UNIFIED LANGUAGE AND OPERATING SYSTEM SUPPORT FOR PARALLEL PROCESSING

by

Donald Acton
B.Sc. (Computer Science) University of British Columbia, 1982
M.Sc. (Computer Science) University of British Columbia 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
April 1994
©Donald Acton 1994
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of _COMPUTER SCIENCE_

The University of British Columbia
Vancouver, Canada

Date _April 26/1994_
Abstract

The programming of parallel and distributed applications is difficult. The proliferation of networks of workstations, combined with the advent of shared memory-machines for desktop and office use, is making parallel and distributed environments more commonplace. And, there is an increasing demand for general purpose applications to be able to make use of these extra processing resources.

This thesis explores the adding of language and system support for general purpose parallel and distributed programming to the object-oriented language and system Raven. The system is targeted for operation in shared-memory and distributed-memory environments where the processors are general purpose and execute their own independent instruction streams. Support for parallelism forms a fundamental part of Raven. To simplify the creation of parallelism Raven introduces the concepts of class-based and user-based parallelism. Class-based parallelism is parallelism created within a class and it is realized through early and delayed result. User-based parallelism results through the use of a class and is realized through companions and certificates. Raven also supplies automatic concurrency control, even across machine boundaries. Concurrency control is attached to an object through the more general property mechanism. Properties are a new way of supplying system services to objects on an individual object basis. Raven also introduces the notion of invoke streams which are a way for a third party to sequence invocation requests targeted at the same object.

To demonstrate the viability of these ideas, an extensive implementation was performed. Raven runs on several different machines and operating systems including a 20 processor shared-memory machine. To demonstrate the usability of the parallel constructs, and the efficiency of the implementation, several parallel applications were written and performance measurements made. Included are implementations on shared-memory and distributed-memory machines that are identical except for a few lines of code specifying the system configuration. A distributed mail system is also presented to highlight the support for writing distributed applications.
CHAPTER 3 The Raven System and Language Overview

3.1 The Passive and Active Object Models
3.2 Raven
3.2.1 The Raven Language
3.2.2 Raven Class Library
3.2.3 The Raven Compiler
3.2.4 The Local Raven Environment
3.2.5 The Virtual Machine
3.2.6 Supporting Distribution in Raven

CHAPTER 4 Support for Parallel and Distributed Computation in Raven

4.1 Specifying Parallelism in Raven
4.2 Class-Based Parallelism
4.2.1 Early Result
4.2.2 Delayed Result
4.3 User-Based Parallelism
4.3.1 Certificates and Companions
4.3.2 InvokeStreams
4.4 Properties
4.5 Object Concurrency Control
4.5.1 Locking Costs
4.5.2 Locking Duration
4.5.3 Locking problems
4.5.4 Locking and Early Result
4.5.5 Locking and Delayed Result
4.5.6 Deadlock Handling
4.6 System Resource management
### Table of Contents

#### CHAPTER 4 Implementation Details and Issues

4.6.1 Memory .............................................................105  
4.6.2 Processor Management ........................................105  

4.7 Object Transparency .............................................110  
4.8 The System Object ................................................118  
4.9 Failures and Exceptions ..........................................119  
4.10 Unique Raven Language Features ..............................121  
4.11 Summary ............................................................125  

#### CHAPTER 5 Implementation Details and Issues ...................126

5.1 Object Implementation .............................................126  
5.1.1 Object Organization ...........................................128  
5.1.2 Global IDs ......................................................129  
5.1.3 Class Objects ..................................................132  
5.1.4 Properties .......................................................134  
5.1.5 Method Invocation and Lookup ................................134  

5.2 Memory Allocation and Garbage Collection ....................142  
5.2.1 Modifications to the memory allocation system ..........144  

5.3 The Thread Environment ..........................................149  
5.3.1 The Thread Class ..............................................150  
5.3.2 Companions .....................................................152  
5.3.3 Early Result .....................................................154  
5.3.4 Delayed Result ................................................156  

5.4 Remote Invocation ................................................158  
5.4.1 Class relationships ..........................................160  
5.4.2 Avoiding Copying Loops .....................................166  
5.4.3 Object Encoding ..............................................168  
5.4.4 Locating Objects .............................................169  
5.4.5 Name Server ...................................................170  
5.4.6 InvokeStreams ................................................172  
5.4.7 Lock Management .............................................174  
5.4.8 Remote Invocation Failures ...................................175  

5.5 Summary ............................................................176  

#### CHAPTER 6 Performance ............................................177

6.1 Performance Metrics ..............................................177  
6.1.1 Speedup and Efficiency ......................................179  

6.2 Example Problems ................................................183  
6.2.1 Mandelbrot Set ...............................................185  
6.2.1.1 Problem Description ......................................185  
6.2.1.2 Performance Results ......................................187  

6.2.2 Bayesian Search ..............................................188  
6.2.2.1 Problem Description ......................................189  
6.2.2.2 Performance Results ......................................193  

6.2.3 Prime Number Generation ....................................198
6.2.3.1 Problem Description ........................................... 198
6.2.3.2 Performance Results .......................................... 200
6.2.4 A Distributed Version of the Mandelbrot Computation ........... 202
6.2.5 A Distributed Mail Application .................................. 205
6.2.6 Miscellaneous Applications ...................................... 208
6.3 Performance Analysis Overview .................................... 209
6.4 Summary .................................................................... 210

CHAPTER 7  Conclusions and Future Research ....................... 211
7.1 Summary of Results .................................................. 211
  7.1.1 Parallelism at the Language Level ............................. 212
  7.1.2 System support .................................................... 214
  7.1.3 Conclusions ....................................................... 215
7.2 Future Work ............................................................. 216

Bibliography ..................................................................... 219

Glossary ........................................................................... 231

Appendix A  Table of Contents .......................................... 235

Appendix A  The Raven Language ....................................... 237
  A.1 Introduction .......................................................... 237
  A.2 BNF Syntactic Notation ............................................. 238
  A.3 Syntax of the Raven Language .................................... 239
    A.3.1 Basic Building Blocks ............................................. 239
      A.3.1.1 Comments ..................................................... 239
      A.3.1.2 Strings ............................................................ 239
      A.3.1.3 Numbers .......................................................... 239
      A.3.1.4 Reserved Words ............................................... 240
      A.3.1.5 Punctuation .................................................... 243
      A.3.1.6 Identifiers and Types ........................................... 244
    A.3.2 Raven BNF .......................................................... 244
    A.3.3 A Raven Program ................................................ 246
    A.3.4 Declaring Classes and Variables .............................. 246
      A.3.4.1 Parameter and Variable Declarations .................. 246
      A.3.4.2 Type Definitions and Declarations ....................... 248
      A.3.4.3 Self, Me, and Super ........................................... 248
      A.3.4.4 Behavior Declaration .......................................... 249
      A.3.4.5 Class Declaration .............................................. 250
      A.3.4.6 Parameterized Types ......................................... 253
    A.3.5 Control Statements ............................................... 254
    A.3.6 Defining Behaviors and Manipulating Objects ............... 254
      A.3.6.1 Behavior Definition .......................................... 254
A.3.6.2  Behaviors with a Variable Number of Arguments ..........255
A.3.6.3  Method Invocation........................................256
A.3.6.4  Creating Objects ........................................257
A.3.6.5  Casting ......................................................258
A.3.7  Support for Parallelism........................................259
    A.3.7.1  Early and Delayed Result ............................259
    A.3.7.2  Certificates and Companions ..........................261
A.3.8  Type Checking ..............................................262
A.3.9  Locking .......................................................263
A.3.10  Properties ..................................................265
A.4  Support Classes .................................................267
    A.4.1  Object ....................................................268
    A.4.2  Class ........................................................269
    A.4.3  Certificate ................................................270
    A.4.4  CertificateGroup ..........................................272
    A.4.5  Thread ......................................................272
    A.4.6  CompanionThread ..........................................274
    A.4.7  InvokeStream ..............................................275
    A.4.8  System .....................................................276
    A.4.9  NameServer ...............................................277
    A.4.10  Miscellaneous Classes .................................277
A.5  The Raven Compiler .............................................278
List of Tables

TABLE 2.1  Summary of language/system support for parallelism. .......................... 44
TABLE 6.1  SPEC results for selected processors ..................................................... 203
TABLE 6.2  Elapsed time and speedup for distributed Mandelbrot computation. ....... 204
## List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Target environment</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Objects invoking on each other.</td>
<td>51</td>
</tr>
<tr>
<td>3.2</td>
<td>Raven pencil cup example</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Pencil cup class hierarchy</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td><em>Eraser</em> as a subtype of <em>Weighable</em>.</td>
<td>57</td>
</tr>
<tr>
<td>3.5</td>
<td>Class organization</td>
<td>59</td>
</tr>
<tr>
<td>3.6</td>
<td>Tagging a parameter as copyable.</td>
<td>60</td>
</tr>
<tr>
<td>3.7</td>
<td>Objects with different invoke routines.</td>
<td>65</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of a remote invocation.</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td>Example code demonstrating early result</td>
<td>72</td>
</tr>
<tr>
<td>4.2</td>
<td>Use of delayed result</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Examples of certificates and companions.</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Example use of <em>CertificateGroups</em>.</td>
<td>79</td>
</tr>
<tr>
<td>4.5</td>
<td>Use of <em>CertificateGroups</em> and tags.</td>
<td>80</td>
</tr>
<tr>
<td>4.6</td>
<td>Some potential parallel invocations with serial components</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Example stream creation and use.</td>
<td>83</td>
</tr>
<tr>
<td>4.8</td>
<td>Creating objects with different properties.</td>
<td>91</td>
</tr>
<tr>
<td>4.9</td>
<td>Concurrent object access</td>
<td>93</td>
</tr>
<tr>
<td>4.10</td>
<td>Object invoking back on self</td>
<td>98</td>
</tr>
<tr>
<td>4.11</td>
<td>Lock Sharing</td>
<td>99</td>
</tr>
<tr>
<td>4.12</td>
<td>Sequential code to update a display.</td>
<td>106</td>
</tr>
<tr>
<td>4.13</td>
<td>Conversion of sequential code to parallel version.</td>
<td>106</td>
</tr>
<tr>
<td>4.14</td>
<td>Object interactions in a distributed Raven environment.</td>
<td>114</td>
</tr>
<tr>
<td>5.1</td>
<td>Object organization in memory</td>
<td>128</td>
</tr>
<tr>
<td>5.2</td>
<td>Format of global ID information</td>
<td>130</td>
</tr>
<tr>
<td>5.3</td>
<td>Instance data of an object of type class.</td>
<td>133</td>
</tr>
<tr>
<td>5.4</td>
<td>Pseudo code illustrating steps involved in performing an invoke</td>
<td>135</td>
</tr>
<tr>
<td>5.5</td>
<td>Processor efficiency and memory allocation delays when no heaps are used.</td>
<td>145</td>
</tr>
<tr>
<td>5.6</td>
<td>Processor efficiency and memory allocation delays when multiple heaps are used.</td>
<td>146</td>
</tr>
<tr>
<td>5.7</td>
<td>Layers of the Raven system</td>
<td>150</td>
</tr>
<tr>
<td>5.8</td>
<td>Actions undertaken upon companion creation</td>
<td>153</td>
</tr>
<tr>
<td>5.9</td>
<td>Pseudo-code illustrating early result</td>
<td>155</td>
</tr>
<tr>
<td>5.10</td>
<td>Delayed result example and pseudo code describing actions taken by system.</td>
<td>157</td>
</tr>
<tr>
<td>5.11</td>
<td>Remote invocation sequence</td>
<td>159</td>
</tr>
<tr>
<td>5.12</td>
<td>Example of the data fields for an invocation request.</td>
<td>164</td>
</tr>
<tr>
<td>5.13</td>
<td>Creating a name server</td>
<td>171</td>
</tr>
<tr>
<td>5.14</td>
<td>Example of stream usage</td>
<td>172</td>
</tr>
<tr>
<td>6.1</td>
<td>Subset of Raven code to perform the Mandelbrot computation</td>
<td>186</td>
</tr>
<tr>
<td>6.2</td>
<td>Mandelbrot set speedup</td>
<td>188</td>
</tr>
<tr>
<td>FIGURE 6.3</td>
<td>Subsection of multiple-bit adder. .................................................. 189</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.4</td>
<td>The main body of the execute method in the Worker class. .................. 192</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.5</td>
<td>Sequential and parallel control portions of Bayesian search. .............. 193</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.6</td>
<td>Speedup for Bayesian search space. ................................................ 194</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.7</td>
<td>Barrier class to synchronize and time a collection of Threads. ............ 196</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.8</td>
<td>Speedup vs. processors using a modified measuring technique. ............ 197</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.9</td>
<td>Sequential and parallel control portions prime number search. .......... 199</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.10</td>
<td>Speedup vs. processors for prime number searching. ........................ 200</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.11</td>
<td>Efficiency vs. processors for the prime number searching routine. ....... 201</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.12</td>
<td>Partial results of the Mandelbrot computation performed in a distributed environment. ......................................................... 204</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.13</td>
<td>A distributed mail application. ....................................................... 206</td>
<td></td>
</tr>
<tr>
<td>FIGURE 6.14</td>
<td>Example early result usage in distributed mail application. ............... 207</td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgement

I would like to thank my supervisor, Gerald Neufeld, for his help, guidance, and prodding in completing this thesis. My thesis committee of Norm Hutchinson, Jim Little, Alan Wagner, Sam Chanson, and Peter Lawrence, also deserves recognition for their comments and help when requested.

Grad school would not have been the enjoyable experience it was without the many people who have become my friends and colleagues over the last few years. To Ian, George, Terry, Peter, Barry, Murray, Carlin, the Mikes, Andrew, Andy, and all the people I am forgetting, thanks.

A special thanks goes to the Amor family, who have adopted me and become my family away from home. Finally, I would like to thank my parents for their constant support and love during this endeavour.
Parallelism and distribution of operations and functions are ubiquitous in everyday tasks in the real world. A person might wash dishes while watching television, dry one load of clothes while washing another, or simultaneously cook the peas, carrots, corn, and potatoes for a meal. However, parallelism is not confined to daily activities: computer systems use it too. A computer system itself typically engages in parallel activities like performing simultaneous reads and writes to disk drives, or accessing a network while displaying information on a console. In addition, a timesharing system extends the notion of parallelism by providing the illusion of running multiple programs at once. After leaving the domain of the operating system, however, and advancing to the application program, any appearance of parallelism has usually disappeared and the programmer is provided with a sequential model to program in. Of course, operating in a sequential environment makes the task of programming easier for many applications; however, those tasks that inherently involve some degree of parallelism are more difficult to program when parallel activities are constrained to operate sequentially.
The need for programs to be able to express and harness parallelism is not strictly confined to applications that have a natural parallel expression but are being forced to adapt to a sequential environment. The increasing power of microprocessors, the proliferation of computer networks, and the development of multiprocessors is increasing the need for applications to be able to deal with parallel and distributed computing. Problems once confined to supercomputers are now being tackled by workstations, and the processor cycles in idle computers attached to networks represent untapped resources. Some mechanism is needed to operate in these types of parallel and distributed environments.

