satisfy the requirement, the RMIC compiler can also be invoked during runtime, creating serialization routines for the new subclasses.

The Manta system is a very fast implementation of RMI, approximately 36 times faster than Java 1.1.4 [6]; however, because the entire system is written in C, the system must be ported for each different platform. This portability problem can creates a large amount of work for maintaining and upgrading the Manta system.

3.4.2 KaRMI by Christian Nester

The KaRMI system uses a number of the strategies mentioned in Section 3.3 - Discussion of Java RMI, however the main modification is to the serialization mechanism [12]. By implementing a serialization mechanism which is optimized for RMI, the time to serialize and deserialize objects is drastically reduced, 81%-97% [12].

The serialization optimizations used in the KaRMI system are: less encoding information, namely the serialization codes and the class descriptors, and more efficient buffering.

It is possible to reduce the type information that is serialized in two ways. The first way is by not writing the serialization codes used in Java serialization. The second modification is to not serialize the class descriptors. KaRMI only sends the class name of the class being serialized, no serialization codes are used and the class descriptors are not serialized. This strategy improves performance by not sending as much data across the network, as well as requiring less serialization time, because the class descriptors do not need to be serialized.

The buffering used by Java serialization is general and does not attempt to exploit the byte-representation of objects. The serialization buffering used by KaRMI is optimized and the performance is much improved over the buffering strategy used by Java serialization.

The principal disadvantage to the KaRMI system is that it was developed for cluster systems. When an object is being deserialized, it is assumed that the bytecode used on the deserializing node is that same bytecode used on the node that serialized the object. No checks are made to ensure that the bytecode used on each node is compatible. This may be acceptable on a cluster system, but for networks that are more general, where

different versions of a class may exist, or even two different classes with the same name, the assumption can cause major problems.

A second disadvantage of KaRMI is that, as the data sets of the parameters get larger, the performance gain over Java RMI is lost. The results presented in the paper by Nester [11] only used parameters with small object graphs. Experience in this thesis has shown that the performance gains obtained using strategies similar to those used by KaRMI, are lost when object graphs of the parameters become very large. As the object graphs become larger, the time required to send the parameter data becomes longer, reducing the overall performance improvement. The second reason for the loss in performance improvement over Java RMI is that one of the strategies used by KaRMI is to reduce the amount of serialization data. However, as the object graph becomes larger, the proportion of extra data included by Java RMI decreases and so does the performance improvement of KaRMI.

While KaRMI is an excellent RMI implementation, very efficient and flexible, it has obviously been designed for very tightly networked systems but without some mechanism for guaranteeing that a compatible bytecode of a class is used on all nodes, the system is not useful for general applications.

Since KaRMI has very similar goals as C-RMI, the performance of the KaRMI RMI implementation is compared to the performance of C-RMI in Chapter 5. As shown in Chapter 5, KaRMI is a very efficient RMI implementation and exhibits large performance gains over other standard RMI implementations, such as Sun's RMI implementation or IBM's RMI implementation. C-RMI is shown to provide even better performance than KaRMI as shown in Chapter 5.

3.4.3 Object Caching RMI Implementation by Krishnaswamy, Walther, Bhola Bommaiah, Riley and Topol

This RMI implementation uses two main strategies to gain improved performance: remote object caching and using a UDP protocol instead of the IP protocol. Similar to the Manta system, this implementation is intended for use primarily on parallel systems, although it can be used on TCP/IP networks as well.

The UDP protocol used was implemented by these researchers specifically for use with RMI. Unfortunately, due to unforeseen complications, the performance of RMI using the new UDP protocol was worse than the performance using the normal TCP/IP protocol. However, this new RMI implementation did observe improved performance using TCP/IP as a network protocol.

The concept behind object caching uses the fact that local method invocation is much faster than remote method invocation. To reduce network traffic, the client downloads the remote object from the server and uses the local copy for all future invocations. The benefit of using the local copy is that no network access is needed to make the method invocation, resulting in a giant performance gain. The disadvantage of using object caching is that consistency between cached objects must be maintained.

This object caching system uses read and write locks to maintain consistency among objects. The consistency management is handled in a manner that is transparent to the client. The remote reference on the server is responsible for managing lock distribution and tracking. The reference maintains a list of clients with read locks so that the reference can invalidate the cached objects if needed.

The semantics of a read lock allows only operations that do not affect the state of the locked object may be executed. Because the state of the locked object is not

modified, many read locks may be held simultaneously. A write lock must be obtained if an operation is to be performed that modifies the state of the remote object. Only one write lock, and no read locks, may be held at one time in the system. Methods that modify the remote object's state are referred to as write methods, and methods that do not modify the object's are referred to as read methods.

A read lock must be obtained when a client wishes to invoke a method that will not modify a remote objects state. When a read lock is obtained the state of the cached object is set to the current state of the remote object. Once a read lock is obtained, further method invocations on the object do not require any network communication, as long as the methods invoked are all read methods. Read and write methods are declared by the remote object developer by declaring the methods as throwing either a read or write exception.

When a client makes an invocation request for a write method, the client must obtain a write lock. When a write lock is requested, the server sends invalidate messages to all the clients holding read locks. All invalidated objects must obtain a read or write lock before a method may be invoked on that object.

The best performance of this object caching implementation occurs when all the clients are invoking read methods. In that case, all the clients can use cached copies, resulting in very good performance, up to 12 times as fast as Java RMI.

Unfortunately, the excellent performance comes at a cost. The first disadvantage of this implementation is that developers of remote objects are restricted to using read and write locks with a minimum granularity of a method. Standard Java concurrency has a granularity of any block of code. The increased granularity of this RMI implementation can have an adverse effect on the performance of a remote invocation. For example, if a

method has one line that modifies a state variable, with the new locking granularity each client must finish executing the method before another client may start execution. Using Java synchronization and locking many clients could execute the method simultaneously and only the one line would be locked. In such a case, the performance would be seriously compromised if the method requires a large amount of execution. In fact, the performance could be worse than performance of Sun's RMI implementation.

There is a second case where performance may be worse than the performance of Sun's RMI. The situation arises when many clients hold read locks and a client wishes to obtain a write lock. The reason that the performance is poor is because all the client's read locks must be released before the write lock can be obtained. At times of high traffic, a very large number of messages may be generated.

A third disadvantage to this mechanism occurs when a client only wishes to make a single request to the remote object. With this implementation, an object must be cached; this results in the single invocation performing very poorly, requiring at least double the invocation time.

This RMI implementation uses a very interesting approach for improved performance; unfortunately, there are still many issues that must be resolved before this implementation is a viable alternative to the current implementation. The current Manta [4] implementation also includes an object caching protocol. However, because the principles are the same, the protocol is not discussed in this thesis.

Chapter 4 - C-RMI – A New Implementation of RMI

This chapter presents a new RMI implementation called C-RMI or Compiled-RMI. C-RMI is intended to retain all the functionality and features of Sun's RMI implementation but with improved performance. To accomplish that goal, C-RMI provides a compiler that generates routines for marshalling parameters. However, testing showed that while extremely good performance was obtained for parameters containing small or heterogeneous object graphs, the performance gain over Java RMI decreases with the size of the parameter object graph. The larger the object graph becomes, the less improvement is observed. To improve performance, the serialized data of large object graphs is compressed before transmission.

This chapter presents the design and implementation of C-RMI, discusses the limitations and correctness of C-RMI and finally, compares C-RMI to the implementations discussed in Chapter 3.

4.1 System Requirements

There are certain design considerations that were considered important when developing C-RMI. The first requirement is to ensure that any application developed for Java RMI would maintain its correctness if C-RMI was used instead. Similarly, an application developed for C-RMI should operate correctly with Java RMI as well. This is an important consideration for two reasons. The first reason is for increased portability and reusability. It is never a good policy to limit your options. The second reason is so that developers are not required to adopt different programming conventions when using

C-RMI. Developers may not consider the performance gain to be worthwhile if they must adopt new conventions in order to use the new implementation.

A related requirement is to ensure that C-RMI is backward compatible with Java RMI. In other words, C-RMI must be able to support remote objects that are compiled with Java's RMIC compiler. This requirement is necessary so that servers running Java RMI can be replaced with C-RMI, and the clients can still use the remote objects on the server without recompiling and redistributing the remote object stubs. Naturally, remote objects that do not recompile and redistribute the remote objects would not gain the benefits of C-RMI, but it does allow incremental upgrades to be made to a system.

A third requirement is to preserve the semantics of method invocation. In Java RMI, the concrete object of a parameter may be a subclass of the declared parameter type. This is one of the strengths of the Java RMI implementation and, therefore, an important feature to retain.

A fourth requirement is to ensure that the performance gains of C-RMI are independent of the transport layer. The only protocol used by the current Java RMI implementation is the TCP/IP protocol. However, any network protocol can be used by providing an alternate SocketFactory object when exporting the remote object. C-RMI must retain that simplicity and flexibility but must also observe the same performance gains over Java RMI using any network protocol or transport layer.

The fifth requirement is that C-RMI must guarantee correct operation in any environment, such as a wide area network or a tightly-coupled cluster system. The optimizations made in C-RMI must not compromise the correctness of an application. The KaRMI implementation discussed in Section 3.4.2 assumes that all nodes use the

same bytecode for all classes. However, this assumption limits the usefulness of the implementation.

A sixth requirement was to implement C-RMI only in Java code so that the implementation is platform independent. C-RMI has been tested on both Windows and Linux systems. No modifications were required to port the implementation from Windows to Linux. Tests executed on both platforms exhibited comparable performance gains.

A seventh and last system requirement is that C-RMI should be a simple drop-in component. To use C-RMI, the C-RMI jar file need only be added to the Java bootclasspath. Once this requirement is met, C-RMI can be used. However, because C-RMI is backwards compatible with Sun's Java RMI, the standard Java RMI can also be used. See the second requirement for a discussion about the backward compatibility.