The increased power of microprocessors is also creating the demand for applications that have substantial parallel components. For example, machines targeted at providing multimedia environments are now available. With this environment a program can be expected to record an audio message while playing back a stream of audiovisual data in real time. This data might be obtained from a local CD-ROM or from a continuous-media file system accessed through a high speed network. Certainly, these types of environments will demand more of the programmer with respect to the number and variety of devices that a program will need to control simultaneously. This correspondingly increases the demand for facilities to manipulate and control parallelism from within the application. Additionally, limiting an application program to using a serial approach needlessly restricts the problem-solving techniques that can be used on inherently parallel applications. For example, the processing of a video stream that needs to be synchronized with an audio stream involves processing operations that are logically separate, yet require synchronization for playback. Although applications of this nature are in their infancy, they will become more commonplace as the processor speeds increase and more sophisticated applications are developed. The result is an increasing need to support parallel and distributed computing.
1.1 The General Problem

To harness the power of parallelism it is imperative that parallel systems be easy to use while simultaneously providing the developer with the software tools and run-time environment required to achieve maximum performance. The advent of small, inexpensive, powerful microprocessors has made it possible to assemble collections of processors into parallel machines with potentially supercomputer performance. The result is that parallel machine environments are becoming more accessible and commonplace.

For parallel machines to come into common usage, a change in the development environment and run time support for parallel systems is required. Changes will be needed, similar to those that revolutionized the uniprocessor world as its programming environment advanced from toggle switches to high level languages. If the parallel programming revolution succeeds it will result in a computing environment capable of allowing a large class of programs to be programmed easily and run efficiently. Until the task of programming a parallel application is comparable in simplicity to that of programming a sequential application, the use of parallel and distributed computing will remain restricted.

1.1.1 Hardware Boundaries

Within the class of general purpose machines, two trends are emerging with respect to computing environments. One trend is the proliferation of workstation or desktop class machines connected via networks; the other is the emergence of shared-memory multiprocessor machines. Several vendors are already offering, or are expected to offer, shared-memory systems targeted for desktop and office use [85][90][25]. Both the shared-memory machines and networks of connected machines can be characterized as collections of independent processors that execute their own instruction streams. Machines of this type are called multiple instruction multiple data (MIMD) machines. The processors can communicate with each other either by shared memory, a network, or both. Figure 1.1 depicts one possible computing configuration. In the figure, circles represent the independent processors, the large rectangle represents shared mem-
ory, and the thick black line represents a network. The system essentially consists of a shared-memory machine with four processors sharing a network with three other processors.

As high-speed networks become prevalent, and shared-memory machines become common, it is reasonable to expect that the computing environments of the future will consist of some collection of shared-memory processors connected to other machines by a high-speed network. With the development of such systems, there will be a demand for applications that effectively use the processor resources of such systems. The problem of providing support for parallel and distributed computing, as addressed by this thesis, is narrowed by restricting the systems to operate in a shared- or distributed-memory system where each processor is autonomous and executes its own instruction stream.
1.1.2 Software Boundaries

When writing a program, the programmer must decide upon the general organization of the program and the degree and type of functionality the application is to have. Once the basic operating parameters for the application have been defined, the programmer must then decide upon the data structures, the functions and procedures, and, ultimately, the actual code that implements the solution. These tasks are difficult in their own right, and the programming of parallel and distributed applications greatly increases the magnitude of the problem. In parallel and distributed programming the programmer must also be concerned with decomposing the application into processes, deciding how processes are to communicate with each other, synchronizing process execution, and controlling access to shared resources. In addition, the programmer may also be concerned about the particular hardware environment the application is to run in. The approach used to solve a particular problem may be highly dependent upon whether the application will be running in a shared-memory or distributed-memory environment. For example, in a distributed environment the programmer may have to be concerned with how data is exchanged between machines and how to deal with any incompatibilities that result from running in a heterogenous processor environment, and in a shared-memory environment the potential for data access conflicts is greatly increased.

In both shared-memory and distributed-memory environments, parallel programming packages have concentrated on programming facilities for scientific applications where high performance is crucial. Systems like the NX/2 operating system for Hypercubes [82], and Chameleon [6] and vendor specific packages [89] for shared-memory machines illustrate this point. These systems sacrifice ease of use for performance. Little support is available to allow more general purpose application to make use of parallelism. Yet, the proliferation of threads packages [33][79] strongly suggests the desire to use multiple threads of control as a way for general purpose applications to use parallelism as a structuring technique for inherently parallel tasks and as a way to improve performance for desktop shared-memory multiprocessor machines. Although threads packages provide the bare tools for writing parallel programs, they address
few of the difficult problems associated with writing parallel and distributed applications described in the previous paragraph.

A good example of a general purpose application that could make better use of parallelism is a graphical user interface to a system. On the display, when a button is pressed and a long running action initiated, it should be possible to press other buttons and have additional requests serviced. Currently, applications generally freeze and do not accept additional input until the current task is completed. If parallelism were easier to create and manage, the task of servicing multiple requests simultaneously would be simplified.

The problem of providing support for parallel and distributed computing as addressed by this thesis is narrowed by restricting the systems to operate in a shared- or distributed-memory system where each processor is autonomous and executes its own instruction stream. In the software environment, this work will be restricted to providing support for parallelism for general purpose applications where the emphasis is on moderate performance improvements in the hardware environments described in the previous section. This should not preclude having programming language constructs that allow the writing of classical performance oriented scientific applications; rather, the focus is on ease of use in writing the general purpose system level software and not on raw performance.

1.2 Thesis Statement

The software currently existing for the programming of general purpose applications in shared-memory and distributed-memory environments that is independent of the base operating system and the underlying hardware is in a primitive state. The programming abstraction used in programming these environments can be greatly simplified by developing an object-oriented language and system that provides a single unified object-view of system services, language constructs, and object location. Furthermore, the additional processing power available from these processor configurations can be harnessed by embedding support for parallel and distrib-
uted processing directly into the language and system so that any object can be the target of parallel requests. The requirements for such a system can be met by:

- **Taking advantage of the object-oriented paradigm to identify the unit of parallelism and the type of actions that can be performed in parallel.**

- **Integrating operating system services directly into the language, thereby freeing the application program from direct interaction with any operating system interface that is dependent upon the underlying system.**

- **Providing a uniform object access environment across shared-memory and distributed-memory environments, thereby freeing the programmer from the need to manage how objects in different environments interact.**

- **Implementing the language/system on shared-memory and distributed-memory hardware, thereby demonstrating the viability of the ideas, the portability of the system, the capability to operate in a heterogeneous environment, and the ability of all the components to operate together.**

- **Undertaking performance measurements, thereby demonstrating the effectiveness of the system for parallel and distributed computation.**

### 1.3 Contributions

The research contributions of this thesis work are detailed below, with further explanation appearing in section 1.4.

1. The development of a model for parallelism in object-oriented systems. The model exploits the strong data encapsulation of methods and the method invocation interface to specify parallelism. The parallelism can be specified either by the user of a method (user-based parallelism), or by the implementor of a method (class-based parallelism).

2. Class-based parallelism is realized through early and delayed result, while user-based parallelism is realized through certificates and companions.

3. The introduction of invocation streams as a way to serialize invocation requests targeted for the same object.

4. The introduction of the notion of properties as a convenient way to provide operating system services to objects on an instance-by-instance basis, while avoiding the class explosion problem.
5. The specification of a target virtual machine interface to aid portability. The usefulness of this was demonstrated by porting the Raven system to several different types of hardware running different operating systems. The porting operation was accomplished by efficiently implementing the virtual machine, using the services of the native system.

6. The provision of system-level support for automatic concurrency control to relieve the programmer of the task of managing concurrent access to objects. This is demonstrated by developing single-threaded applications in a uniprocessor environment and converting them to run in parallel on shared-memory multiprocessors, without adding additional code to manage concurrency.

7. The specification of a uniform programming interface that allows applications to be programmed independent of knowledge about the underlying hardware, be it a shared-memory or distributed-memory environment.

8. Providing an implementation of the proposed system to demonstrate the viability of the features required to support parallel and distributed programming.

9. The programming of several examples to demonstrate the usability of the system for writing parallel and distributed applications in both shared-memory and distributed-memory environments.

1.4 Thesis Overview

In this section, a broad overview of the contents of the thesis are first presented. The overview is followed by a more detailed description of contents of the various chapters. The remainder of the thesis is organized in the following way:

- Chapter 2 provides an overview of the work done in parallel and distributed systems relevant to the work of this thesis. The strengths and weaknesses of the various systems are examined to arrive at the features important to supporting parallel and distributed processing.

- Chapter 3 introduces the Raven language, which serves as platform for the ideas and features introduced by this thesis.

- Chapter 4 introduces the design features introduced into Raven to support parallel and distributed computing.

- Chapter 5 provides details of the implementation of the features described in Chapter 4.
• Chapter 6 demonstrates Raven's support for parallel and distributed computation, with several examples.

• Chapter 7 is a summary of the thesis with emphasis on the thesis results and future areas of research.

In general terms, the first three chapters of this thesis are introductory in nature. Chapter 1 introduces the problem, Chapter 2 introduces the important terminology with respect to parallel and distributed computing, and Chapter 3 introduces Raven, the language and system that the work this thesis describes is implemented in. Chapter 2 also provides an overview of related work.

Chapters 4, 5, and 6 form the main body of the thesis, organized into three major parts. The first part, Chapter 4, describes the features to support parallel and distributed programming. Chapter 5, the second part, describes the implementation issues and problems associated with the implementation. The third part, Chapter 6, illustrates the use of Raven's support for parallel and distributed programming with some programming examples. A more detailed description of the contents of the remainder of the thesis follows.

Chapter 2 presents the background and related work relevant to the understanding of the thesis. It begins with a discussion of the distinction between distributed and multiprocessor environments (Section 2.1), which includes a discussion of the granularity of parallelism. From there, a review of the various programming models used in developing parallel programs is undertaken (Section 2.2). An overview of the support for distributed and parallel programming provided by other systems then follows (Section 2.3). The systems examined are classified according to how they provide support for parallel and distributed computing. Some systems extend sequential systems, some are based on distributed systems, some use language extensions, and some are object based. By examining these systems, a list of important features required to support parallel and distributed computations is created. This list forms the basis for determining the feature set needed by Raven to support parallel and distributed programming.
Chapter 3 introduces the Raven language and system. The chapter starts by introducing the passive and active object models (Section 3.1) and how the Raven programming model differs from these models. A detailed overview of the Raven language follows (Section 3.2). The overview consists of defining the various terms used to discuss programming in Raven and a program example to illustrate the features of Raven important to understanding this work. To do this, the basic syntax of the Raven language is presented, along with an overview of the various components that comprise the Raven system. These components consist of the compiler, class library, and the underlying virtual Raven machine. Additionally, the concept of a local Raven environment and the support for distribution are also introduced.

Chapter 4 introduces the key problems and the application level solution to supporting parallel and distributed computation. To provide an environment that handles the problems associated with parallel and distributed processing described earlier in Chapter 1, a number of issues had to be addressed. The initial decision that needed to be made concerned the programming language or environment to be used. The choice was to extend the object-oriented language and system, Raven, to support the programming of parallel and distributed applications in shared-memory and distributed-memory environments. To do this, several key problems had to be solved. The first problems concentrated on creating parallelism and determining what the unit of parallelism should be (Section 4.1). The strong data encapsulation presented by objects, combined with a well defined and restricted procedure interface (methods) to manipulate the data strongly suggested that the method interaction point should be the location where parallelism is created, and that the unit of parallelism should be the method. This observation leads to the introduction of class-based (Section 4.2) and user-based parallelism (Section 4.3). Class-based parallelism is parallelism that results from actions on the callee side of the invocation. The parallelism is expressed through the new language constructs of early and delayed results. User-based parallelism is parallelism initiated on the caller side and takes the form of certificates and companions. Certificates and companions are used to create, monitor, and manage parallelism on the caller side. Part of the ability to manage companions takes the form of invo-
cation streams, which are a new way of imposing third-party sequencing on requests made of the same server.

With the facilities for creating parallelism firmly established, Chapter 4 proceeds to discuss properties (Section 4.4). Properties are a new way of providing operating system services to individual objects. Properties stop the class explosion problem that results when the same class needs to make use of specific system services, such as concurrency control, yet was not initially programmed to use those services. Properties also eliminate the need to provide a mechanism that allows the programmer to make direct calls to the operating system for specific services.

One of the main motivating factors for the concept of properties was the need to supply concurrency control (Section 4.5) for objects that were not originally programmed to be used that way. The concurrency control provided in Raven is accomplished at the method level, and it permits multiple reader methods and a single writer method. Combining the object-oriented programming style encouraged by Raven with facilities to encourage use of parallelism creates challenging concurrency control problems. In particular, the problems associated with objects calling methods on themselves, locking conflicts that result from using class-based parallelism, locks that need to cross machine boundaries, and the handling of deadlocks are explained and solutions described.

With the ability to easily create parallelism come problems associated with managing system resources (Section 4.6), in particular main memory and the number of active processes. Garbage collection is used to help manage memory and restrictions on the creation of processes are used to control the number of active parallel execution threads. The number of active parallel threads a system can have is considered a system configuration issue. By using this approach the programmer can write an application without regard to how many processors the target system actually has.
One of the goals of the work in this thesis is to provide a uniform application programming interface for programs that run in a shared-memory or distributed-memory environment. This is accomplished by using the object and method invocations as the unit of work. With system support for making an object location transparent (Section 4.7), the application cannot distinguish remote objects from local objects. This transparency is crucial to making it possible to write applications that can run in shared-memory and distributed-memory environments without modification.

Raven provides support to configure the running system through the System object (Section 4.8). Chapter 4 concludes with an overview of the facilities Raven provides for processing and handling exceptions and failures (Section 4.9).

Whereas Chapter 4 introduces the various features of Raven to support parallel and distributed computation, Chapter 5 describes the underlying implementation of the Raven system and details the various problems that were encountered and solved while implementing the support for parallel and distributed processing. The first topic covered is how objects are organized in Raven (Section 5.1). Particular attention is given to how methods are invoked, how properties are associated with objects, and how remote objects are represented locally. When Raven was first run on a shared-memory multiprocessor, a performance bottleneck in the dispatching of methods was encountered. The diagnosis of this problem and the resulting solution are also described in this section of Chapter 5.

The memory allocation and garbage collection system used on a shared-memory multiprocessor also introduced some unexpected performance problems (Section 5.2). The extensive modifications made to improve the operation of the memory management system, along with proposals on how to improve the overall performance of the memory management system are also discussed.

The next section of Chapter 5 is devoted to a discussion of the role of threads (Section 5.3) in the Raven environment. All parallelism within Raven is accomplished through
the creation and management of new threads of control. Threads are system level entities and they are not used by the application. Applications achieve parallelism by using companions, early result, and delayed result. How these language level features are translated into real parallelism is described in this section.

To support a uniform programming environment for shared-memory and distributed-memory machines, special system support is needed to make remote objects appear like local objects (Section 5.4) and this topic is the last topic of Chapter 5. In providing this support, the problems associated with copying objects, operating in a heterogenous processor environment, locating objects, managing locks across machines, sequencing invocation requests, and handling failures are discussed and addressed.

Chapter 6 demonstrates the usability of the Raven system and its support for parallelism by describing the implementation of several applications and the performance results. The chapter starts with a discussion of the performance metrics (Section 6.1) to be used, and continues with some specific examples (Section 6.2). Three examples of Raven’s performance on a shared-memory multiprocessor are presented. These examples, the Mandelbrot set, a bayesian search, and a prime number generator are each presented and the performance results described. A distributed implementation of the Mandelbrot set computation is also discussed. Of particular note is the fact that the shared-memory and distributed-memory implementations are identical except for a few lines of code involved with system configuration issues. The distributed Mandelbrot example was run in heterogenous environment. To illustrate that Raven can be used to develop larger distributed applications, a distributed mail application is described. Taken together, these applications illustrate Raven’s support for parallelism through companions, early result, delayed result, and the ability to provide a uniform operating environment for shared-memory and distributed-memory environment.

Chapter 7 presents a summary of the results (Section 7.1). The summary includes a review of the way parallelism is supplied in the Raven language and of the system support sup-
plied for parallelism. The thesis closes with a description of the research contributions of this thesis and the identification of some future areas for research.

In summary then, Chapter 2 presents an overview of work done in parallel and distributed systems relevant to this thesis and sets the basis for the work. Chapter 3 introduces the Raven language and system. Chapter 4 describes the new features for supporting parallelism provided by Raven. Chapter 5 describes the implementation, with Chapter 6 describing the sample problems programmed in Raven. Chapter 7 presents a review of the thesis work. The relevance of this work, its contributions to the body of computer science research, and future research areas are highlighted.
The purpose of this chapter is to provide a foundation for the discussion of the work presented in the remainder of this thesis. To accomplish this, the chapter first discusses the differences between distributed and multiprocessor programming environments. Since the granularity, or unit of parallelism will affect how a parallel application is written, the next section is devoted to a discussion of granularity. The following section provides a design approach for creating a parallel program. It starts with a description of the steps involved in creating a parallel application and concludes with an enumeration of some of the common approaches to organizing a problem to achieve parallelism. To this point, the chapter has provided the background material used when discussing parallelism, and insight into ways to design parallel programs. Building on this material, the next section reviews and classifies some systems that support parallel and distributed programming. The chapter closes with a summary of the material that was presented.
2.1 Distributed and Multiprocessor Environments

When dealing with parallel and distributed computation, there are two main types of programming environments. One type is a distributed environment and the other is a multiprocessor environment. Each of these environments presents a particular system view to the programmer, with the result that the similar problems are solved using different approaches, and the environments themselves affect the sort of problems being solved. Bal et al [17] define a distributed system as follows:

A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a network.

A further elaboration of this definition explains that each system executes its own instruction stream and has its own local data. By its very nature, such a configuration is subject to partial failures where one part of the system fails and the others continue to run. These types of systems also tend to be loosely coupled, and there is the possibility that such a system is heterogeneous and consists of multiple processor types.

This is contrasted to a multiprocessor, where processors communicate through shared-memory, like the Sequent [89], or through fast processor to processor connections, such as might be found in a Transputer farm [54][42] or Intel Hypercube machine [58]. These systems differ from a distributed environment in that they tend to consist of tightly coupled homogeneous processors. The processors and interconnections are not as autonomous as a distributed system and the system lacks independent failure modes. That is, if one processor node or memory component fails, the whole system fails, just like a uniprocessor machine.

The types of programming models used in these two environments is often quite different. Distributed-system software tends to use remote procedure calls to communicate between the different processors, with the result that the applications remain sequential in nature. Remote applications are often involved with the managing or sharing of resources amongst a collection of machines; some examples of this are the sharing of printers, providing file access
services, or maintaining a database. A distributed application tends to be much more concerned with the handling of failures than multiprocessor applications. When a component of a distributed system fails, the application needs to have some level of fault tolerance. Using printing as an example, if the print server fails client processors should not crash. It should, instead probably wait for the print server to become available again. Since distributed environments can be heterogenous, a distributed application must also address the problems of exchanging data and requests between machines with different internal data representations. Increases in performance of a distributed system are often more a by-product of the organization of the application than by design.

In contrast, applications developed for a multiprocessor environment, and in particular for a shared-memory environment, tend to have as their primary goal increasing the performance of the application. In a distributed application an increase in performance may result for an application, but that is not the sole goal. Especially in a shared-memory multiprocessor, the applications, typically, do not concern themselves with issues associated with heterogenous environments or fault tolerance.

Obviously, the physical constraints of an environment play an important role in the system and affect how applications are developed in distributed and multiprocessor environments. However, these systems do have to address many of the same sorts of problems. For example, how are other entities in the computation located and communicated with, how are shared resources protected from improper concurrent access and how are the various components of the application defined, created and assigned to processors.

2.1.1 Granularity of Parallelism

Another consideration when discussing parallelism is the unit of parallelism. This is referred to as granularity. Bal et al [17] provide a definition for the granularity of parallelism. They define a parallel application’s granularity by the amount of computation done between communications. With such a definition, the natural inclination is to consider communications to mean
the exchange of messages between processes running in a distributed memory environment. This is not the case, since a communication can take place in a variety of ways. In particular the communication can consist of the exchange of information through shared memory, or the synchronization of processes.

The terms fine-grained and coarse-grained are used to describe the length of time between communications. In a fine-grained parallel application the time between communications is small; in a coarse-grained parallel application the time between communications is large. For a fine grained application a communication might occur after every instruction, whereas in a coarse-grained applications the communications may be separated by hours. The hardware an application is using will, obviously, have some effect on the granularity of parallelism. For example, a distributed environment is not likely to support an extremely fine-grained model of parallelism, since the cost of synchronizing across the network would be high compared to the amount of computation done between communications.

### 2.2 Parallel programming models

There are several ways that parallelism can be used to solve a problem. The technique that Carriero and Gelertner [29] advocate for writing parallel programs consists of the following steps:

- First choose an approach to parallelism that most naturally maps to the problem.
- Write the program based on this choice.
- Run the program, and, if it is not efficient enough, transform the solution into a more efficient solution by changing the approach to parallelism.

This is the approach to parallel programming advocated in Raven. It is important to have a good understanding of a problem and its solution, and one of the best ways to achieve this is to implement the solution in its most natural form. By analyzing the results and the way the program behaves, these additional insights into the problem can be used, if needed, to modify the solution to be more efficient.
Carriero and Gelertner also present three common ways to organize applications to achieve parallelism. These approaches are classified as follows:

1. Result parallelism which focuses on the finished result.

2. Agenda parallelism which focuses on the list of activities that must be performed.

3. Specialist parallelism which focuses on specialized processing that needs to be performed.

This classification is based on the techniques used to extract parallelism from a solution. Although these forms of parallelism are identified, an application seldom uses a single form of parallelism. An application will tend to have one dominant form of parallelism, with the other forms of parallelism being present in varying degrees.

With result parallelism, the focus of the approach is on the format that the final result is to take. Parallelism is achieved by computing all the component parts of the result simultaneously. Result parallelism is often used when the result consists of a series of values that are predictably organized and/or have well-defined dependencies. When agenda parallelism is used, the application concentrates on producing a list of activities that need to be performed for a solution to be reached. Parallelism is then achieved by using multiple worker processes. These workers may cooperate to complete one agenda item or, if possible, several agenda items can be worked on simultaneously. Finally, with the specialist approach, a group of specialized workers are connected together to form a logical network. Parallelism results when multiple specialists are active at once. Typically, each specialist is responsible for some sort of computation and then moves the results of its computation, via the network, to the next specialist.

The three formats for organizing parallelism described in the previous section provide a broad framework and general approach that can be used when trying to write a parallel program. They do not, however, provide any guidelines on how to recognize the parts of an application that can be parallelized and the techniques that can be used to extract parallelism. Lester
[65] provides a summary of some of the more common approaches to extracting parallelism and how to recognize it. A brief overview of some of these techniques follows.

2.2.1 Data Parallelism

It is often the situation that the form of the input data and output data has a strong influence on the approach to achieving parallelism. The term data parallelism is applied to approaches where there are a large number of data objects subject to similar processing. To accomplish this, the computations on the data items need to be independent. The manipulation of large vectors or matrices, typical of many numerical algorithms, can often make good use of data parallelism.

2.2.2 Data Partitioning Parallelism

Closely associated to data parallelism is parallelism that results from data partitioning. With data partitioning, the data space is divided into regions, and each region is assigned a processor so that the computation can proceed in parallel. What constitutes a region usually has some natural definition within the problem being considered. This region might be a block of a matrix in a numerical algorithm, or it might correspond to a region of space in a simulation of a gas. This approach differs from the data parallel approach in that there is the occasional need to exchange data across region boundaries. Except for these occasional data exchanges, each processor concentrates on performing the computation within its region.

2.2.3 Synchronous Iteration Parallelism

Another technique for parallelism, closely associated to the ones already discussed, is synchronous iteration. In both data partitioning and data parallelism, each processor proceeds on its own with minimal interaction with the other processors. In synchronous iteration, the same computation is performed on each data item, this is similar to the approach used by data parallelism, except that after each step in the computation all the processors synchronize and exchange data. It is the use of the results of one step of the computation by other processors
performing subsequent steps of the computation that mandate the synchronization, and differentiate synchronous iteration from both data parallelism and data partitioning.

2.2.4 Replicated Worker Parallelism

With replicated worker, or processor farm, techniques for parallelism, there exist a number of worker processes that are constantly requesting work to perform from a pool of possible tasks. The algorithm terminates when all the tasks in the pool have been completed. The approach proceeds by having each processor ask for a task from the pool, perform the task, and possibly registering the results before asking for the next task to perform. During the execution of a task, a processor may generate new tasks that need to be computed and these are added to the pool. Tree and graph searches can often make use of this approach to parallelism, since this approach is well suited to problems where the amount and nature of computation is dynamic and cannot be accurately determined in advance.

2.2.5 Pipelined Parallelism

The final technique for parallelism that will be introduced is pipelined parallelism. When using the pipeline approach to parallelism, the processors to be involved in the computation are arranged into a regular interconnection pattern, of which a mesh, ring, and pipeline are some common types. The computation is broken into a number of different component tasks, and each one is assigned to a processor. Just as in a pipeline, the data flows through the network of processors with each node performing some computation on the data. Parallelism results when multiple stages of the pipeline are executing simultaneously. Execution on a problem proceeds by having the first stage acquire the information needed to start processing. It processes until its allotment of work is finished, and then it passes it to the next stage. While the second stage of the pipeline works on the problem, the first gets more work to perform. This approach to parallelism is like an assembly line, and has similar concerns. For example, to maximize parallelism each stage of the pipeline should use the same amount of time as the other stages. If this is
not true then some stages will have no work to perform while other stages will have work queuing up, and the parallelism will not have been maximized.

Although the approaches and techniques for dealing with parallelism have been presented as being disjoint, in many situations more than one technique will be used. A solution might combine the approaches of data parallelism with replicated workers, or perhaps replicated workers will be the front end for a computation also using pipelining.

2.3 Supporting Parallel and Distributed Computing

The previous sections provided a broad overview of what constitutes parallel and distributed computing and some of the design techniques and approaches that can be used when developing a parallel or distributed application. This section presents an overview of various approaches to addressing the problem of actually realizing a design in a programming system. The various approaches have been divided into the following five groups:

1. Extending existing sequential systems.
2. System systems for distributed processing.
3. Changes or adaptations of existing languages to support parallelism.
4. Systems for parallel and distributed processing based on Smalltalk.
5. Support for parallelism in object-oriented systems that are not Smalltalk-based.

Each one of these systems, or approaches, will be briefly described with particular attention being paid to the issues of the granularity of parallelism, how parallelism is created, and the level of support for concurrency control. Taken together, these systems will provide an overview of some of the techniques for supporting parallel and distributed processing and how these techniques have evolved.
2.3.1 Extending Sequential Systems

In this section techniques for achieving parallelism that exploit existing sequential systems are provided. By taking a sequential system and extending it, experience on providing services for parallelism is gained, and this serves to drive the development of new approaches to managing and supporting parallelism. The first three examples describe changes that have been made to the procedure calling mechanism to support parallelism. The final example describes the support for parallelism available on multiprocessors.

2.3.1.1 Parallel RPCs

The procedure call mechanism is a well understood programming tool. In distributed systems, the remote procedure call (RPC) has been developed to manage inter-process communication. Work by Martin [71][72][73] has concentrated on extending a single remote procedure call to run on multiple machines.

Two extension to the remote procedure calling interface were made. These are:

1. A mechanism to identify \( N \) remote hosts to execute the remote procedure call was added.
2. An optional results statement, which is given control each time one of the \( N \) procedure calls terminates, was introduced.

The general form of usage is to have the remote procedure call specify both the processors to make the call to, and the result statement to be executed. As each call completes, the results statement is given control. This mode of processing continues until either all the calls have completed or the decision to break out of the wait is made. When a results statement is broken out of, the results from any calls which have not completed are lost. The only way results can be returned is through the parameters of the remote procedure call. Therefore, these calls cannot be used as functions. The completion of an RPC and the execution of its results statement are atomic.

The advantage of using this approach is that the procedure call, and by implication the remote procedure call, is a language construct that is well understood by the programmer. Con-
ceptually, there is only one thread of control, thus making it easier to understand what is happening. Parallelism in this system is coarse-grained and only accomplished when the same remote procedure call can be targeted at multiple sites. Synchronization support is supplied to manage the potential simultaneous return of results, however, concurrency control on the RPC server side remains the responsibility of the programmer.

### 2.3.1.2 Streams and Pipes

Gifford [46][47][48] also looked at remote procedure calls and suggested changes to improve their speed while at the same time increasing the parallelism between the caller and callee. Gifford recognized that one of the bottlenecks to parallel processing using the remote procedure call model occurs when the caller blocks waiting for the procedure to complete. To overcome this problem, the pipe is introduced. A pipe is essentially a remote procedure call, except it does not block the caller nor does it return any results. Because the caller does not block, parallelism between the client and server results when a pipe is used. The granularity of parallelism is coarse-grained and at the procedure call level. Concurrency control is the responsibility of the programmer and it is assumed that something like monitors [53] is used.

In this system, remote procedure call is kept and optimized for low latency. Since no ordering is implied between multiple requests not directed at the same pipe, the group and sequence constructs were introduced to allow a mechanism to specify the ordering of requests on a channel. (A channel is the connection between the source of the request and the destination. Both pipes and RPCs use channels.) By splitting the familiar remote procedure call model into two different classes it is possible to develop and optimize protocols for the function they are to perform. RPCs can be optimized for quick sending and receiving of results, and pipes can be optimized for bulk data transfer.

### 2.3.1.3 Promises and Claims

Argus [66][67][68] incorporated both the futures notion from Multilisp (Section 2.3.3.1) and Gifford’s work with pipes into promises and claims [69]. Like a future, a promise allows the results of a call to be claimed at a later time. To use a promise, the notion of a call stream is
introduced. A stream is like a pipe in that the sender does not wait for replies. Unlike Gifford’s work, however, a call on a stream may return a result. To capture this result each call returns a promise that can be used to claim a result in the future. A promise can be viewed as an object like data structure reserving space for the results of the method invocation, which will be arriving at a later time. To get the results of the promise, a claim call is made on the appropriate promise and the process blocks until the promise is fulfilled. Other functions to check on the completion status of a promise, and functions to iterate and wait on collections of promises, are also provided. The promise and the claim provide a way for RPCs to proceed in parallel while at the same time allowing results to be returned. This overcomes the restriction of the previously described parallel RPCs being unable to execute different types of procedure calls, and the inability of pipes to return results. Parallelism, therefore, results from the ability to overlap client and server processing. The parallelism is at the procedure call level and is coarse-grained. Being written in Argus, the protection required in the presence of concurrent accesses to objects is automatically provided through Argus’s atomic transaction facilities.

2.3.1.4 Multiprocessors

The previous examples have concentrated on some of the methods that have been used on distributed systems to provide parallelism. Similar incremental techniques have been in use to extract performance improvements in multiprocessor systems. This processing environment can be described as one in which more processors are added to an existing general purpose system while hiding the multiple processors from the user and continuing to present a uniprocessor environment. Parallelism is accomplished by having multiple coarse-grained processes run simultaneously on the individual processors. Typical of the machines in this class are the Sequent machines running a variant of the UNIX operating system [96][89] and Helios [42] running on transputers.

These machines typically consist of some number of processors on a common high speed bus, along with some amount of shared memory. When a process is ready to run, it is assigned to one of the processors. Parallelism is achieved by having multiple processors active at the
same time. This leads to two major techniques for using parallelism on these machines. One is system wide parallelism that can easily be achieved by assigning the various runnable processes in a time shared environment to different processors. Although this may increase the throughput of the system as a whole, the elapsed time for any one process will not be less than if the machine were idle and consisted of a single processor.