4.2 Performance Improvement Assessment

This section describes the strategies employed by C-RMI to improve the efficiency of RMI. The five strategies used by C-RMI are detailed in Sections 4.2.1 to 4.2.5.

C-RMI differs from Java RMI mainly in how objects are serialized. The serialization mechanism in C-RMI has been optimized for RMI. The serialization mechanism in C-RMI consists of compiled class descriptors, a stub, a skeleton and specialized RMI streams. These components are discussed in detail in Section 4.3, but a brief explanation is offered here. The class descriptors, the stub and the skeleton classes are generated by the rmic compiler. The rmic compiler used by C-RMI is similar to the

stub compiler used by Java RMI but has been modified to also generate stubs, skeletons and class descriptors. The class descriptors provide routines for serializing and deserializing a class. The stub provides routines for marshalling the method parameters into a byte array and invoking the remote method. The skeleton contains routines for unmarshalling method parameters. The RMI streams are simplified data stream objects. The RMI streams manage an internal buffer and provide routines for serializing primitives. The RMI streams also manage data compression if there is a large amount of serialized data.

4.2.1 Compiled Serialization Routines

Compiling the serialization routines is the first strategy that C-RMI uses to improve performance. Compiling serialization routines increases performance in two ways. One way that time is saved is by reducing the amount of reflection required to serialize an object. During compile time, the parameters that must be serialized are analyzed and the serialization routines are generated. The generated serialization routines contain the instructions for serializing all the fields of an object. In Java Serialization, reflection is required to determine what class fields exist, what type the fields are and what super classes the class has. With compiled serialization routines, all of the analysis is performed at compile time.

The second way that compiled serialization improves serialization performance is that the compiler can create a serialization method to serialize objects in a manner specific to the class being serialized. For example, data of the same types can be grouped together during serialization so that the serialized data can be compressed more efficiently

and the data can be buffered more efficiently. See Section 4.2.3 for more information about the customized buffering used by C-RMI.

4.2.2 Fewer Serialization Codes

Using few serialization codes provides a second method to improve performance and is the second strategy used by C-RMI. Section 3.2 explains that serialization codes are used to control deserialization and ensure that the object data can be obtained from the serialized data even if the bytecode of the class is not available. A serialization mechanism for RMI does not require such a general solution. In RMI, the bytecode of a class is always available either on the client or on the server. Only existing objects can be serialized and an object can only be instantiated if the bytecode is present; therefore, the node that is serializing the object must have the bytecode. If one node does not have the class's bytecode, that node can download the bytecode from the serializing node.

Since the bytecode is always available, less information needs to be serialized with the object data. C-RMI reduces the amount of serialization data in two ways. Java Serialization serializes one class descriptor object for every class that is serialized. To reduce the amount of overhead, C-RMI does not serialize class descriptors. The second way that C-RMI reduces the amount of serialization data is by reducing the number of control codes. Java RMI uses codes to indicate the beginning and the end of an object. Because C-RMI uses compiled serialization routines, codes are not needed to indicate the beginning and the end of each object. The two ways that these changes improve performance are sending less data to the server, which takes less time, and not serializing the class descriptors.

4.2.3 Improved Buffering

The buffering strategy used by Java RMI is very general. Being a general solution, there are inefficiencies when used for parameter marshalling for RMI. The third strategy that C-RMI uses to improve performance is to customize the buffering used for parameter marshalling. Because serialization is a linear process, deserialization can begin before serialization is complete. To obtain maximum performance, deserialization must occur in parallel with serialization. Unfortunately, network accesses are slow, so the number of accesses must be limited. A balance between parallel serialization/deserialization and few network accesses must be found. The serialization used by Java RMI does not attempt to reach a balance, which results in more network accesses than are necessary. To balance parallel serialization/deserialization with network accesses, C-RMI buffers a set amount of data before sending the data for deserialization. Figure 4-1 shows the impact that buffer size has on performance. The experiment used to obtain results for Figure 4-1 was a simple remote method invocation with a tree data structure as the parameter. The remote method invoked in this experiment has no functionality and no value is returned. The nodes in the tree data structure are shown along the x-axis and the y-axis shows the time taken to perform one remote method invocation. Each data series in Figure 4-1 shows the results of the method invocations for different sized buffers. These experimental results determined the size of C-RMI's internal buffer.

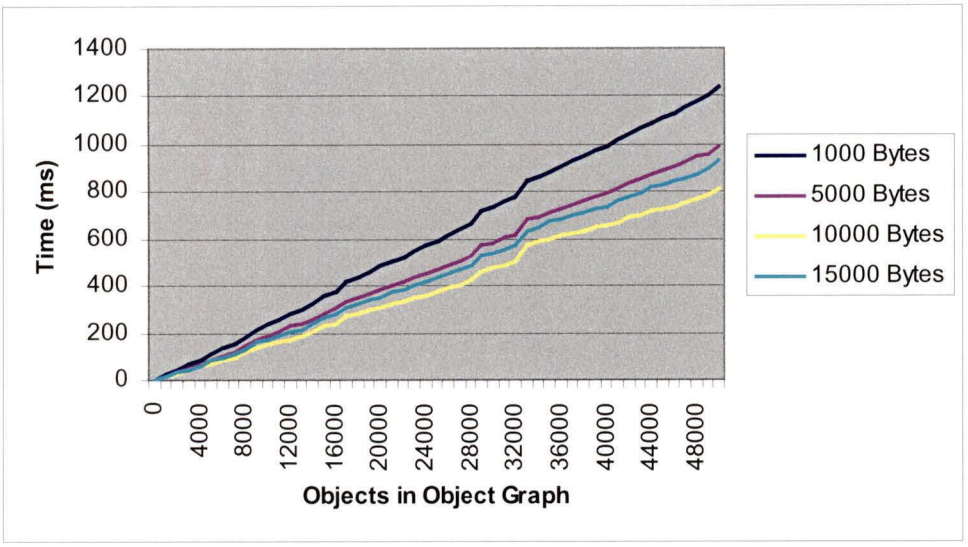

Figure 4-1 - Performance Comparison of C-RMI with Different Buffer Sizes

The last customization to buffer handling is how buffer overflow is handled. Instead of checking that the buffer has sufficient space each time a primitive is serialized, the class descriptors check whether there is sufficient space in the buffer for serializing a group of primitives. Performance is gained because buffer overflow is checked less often.

4.2.4 More Efficient Execution Flow

One of the criticisms of Java RMI in Section 3.3 is the inefficiency introduced to keep the stub class small. In Java RMI, the stub method called by the client wraps all the parameters in an object array and invokes the remote reference's invoke methods with the object array as one of the parameters. If one of the parameters is a primitive, the primitive is wrapped in an object and added to the object array. The invoke method applies reflection to each of the parameters to see what type the object is. If the object

wraps a primitive, the primitive is extracted and serialized; if the object is not a primitive then it is serialized by the ObjectOutputStream. The fourth strategy used to improve performance is to streamline the remote method invocation execution flow. C-RMI serializes the parameters in the stub before calling the invoke method. As a result, the inefficiencies that are removed by C-RMI are the following: the primitive object wrapper objects are not instantiated, the object array is not instantiated and the reflection performed in the invoke method is no longer required.

4.2.5 Compress Serialized Data for Rapid Data Transfer

The disadvantage of all the other improvements employed by C-RMI is that the performance gain decreases as the volume of data to be serialized increases. Performance gain is lost because the relative overhead of creating and serializing class descriptors lessens as the object graphs get larger. To serialize an object of a type that has not been previously serialized, a class descriptor associated with that object's class must be created and serialized. However, serializing two objects of the same type only requires one class descriptor object to be created and serialized. As a result, the relative performance gain of C-RMI decreases as the number of homogenous objects increases.

The fifth strategy used to improve performance is to compress the serialized data when the amount of serialized data becomes large. By compressing the serialized data, the amount of data that is sent to the server is reduced, resulting in less network traffic and time.

The current implementation of C-RMI uses the ZLIB compression library, which is provided with Sun's Java implementation, for compression and decompression. The

ZLIB library is a generic library originally developed as part of the PNG graphics standard. C-RMI uses the ZLIB library because an implementation of the library is part of Sun's Java implementation. In the future, a customized compression scheme will likely be used to provide improved performance. ZLIB is a very general solution and a customized solution would likely result in further performance improvements.

4.3 C-RMI Design

This section presents the design of C-RMI. The presentation is divided into four parts. The first part discusses the architecture of C-RMI by briefly presenting the different components of the system. The second section discusses the C-RMI serialization support classes, and the third section presents the compiled serialization classes. The final part explains how the system is kept thread-safe.

4.3.1 Overview of C-RMI Architecture

The architecture of C-RMI is similar to the architecture of Java RMI except the serialization streams are different and the roles of the stub and skeleton are very different. In Java RMI, the stub performs almost no duties and there is no skeleton object. In contrast, the stub and skeleton play a pivotal role in C-RMI. In this respect, C-RMI more closely reflects typical RPC architecture than Java RMI. In RPC architectures and C-RMI the stub and skeleton are responsible for parameter marshalling and unmarshalling and the skeleton invokes the remote method. Like Java RMI, the remote references in C-RMI still control network access and garbage collection. Figure 4-2 shows the general architecture of C-RMI.

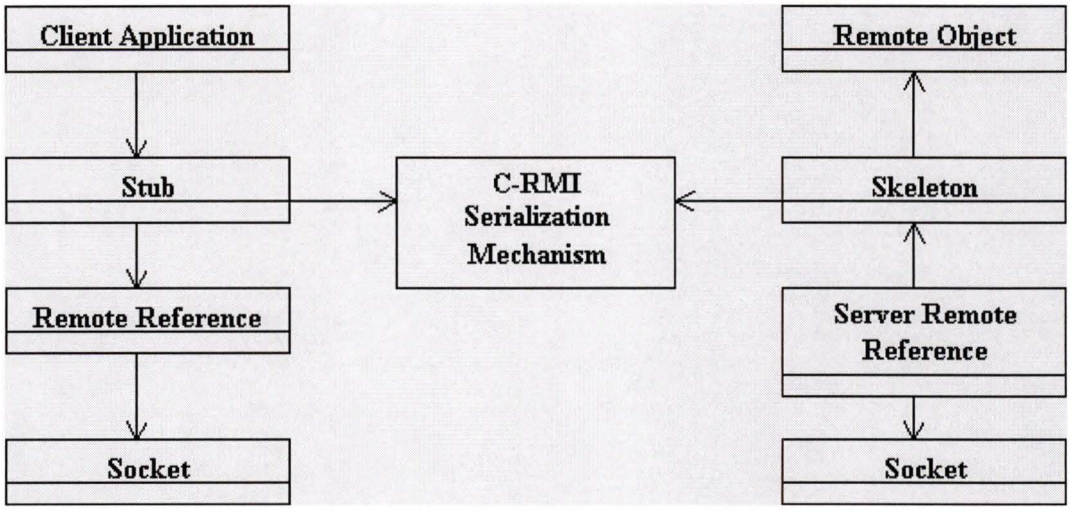

Figure 4-2 - C-RMI Architecture Class Diagram

The primary difference between the architecture of C-RMI and the architecture of Java RMI is where parameter marshalling is handled. In Figure 4-2, it is shown that the skeleton and the stub objects manage parameter marshalling. In C-RMI, the stub and skeleton are drastically different than those in Java RMI; Java RMI version 1.2 does not even have a skeleton class. The serialization mechanism for C-RMI has little in common with the Java Serialization mechanism. Figure 4-3 shows the architecture of the C-RMI serialization mechanism.

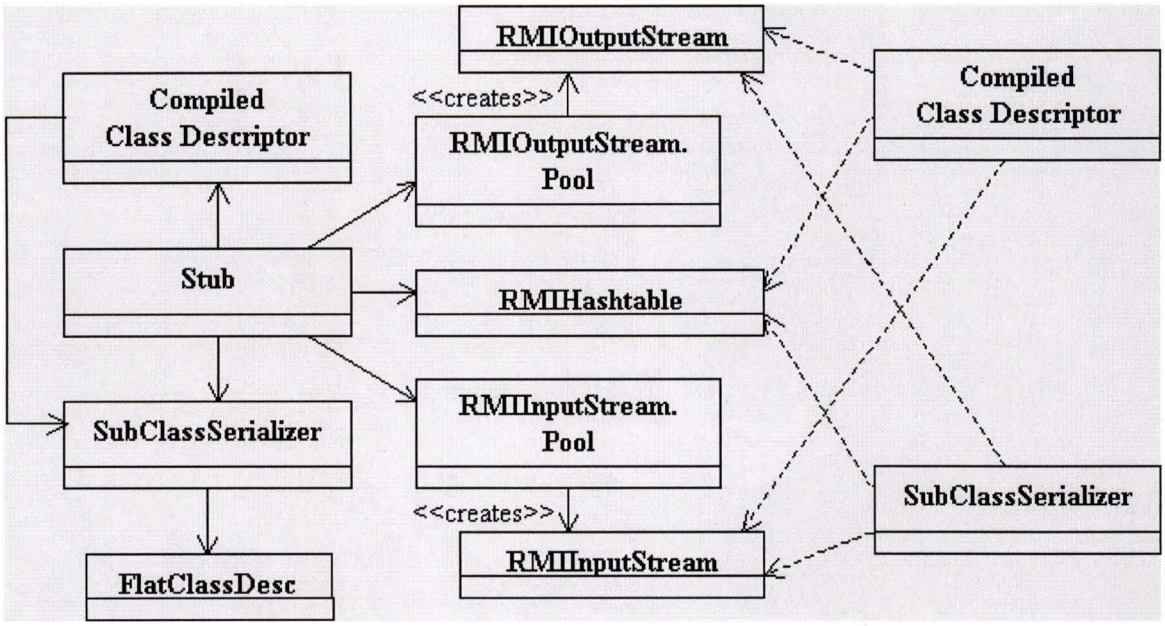

Figure 4-3 - Class Diagram of the C-RMI Serialization mechanism

Other than the serialization, the stub and the skeleton, C-RMI is virtually unchanged from Java RMI. The remote reference classes are the same as in Java RMI except that parameter marshalling is no longer managed. Instead, the stub or the skeleton writes to a byte array and the reference writes the contents of the byte array to the socket stream. To unmarshal parameters or return values, serialized data is read into a byte array by the remote reference and the byte array is passed to the stub or skeleton for interpretation. The Socket class in Figure 4-2 represents all the underlying communication classes used by RMI, all of which are unchanged.

Figure 4-3 shows all the components in C-RMI's serialization mechanism. The core parts are the RMIOutputStreams, RMIInputStreams and RMIHashtable. The Stub, Skeleton and class descriptors are compiled by C-RMI's rmic compiler. The SubClassSerializer and the FlatClassDesc classes are used when the concrete parameter object is a subclass of the declared parameter class. RMI streams provide serialization

routines for primitive data, the stub and skeleton control parameter serialization, the IClassDesc implementations control serialization for the associated class, and the RMIHashtable is used to keep track of serialized objects. The RMIOutputStream.Pool and RMIInputStream.Pool classes help maintain thread safety in the stub and skeleton objects.

4.3.2 C-RMI Serialization Support Classes

One of the topics discussed in Section 2.2 is the different ways to serialize an object. The two ways discussed were compiled serialization and interpreted serialization. One of the disadvantages of compiled serialization was that the large number of instructions could create caching problems during periods of high traffic. To reduce the number of instructions required for serialization, RMIOutputStream and RMIInputStream classes were created. The RMI streams provide routines for serializing primitives, handling data buffering and compressing serialized data when necessary. The class descriptors, stubs and skeletons, which are compiled classes, all use the RMI stream classes heavily. Using the RMI streams makes the classes smaller and less likely to cause caching problems. To further reduce the problem, stubs and skeletons share class descriptors. Improved performance could be obtained by in-lining all serialization routines instead of sharing the routines and requiring extra method calls. It was decided that the improved performance is not worth the caching problems that would be caused.

RMIInputStream and RMIOutputStream are similar to ByteArrayOutputStream and ByteArrayInputStream. Both output streams write to an internal buffer and return a copy of the buffer when requested. RMIOutputStream includes methods for serializing

any primitives, such as writeInt() and writeFloat(). If the amount of data in the buffer is large, greater than 10000 bytes, RMIOutputStream compresses the data before returning the byte array. To indicate whether the byte stream has been compressed, RMIOutputStream adds a flag to the byte array before returning the array. If the byte array passed to RMIInputStream has been compressed, RMIInputStream decompresses the data.

The major support class used during serialization is the RMIHashtable. The RMIHashtable is used for keeping track of objects that have been serialized. Each entry in the hashtable has three fields: a reference to the serialized object, the hashcode of the object, and an integer that is assigned to the object when the object is added. The hashcode of the object is used as a key for the hashtable during serialization and the assigned number is used as a key during deserialization.

When an object is to be serialized, the RMIHashtable is checked. If the object is not in the hashtable, then a flag is serialized to indicate that an object is about to be serialized. If the object is present in the hashtable, then the object has previously been serialized and a flag indicating that the object has already been serialized is serialized, then the assigned number is serialized. During deserialization, each object is added to an RMIHashtable before deserialization. If a flag is received indicating a reference then the assigned number is deserialized and used as a key to locate the object in the hashtable.

4.3.3 C-RMI Compiled Classes

C-RMI has three types of compiled classes: stubs, skeletons and class descriptors. The stub classes control the parameter serialization and deserialization of the return data.

The skeleton performs the equivalent duties of deserializing the parameters and serializing the return data. Class descriptor classes are compiled only if one or more parameters are objects. Each class descriptor provides a method for serializing objects of a class and another method for deserializing objects of the same class. An alternative approach could be to have only one class. The class would contain one serialization method and one deserialization method for each class needing to be serialized for a remote object. The decision to have separate class descriptors was based on a number of reasons. The first advantage of class descriptors is that a class descriptor can be used by separate remote objects. The second advantage is maintainability. If a client does not have a class descriptor or has an incompatible version, the class can be downloaded from the server. If a class is changed and the class descriptor is recompiled, downloading a class descriptor is faster than downloading one large serialization class.

One disadvantage to having compiled classes is the compiled classes can have different versions or become outdated. Problems can occur in three ways in C-RMI. The first is the stub on the client is a different version than the skeleton on the server. When the stub and skeleton are compiled, a unique hashcode is calculated for each. When a client looks up a remote object, the registry sends the skeleton's hashcode to the client. The client compares the hashcode to the local stub class. If the hashcodes differ then the local stub class is replaced with a copy loaded from the server. This guarantees that the stub and skeleton are always of the same version.

The second possible incompatibility that can occur is that a class descriptor is incompatible with the class on the local machine. This can occur when a class is modified but the class descriptor is not recompiled. However, certain changes can be made to a class without the class becoming incompatible with the class descriptor. For

example, if a method is added, removed or changed the class descriptor will remain compatible. However, if a field is added, removed or modified, the class descriptor will become incompatible and will need to be recompiled. During initialization, the class descriptor checks the fields to ensure that no fields have been added or removed and to ensure that the fields remain the same type. If a discrepancy is detected then the class descriptor is marked as invalid. If the class descriptor is marked as invalid then a new class descriptor is generated dynamically. Note that only class descriptors for declared parameter classes are generated. Class descriptors for subclasses of declared parameter classes are not generated. See Section 4.3.4 – C-RMI Interpretive Serialization for information on why generating class descriptors for subclasses of declared parameters is not beneficial.