A second type of parallelism, that a user is more likely to encounter, results from the UNIX structures of pipes and background process. In this case the background processes or constituents of a pipe are assigned to separate processors, if possible, in an attempt to take advantage of the parallelism implied by the user. Under these situations it is quite likely that a user will experience a speedup in his job mix attributable to this parallelism.

However, it can be argued that the above really is not what is meant by programming a parallel system. In the multiprogrammed environment of UNIX and other operating systems, the pseudo parallelism, provided by the operating system in an attempt to make better use of a single processor, is simply extended to use more than one processor. The creation of multiple processes, in the background or in a pipeline, does not necessarily imply parallelism. Typically, the process model of programming is being applied and processes are being created to handle logically separate components of a task. These separate processes are part of a design methodology that makes it easier to construct a system by interconnecting smaller, easier to debug programs, and not a technique intent on improving system performance through parallelism. The pipeline form of parallelism represented by this approach is stylized and restricted in functionality.

To allow the user to take direct advantage of the shared memory and multiple processors, the operating system typically supports some form of threads package [89][70]. These packages provide ways for the program to create new threads of control and services to allow processes to synchronize within a shared address space.
A logical progression from the shared-memory approach is to make the processors self contained with their own memory and then provide some form of interconnection. An example of this is the Intel Hypercubes type machines [16][57][58]. Each processing node is independent and requires programmers to explicitly write and assign code to the individual processors. Parallelism is then achieved by having the multiple processors work concurrently. Simple communications primitives like send and receive are used to exchange data between processors. To achieve parallelism great care must be taken to ensure that once a processor sends a message it is possible for both the sender and receiver to execute simultaneously.

In all of these systems, the granularity of parallelism would be described as relatively coarse-grained, since it tends to be at the process level. The creation of parallelism is done explicitly by the programmer and data protection is, also, the explicit responsibility of the programmer.

2.3.2 Distributed Systems

Programming distributed memory systems is characterized by two notable problems. One problem is deciding where to place processes; the other problem is establishing a mechanism to simplify the problem of identifying and communicating with the processes of the application. The systems examined in this section explore the issues associate with providing solutions to these problems.

2.3.2.1 SR

In a distributed system, RPCs are often too inflexible for efficient distributed computation, and pure message passing can be difficult to program. SR (Synchronizing Resources) [8] [9] [10] investigates these problems through processes, resources, and a full complement of message passing primitives. In SR, a resource possesses some of the abstract qualities of an object by providing encapsulation of data and specifying the operations that must be used to interact with this data. A resource is comprised of one or more processes, which represent the operations that a resource is willing to perform. All the processes associated with a resource must execute on
the same processor, and this leads to the property that all the process within a resource, and only those processes, may share data. This organization captures aspects of both the distributed programming environment and the shared-memory multiprocessor environment. A single resource could be viewed as a multiprocessor, and multiple resources can be viewed as a distributed environment.

In SR, communication is accomplished on the sender side using the call and send statements. These correspond to synchronous and asynchronous message sends, respectively. Each call, or send, action specifies the name of the request and its parameters. On the receiving side, the requests are accepted with the in statement, which specifies the name of the request to receive and the parameters to be retrieved. The in statement also has an optional guard clause that can be used to control or prioritize the reception of requests. The results of a request can be returned either through a result statement or reply. When a reply is done, the result is returned to the sender and execution continues with the statement after the reply.

SR also has a co/oc delimited block to specify invocations to be done in parallel. Each invocation can also have a post processing block associated with it. This allows the programmer to directly specify which activities are to be executed concurrently. Concurrency control on any shared resource is the responsibility of the programmer, and SR provides semaphores for this purpose. The granularity of this system is coarse with parallelism being at the procedure call level. Although the user may create new processes, a substantially component of an SR application is static, with the processes being defined at start time.

2.3.2.2 Poker

Researchers at the University of Washington recognized that another difficulty with distributed memory systems was assigning processes to processors. To explore these problems, they developed the Poker system [81][94][95]. The Poker system consists of both hardware and software, with the hardware being a collection of processors with fixed interconnects to adjacent processors. One of the basic underlying assumptions of the research project was that the parallelism in a program is best handled explicitly by the programmer.
The designers of Poker identified five functions common to the specification of parallel programs for MIMD machines. These are:

1. The specification of a communication graph to indicate how the various processing elements are interconnected.

2. The specification of the program(s)/code that will ultimately be loaded to a processing element.

3. The specification of the assignment of object code to processing elements.

4. The assignment of names to the edges (wires) connecting processing elements. This specification allows a mapping between the named edges used in the program and the physical wires to be carried out.

5. A specification of where input is to be taken from and where output is to be placed.

Although these points are valid, they highlight the problems that prevent parallelism from moving into common usage. Programmers are not particularly interested in taking on the task of specifying the communications graph, decomposing the problem into parts for assignment to processors, any more than they are in specifying where in main memory their program should reside. The addition of these extra dimensions to the programming task detracts from the usefulness of these parallel environments for programming general purpose tasks.

In addition to these types of problems, there were also restrictions on the how the processors could be organized. For example, the processing elements had to be placed on a lattice, thus restricting the interconnection patterns. Only one sequential process can be assigned to each processing element: that is, it is not possible to have multiple process on the same processor. The Poker system is hardware-driven, forcing the user to be intimately involved in the specification of a program's parallelism. Parallelism in this system results by having multiple nodes active at one. Since there can be only one process active on a node, and since the processes must explicitly accept incoming messages, concurrency control is implied. The granularity of this system will vary depending upon the speed of the communication primitives. The system would not be defined as fine-grained, however, the time between communications is
much smaller than the systems examined to this point and is probably best classified as medium grained.

### 2.3.2.3 MUPPET

In many respects MUPPET [77], the multi-processor programming environment of the West German SUPERNUM computer project, is a lot like Poker; but, instead of facilitating the development of programs for a particular hardware configuration of a machine, MUPPET attempts to remove this restriction through the introduction of abstract machines that hide the underlying processor interconnection patterns. With MUPPET, when a program is designed, the programmer is permitted to specify a collection of abstract machines and the interconnection pattern to be used for processing. These abstract processors, called LAMs (local memory abstract machines), represent virtual disjoint processors that do not share memory. Each LAM provides facilities for message passing, process creation and process destruction. Once a program has been specified for a given collection of LAMs the MUPPET software will attempt to configure that representation on the available set of processors. Depending upon the relationships between the abstract machine specified, and the actual machine, this mapping may be a non-trivial problem.

The major difference between Poker and MUPPET is that in Poker one specifies the actual machine configuration to be worked with, and designs an algorithm for that configuration. In MUPPET, one designs an algorithm for an abstract machine and maps the abstract machine to the physical processors. The effective parallelism will be a function of how many LAMs can be assigned to different processors. Concurrency control remains the responsibility of the programmer.

### 2.3.2.4 PVM

PVM [43][44] stands for Parallel Virtual Machine, and has as its goal the desire to make a collection of machines on a network appear as a single concurrent computation resource. PVM does many of the same things as Poker, except it does not assume any special underlying hardware. PVM does assume:
• A collection of heterogenous machines accessible through a network.
• Programming using an imperative language.
• Access to system services through procedure calls.
• Operating system support for limited inter-process communication within a machine.
• Support for the unreliable delivery of data between machines.

PVM can basically be described as a software system that tries to make a collection of machines appear like one. Support for this is provided through a library package that supports the creation of processes and the sending of messages within the PVM environment. This is partly accomplished through a daemon process run on each node in the PVM system. These daemons essentially cooperate to keep each other informed as to the state of their part of the PVM. Each PVM daemon can be viewed as a mini operating system that maps the PVM calls, for things like process creation and message passing, from the PVM form to the local machine. Parallelism in PVM is coarse-grained at the process level, and explicitly created externally by the PVM user or internally within a program. Being a message based system, concurrency protection is implicit since a process encapsulates data within its address space. Shared resources outside of a process's address space must be managed as in any other system.

2.3.3 Language Extensions or Adaptations

The previous sections have all examined providing support for parallel and distributed computation primarily from the system level. That is, the support for parallelism encompasses something about how the complete system is built. In this section some changes to programming languages to support parallelism will be looked at. The first two examples Multilisp and Actors look at language-level support for parallelism in a fine-grained environment. The third example, Linda, presents language-level parallelism from a coarser-grained perspective.
2.3.3.1 Multilisp

Multilisp [50][51][52] is a variant of Lisp that explores the possibility of evaluating multiple expressions in parallel. The initial parallel construct in Multilisp, \texttt{pcall()}, is used in the following manner:

\[(\texttt{pcall } F \ A \ B \ C \ldots)\]

The expressions \(F, A, B, C\) and \(\ldots\) are evaluated in parallel with control passing to the next expression when all the expressions have been evaluated. Another way to exploit expression parallelism is to separate the computation of a value from the point at which it is used. This type of concurrency is captured by futures. An example use of future is:

\[(\texttt{cons (future A) (future B)})\]

In this situation the computation of \(A\) and \(B\) is started in parallel with the main flow of control, but the results are not needed immediately. At some later time, when the actual values are required, either they will be ready or they will still be being computed. If they have not finished, then the computation will stop and wait for the results to be returned; otherwise, the computation can continue immediately. Concurrency is explicit through the \texttt{pcall} and \texttt{future} functions and is relatively fine-grained. Concurrency control is not an issue in a language like this since only values are manipulated.

2.3.3.2 Actors

The Actor [2][3][4][40] languages also have a Lisp like flavour and are designed to support large scale concurrent symbolic computation. These systems are characterized as being highly dynamic, with constantly changing structure and computing requirements. One of the assumptions for parallelism in the later Actor languages is that all the expressions forming part of a function can be evaluated in parallel. Everything in an Actor system --this includes functions, data and messages-- is an actor. Expressions and functions are evaluated by sending messages to the message queue associated with the actor specification for that function. These messages contain information indicating exactly what function is to be performed. To actually do some-
thing, each actor has a script which tells it what to do when a message is received. An actor may work on only one message at a time, and as soon as an actor is finished its computation it disappears.

As part of its execution an actor computes a replacement for itself. In many cases the replacement actor is a copy of the existing actor, but there is nothing to prevent the replacement from being a totally different actor with a totally different behavior. The message queue associated with an actor is inherited by its replacement. Some control over parallelism can be exercised by specifying when the replacement actor is created. If the replacement is specified immediately, then a message from the inherited queue can be retrieved for processing and computation between the original actor and its replacement can proceed in parallel. If instead the specification of the replacement is delayed until the computation is nearly completed, than there is little opportunity for parallelism.

Actor languages have the advantage that parallelism is automatic; however, no ordering can be imposed on the sending of messages, thus increasing problems resulting from nondeterminism. Also under certain circumstances some parts of an expression may continue to be evaluated even after the value of the expression is known, thus wasting computing resources. (Example: Two expressions are or’ed. One of the expressions evaluates to true while the other is still computing. At this point the value of the expression, --true--, is known and there is no need to continue to evaluate the other expression.) The actor approach is targeted towards functional languages which will run on large fine grain machines. Unlike Multilisp, the parallelism is implicit in the Actor languages since every expression is viewed as running in parallel, and like Multilisp concurrency control is not an issue since computation are always performed on values.

2.3.3.3 Linda

The Linda programming language [64][5][28][29][45] has adopted the approach of adding new functions to an existing programming language, along with underlying software support, to provide parallelism. The goal of Linda is to achieve parallelism by allowing the user to con-
centrate on programming and not on the hardware. When programming, the user is expected to explicitly decide which parts of a program can be run in parallel. The Linda language then provides the mechanism to implement this parallelism.

In Linda the programmer is presented with a tuple space to which processes perform operations in an effort to achieve parallelism. The tuple space is common to all processors in the system. Linda provides operations to add, remove or read tuples from the tuple space. One justification for this computational model is that processes in a parallel environment should not concern themselves with how other processes are using data. They should essentially take some data, compute with it, and then, maybe, generate some new data for another process to work with. The parallelism is then achieved by having multiple processes perform requests of the tuple space. The programmer must either create these new processes within the application, or start them and then associate them with a running tuple space. Linda can also be viewed as a collection of global mailboxes that are emptied and filled with data. The fields comprising a tuple serve as the mailbox or tuple identifier.

Parallelism in Linda is coarse-grained and at the process level. Parallelism is achieved by using the tuple space as a buffered send area. The tuple space, which is a the only shared data structure in Linda is maintained in a consistent state by the system by ensuring that the operations to add, read, and remove tuples from the tuple space are atomic.

2.3.4 Smalltalk-Based Systems

Object-oriented systems, with their strong data encapsulation and well-defined method interface, seem like they should provide good environments to support parallelism. The classical object-oriented programming language, Smalltalk, has also been the target of modifications to support parallelism. This section will look at some of the ways that Smalltalk and Smalltalk derived languages have been used to support parallelism. The first system examined is an implementation of Smalltalk for a distributed environment; the second system examined is an
implementation of Smalltalk for a fine-grained environment; and the third system examined is an implementation of Smalltalk for a multiprocessor.

### 2.3.4.1 Distributed Smalltalk

Distributed Smalltalk [19][20] is a system that allows objects on different machines to exchange messages. It also permits objects to be shared amongst different users. The functionality of Distributed Smalltalk can be characterized as being similar to the situation of users sharing files or server processes in a more general purpose environment. It lacks the tight coupledness and cohesiveness of an environment designed for parallel programming. The supported parallelism is extremely limited, is system wide, and is available only at the user process level. Individual users cannot have multiple threads of control active at once and the programmer is responsible for controlling concurrent access to objects.

### 2.3.4.2 CST

CST, which stands for concurrent Smalltalk [36][37][104], is a proposed dialect of Smalltalk developed to run on the J-machine. The J-machine is a fine-grained concurrent computer with a large number of special purpose processing nodes. The processing nodes each have 4K words (36 bit words) of memory, each forming part of a global virtual address space, and a special purpose communications controller to perform message routing. Concurrency is achieved in this system through distributed concurrent objects, asynchronous message sends and multiple concurrent methods on an object.

This system is heavily reliant upon the global virtual address space to support a virtual object space. Invocation on objects can be supported by moving the objects to the requesting location or by sending the request to the object. The virtual object space allows for easy location or movement of objects. Depending upon the nature of the request, the method will be transferred either to the object, or the object to the request.

One of the ways to achieve parallelism in CST is to use a distributed concurrent object. A distributed concurrent object is an object which has a number of constituent parts, which are
distributed over the collection of processing nodes. Some Smalltalk objects that fit this classification are arrays, collections and bags. The creation of a distributed object requires a list of nodes at which constituent objects are to be placed. The distributed objects are constructed, by the system, in such a fashion that when a method is sent to a distributed object it may be delivered to any one of the constituent objects. A distributed object is viewed as being composed of many objects, each of which may be processing a method, thus achieving parallelism. In this situation it is the programmer’s responsibility to control access to any of the shared data that the distributed objects may be trying to modify.

A second way to achieve parallelism is through asynchronous message sends to objects. This allows several methods to be sent without waiting for replies. Finally, multiple methods may access an object in parallel. Lock primitives are provided to allow the programmer to protect the integrity of data in this situation.

2.3.4.3 Actra

Actra [102][103][18] is an Actor based extension to the Smalltalk programming language. The design goal is to provide a multi-user, multiprocessor, object-oriented program development environment. Unlike the Actor model proposed by Hewitt, these Actors are of a much coarser granularity. Actra is intended to run on a shared memory multiprocessor. It is hoped that this uniform memory view will simplify referencing and communication between objects.