The final danger of incompatibility occurs when the parameter class on the client side is of a different version from the class on the server side. To detect these occurrences, C-RMI uses the same strategy as Java RMI. The class descriptor class used by Java RMI has a method to calculate a class's serial version UID. This calculation uses all the aspects of the class declaration to generate the serial version UID. The serial version UID is a long. The class name, the extended superclass, the implemented interfaces, the field types, and the method and constructor signatures are all used to generate the UID. If the same serial version UID is calculated for two classes then the two classes must have the same fields types, the same field names, the same constructors and the same methods. Two classes with the same serial version UIDs are therefore completely compatible and, under most circumstances, are the same classes.

Class descriptors are stateless and are not instantiated. The two class descriptor methods, writeObject() and readObject(), are static and the fields are all static and

constant. As parameters, the writeObject method takes the object to be serialized, the RMIOutputStream object to write to, and an RMIHastable instance. The first task done by the writeObject method is to write the serial version UID of the class being serialized. The serial version UID in C-RMI is computed the same way as the serial version UID computed in Java Serialization, Section 3.2 discusses Java Serialization. When the readObject method executes, it compares the serial version UID of the class on the local machine to the serial version UID written by the client. If the UIDs are not the same, an exception is thrown.

To serialize primitive fields, the writeObject method calls the RMIOutputStream methods; to serialize an object, the writeObject method of the associated class descriptor is called. To obtain the value of each method, the Java Reflection mechanism is used; either reflection or a native method must be used to retrieve a field's value because Java does not allow private fields to be accessed by other objects. To keep C-RMI a Java only solution, it was decided to use Java Reflection instead of a native method. The readObject method is the exact equivalent of the writeObject. The fields are read in the same order that the objects were written. Program 4-1 and Program 4-2 are code fragments from a class descriptor's writeObject and readObject methods, respectively.

```
1  int i = _field.getInt( obj );
2  out.writeInt( i );
```

Program 4-1 - Fragment of writeObject(). Obtains a field's value from obj and serializes the value.

```
1  int i = in.readInt();
2  _field.setInt( obj, i );
```

Program 4-2 - Fragment of readObject(). Reads field value and sets field of obj.

The variable *_field* is a Field object, which is part of Java Reflection. The value of the field is obtained from the object using the Field object and is serialized by calling RMIOutputStream's writeInt method. In the readObject method, the equivalent operation is performed. The field value is deserialized by calling RMIInputStream's readInt method and the field is set by calling the Field object's setInt method.

The stub and skeleton methods serialize each parameter for the associated remote method. The structure of the serialization instructions in the stub and skeleton methods is similar to the structure in the class descriptor. The difference is that the stub and skeleton do not use reflection because they do not set or get fields in an object. When an object needs to be serialized, the associated class descriptor's writeObject() method is invoked. As with class descriptors the RMIOutputStream class's primitive serialization routines are used to serialize primitives. Deserialization uses the readObject method and RMIInputStream's primitive deserialization routines.

4.3.4 C-RMI Interpretive Serialization

The serialization mechanism described in Section 4.3.3 is very efficient but to satisfy the design requirements, C-RMI must be able to serialize an object that is a subclass of the declared parameter. There are two possible ways of dealing with an object that is a subclass of the expected type. The first solution is to generate a class descriptor for the concrete class during runtime. The generated class descriptor can be either stored or temporary, just lasting the duration of the virtual machine. The primary disadvantage of generating a class descriptor is the large amount of time required to generate and

compile a class descriptor. If the class descriptor is compiled, both the client and the server must obtain a copy of the class descriptor; this requires that both the server and the client compile the class descriptor, which is very inefficient because they cannot be compiled in parallel. To be more efficient than Java RMI, a class descriptor generated dynamically must be used many times. If a class descriptor is not persistent beyond the execution of a virtual machine, then it is unlikely that the class descriptor will be used often enough to result in a significant performance gain. Storing the class descriptor is better for overall performance, but has disadvantages because a large number of rarely used class descriptors may end up being stored. This is a particularly large problem for servers because they would be required to generate and store more class descriptors than a client.

The alternative to generating a class descriptor for a subclass is to use an interpretive serialization mechanism for subclasses of the expected class. This method limits the worst-case performance to that of Java RMI. However, C-RMI attempts to maintain an average case performance that is better than Java RMI even when serializing a subclass object.

C-RMI uses an interpretive and compiled serialization mechanism for serializing objects of subclasses. Because an object of a subclass still contains all the fields of its superclasses, the class descriptors for the superclasses are still used to serialize the object. In Figure 4-4, the class ExpectedClass is the declared parameter class. If an object of class SubClass is the concrete parameter object, then C-RMI uses an interpretive serialization mechanism to serialize the fields of SubClass and uses the compiled class descriptor object to serialize the fields of ExpectedClass.

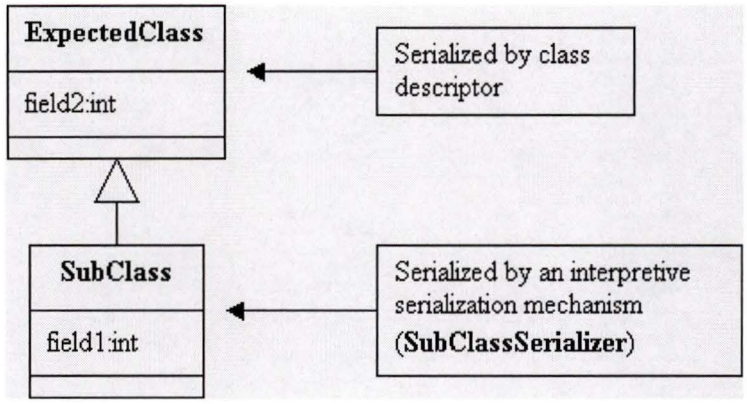

Figure 4-4 - C-RMI uses both compiled serialization and interpretive serialization to serialize an object that is a subclass of the expected class

The SubClassSerializer and the RMIStream classes in Figure 4-3 make up the interpretive serialization mechanism in C-RMI. The SubClassSerializer serializes the subclass fields of an object. ObjectStreamClass is the class descriptor class used in Java Serialization by the ObjectOutputStream and ObjectInputStream classes.

SubClassSerializer has readObject and writeObject methods. These methods serialize or deserialize an object using the provided RMI stream. When calling the read or write methods, the expected class is provided and SubClassSerializer only serializes the fields of the expected class's subclasses. SubClassSerializer, unlike Java's object streams, is customized for RMI. Similar to the rest of C-RMI's serialization mechanism, less serialization data is included. The class descriptors are not serialized and fewer serialization codes are used in C-RMI's interpretive serialization mechanism than in Java RMI. The support classes, the RMI streams and the RMIHashtable class are used by SubClassSerializer to ensure that the C-RMI's interpretive serialization component also has the compression and custom buffering advantages that the compiled serialization component enjoys.

4.3.5 Thread Safety in C-RMI

One of the considerations while developing C-RMI was whether the system was thread-safe. The most important rule for maintaining thread safety is that shared resources can cause problems. If the shared resource has internal state then the resource must be locked before the internal state is modified. If a shared resource does not have internal state then multiple threads of execution can execute in the resource. The class descriptor classes are examples of shared resources without state and RMI streams are examples of classes with state. Fortunately, RMI streams do not need to be singleton objects. To keep the system thread-safe, there are two options available for RMI stream objects: create a new object for each thread or maintain a pool of objects. Because RMIOutputStream and RMIInputStream have potentially large internal buffers, it was decided that maintaining an object pool for RMIOutputStream objects and another pool for RMIInputStream objects would be more efficient than creating a new object for every invocation. The RMI streams each have a static innerclass called Pool, which provides synchronized methods for obtaining and releasing RMIInputStream objects or RMIOutputStream objects. In Java, each object is associated with a lock. During normal operation, the lock is ignored. However, it is possible to "synchronize." When a method is synchronized, the object lock must be obtained before the method can be executed. Because the Pool release and getStream methods are synchronized, only one thread can execute in the release or getStream methods. This restriction guarantees that threads cannot share RMI stream objects.

Even though the remote reference objects have state, they are shared because the methods executed during a remote method invocation do not change the internal state of the remote references; similarly, remote method invocations do not change the state of the stubs or skeletons, so they can be shared as well.

4.4 Correctness and Limitations

The current implementation of C-RMI satisfies the requirements specified in Section 4.1. C-RMI is an implementation that is completely compatible with Java RMI. A remote object using C-RMI obtains exactly the same results as a remote object using Java RMI and no changes to the remote object or the client are required when switching between architectures, other than recompiling the stubs and skeletons. C-RMI performs checks to ensure that the class will perform correctly under all circumstances. Before an object is serialized, a class descriptor checks to ensure that the fields of the associated class have the same types and there are as many as expected. This is to ensure that the class descriptor is compatible with the version of the object being serialized. Then a serial version UID is calculated for the object and the UID is sent to the server. The class descriptor on the server also checks that the class descriptor is compatible with the class version on the server. Once it is known that the class descriptors on the server and the client are compatible with the associated local classes, compatibility between the class version on the server and the class version on the client is checked. This check is performed by comparing the serial version UID sent by the client and the serial version UID calculated by the server. If the two UIDs are the same then the two classes are compatible. The method used to check class version compatibility is the same as the

method that is used by Java RMI; therefore, the method is sufficient for ensuring class compatibility.

The semantics of a remote method invocation using C-RMI are the same as the semantics of a remote method invocation using Java RMI, and all data types that can be marshalled by Java RMI can be marshalled by C-RMI, including primitives and user defined objects. However, neither pass-by-reference semantics nor remote class loading has been implemented in this version of C-RMI. These features were not implemented because, in general, they would have a negligible impact on the performance improvement over Java RMI.

One unexpected, but beneficial, side effect observed in the behavior of C-RMI was a large increase in the depth of the object graphs that can be serialized by C-RMI. Because C-RMI is more streamlined than Java RMI and all of the other implementations tested, C-RMI is capable of serializing objects with a much deeper object graph than the other implementations. One of the tests performed was the serialization of a linked list. Java RMI was only capable of serializing linked lists with fewer than 1,100 elements; however, C-RMI was capable of serializing linked lists with over 5,500 elements. Other RMI implementations were tested as well and all implementations threw exceptions if the list size was over 1,100 elements. The IBM implementation was the worst with a limit of less than 500 elements in the linked list. Using C-RMI, the depth of an object graph can be up to five times deeper than when Java RMI is used and more than ten times deeper than when IBM RMI is used.

Chapter 5 - Performance Evaluation

The goal of this chapter is to explain the performance tests used, the results of the tests, and what the test results mean. The first section in this chapter, Section 5.1, explains each test performed and presents the reasons the tests were chosen. The second section presents the performance test results and briefly discusses the meaning of the results. Finally, in the last section, Section 5.3, the results are thoroughly discussed and the implications of the tests are presented.

To prove the validity of the C-RMI implementation, three other RMI implementations were tested and used as a basis for comparison. All the implementations ran the same tests in exactly the same environment, even using the same computers. The implementations that are used for comparison are Sun's Java 1.3 RMI implementation, IBM's Java 1.3 RMI implementations and KaRMI [11].

5.1 Experiment Specifications

The programs used to obtain performance measures are extremely simple systems. Each test program has a remote object and a client. The methods of the remote objects have no functionality because the only data of interest is the time required to marshal and unmarshal parameters. Each remote method has only one parameter that is a simple data structure. The parameters are data structures so that the size of the parameter can be easily varied. To obtain a good understanding of C-RMI's performance under different circumstances, the sizes of the parameter data structures were varied by a large amount. Data structure sizes of between 0 and 50,000 nodes were used. In this section, the term

parameter size is often used. The term refers to the size of the data structure being passed to a method as a parameter.

Four remote methods were designed for testing, none of which contain code and all accept a single parameter. The data structures used as parameters are shown in Figure 5-1.

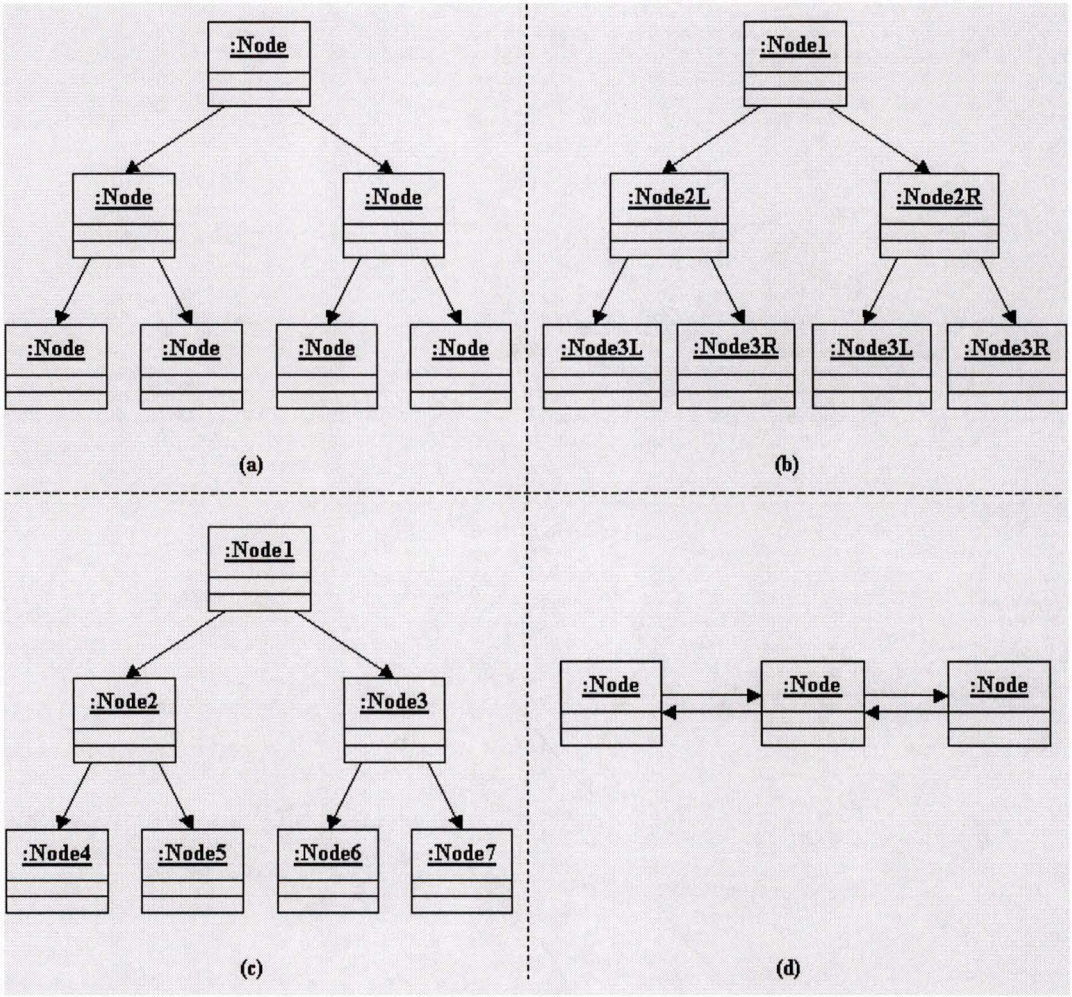

Figure 5-1 - Data Structures Used as Parameters in Performance Tests (a) A homogeneous binary tree (b) A partially heterogeneous binary tree (c) A fully heterogeneous binary tree (d) A homogeneous doubly linked list

The data structure in Figure 5-1(a) is a control test; all the nodes in the tree are of the same type. Because C-RMI was expected to provide the largest performance gain when data parameters consist of a collection of heterogeneous objects, the performance gain of C-RMI was not expected to be large when using the homogeneous binary tree shown in Figure 5-1(a). The tests using the data structure shown in Figure 5-1(b) are expected to demonstrate a large performance gain because object graph contains a number of different object types, this data structure is referred to as a partially heterogeneous tree. The nodes in this tree are not all of the same type, as are the nodes in Figure 5-1(a), but they are not all unique either. As shown in Figure 5-1(b) the two children of a node are of different types, but each level in a tree consists of only those two node types. Therefore the number of object types in a tree is $2*d-1$, where d is the depth of the tree. In the largest tree tested, a tree with 50,000 nodes, there are only 29 different object types.

In Figure 5-1(c), each node in the tree is of a unique type. However, there are only 500 classes defined, so, for the larger data structures, not all nodes are unique. The performance of C-RMI, when tested using the heterogeneous tree shown in Figure 5-1(c), is expected to exhibit the largest performance gains. This tree is termed the heterogeneous tree because of the high level of heterogeneity.

Figure 5-1(d), the final test, was designed to measure the performance of C-RMI when required to marshal a graph with cycles. The results from the doubly linked list tests were expected to show a similar performance gain as the homogeneous tree tests.

To measure the time required for a remote method invocation, each remote method invocation was executed many times and the cumulative time was measured using Java's System.currentTimeMillis() method. Times are sampled before and after the

tests are performed, and the difference between the two times is the cumulative time in milliseconds. The cumulative time is then divided by the number of method invocations performed. Table 5-1 summarizes the data collected to create the graphs described in Section 5.2.

Largest Data Structure	Sample Size Difference	Method Invocations
50,000	1,000	200
500	10	10,000

Table 5-1 - Specification for data samples used by graphs

Each sub-section in Section 5.2 shows two very different graphs. One graph compares the general performance of the various RMI implementations over a very broad range of parameter sizes, 0 to 50,000 nodes. The graph is composed of samples taken at intervals of 1,000 nodes and the sample is calculated from 200 remote method invocations. The other graph focuses on smaller parameter sizes, 0 to 500 nodes with samples taken at intervals of 10 nodes. The sample time is calculated using 10,000 method invocations instead of only 200.

The tests were repeated in two environments. The first environment was two virtual machines running on the same computer and the other environment was a private network between two computers. The network was a 10/100bps Ethernet. The performance ratios between the different implementations were similar in both environments.

5.2 Experimental Results

This section presents the experimental results in three sections. The first section presents the results obtained when the parameter is a heterogeneous tree, Figure 5-1(c). The second section presents the results obtained when the parameter is the partially heterogeneous tree shown in Figure 5-1(b). The third section presents the results obtained when the parameter is a homogeneous binary tree as shown in Figure 5-1(a) and the last section presents the test results when the parameter is a doubly linked list, Figure 5-1(d).

The sections are all organized in the same manner. A data set is presented using a graph and a table. A discussion of the data follows the graph and the table of each data set. Section 5.1 discusses the data gathered.

5.2.1 Heterogeneous Binary Tree

The greatest performance gain of C-RMI is observed when the parameter's object graph is largely heterogeneous. This section compares the performance of IBM's Java RMI, Sun's Java RMI, KaRMI, and C-RMI when the parameter is a heterogeneous binary tree. Figure 5-1(c) shows an example of the data structure. Because the objects in the tree are mostly heterogeneous, the performance of C-RMI is at its best. As the heterogeneity of the binary tree increases, the performance gain of C-RMI increases. Since there is a maximum of 500 classes, the observed performance gain decreases as the tree size gets larger than 500 nodes.

Figure 5-2 compares the performance of four RMI implementations with a parameter that is a heterogeneous tree like the tree shown in Figure 5-1(c).