In Smalltalk, processes are not permitted to receive messages, thus limiting the way in which they can be used. To overcome this problem in Actra, a new class of objects, --actors--, is introduced. An actors object is a community of objects. Each new actors object is implemented as a separate task. Multiple actors residing on separate processors may execute in parallel. Communication between actors (tasks) is accomplished by adding the blocking send/receive/reply methods to the actors class. The blocking nature of these new constructs maps easily to the way methods are normally invoked in Smalltalk. This permits the interaction between objects and actors to have the same surface level appearance.
Parallelism is achieved by permitting the receiving actors object to perform a reply and then continue processing. Only one method at a time may be active in an object. With only one object method active at a time, there is the possibility that long lived requests could prevent other methods on the object from executing. As a result a mechanism allowing a method to voluntarily relinquish control is provided. The system can be viewed as collection of Smalltalk environments running in their own environment on their own processor. These environments then communicate to each other through actor objects and are, perhaps, better thought of as distributed environments that occasionally exchange data or make requests of one another.

### 2.3.5 Other Object-Oriented Systems

Object-oriented systems and programming languages offer such attractions as code reuse and a structured programming environment to the programmer. Smalltalk, however, is often viewed as being too inefficient for production programming. In an effort to overcome these problems, new languages and systems that provide many of the benefits of an object-oriented language have been developed. In this section four approaches, SINA, POOL-T, Eiffel and Choices are examined.

#### 2.3.5.1 SINA

SINA [15][105] is an object-oriented language being designed to study issues associated with concurrent and distributed programming. In general terms, SINA would have to be regarded as being typical of large grained object-oriented languages. An object consists of its instance data and methods. Method invocations are realized through the send, receive and reply message passing primitives. Each method in an object is associated with a queue of message invocation requests. Even though an object may have multiple methods, only one method is active at a time, thus ensuring it has exclusive access to the object instance data. Within an object, there are three system calls, hold, accept, and detach which can be used to control method processing. A method can be in either one of two states, held or accepting. If the method is in the held state, then it is not allowed to execute new requests, and must hold them in a queue for subsequent execution. In the accept state, the method is permitted to execute requests from its mes-
sage queue provided another method of this object is not active. Initially, all method interfaces are in the accept state, but as processing proceeds the object may instruct the runtime support system to change the activity state of its method or methods of other objects of the same type. (An example when this is useful can be found in the implementation of a stack object. When there is nothing on the stack one might decide to stop processing pop requests until some more data is put on the stack.) With the facilities described so far, there is little opportunity to achieve parallelism. The detach command is an attempt to overcome this problem. If while executing a method, a detach call is made, the current execution thread detaches itself from the object. To do this it makes copies of all the object and local data and retains sufficient information about the object performing the invoke so that results may be returned. When the detach is performed this releases the block on the method interface and a new method request can be removed from the queue and executed.

The detach construct, which is similar to a fork(), allows limited concurrency within an object. The ability to turn method interfaces off and on allows for control over the order in which methods are invoked. However, there are a number of problems with the approach taken in SINA. For example, since the detached thread only has copies of the object's instance data it cannot make any lasting changes. The detached thread is a short-lived cloned version of the original object, and once it has finished its computation, it ceases to exist. There is no ability for a detached object to synchronize with other detached objects, or even its parent.

The detach statement is a heavyweight construct since it must create and initialize a new process before there is any parallelism. The parallelism in the system is therefore coarse-grained and programmer controlled. By permitting only one method at a time to execute, concurrency control is automatically provided.

2.3.5.2 POOL-T

POOL-T [11][12][13][14] is a language written in part to explore the possibilities for parallelism that may exist in larger-grained object-oriented systems. To accomplish parallelism a method is broken into two parts:
1. The body, which constitutes the processing that must be performed to determine completely the result of the method.

2. A post processing part, which is processing that is logically part of the method but does not affect the result to be returned. Once the body of the method has completed and returned the results to the invoker, the post-processing part of the method is executed.

The existence of the post processing part of a method allows the client to execute in parallel with the server, once the post processing part is entered. This form of parallelism is restrictive and dependent upon how much, if any, of the method’s processing may be done in the post processing part.

The underlying implementation strategy uses message passing and a single message queue is associated with each object. To assure a degree of fairness, messages from the queue are serviced on a first in, first out basis. Only one method from an object can be executed at a time and the post processing part is considered part of the method. Concurrency control is therefore automatically provided by the system, but parallelism depends upon how many large-grained objects are created. Parallelism is at the procedure call level and is relatively coarse-grained.

### 2.3.5.3 Eiffel

Eiffel [75][55] is an object-oriented class-based language supporting multiple and repeated inheritance. It is a strong statically-typed language with support for parameterized classes and dynamic binding. One of the design goals that has had a significant effect on Eiffel is the concern with software correctness and robustness. This has lead to the ability to associate preconditions and postconditions with a feature’s (method’s) execution. The preconditions specify the set of conditions that must be satisfied when a routine is called, and the postconditions specify the conditions that must be true when the method returns. If either a precondition or postcondition is violated, an exception is triggered. Eiffel was initially released as a sequential language, but since then, work has been undertaken to add support for parallelism [76].
Support for parallelism is provided in two ways. One is through the addition of the keyword *separate*, that can be used to indicate whether or not a method can be run on a different processor than the one the request comes from. The second modification is to the way preconditions are handled with respect to separate objects and methods. Preconditions are used, in the parallel case, to specify conditions that must be met before execution can proceed. Failure to meet these conditions suspends the execution until they are met, and does not result in an exception.

What is particularly interesting about this approach is that the parallelism is captured in the declaration of objects and methods and not explicitly through an execution time statement. This means, that by examining a piece of code, it is not possible to determine the potential for parallelism without examining the object's definition.

The granularity of parallelism is at the procedure level and is therefore medium-grained. Concurrency control is automatically provided by letting only one method access an object at a time, and parallelism is created by marking objects and methods as separate.

### 2.3.6 Choices

The approaches to parallelism that have been examined in this section have all concentrated on language-level support for parallelism. Choices [26][27][88], an object-oriented operating system written in C++[80] uses a different object-oriented approach to providing parallelism. Instead of addressing the issues of parallelism from the language perspective, Choices addresses the problem by constructing operating system interfaces that can be tailored to the application. Briefly, Choices is designed for distributed and shared-memory multiprocessors and has generic support for parallelism that can be customized for the application. This customization is termed problem-oriented concurrent programming.

Choices is itself object-oriented and presents an object interface of the system. Five groups of functions (objects) needed to write a parallel application have been identified. These objects supply services for:
1. Process creation. The process interface to create new processes for execution.
2. Naming. The name server interface used to locate and share objects.
3. Messaging. The message passing interface for the sending and receiving of messages.
4. Memory sharing. The memory interface that permits the sharing of memory between processes.
5. Synchronization. The synchronization interface provides a way for processes to synchronize.

Applications use these interfaces to create and manage the parallelism within the application. The advantage of this approach is that applications can be programmed in a particular way without regard for the underlying hardware implementation. The service classes can then be modified or subclassed to take advantage of a particular system, without need to modify the application. For example, a parallel application might be using message passing. On a shared-memory machine it might be possible to change these primitives to use shared memory to speed execution. In this situation the interface remains the same, but the underlying support has changed.

Choices supports medium- to coarse-grained parallelism at the procedure or application level. Parallelism is explicitly created by the user and concurrency control is also the responsibility of the programmer. However, the object-oriented nature of the system services provide substantial capability for some system support by including concurrency support directly within a class.

### 2.4 Support Requirements

The systems described in the previous section outline various approaches, across different environments, to the problem of supplying support for parallel and distributed computations. By surveying these systems it was possible to identify the key system and language features needed to provide comprehensive support for parallel and distributed programs in shared-memory and distributed-memory environments. A list of the identified features, and what each one
contributes to the programming environment, is given below. The classification of the systems examined in the previous section, with respect to the identified features, can be found in Table 2.1. The names used to identify the features in the columns of the table are highlighted at the end of each feature description. The identified features are:

1. The model for achieving parallelism should be uniform across both shared-memory- multiprocessor and distributed-memory environments. The application should not have to use one method to create and manage parallelism in a shared-memory environment and a different method in a distributed-memory environment. (Uniform interaction)

2. The size of the entities being manipulated should be small enough to support and encourage code reuse. For example, functions and routines to manipulate a generic list are more reusable than a complete database system. (Code reuse)

3. It should be possible to create processes dynamically. This permits an application to adjust the number of parallel components to take advantage of different hardware environments or changing service demands. (Dynamic process creation)

4. It should be possible to make a parallel request and collect the results at a later time. This permits parallelism between a client and server, regardless of the server implementation. (Delayed result acceptance)

5. Conversely, the ability to generate parallelism on the server side is greatly enhanced by the facility to return a result to the client while the server continues to execute. This allows the supplier of a service to create parallelism independent of the manner the client is making the request. (Early reply)

6. It should be possible to monitor the completion status of outstanding parallel requests. This will allow the client to maximize parallelism with the server by performing additional processing while waiting for the request to finish. (Request monitoring)

7. It should be possible for a client to make a parallel request of a server, and then ignore the results. This recognizes that a client may not need a result and that accepting the result introduces a needless synchronization point between the client and server that reduces parallelism. (Ignore results)

8. It should be possible to serialize requests to a server. A major problem for certain applications is nondeterminism with respect to requests. Given communications delays and changing load patterns, requests can easily arrive out of order at the server. Under certain circumstances, a collection of requests needs to be executed serially with respect to each other, but they can be executed in parallel with the initiating thread of execution. An example of this is screen updates. The
updates may need to be performed in a specific order relative to each other, but there is no need to execute the requests sequentially with respect to the initiating thread. *(Serialize requests)*

9. Concurrency control should be automatically supplied by the system. The application programmer should not have to explicitly manage concurrent accesses in the system. *(Automatic concurrency control)*

10. Support for parallelism should be well integrated into the system. That is, the support for parallelism should seem to be a natural part of the language used for programming. Typically, this means that support for parallelism is done at the language level to avoid presenting one programming model for the language and another for the external system responsible for the parallel and distributed processing support. *(Language support)*

11. Servers should be able to control the service order of requests. Without the capability to order service requests, some requests may be forced to fail even though a different processing order would have allowed them to succeed.Suspending a request and completing it later, or using guard conditions allows the server to determine the order of processing and subsequent replies. *(Selective request processing)*

12. It should be possible to group requests together. Certain requests, such as requests to update a collection of terminal screens, can conceptually be viewed as one request. It should be possible to group these requests together and treat them as one. This facility allows groups of requests, which are logically related, to be treated as one request. *(Request grouping)*

13. The units of parallelism should be easy to identify and use. To effectively harness parallelism, the unit of parallelism should be naturally defined within the environment, and the mechanism for extracting the parallelism should act directly on the unit of parallelism. For example, creating parallelism with a `fork()` call is not an easy way to create parallelism because it is difficult to identify the unit of parallelism and then start it executing. *(Easy to identify units of parallelism)*

As mentioned earlier, Table 2.1 provides a summary of the features supported by the systems that were surveyed. The rows of the table correspond to the language or programming system of interest, and the columns identify the feature of interest. In the table a "✓" indicates a feature is supported; a blank is an unsupported feature, and a "-" indicates that a feature is not applicable. The entries in the table attempt to capture the "spirit" of the system, with a negative entry meaning that a feature might be supported, but normally one would not think of it as being sup-
TABLE 2.1 Summary of language/system support for parallelism.

<table>
<thead>
<tr>
<th>Language/System</th>
<th>Granularity</th>
<th>Shared memory</th>
<th>Distributed memory</th>
<th>Uniform interaction</th>
<th>Code reuse</th>
<th>Dynamic process creation</th>
<th>Delayed result acceptance</th>
<th>Early reply</th>
<th>Request monitoring</th>
<th>Ignore results</th>
<th>Serialize requests</th>
<th>Automatic Concurrency Control</th>
<th>Language support</th>
<th>Selective request processing</th>
<th>Request grouping</th>
<th>Easy to identify units of parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel RPC</td>
<td>C</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Streams and Pipes</td>
<td>C</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiprocessors</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Promises</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>SR</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Poker</td>
<td>M</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>MUPPET</td>
<td>C</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>PVM</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Multilisp</td>
<td>F</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Actors</td>
<td>F</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Linda</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Distributed Smalltalk</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>CST</td>
<td>F</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>ACTRA</td>
<td>C</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>SINA</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>POOL-T</td>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Eiffel</td>
<td>M</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Choices</td>
<td>M</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

ported. For example, a system like Parallel RPCs would work in a shared-memory environment, but that is not the environment it was designed for; therefore, it was given a negative entry for shared-memory support.

In addition to features enumerated above, three other entries are also shown in the table. These are:
1. **Granularity.** This column specifies the granularity of the parallelism supported by the system. C is for coarse-grained, and can be associated with parallelism at the process level and long communication delays. M is for medium-grained, and can be characterized as parallelism at the procedure level. F is for fine-grained, and can be associated with parallelism at the statement or instruction level.

2. **Shared memory.** The system is designed primarily for use in a shared-memory environment

3. **Distributed memory.** The system is designed primarily for used in a distributed-memory environment.

In order to support programming in the types of parallel and distributed environments described in Chapter 1, all the features identified in this section need to be present. If a feature is not present, then either the expressiveness of the system is lacking, and it is unable to handle parallel and distributed computation in a particular way, or else the programming and administrative burden of a commonly used feature is made the responsibility of the application programmer. The effect is that needless constraints are placed on the programming techniques usable for programming in parallel or distributed environments. None of the systems surveyed provide support for all the identified programming features. Most of the systems support about half the features, which indicates a concentration on some subset of how to manage parallel and distributed processing.

The programming environment of interest in this work is comprised of both shared-memory and distributed-memory machines. To simplify programming in these types of environments, a single programming model must be supplied to the programmer. The programmer needs to be able to develop applications that do not depend upon, or make assumptions about, the underlying target hardware environment.

Based on a review of these features and the above observations, a broad outline of a system to support parallel and distributed programming can be developed. The requirements for a high-degree of code reuse, a uniform interaction environment for both distributed-memory and shared-memory environments, and easy-to-identify units of parallelism suggest using an object-oriented language. Objects, with their strong data encapsulation and well defined inter-
action points, are well suited to be the unit of parallelism and automatic concurrency control. Support for parallelism needs to be embedded in the language to ensure a consistent and uniform programming environment. This uniform environment applies to the programming environment presented for both shared-memory and distributed-memory machines. The system should also support the generation of parallelism from both the client and server side, and when parallel requests are made the client should be able to ignore the results or monitor and collect them later. For server objects, it is extremely important that some mechanism be provided to delay or suspend a request. Without such a facility, it is difficult to synchronize multiple threads of control. Taken together, these features can be used to develop an easy-to-use, consistent and uniform environment for parallel and distributed programming on shared-memory and distributed memory machines. To support these identified features, the object-oriented system and language Raven was developed. Raven has built-in support for parallel and distributed programming. By using an object-oriented language, issues associated with reusing code, identifying the unit of parallelism, and providing a single uniform programming environment for shared-memory and distributed-memory machines are addressed. Some of the functionality naturally arises through the use of an object-oriented system, while others require explicit support from the underlying system. Adding system support does not affect the application visible facilities available to manage parallelism and distribution. Instead, it affects the programming effort by relieving the programmer of having to perform certain system administration related tasks. The system performs the administration automatically instead of making it the programmer’s responsibility.