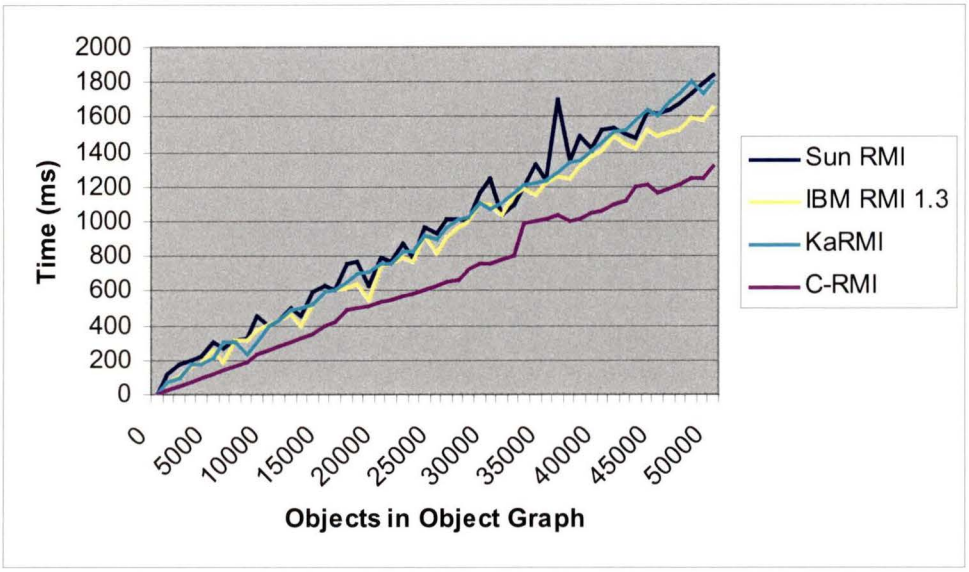

Figure 5-2 - Performance of RMI implementations when the parameter is a heterogeneous tree with 0 to 50000 nodes in the tree.

The comparison in Figure 5-2 shows a large performance gain when the tests are performed with C-RMI, the average improvement and maximum observed improvements are summarized in Table 5-2.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	35%	29%	30%
Maximum	78%	65%	62%

Table 5-2 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a heterogeneous tree with 0 to 50000 nodes.

Table 5-2 and Figure 5-2 report the large improvement obtained with C-RMI. C-RMI shows an average of a thirty percent improvement over the other RMI implementations and a thirty-five percent improvement over Sun's Java RMI. A twenty

to thirty percent improvement is typical for large parameters independent of what data structure is used. Other RMI implementations have only one strategy for sending data, which does not scale up when there is a large amount of parameter data. Because C-RMI compresses the serialized parameter data when the parameters are large, C-RMI has much better performance than other implementations when the parameter size is large.

Figure 5-3 shows the performance of C-RMI when the serialized data is not large enough to be compressed. The parameters used for the tests shown in Figure 5-3 are still heterogeneous trees, see Figure 5-1(c).

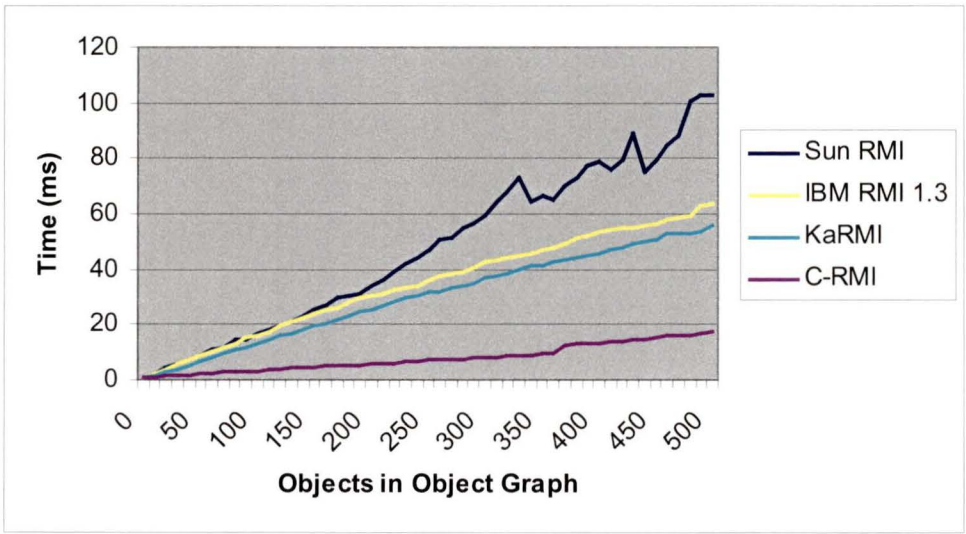

Figure 5-3 - Performance of RMI implementations when the parameter is a heterogeneous tree with 0 to 500 nodes in the tree.

The results in Figure 5-3 show the enormous performance gain that can be obtained by using C-RMI. Table 5-3 summarizes the average and maximum performance gain of C-RMI over the other tested implementations. Using the heterogeneous tree as a parameter, C-RMI executed remote method invocations eighty percent faster than Sun's

Java RMI performed the same method invocation and showed a seventy-six percent performance gain over IBM's RMI implementation.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	81%	76%	71%
Maximum	88%	82%	79%

Table 5-3 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a heterogeneous tree with 0 to 500 nodes.

5.2.2 Partially Heterogeneous Binary Tree

The results shown in Section 5.2.1 showed C-RMI performing at its best; this section and the following sections prove that C-RMI is an improvement over other implementations in all cases. For the tests used to obtain the results shown in this section, the parameter passed was again a heterogeneous tree but with only a few different object types. The formula for determining the number of object types in the tree in terms of the number of nodes is $2\text{ceiling}(\log_2 n-1)-1$, where n is the number of nodes in the tree. Therefore, if there are 129-256 nodes in the tree, there are 13 different object types in the tree. Because the data structure used as a parameter is only partially heterogeneous, the expected performance gain should not be as large as in Section 5.2.1; however, the performance gain should still be substantial. Figure 5-4 and Table 5-4 compare C-RMI with the other RMI implementations when the parameter size is large and when the parameter is the partially heterogeneous binary tree described earlier.

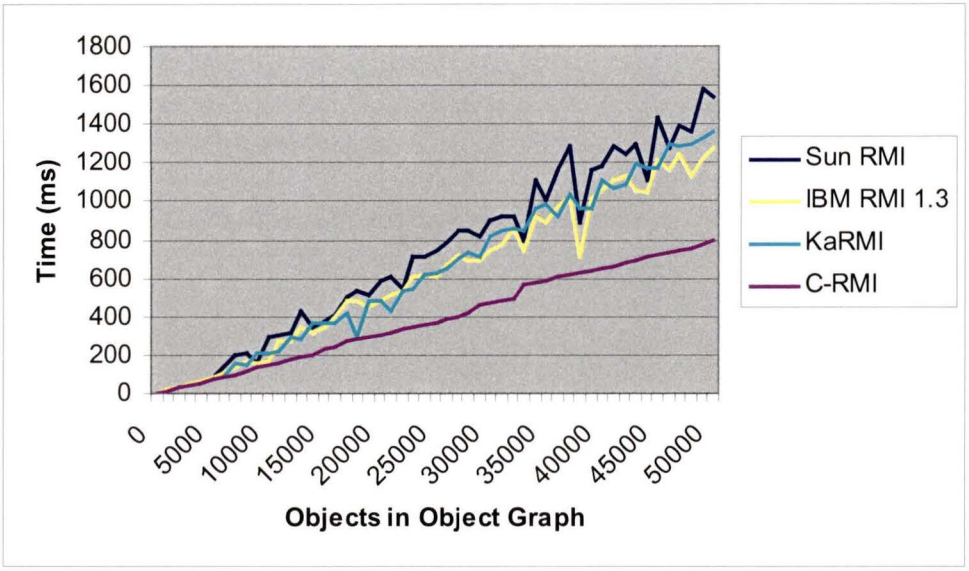

Figure 5-4 - Performance of RMI implementations when the parameter is a partially heterogeneous tree with 0 to 50000 nodes in the tree.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	42%	34%	32%
Maximum	57%	46%	45%

Table 5-4 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a partially heterogeneous tree with 0 to 50000 nodes.

Because none of the other RMI implementations have a strategy for marshalling very large parameters efficiently, C-RMI is more efficient than the other implementations independent of the parameter type. Figure 5-5 and Table 5-5 compares the performance of C-RMI to the other implementations, and, while there is a large improvement over the performance of the other implementations, the performance of KaRMI shows a much-improved performance.

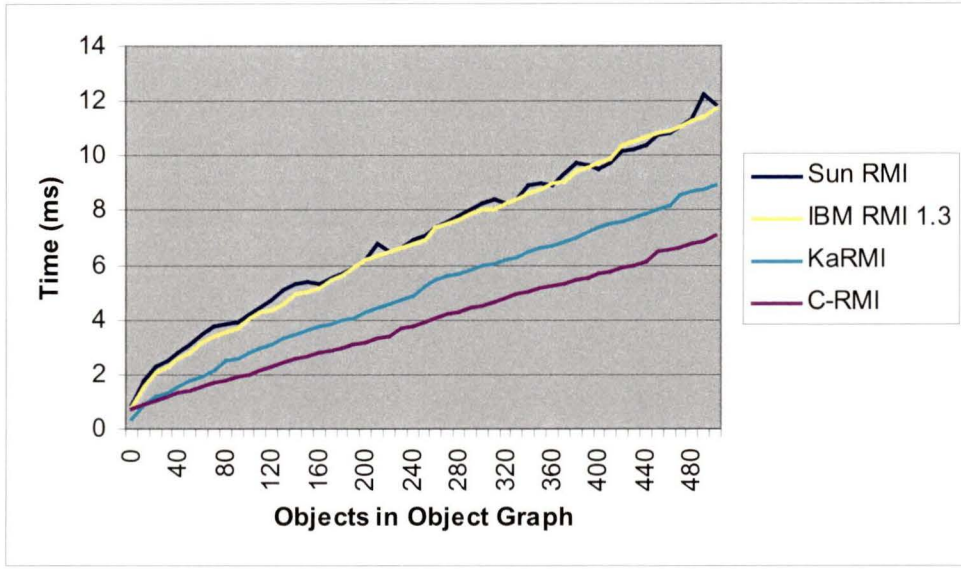

Figure 5-5 - Performance of RMI implementations when the parameter is a partially heterogeneous tree with 0 to 500 nodes in the tree.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	46%	44%	20%
Maximum	55%	50%	28%

Table 5-5 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a partially heterogeneous tree with 0 to 500 nodes.

Table 5-5 shows the average improvement of C-RMI over Sun and IBM's RMI implementations to be forty-five percent, but the improvement over KaRMI, while significant, is only half that. However, it was expected that KaRMI would exhibit better performance than either Sun's implementation or IBM's implementation.

5.2.3 Homogeneous Binary Tree

When the parameter being passed between the client and the remote object is a homogeneous binary tree, C-RMI is expected to perform well but is not expected to demonstrate the huge performance gain observed when a heterogeneous tree is passed. The performance gain is not expected because C-RMI derives a large part of its performance gain by not needing to analyze the parameter classes during runtime. Other implementations cache the analyzed data so when an object graph only has one class type the runtime analysis is only performed once, causing C-RMI to lose the gain achieved by compiling serialization data. Figure 5-6 compares the performance of the RMI implementations. However, the more streamlined execution and optimized buffering still provide significant performance gains.

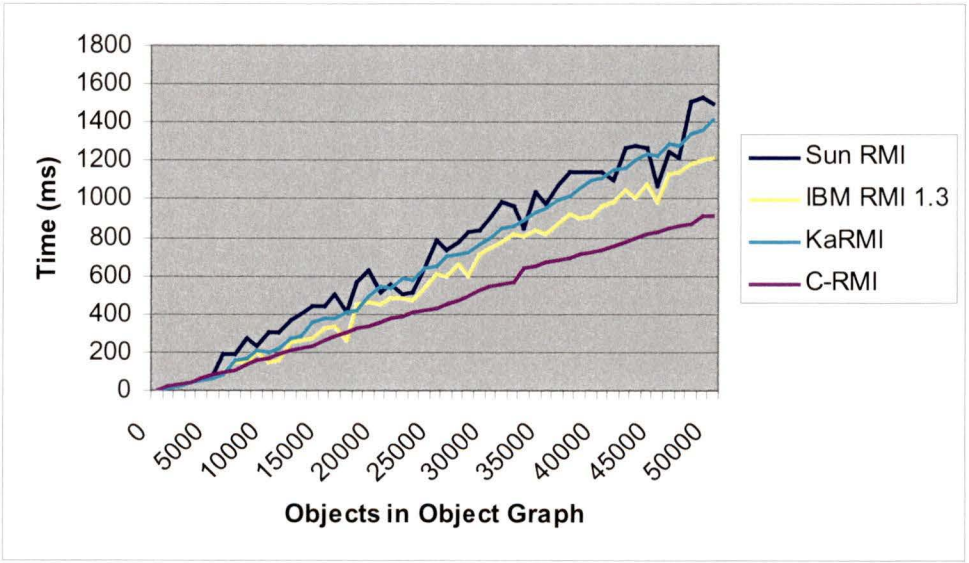

Figure 5-6 - Performance of RMI implementations when the parameter is a homogeneous tree with 0 to 50000 nodes in the tree.

While the performance gain is not as large as when the parameter is a heterogeneous tree, there is still a major improvement. As the parameter size increases, so does the performance gain observed with C-RMI, thanks to the customized data buffering and data compression. Table 5-6 shows the average and maximum observed performance gain of C-RMI over the other implementations.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	33%	16%	23%
Maximum	50%	30%	36%

Table 5-6 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous tree with 0 to 50000 nodes.

When the parameter size is small, some of the buffering advantages and all of the data compression advantages are lost; as a result, the performance improvement is not as great as when the parameter size is large. Figure 5-7 compare the performance of the RMI implementations when the tree size size ranges from 0 nodes to 500 nodes.

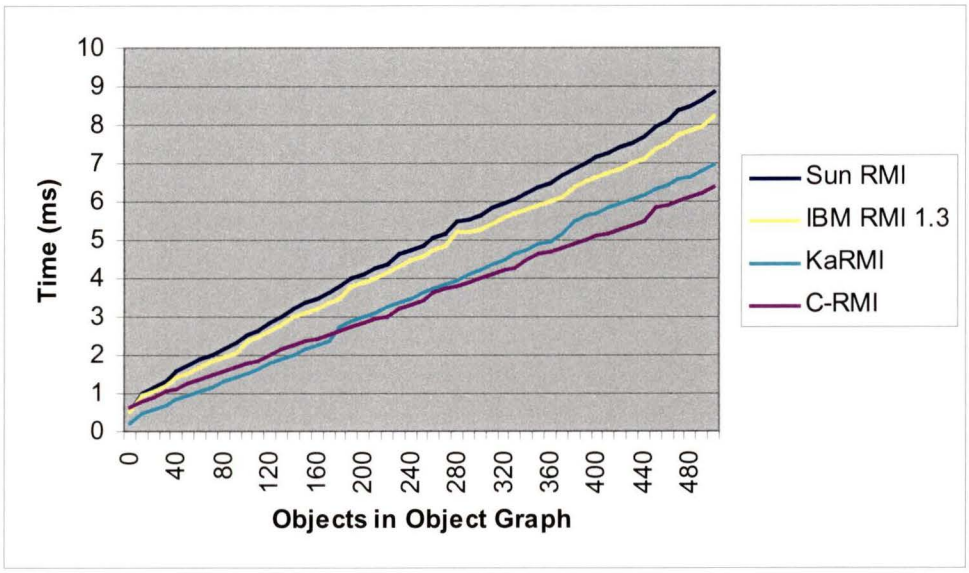

Figure 5-7 - RMI implementation performance when the parameter is a homogeneous tree of size 0 to 500 nodes.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	27%	22%	-3%
Maximum	32%	28%	12%

Table 5-7 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous tree with 0 to 500 nodes.

When smaller parameter sizes are passed to a remote method, KaRMI is very efficient and the performance of C-RMI is a little worse than for KaRMI. However, as the parameter size increases KaRMI loses its efficiency and performs worse than C-RMI. As seen in Sections 5.2.1 and 5.2.2, KaRMI also performs poorly when more classes are introduced to the parameter data structure. Table 5-7 presents the gain observed in C-RMI. The performance gains observed in KaRMI are a result of the great deal of optimization made for efficient serialization of small homogeneous data structures.

5.2.4 Doubly Linked List

The doubly linked list was developed to test the performance of C-RMI when the object graph contains cycles. Since the doubly linked list is entirely homogeneous, the performance was expected to be similar to that of the homogeneous binary tree. The results obtained, however, were completely unexpected. When the parameter size was small, the different implementations exhibited similar performance as expected, except for IBM RMI, which performed poorly. When the parameter became large, the surprising feature became apparent. It was discovered that C-RMI was capable of marshalling and demarshalling lists that are five to ten times longer than the other implementations. Figure 5-8 compares the performance of the RMI implementations and Table 5-8 shows the average and maximum performance gain of C-RMI compared to the other implementation. The numbers in Table 5-8 are calculated from the data samples where all implementations completed the tests.

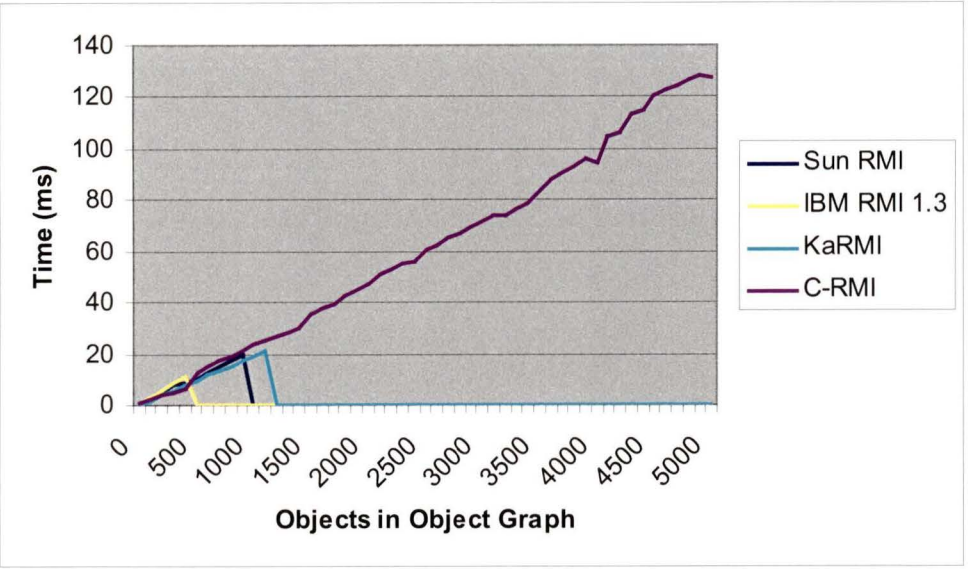

Figure 5-8 - RMI implementation performance when the parameter is a homogeneous doubly linked list of size 0 to 5000 nodes.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	3%	26%	-16%
Maximum	37%	41%	17%

Table 5-8 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous doubly linked list with 0 to 5000 nodes.

From Figure 5-8, it is clear that IBM's Java RMI was unable to marshal a doubly linked list when the list is longer than 400 nodes, Sun's Java RMI was capable of marshalling lists as long as 1,100 nodes and KaRMI was capable of marshalling lists as long as 1,400 nodes. C-RMI is capable of serializing a doubly linked list of 5,600 nodes.