Raven addresses the issues associated with making parallelism accessible by incorporating support for a parallelism directly into the language. Raven identifies a single execution point within the language where parallelism can occur (Section 4.1). The result is class-based parallelism (Section 4.2), which is parallelism implemented on the server side, and user-based parallelism (Section 4.3), which is parallelism implemented on the client side. These two facilities provide support for early reply, dynamic process creation, and the ability to perform selective request processing. Creation of parallelism is supplemented with the introduction of the
notion of Certificates (Section 4.3.1). Certificates provide the mechanism to monitor, track, and group the units of parallelism resulting from user based-parallelism. Properties (Section 4.4) are a way of associating system services with individual objects and are the mechanism used to supply automatic concurrency control. InvokeStreams (Section 4.6) interact with Certificates to serialize certain types of parallel requests. By confining the creation of parallelism to a single point and providing ways to manage and monitor parallelism, the creation of parallelism at the client and user side can be addressed.

If the Raven System were included in Table 2.1 it would be classed as a system that supports medium-grained parallelism and provides a uniform interface for programming in a shared memory and distributed memory environment. Its object-oriented nature encourages code reuse while providing a well defined interaction point for creating parallelism. The well defined interaction point simplifies the identification and management of the units of parallelism. Because the Raven language has built-in language and system support for parallelism, all the other features for supporting parallel and distributed computation would be marked as present.

2.5 Summary

This chapter began with a discussion of the differences between distributed and multiprocessor environments and how they affect the programming of an application. This was supplemented with material discussing granularity and how to construct parallel programs. A number of different parallel and distributed systems were then surveyed to provide a review of the work that has been undertaken in this area. Each of these systems uses a different technique for managing and dealing with parallel and distributed computing. Some of the systems dealt specifically with supporting parallelism, while other approaches were more general purpose in nature. From these systems, the most important features and useful features for supporting parallel and distributed programming in shared-memory and distributed-memory environments were identified. None of the surveyed systems supported all the features, thereby indicating a bias
towards supporting certain types of computations. To overcome this bias, the Raven language
and system with built-in support for parallelism was proposed as a way to provide this support
in a single, consistent programming environment.
CHAPTER 3  
The Raven System and Language Overview

Raven is an object-oriented system that runs on a number of different hardware platforms, ranging from a collection of workstations connected via a network to shared-memory multiprocessors. In the past, these types of environment have used entirely different programming models and primitives designed to take advantage of the particular environment. By using objects as the primary interaction mechanism, Raven makes it possible to reduce greatly the distinction between programming for a distributed-memory environment and programming for a shared-memory environment. This chapter introduces the Raven programming language and environment that form the basis of the work described in the remainder of this thesis.

3.1 The Passive and Active Object Models

In object-oriented systems, two programming models [31] are predominant, the passive model [83][38] and the active model [67][68]. Each can be related to a particular way of implementing an object system. The distinction between the two models is based on how the meth-
ods of an object are executed and the effect that has on the way objects are programmed. Methods, or behaviors, are the routines that manipulate the data an object encapsulates.

Conceptually, the active model associates a process with each object. Invocation requests are accepted at the object and processed sequentially. The programmer is, therefore, presented with a system that is seen as accepting the request, processing the request, returning a result, and waiting for the next request. The effect is that there is no concurrency within the object and the programmer does not need to take explicit measures to provide concurrency control.

In the passive model, there is no process associated with an object. Instead, a thread of control takes execution to the object, selects the method to execute and executes the code. The result is a model that has many similarities to the subroutine call in a local environment or a remote procedure call in a distributed environment. Since an execution thread goes to an object, it is possible for multiple threads of control to be active within an object at once. This has obvious programming ramifications. It is now the programmer’s responsibility to ensure the consistency of an object’s instance data by explicitly identifying critical code sections and protecting them with mutual exclusion primitives like semaphores or monitors.

The passive and active object models differ primarily in how concurrent access to an object is handled. This directly affects the programmer by dictating whether or not explicit management of concurrent access to an object is required. The models also affect the overall design of an application by placing constraints on the type and scope of objects, along with how the objects can interact with each other. Consider Figure 3.1, which shows two objects invoking on each other: objects are ovals, the dotted line indicates the flow of execution, and the rectangles are particular methods. In this example Method_1 of object A invokes Method_2 on object B. Method_2 of object B then proceeds to invoke Method_3 of object A. If this scenario were programmed in an active model, a deadlock would occur, since Method_3 could not complete until Method_2 had finished. In a passive model the execution sequence would be permitted, but it would be the application’s responsibility to manage the concurrent access to the object. Depending upon the concurrency control implemented by the programmer, this scenario
might also deadlock in the passive model. It is clear that the programming model affects more than just the implementation of a single object. The programming model affects the way in which a problem is decomposed into objects and how those objects are organized. Instead of focusing on how an object system is implemented, and the resulting programming model consequences, the Raven system has adopted an object model that concentrates directly on the issue of concurrency.

In the Raven model, concurrency control is performed automatically at the method level. Whether a method can execute on an object depends upon the other methods executing in the object and the execution source of a method (see Section 4.5). Basically, methods are classified as read methods or write methods, with Raven allowing multiple readers or a single writer to be active in an object. However, when the read and write methods are associated with the same thread of control, both methods are permitted to be active in the object. In this case the read and write method interactions are viewed more on the level of a subroutine call than as a new request emanating from a different thread of control. Within the context of the scenario depicted in Figure 3.1, the invocation of Method_3 on object A by object B would not block.

In many respects, the Raven model of programming is a hybrid of the passive and active mod-
els. Like the passive model, Raven has the notion of a thread of control. The thread of control establishes relationships between objects at runtime. Within an individual object, however, the notion of a thread of control does not force the programmer to explicitly take action to protect an object’s instance data with critical regions. Like the active model, the programmer can assume that only one method is active in an object at a time, since the system provides the concurrency control.

The Raven object model is independent of the underlying object implementation model and makes no assumptions about the implementation. Both active and passive implementation models were considered when Raven was being designed. Ultimately, a passive implementation model was selected because it was perceived to be easier to implement in the local case, would be better able to take advantage of a shared-memory multiprocessor environment, and would have less execution overhead since context switching on a method invocation would not be required. Although the selection of a passive implementation model affects how certain services are supplied, it does not manifest itself at the application program level.

### 3.2 Raven

One way to explore the system and language issues associated with supporting and using parallelism is to take an existing system and modify it to provide the required services. This approach has several disadvantages with respect to some of the goals of this work. Two major requirements of an environment to explore these goals are:

1. **Portability**: To demonstrate the usefulness of the system, the system needs to be portable across architectures and machine types. In this context, the cost of porting the software, both in terms of money and time, must be considered. Commercial software with its cost, and free software which is often designed to run in only certain environments, cannot meet these portability requirements.

2. **Changeability**: To be able to support new concepts and to deal with performance and implementation issues, the system needs to be changeable. With an integrated approach to supplying parallelism it is possible that new services or changes to existing ones will require changes to the operating system, runtime system, or language. When problems arise with the implementation it should be
possible to make the changes to the component(s) of the system most responsible for the problem. In a restricted software environment this is not possible.

Combining the above requirements with the goal of developing a system to explore the issues associated with building parallel and distributed systems in an object-oriented environment, it was decided that these goals could best be met by developing a new system. The result is the Raven System [1], a project in the Computer Science Department at the University of British Columbia. To speed up the development of such a system it was decided that, where possible, existing software would be used. Besides the obvious implementation speedup advantage, this also permits individual software components to be evaluated independent of the Raven system. This evaluation will identify what software can be used and where design and programming effort needs to be concentrated to improve system performance. This approach makes it possible to concentrate the development effort on support for the new features of interest and on improving performance in the existing system components.

The Raven system consists of four main components: a compiler, runtime system, class library, and virtual machine (threads). The separation of Raven into these different components represents a logical breakdown of the functionality of the various parts of the Raven system. The separation also reflects how likely a component of the system is to change. The virtual machine component of the system is expected to be the most stable since it forms the foundation of the system, and changes to it can ripple through the rest of the system. This would be followed by the runtime system, the class library and finally the user's application.

3.2.1 The Raven Language

The programming language, also called Raven, is heavily based in its syntax on the C programming language. The Raven class system is similar to that of Objective C [35] and Smalltalk [49], but unlike these languages it is statically typed like C++ [97]. Although the language is statically typed, the method binding is done at runtime, thus providing dynamic method binding.
To illustrate some of the basic constructs in the Raven language and to provide a basis for introducing the basic terminology and concepts associated with an object-oriented system, consider Figure 3.2. This is a complete Raven program that performs a small simulation of a pencil cup. Common stationery items can be added to and removed from the pencil cup and the weight of the pencil cup computed. Raven keywords and predefined methods have been highlighted in the example.

Object-oriented languages accomplish work by manipulating objects, and by relying upon properties of encapsulation, inheritance and organization [78] to do this. Conceptually, an object is an entity that has its own private data and a collection of functions to operate on that data. An object is in effect the live data structure that changes during program execution. How an object is organized and behaves is determined from its class definition. A class definition specifies the exact organization of the data and the routines to manipulate the data. The class can be viewed as a prototype or template for a particular data type. Classes are themselves further organized into a subclass/superclass hierarchy. At any level within the hierarchy a class inherits all the attributes from the classes it is descendant from. Some languages support multiple inheritance, but Raven does not. In Raven each class is directly descendant from one class. At the top of the Raven class hierarchy is the class Object from which everything is descendant. Figure 3.3 provides a pictorial representation of the relationship between the programmer-defined classes in the pencil cup example. Object is at the top of the hierarchy and has as its direct descendents the classes Cup, Weighable and Main. The class Weighable has two other descendants, Pencil and Scissor. At any level the classes in the hierarchy inherit all the attributes of the higher level objects they descend from. This includes both the higher level objects’ instance data and their methods.

Although inheritance ensures that a subclass is a subtype of a super class, Raven supports sub-typing through contravariant type-checking (a conservative type equivalence policy) [1]. Raven’s type checking rules are defined as follows:
```plaintext
#include <Basic.r>
#include <Array.h>

class Weighable {
    wght: Int;
    behavior weight() : Int;
        constructor(obj_weight : Int);
    }
    constructor {wght = obj_weight; }
    behavior weight {return wght;}

class Pencil <- Weighable {
    constructor();
    }
    constructor {
        super.constructor(5); }

class Scissors <- Weighable {
    constructor();
    }
    constructor {
        super.constructor(20); }

class Cup {
    items : Int;
    cup : Array[Weighable];
    constructor();
    behavior add(item : Weighable) : Int;
    behavior remove() : Weighable;
    behavior weight() : Int;
    }
    constructor {
        cup = Array[Weighable].new(1);
        items = 0;
    }
    behavior add {
        items = items + 1;
        cup.atPut(items, item);
        return 1;
    }
    behavior remove {
        if (items > 0) return cup.atGet(items--);
        else return nil;
    }
    behavior weight {
        var total_weight, i : Int;
        total_weight = 40; // weight of cup
        for (i = 1; i <= items; i++)
            total_weight += cup.atGet(i).weight();
        return total_weight;
    }

class Main {constructor(); }
    constructor {
        super.constructor();
        behavior start {
            var cup : Cup;
            cup = Cup.new();
            cup.add(Pencil.new());
            cup.add(Pencil.new());
            cup.add(Pencil.new());
            "The pencil cup weights".print();
            cup.weight().print(); "n".print();
            cup.add(Scissors.new());
            "The pencil cup weights " .print();
            cup.weight().print(); "n".print();
        }
```

FIGURE 3.2 Raven pencil cup example
A variable of type $T$ can reference any object of class $S$, if $S$ is a subtype of $T$. $S$ is a subtype of $T$ if $S$ is identical to $T$ or if the following conditions hold:

1. $S$ provides at least the behaviors of $T$.
2. For every behavior in $S$ that has a corresponding behavior in $T$, the corresponding behavior has the same number of parameters and results. (This is exclusive of the constructor method.)
3. The type of the result returned by $S$'s behavior is a subtype of the result returned by $T$'s behavior.
4. The parameters of $T$'s behaviors as subtypes of the parameters to the corresponding arguments of $S$'s behavior.
5. $S$ contains at least the public instance variables of $T$. (Public instance variables, are instance variables visible outside the class.)
6. The corresponding public variables in $S$ and $T$ have identical types.

This means that a class can be a subtype of another class even if it is not part of that class's inheritance hierarchy. These typing rules are illustrated in Figure 3.4. In the example, the class Eraser is defined, but it does not inherit from Weighable. It is a subtype of Weighable because all the methods it has in common with Weighable (i.e., weight()) have the same number and
CHAPTER 3: The Raven System and Language Overview

```plaintext
class Eraser {
    wght : Int;
    wght_KGs : Int;
    behavior weight() : Int;
    behavior weight_KG() : Int;
    constructor();
}
constructor { wght_KGs = wght / 1000; wght = 10000; }
behavior weight { return wght; }
behavior weight_KG { return wght_KGs; }
...
cup.add(Eraser.new()); // Valid usage of instance of Eraser

FIGURE 3.4 Eraser as a subtype of Weighable.
```

type of parameters, and return the same result. Additionally, the common public instance variable (wght) has the same type. Since instances of Eraser are sub-types of Weighable, they can be used anywhere that an instance of Weighable can. Implementation inheritance is, therefore, the main use of inheritance in Raven.

In addition to explicitly defined classes, Raven supports a second type of object, the basic or primitive object. These are basic Raven entities that cannot be decomposed further, are supported and manipulated by the compiler, and are manipulated directly by the hardware. Floating point numbers and integers are the current primitive objects. These are denoted as Int, and Float, within Raven declarations. User-defined classes are complex objects and are composed of primitive objects and references to other complex objects.

Class definitions in Raven start with the keyword class and terminate when a new class definition begins or when the end of the file is reached. The first class defined in the pencil cup example (Figure 3.2) is the class Weighable, and its definition terminates when the definition for the class Pencil begins. A class definition consists of two parts: one part describes the data items
for the class, and the other provides the definitions of the routines that must be used to manipulate the data in the objects of a particular class. The data defined within a class is referred to as the object’s instance data. The data defined within a method is referred to as method data.

Unlike a struct in C, a class also defines the routines that can be called to manipulate the instance data. The routines to manipulate the instance data are called methods or behaviors. It is only through methods that an object’s instance data can be modified. The class Cup defines the methods add(), remove(), weight() and constructor(). An object, also referred to as an instance, is a particular instantiation of a class. To provide a consistent object view, classes themselves are instances of classes. When a class, like Cup, is defined, the Raven system automatically defines a meta-class for the defined class. The meta-class has its own predefined methods. One of the class’s predefined methods is new(). The method new() is invoked on a class object to create an instance of that class.

All objects in Raven are identified by an object ID (OID), and variables in Raven always reference a valid object. Instance and method variables are initialized by the system to reference the predefined object, nil. An object reference (OID) is an identifier, or capability, that can be mapped to the internal representation of an object. The internal representation contains information needed to locate the object’s instance data and to lookup methods.

Objects support the is-a relationship in that each object is a particular type. Figure 3.5 provides a pictorial representation of the is-a relationships between instances of objects, class definitions, and meta-classes along with the inheritance hierarchy and class organization of a Raven environment at runtime. In the pencil cup example the classes Pencil and Scissor inherit from the class Weighable.

Each Raven class must define the method constructor(). This method is automatically executed when an instance of a class is created. The code within the constructor is responsible for performing any object-specific initialization and for calling the parent object’s
CONSTRUCTOR. The parent object is referenced through the predefined object super; examples of this are shown in several of the constructor methods of the pencil cup example.

Every Raven program must also define the class Main with the method start(). This is analogous to main() in a C program in that execution starts with the start method, after the constructor for Main has been called. In the start() method of the pencil cup example, a local variable of type Cup is defined and a new instance of Cup assigned to it. Three pencils are created, and added to the cup. The cup is weighed and the weight is printed. After that, an
instance of Scissor is created and added to the cup. The cup is again weighed and the cup’s weight printed.

In summary, the pencil cup example defines a pencil cup object which can have other objects added to or removed from it. That is, there are a number of operations that can be performed on this pencil cup. These operations are known as methods or behaviors. The actual cup definition itself is defined by the class Cup, which is a template for the canonical cup. In more sophisticated systems the new operator could be parameterized to impart different qualities onto the pencil cup. Cup contains two instance variables: items, an instance of the primitive class Int, and cup, an instance of the parameterized class Array. The parameter in this last case indicates that the array contains items of the type Weighable.

Although not shown in the pencil cup example, Raven methods can also support copy semantics. The arguments to methods are always references to objects. If the same reference is passed to multiple methods the same object is updated. Under certain circumstances a method may want its own private copy of an object. To support copying, individual parameters can be marked as copy parameters. Figure 3.6 shows how the add() method of the class Cup could

```ruby
behavior add(copy item: Weighable) : Int;
```

FIGURE 3.6 Tagging a parameter as copyable.

have the item parameter marked as being copyable. (A returned result can also be marked as copyable in a similar fashion.) When a parameter is marked as copyable, a copy of the object is made, and it is the reference to the copied object used in the method. When an object is copied like this, the copy is a deep or recursive copy that may cross machine boundaries. In a recursive copy all top level objects are copied, as are all the objects referenced by the copy and so on.
### 3.2.2 Raven Class Library

Part of the power of an object-oriented system results from the reuse of code. To address this issue, a Raven class library is provided. The class library is written in Raven and builds complex objects from primitive Raven objects and other existing classes. This provides a basis for a class system that the application programmer can work with. The Raven class library also defines a number of classes used by the compiler. For example, some of the support for parallelism is provided through Raven classes and the compiler generates code to make direct use of these classes. Raven classes in this category are `Thread`, `Companion`, `Certificate`, and `System`. Although these classes are accessible to the Raven programmer, and can be treated like any user defined class, their primary purpose is to support the class environment expected by the compiler.

Some of the supplied Raven classes also interact with the Raven runtime support system. Because the runtime system is written in C, the classes are a mixture of Raven code and special escapes to make procedure calls, using C, into the runtime environment. To be able to do this properly, the programmer needs to have an intimate understanding of the underlying structure of Raven objects and the runtime environment. Unlike other languages, which often provide support for inter-language procedure calls, Raven does not supply such support. The output from the Raven compiler is C code and the Raven compiler provides a mechanism that allows arbitrary code to be inserted into the output file. Raven classes that need to use the Raven runtime support system must use this mechanism to insert C code directly into the compiler output. To be successful, the programmer needs a thorough knowledge of the intricacies and representation of objects and how they are supported by the runtime system. Clearly, classes requiring this level of understanding of the construction and implementation of the Raven system need to be provided as part of the class library and not as user-written code.

Not all the classes in the Raven class library interact with the runtime system. Some classes are included because they are expected to have considerable reuse by other classes. The classes `Queue`, `List`, and `Stack` are examples of such classes.
The class library also makes a contribution to defining what a Raven environment is. Depending upon how the classes are defined and the behaviors the classes have, different Raven environments can be constructed. To work, the class library has to supply only a fixed set of classes and behaviors expected by the compiler. Any classes and behaviors that the compiler expects to use must also provide the types of results and perform the expected actions. For the work described in this thesis, the term Raven refers to a particular environment defined by the compiler, class library, runtime system and the virtual machine support. Since Raven is a changing language, changes across Raven releases are to be expected and have occurred.

3.2.3 The Raven Compiler

Prior to the existence of the Raven compiler, all the support for objects and their use was done in the C programming language, and required a preprocessor, runtime support library and considerable effort by the programmer. Although this combination of development tools made it possible to develop and test many of the ideas and feature implementation approaches ultimately used in the Raven system, it was difficult to use. The preprocessor and macro package lacked the sophistication to handle the manipulation of complex expressions without becoming a compiler. Taken together, all these idiosyncracies made the existing Raven environment difficult for other people to use. To add to the confusion, this environment also presented two different programming models that can be difficult to reconcile within a program: the regular C language, and the Raven object model with its own programming rules and regulations. As more people became involved with Raven, the desire for a compiler, and the improved language support it would provide, increased until a compiler was actually written [1]. The introduction of a compiler helped to provide a unified programming model where all data manipulation is done through objects. Stronger type checking and rule enforcement also provides more feedback to the programmer at compile time, thereby reducing the number of runtime errors. During compilation, compilers maintain more information about the current environment than a preprocessor and macro package does; therefore, the compiler can pass
along this information to the runtime system to improve performance and increase functionality.

The Raven compiler is based on an LALR(1) grammar, using the compiler development tools yacc and lex [60]. All the supporting software is written in ANSI C and is compiled using the Free Software Foundation's C compiler, gcc. The output from the Raven compiler is ANSI C with the final compilation and linking for a particular environment being done with gcc. Any platform that has gcc running on it should be able to build and run the Raven compiler.

3.2.4 The Local Raven Environment

The Raven runtime system is a layer of code between the Raven language and the underlying threads system. It is designed to be application-independent and provides a set of runtime services that the Raven compiler can use. In essence, it is a collection of functions that provides the basic character and functionality of the Raven system. The runtime system is the heart of Raven and taken together the services provided by the runtime system define the nature and character of the Raven environment. Among other things, the runtime system:

- Defines the bootstrap level objects;
- Defines the routines used to specify class definitions;
- Defines what an object looks like in the running system;
- Provides the method dispatch code;
- Provides support for remote objects (Section 3.2.6);
- Defines the Raven environment.

A short elaboration of each of the above points will now be undertaken. One of the major functions of the runtime system is to provide the mechanism to be used in defining new classes. The bootstrap classes defined by the runtime system are Object, MetaObject, Class, MetaClass, and MetaMetaClass. These are the objects and classes (Figure 3.5) at the top of the class hierarchy, and all other objects descend from them. Being the base classes, these initial classes pro-
vide the methods called to construct new classes and instantiate objects. Consequently, these classes must be created as soon as the Raven environment begins execution.

The Raven compiler is responsible for emitting the code needed to construct the class hierarchy for the application during the initialization of the Raven system. From the runtime perspective, the class hierarchy is a collection of data structures that provide the information to construct an instance of an object. As well as providing information about the basic data layout of an object, a class also specifies the routines associated with the object.

Raven supports dynamic method lookup and this is done by the Raven method lookup routine that is called when an invocation is performed. Depending upon the type of object and the way it is being used, different invocation scenarios are possible, and each invocation scenario has an invoke routine associated with it. To make use of this flexibility, the runtime system assigns an invocation handler to each object when it is created. The selection of the handler is based on the special processing needs of the object, how the object is created, and the current runtime environment. A default generic invocation routine can be assigned to objects that require special services not supported by the existing invocation routines. The default handler is general purpose and supports all types of invocations, but at an increased execution cost. This approach has the advantage that objects which need complicated invocation handlers can have them without introducing a performance penalty for the objects that do not. Figure 3.7 provides a pictorial representation of how invocation handlers are assigned to objects. In the figure there are 5 different objects and 3 types of invocation handlers. Each object is assigned its own handler. In this case, objects A and C use a generic invocation scheme, object D uses a remote invocation handler, and objects D and B use a simple invocation handler.

In its most generic implementation, an invocation consists of several parts. As soon as the invocation routine is entered, the method that needs to be run is looked up. A method cache is maintained to make this operation fast. Once the method is located, a series of pre-invoke handlers is called. Next, the actual code to execute the actions of the method is called. Upon the method's return, a series of post-invoke handlers is called. Each object has its own require-
ments for pre- and post-invocation method handlers. At object creation time, the Raven runtime system selects the invocation handler best suited for dealing with the invocation needs of the object.

### 3.2.5 The Virtual Machine

The threads portion of the Raven system, which is derived from the work done in [80], defines a virtual machine, or more accurately an operating system interface, that the other layers of the Raven system can use. Through this mechanism the threads package provides a set of system calls that insulate the other layers of the Raven system from the particular machine architecture, while at the same time providing the functionality of an operating system. The threads library, however, does more than map its calls to the matching calls of the native operating system: it also provides support for memory management, thread creation, scheduling of threads, semaphores, and send/receive/reply primitives.
The threads library also contains all the operating system and hardware-specific code for the Raven system. Porting Raven to a new environment consists primarily of porting the threads package. Depending upon the underlying architecture and operating systems, the amount of effort to port the threads software will vary, but other parts of the Raven system are unaffected. As an example of this portability, Raven is implemented within a process on machines running UNIX and as a collection of Mach threads under Mach 3.0.

3.2.6 Supporting Distribution in Raven

The previous two sections have provided an overview of what a local Raven environment appears like. In Raven, invocations on objects are not confined to the local environment, and it is possible to invoke methods on objects managed in other Raven environments. The approach to dispatching methods makes support for remote object invocation relatively straightforward. Within a Raven environment, each remote object used in the environment has a proxy. A proxy is a local object that acts as the agent, or representative, for the remote object. To the application, a proxy is identical to a local object. To the runtime system, however, the proxy is an object with remote instance data; therefore, the method invocations need to be run remotely. Since a proxy is a real object, it has an invocation routine associated with it. To support remote invocation a proxy's invocation handler is set to the invocation routine responsible for remote execution. Remote invocation requires additional system support to manage the network communication between the source and the destination sites. This includes managing of a method invocation protocol and support for the marshalling of data between hosts. Figure 3.8 shows the relationships between a proxy object and a remote object. In the example, an initiating object in environment A performs an invocation on the proxy for object A. To the initiating object, the proxy appears like any other object. The proxy identifies the remote request handler as its method invocation routine. The remote request handler packages the request and has it communicated to the remote site. At the remote site the request is unpacked and the method invoked on object A by a remote worker. Results are returned by the reverse path. The low-
level transferring of data between Raven environments is handled by the communication managers which exist in each local Raven environment.

The terms Raven environment and Raven system are used throughout the thesis. The meaning is usually apparent from the context. Figure 3.8 shows two Raven environments interacting with each other. The terms are used to convey information concerning the implementation. An environment is a single address space with a particular configuration. A Raven system is a collection of Raven environments.

3.3 Summary

In this section the Raven language and system have been introduced. Raven consists of a compiler, runtime system, class library, and virtual machine environment. Each of these compo-
nents has been briefly described to provide an overview of the system and the type of functionality assigned to each component. An example program is used to introduced the syntax of Raven and some of the underlying concepts of object-oriented systems and how they relate to Raven. Subsequent chapters describe the extensions and modifications made to the Raven system to support parallel and distributed programming.
Most operating systems provide support for the creation of processes, and, in its most basic sense, that is all that is required to supply some form of parallelism. However, support for parallelism encompasses more than a one-dimensional way of creating new threads of control. To be useful, there needs to be cooperation, or common purpose, between the execution threads, and the programming environment needs to make that possible. To provide support for parallelism in an integrated fashion, a whole approach needs to be developed, extending from the creation of the multiple threads of control to the mechanisms that exist for the cooperating execution threads to interact and synchronize with one another. Thus, to provide a cohesive integrated environment for supporting parallel and distributed computation, one needs to question the whole premise that the operating system and programming language are separate entities. Current environments require the programmer to maintain two different programming perspectives. At one level there is the programming model presented by the programming language; at a second level is the view provided by the operating system. Depending upon the aspect of the programming problem being examined, the programmer must interact with different program-
ming models and then reconcile them within the programming language. This also means that the end programming environment varies from machine to machine, depending upon the operating system and functions it provides. This certainly does not facilitate the development of portable code.

Although parallelism can be introduced simply by providing a mechanism to create multiple threads of control, this does not make the task of programming in a parallel or distributed environment easier. To simplify the task of programming in a parallel or distributed environment we exploit the strong data encapsulation and well defined object-access interface provided by an object-oriented system to merge the facilities for parallelism with the general programming techniques used in object-oriented systems. The motivation for this approach stems from the observation that, at least in the near term, compiler-generated parallelism suffers from the problem of making parallelism either too fine- or too coarse-grained. To combat this problem the programmer needs to be actively involved in specifying and controlling parallelism. As a result, the decision was made to incorporate features for coarse-grained parallelism (which assumes that any parallel activity will involve several thousand instructions) directly into the Raven programming language. This has resulted in a multipronged approach to supplying and supporting parallelism.

The result is new syntax within Raven for specifying parallelism, the introduction of properties to specify concurrency control and to manage the class explosion, along with modifications to the runtime system to manage and control resource allocation. To support distributed computations, the system provides some for object location transparency.

### 4.1 Specifying Parallelism in Raven

Existing approaches to parallelism require the designer to decompose a program into functions and procedures as well as processes. As a result, the programmer must deal with the issues associated with interprocess communication and process synchronization. This additional step adds complexity to the task of programming and increases the design, programming
and debugging effort needed to get a program running. Raven supports the explicit creation of processes through the supplied Thread class. But, such an approach does not adequately address the issues associated with providing parallelism by integrating it at the language level. The encapsulation of the low-level system thread entities into a Thread class ensures a uniform object view for process creation at the application level. The actual act of creating the thread and associating an object and a method with it, however, involves considerable effort by the programmer. Under many circumstances it should be possible to identify parts of the code that can run in parallel and then delimit those regions, without having to manage the thread creation explicitly. Ideally, it should be possible to take an existing program and make only minor changes to achieve parallelism.

Since methods provide the mechanism for object interaction, it is logical that a modification to the way methods are used or specified would provide an easy-to-understand way of achieving parallelism. Two possibilities arise. First, parallelism can be specified directly within the methods of the class: this is referred to as class-based parallelism, since the parallelism is the result of the way the class is implemented. A second possibility is that the invoker creates the parallelism at invoke time: this is user-based parallelism, since the user of the class is responsible for specifying the parallelism.

### 4.2 Class-Based Parallelism

Class-based parallelism refers to parallelism that results from the way an object is implemented rather than the way it is used. With class-based parallelism the class implementor designs methods in such a manner that the method’s use results in parallelism between an instance of the class and its invoker. This parallelism is accomplished through the use of Raven’s early result and delayed result constructs.
4.2.1 Early Result

The return statement in most programming languages serves two functions: it returns execution control to the caller, and it returns a result. In Raven these two functions can be separated. A result statement can be executed which returns a result to the invoker, thus unblocking the invoking object while continuing execution in the called method. This results in an additional thread of execution. A method’s execution completes when a return statement is executed, either explicitly or by completing the method. Once a result statement has been executed, however, the value of any subsequent return statement is ignored.

```
behavior postMail {
    return_code = message.validFromField();
    result return_code; // added result line
    if (return_code == OK)
        message.sendMessageToRecipients();
    return;
}
```

**FIGURE 4.1 Example code demonstrating early result**

Figure 4.1 illustrates how, through the addition of a single result statement, a formerly sequential method is parallelized. For this approach to be effective, the code following the result needs to be sufficiently coarse-grained to amortize the cost of thread creation.

4.2.2 Delayed Result

When an object is being accessed concurrently, it is conceivable that the received method invocation order does not correspond to the order the invocations need to be performed in. For example, in the readers/writers problem a read request may be made before there is data available for reading. In these circumstances, it is desirable for the processing of a method to be delayed or suspended. An approach used in other systems is to place guard conditions on the
execution of a method. The guards specify the conditions permitting method execution. One problem with the guard approach is that once the execution of a method has started it must be completed. Therefore, all the conditions needed for a successful execution must be determined in advance and only those methods satisfying the guard conditions can be permitted to execute. Depending upon the complexities of the execution restrictions, setting up guards can be an onerous task. Another potential problem with guards concerns their implementation and how often the guard conditions must be rechecked to determine whether any of the pending methods can now be executed.