C-RMI is capable of marshalling a doubly linked list that is at least four times as long as the other implementations because the serializing strategy used is much more streamlined than the other serialization implementations. The other RMI

implementations crash when the call stack overflows. Because C-RMI only pushes one method call onto the stack for each level of recursion, it is capable of serializing lists that are at least four times longer than the other RMI implementations. Another test performed to support this theory passed a normal list as a parameter and all the implementations performed the same, see Section 5.3. IBM RMI still crashed when the list became longer than 400 nodes.

As mentioned earlier, the performance of the RMI implementations, before they crash, are approximately the same as for a homogeneous binary tree. Figure 5-9 and Table 5-9 compare the performance of the different RMI implementations. Only IBM RMI performs differently from when a homogeneous tree is passed as the parameter.

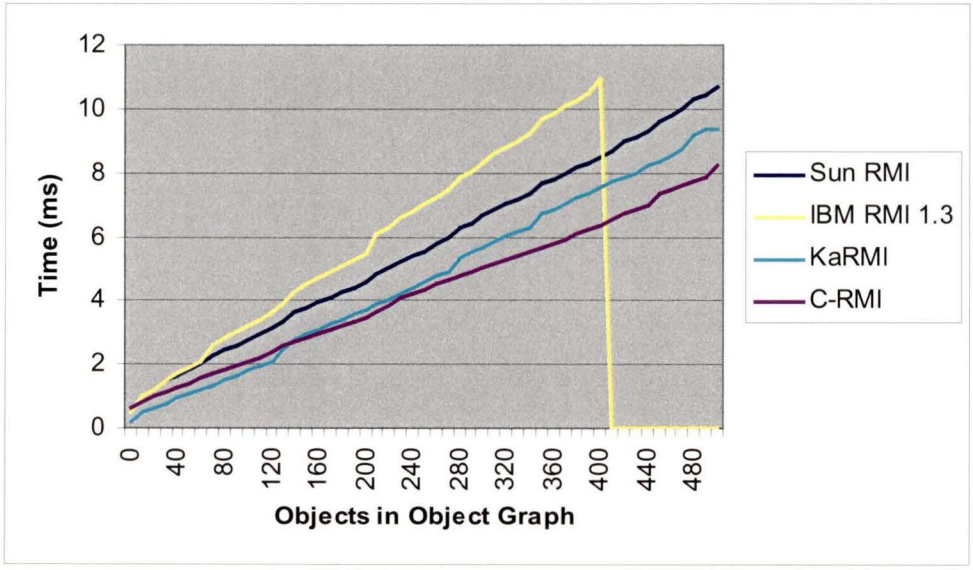

Figure 5-9 - RMI implementation performance when the parameter is a homogeneous doubly linked list of size 0 to 500 nodes.

	Sun RMI	IBM RMI 1.3	KaRMI
Average	23%	35%	1%
Maximum	26%	42%	17%

Table 5-9 - Average and Maximum observed performance gain of C-RMI over other RMI implementations when the parameter is a homogeneous doubly linked list with 0 to 500 nodes.

5.3 Discussion

It was intended that the results reported in this chapter would show the areas of strengths and weaknesses of each RMI implementation, especially C-RMI and KaRMI because both of these implementations have been greatly optimized. KaRMI is most efficient when the parameter size is small and the parameter is a homogeneous object graph while, in contrast, C-RMI is most efficient when the object graph is heterogeneous. As the size of the parameter grows larger, the performance gain of KaRMI compared to Sun's RMI or IBM's RMI decreases; for C-RMI, large performance gains are observed under virtually all circumstances. When the parameter size is small, the performance is the same as that of Sun's RMI or IBM's RMI; however, as the object graph grows larger, C-RMI exhibits more efficient performance. The speed of the performance increase is related to the heterogeneity of the object graph. Object graphs that have a large number of object types exhibit a larger performance gain than object graphs that have a small number of object types. As indicated above, C-RMI, compared to the other implementations, maintains a large performance gain even when the object graph is large.

A source of interest is the incredible performance of KaRMI when the parameter size is small. All the implementations, Sun's RMI, IBM's RMI and C-RMI, all take

When C-RMI needs to obtain or set the value of an object variable, the reflection get-and-set methods are used. These methods are very inefficient, so an alternative is advantageous. The suggested alternative would be to develop two native methods. The first method would take an object as its parameter and would return all the object's field values as an object array. The other method would take an object array and an object as parameters and would set all the fields of the object to the values in the array. Instead of making numerous reflection calls to get and set an object's fields, this strategy would make only one call to set all the fields and one call to get all the fields; this strategy should yield some performance improvements.

As observed in Chapter 5, the performance of KaRMI is very impressive when the parameter size is very small, less than ten or twenty nodes. If the reason for the impressive performance could be understood, C-RMI could benefit from a similar optimization.

The current C-RMI implementation uses an interpretive serialization strategy when a subclass of the expected parameter needs to be serialized. A method that could improve the performance would be to compile a class descriptor for the subclass during runtime and use the class descriptor to serialize the object. It is difficult to predict whether this strategy would produce any significant performance gains because the current implementation only uses the interpretive serialization for the unknown classes. The expected parameter and all of its superclasses are serialized using the compiled class descriptors. Experimentation would be required to determine whether this is a useful strategy.

Chapter 7 - Bibliography

[1] Birrell, A.D., and Nelson, B.J.: "Implementing Remote Procedure Calls, " ACM Trans. on Computer Systems, vol. 2, pp. 39-59, Feb 1984

[2] Krishnaswamy V., Walther D., Bhola S., Bommaiah E., Riley G., Topol B., Ahamad M., "Efficient implementation of Java remote method invocation (RMI)," In Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98), pages 19-35, Santa Fe, New Mexico, April 1998. Online at http://www.cc.gatech.edu/~kv/publications/rmicoots98.ps

[3] Maassen J., Kielmann T., Bal H., "Efficient Replicated Method Invocation in Java," Proc. ACM 2000 Java Grande Conference, San Francisco, CA, June 3-4, 2000. Online at http://www.cs.vu.nl/~rob/papers/jg00.pdf

[4] Maassen J., Kielmann T., Bal H., "Parallel Application Experience with Replicated Method Invocation," Concurrency and Computation: Practice and Experience, 2001, John Riley & Sons Ltd. Online at http://www.cs.vu.nl/~rob/papers/cpe01-repmi.pdf

[5] Maassen J., Nieuwpoort R., Veldema R., Bal H., Kielmann T., Jacobs C., Hofman R. "Efficient Java RMI for Parallel Programming," ACM Transactions on Programming Languages and Systems (TOPLAS) 2001. Online at http://www.cs.vu.nl/~rob/papers/toplas2001.pdf

[6] Maassen J., Nieuwpoort R., Veldema R., Bal H., Plaat A.: "An Efficient Implementation of Java's Remote Method Invocation," ACM Symposium on Principles and Practice of Parallel Programming, Atlanta, Georgia, May 1999

[7] Nieuwpoort R.: "Fast Parallel Java," Master's thesis Vrije Universiteit Amsterdam, August 1998. Online at http://www.cs.vu.nl/~rob/papers/scriptie_jason_rob.pdf

[8] Menage P.: "Improved code sharing in dynamically generated marshalling routines," Online at http://www.chiark.greenend.org.uk/~paulm/papers/gc99.ps

[9] Nelisse A., Kielmann T., Bal H., Maassen J.: "Object-based Collective Communication in Java," Proc. Joint ACM Java Grande - ISCOPE 2001 Conference, Stanford University, CA, June 2-4, 2001. Online at http://www.cs.vu.nl/~rob/papers/javagrande2001-jason.pdf

[10] Nelson B.J.: "Remote Procedure Call, " CMU PhD Thesis, 1981.

[11] Nester C., Philippsen M., Haumacher B.: "A More Efficient RMI for Java," *Proc. ACM 1999 Java Grande Conference*, San Francisco, June 12-13, 1999. Online at http://wwwipd.ira.uka.de/~phlipp/mypapers/rmi.ps.gz

[12] Philippsen M., Haumacher B., "More Efficient Object Serialization," *Parallel and Distributed Processing*, LNCS 1586, pp 718-732, International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 12, 1999. Online at http://wwwipd.ira.uka.de/~phlipp/mypapers/betterserial.ps.gz

[13] Rajeev R. Raje, Joseph Williams, Michael Boyles, "An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java, " Concurrency: Practice and Experience, Vol. 9(11), 1207-1211 (November 1997).

[14] Sun Micro Systems: "Java™ 2 SDK, Standard Edition Documentation". Online at http://java.sun.com/j2se/1.3/docs/index.html

[15] Tanenbaum A.S.: Distributed Operating Systems, Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1995.

[16] Veldema R., and Bal H.: "Optimizing Java specific overheads, Java at the speed of C ?," Proc. *HPCN 2001.* Online at http://www.cs.vu.nl/~rob/papers/hpcn01.pdf

UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain by the University of Victoria shall not be allowed without my written permission.

Title of Thesis:

C-RMI: An Efficient Alternate RMI Implementation

Author 

Jesse Dale Eichar

December 1, 2002

VITA

Surname: Eichar Given Names: Jesse Dale

Place of Birth: Burns Lake, British Columbia, Canada

Educational Institutions Attended:

University of Victoria	1999-2002
University of Northern British Columbia	1994-1999

Degrees Awarded

B.Sc.	University of Northern British Columbia	1999

Honours and Awards:

University of Victoria Fellowship	1999-2001

Publications:

_____

_____

_____

_____

_____

_____

_____

_____

UNIVERSITY OF VICTORIA PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain by the University of Victoria shall not be allowed without my written permission.

Title of Thesis:

C-RMI: An Efficient Alternate RMI Implementation

Author 

Jesse Dale Eichar

December 1, 2002