In Raven a different approach, that of a delayed result, is used. With this approach an object accepts and processes methods in the order they are presented, but a method does not have to run to completion. A method may choose to stop executing so that other methods may execute, or to wait for certain conditions to be met, such as resources becoming available. This has the advantage over guards in that the programmer does not need to specify all the conditions needed for a method to execute at method invocation time. This eliminates the need to repeat the guard logic inside the method if there are multiple condition sets permitting execution.

When a delayed result is used, the method executes until it determines that it needs to wait, at which point it executes a `leave` statement (see Figure 4.2). When the delayed result is performed, the method terminates without returning a result, thus keeping the invoker blocked. This contributes to parallelism by permitting multiple methods to be active in an object. The invoker remains blocked until another thread uses the `result` statement to supply the return result.

Since there may be multiple threads of control waiting on a delayed result, each waiting thread must be identifiable; therefore, a thread has a `me` variable associated with it, which identifies the thread of execution. As Figure 4.2 illustrates, when a method wants to use a delayed

---

1. The `controlled` keyword of the `Resource` class definition indicates that instances of this object are subject to system administered concurrency control.
result it saves the value of its me variable before doing the leave so that the suspended thread can be identified. The freeResource() method of the example shows the use of the result statement in returning a value for the method that did the leave. The me variable is the object ID of the currently executing Thread object.

Upon executing result, the specified waiting object (i.e. the object enclosed in brackets) receives the result for its method invocation, and its execution thread is resumed. This implies that no statements in a method which follow leave are executed. In the invoked object, control is not returned to the statement following the leave but to the caller, exactly as if a return had been done. The thread of control responsible for completing the delayed result also continues to execute, thus executing in parallel with the original thread.

4.3 User-Based Parallelism

For some objects it is not possible to construct an operation that can respond with a early result. Additionally, relying upon one object to provide another object's parallelism is not in keeping
with the object-oriented design philosophy of hiding the implementation details of a method from the user of the method. Furthermore, it is risky to trust parallelism to another object, since a change in the implementation technique of the methods for that object could eliminate or reduce the amount of parallelism within the system. To combat this problem, it is essential that the invoker of an operation have the ability to specify whether or not the invocation of a method should be done in parallel with the object’s own execution. Certificates and companions address this problem.

4.3.1 Certificates and Companions

A companion is a mechanism to generate one or more threads of execution running in parallel with each other and with the initial thread of execution. Specifying a companion consists of providing a set (block) of method invocations (requests) to be run in parallel with themselves and the initiating thread. Each time a companion is created, a corresponding Certificate object is generated and is obtainable by the programmer. A Certificate is itself an object. A Certificate object identifies a companion and contains status information about the companion and responds to methods that can be used to monitor the companion or to wait for its completion.

The only statements that may appear in the companion are method invocations (Figure 4.3). For the purpose of determining the arguments to the method invocations, the initiator’s execution is suspended until all the parameters for the methods have been evaluated. This ensures that the variables used in the evaluation are not unexpectedly changed by the initiator. Like C, where the order of parameter evaluation for a function is undefined, the evaluation order of a method’s parameters is also undefined. To aid the understanding of how companions work and are organized, consider a part of an application that has as its goal the displaying of the current time on two workstations through the displayTime() method. From the application’s perspective, the displaying of the two times can be done in any order, or in parallel and the application does not need to wait for the completion of the displayTime() method. Figure 4.3 shows three possible ways to do this. The example starts by
CHAPTER 4: Support for Parallel and Distributed Computation in Raven

FIGURE 4.3 Examples of certificates and companions.

defining an instance variable of type Certificate. The first technique performs a standard invoke
on the workstation objects: there is no parallelism and the application waits for each method to
complete before proceeding. The second approach encloses the method invocations within the
companion delimiters of ![ and !]. Taken together, all the methods between the companion
delimiters form the companion. The methods in the companion run in parallel with one another,
and in parallel with the thread of control that created the companion. Each method within a
companion is executed through an instance of the class CompanionThread. (A CompanionThread
is a subclass of Thread.) When a companion is created, an instance of Certificate is created and
returned to the application. Methods can then be invoked on Certificate objects to manage and
monitor the execution of the companion. The CompanionThreads of a created companion are all
initially suspended. To start the execution of the CompanionThreads, the start() method
must be invoked on the Certificate returned as part of the companion creation. This can be done
as soon as the companion is created, as is shown in the second scenario, or the resulting Certif-
icate can be saved and the CompanionThreads started later, as is shown in the final part of the
example. It should be noted that even if the methods within a companion are targeted at the

```java
cert : Certificate; // Variable declaration
res1 = workstation1.displayTime();
res2 = workstation2.displayTime();
cert = !{ res1 = workstation1.displayTime(),
    res2 = workstation2.displayTime() }!.start();
cert = !{ res1 = workstation1.displayTime(),
    res2 = workstation2.displayTime() }!
    cert.start();
```


same object, the methods still run in parallel. However, the target object may impose its own ordering on the method execution through locking or the delayed result facility.

As indicated earlier, each Certificate contains status information and responds to methods that can provide information about the associated companion. A tag can also be associated with each companion. The tag is more than an identifier for a companion, since it can be used to associate any type of user data with a companion. It is easier to classify and associate data with a Certificate through the tag at invocation time than it is to build special data structures to reverse-engineer the information when the method completes.

The application-level operations supported on a Certificate are:

- `setTag(value: cap)`. This function sets the tag value.
- `getTag()`: cap. This function returns the current tag value.
- `start()`: Certificate. Causes the CompanionThreads associated with this companion to begin execution.
- `wait()`. Causes the current thread of execution to block until the companion completes. A companion completes when all the CompanionThreads associated with the companion have completed execution.
- `status`. This accesses Certificate's public variable, status, and can be used to determine the current status of a companion. A companion can be: idle and waiting for execution of the CompanionThreads to begin, running, or finished.

Given a Certificate, it is possible to get and set the tag value, get the execution status of the companion, or wait for the companion to complete. A companion completes when all the methods forming part of the companion have completed. It is important to know when a companion is finished since only then can a user be assured that all the results returned by the methods in the companion are available.

To further extend the usefulness of companions, and their associated Certificates, Certificates can be collected together in the class CertificateGroup. An instance of class CertificateGroup acts as a grouping or collection mechanism and is similar to a set. An important operation
that can be performed on a CertificateGroup is iteration. Iteration on CertificateGroup objects is not like conventional iteration on a set. When iterating, each time a companion forming part of the CertificateGroup completes, control is returned to the iterator.

To use CertificateGroup an instance of CertificateGroup is created. Certificates from created companions are then registered into the CertificateGroup. Before an iteration is performed setWait() is invoked to setup the CertificateGroup for iteration. This initialization call adds flexibility to the use of CertificateGroups by making it possible to iterate over a CertificateGroup multiple times. After the initializer has been invoked, the method waitForNextCertificate() can be used to collect individual certificates. The semantics of the iterator are:

- Unless an explicit break is done out of the iterator loop, each certificate is selected exactly once.
- A certificate can be selected only if the companion it is associated with has completed. When there is no completed certificate, the invocation of waitForNextCertificate() will block, waiting for a companion to complete.
- The waitForNextCertificate() method will return immediately if any companions in the CertificateGroup being waited on are finished.
- When no more Certificates can be returned, a reference to the predefined object, nil, is returned.

In summary, the methods supported on instances of CertificateGroups are:

- waitForNextCertificate(): Certificate. Wait for the next companion in the CertificateGroup to complete and return the associated Certificate.
- setWait(). Set the CertificateGroup state such that, for iteration purposes, no Certificates have been returned yet.

To develop a better understanding of how Certificates, CertificateGroups and companions work, consider an application based on data collection. The example consists of a collection of
CHAPTER 4: Support for Parallel and Distributed Computation In Raven

observation stations on the west coast of Canada. When an event occurs, the central site is notified, and it records the information and displays the time of the event on the screen of the monitoring station. To keep the example simple, assume that there are only three monitoring stations. Figure 4.4 shows the Raven code and usage of Certificates and CertificateGroups.

cert_group : CertificateGroup;
station1_cert, station2_cert, station3_cert, res: Certificate;
cert_group = CertificateGroup.new();

station1_cert = {station1.waitForEvent()}.startO; cert_group.add(station1_cert);
station2_cert = {station2.waitForEvent()}.startO; cert_group.add(station2_cert);
station3_cert = {station3.waitForEvent()}.startO; cert_group.add(station3_cert);
cert_group.setWaitO;

while((res = cert_group.WaitForNextCertificateO)nil){
  if (res == station1_cert) station1.displayTimeO;
  if (res == station2_cert) station2.displayTimeO;
  if (res == station3_cert) station3.displayTimeO;
}

FIGURE 4.4 Example use of CertificateGroups.

needed to accomplish this task. In this example, after the initial declarations and allocation of variables, three companions are created, started, and added to a CertificateGroup. A call is then made to setWait() to prepare for iteration. Iteration over the CertificateGroup is accomplished with calls to waitForNext(). The iteration terminates when nil is returned. Within the body of the while loop, the returned Certificate is checked against the recorded Certificates so that the appropriate display can be updated. The proposed solution has the undesirable property that as more stations are added it does not scale well. This problem can be overcome by tagging Certificates. The use of tags is shown in Figure 4.5. In this new solution the ability to tag a Certificate has been used to simplify the body of the while loop and to make it easier to scale the solution. In Figure 4.5, the Certificate for station 1 is assigned its tag
CERTIFICATE GROUPS AND TAGS

through the `setTag()` method. The tag value is the object ID of the station. The other `Certificates` acquire their tag by using the delimiters, { and }, to immediately tag the `Certificate` with the station ID. Using the tag value makes it a straightforward matter to update the time on a display by retrieving the stored tag and invoking the appropriate method on the tag.

Together, early result, delayed result, and companions and certificates provide a set of tools for generating, controlling and monitoring parallelism. These facilities, however, introduce minimal changes into the Raven language and isolate parallelism to the method interface. One shortcoming of companions and `Certificates` is their inability to impose some serial ordering on groups of requests that can be run in parallel. To address this problem a new class, `InvokeStream`, is introduced into Raven.

4.3.2 InvokeStreams

In some circumstances, an application that has both a parallel and a serial component is forced to execute serially because there is no way to specify the serial relationship while allowing parallelism. The example code of Figure 4.6 illustrates such a situation. It consists of a series of
requests that update a workstation display in a particular order. More specifically, the `positionCursor()` method is first used to position the cursor and then the data is written out. The positioning and displaying operations exhibit a tight coupling. For the display to look correct, the operations must be performed in the order they are coded. Although the positioning and updating operations cannot be allowed to proceed in parallel with one another, there is no reason why they cannot proceed in parallel with the main thread of control initiating the requests.

Restrictions on the ability to use parallelism in this configuration or across companions result because Raven makes no statement about the execution order of methods in a companion. Two parallel invocations targeted at the same object are not guaranteed to execute in any particular order. If the target object is remote, the vagaries of network connections can easily result in out-of-order execution relative to the coded order. In a parallel machine, the actual scheduling order and competition for local resources could also result in out-of-order execution. In the initial Raven implementation, if order was important the application had to perform the operations sequentially and pay the penalty in the form of decreased parallelism. Some parallelism could be gained by encapsulating the serial component of the requests into a self-contained method and then executing it. Although this would work for a single block of requests, it does not work for multiple blocks of requests executed at various times. To allow an application to extract some parallelism from these types of invocations, while maintaining the required serial characteristics of the invocation requests, a new class, `InvokeStream`, is introduced.
The functionality and behavior of an InvokeStream can be characterized by comparing a parallel invocation to a packet in a reliable datagram service. In such a system the requests eventually arrive at the destination object, but nothing can be said about the order. Continuing with this communications analogy, what we would like to do is establish the invocation equivalent of a TCP/IP stream to an object. The effect would be that the invocation requests would behave strictly sequentially and arrive for execution at the destination object in the order specified by the requesting site, as if they were packets forming a stream. Such a system allows for parallelism between the initiating thread and the invocation requests, while maintaining the inter-invocation sequential constraint.

There are several ways that modifications could have been made to the Raven environment to support InvokeStreams. One approach would be to introduce additional syntax into the base Raven language to differentiate between companions that operate in the way described earlier, and companions that operate with a sequential relationship. This would result in Raven supporting two almost identical language level constructs. To be useful, the constructs to support sequencing would have to support multiple invoke streams, for if only one central invocation stream existed, there would be the potential to introduce a sequential relationship between invocations where none exists or was intended. Adding support for multiple streams would require additional changes to the Raven syntax and base system to provide a mechanism to create and manage the multiple alternate invocation streams.

The approach of modifying the compiler to support invocation streams is extreme, and lacks flexibility, so alternate techniques were explored. Since Raven is an object-oriented system, the decision was made to explore ways to make additions to the Raven class system to support invocation streams. By introducing a new class to handle this problem we were able to leverage off existing Raven classes and provide a flexible and extendable mechanism to explore and experiment with invocation streams.

Ultimately it was decided that the desired level of support for invocation streams could be achieved by adding the new class InvokeStream. This class has a single application-visible
method. The method is `push()`, which takes as its sole argument the `Certificate` of the companion to be streamed. When a `Certificate` is pushed, sequencing information is associated with each `CompanionThread` forming part of the pushed companion. Within a companion the methods are sequenced based on the their syntactic order and the target object of the invocation. Requests targeted for the same object are sequenced relative to each other but not to requests targeted at different objects. Once the sequencing information is attached, the `CompanionThread` is started with the Raven system enforcing the sequencing (see Section 5.4.6).

An example of `InvokeStream` usage is shown in Figure 4.7. In this example the

```java
var stream : InvokeStream; /* Declare an Invocation stream */
stream = InvokeStream.new(); /* Create the actual stream */

bar = l{ workstation.positionCursor(cursor_position1)}; /* CompanionThread */
stream.push(bar); /* Pass the Companion to the invoke stream */
stream.push(l{ workstation.displayData(data1, amount1) });
```

**FIGURE 4.7** Example stream creation and use.

An `InvokeStream` object is first created and a reference to the stream obtained. This causes a `CompanionThread` to be created and associated with the method to position a cursor on the workstation display. Next, the `Certificate` generated by this operation is pushed into the `InvokeStream`. The second push to stream pushes the `Certificate` for the companion responsible for displaying the actual data into the `InvokeStream`. This second push illustrates that there is no need for the application to actually acquire the `Certificate` identifying a companion and that the `Certificate` can be pushed directly into the stream. The requests to position the cursor and then display some data on the screen will be sequenced and both the requests will be run in parallel with the initiating thread.
A side effect of implementing *InvokeStreams* as a class results because an instance of an *InvokeStream* is itself an object. This means that it has all the properties of an object and can be used like any object. In particular, a reference to an instance of *InvokeStream* can be passed, through the normal Raven mechanisms, to another Raven object in potentially another thread and on a different processor. More concretely this means that one object can push some invocations into a stream and then pass the stream to another object. That object can then push its own invocation requests into the stream. The result is that *InvokeStream* objects allow requests emanating from different threads of control, and potentially different objects and separate processors, to be streamed and ordered regardless of where the invoking object resides.

The example in Figure 4.7 could still make use of streams even if other threads of execution were involved. For example, the main thread would create the stream and then push the positioning request into the stream. It might then do an invoke on another object, passing the stream ID and the workstation object ID as parameters. The target object can then perform a computation and push the display request into the supplied stream, and the display request will be sequenced.

Since the stream can be passed, multiple objects and potentially multiple threads can have references to the stream at once. Consequently, two *push()* operations can be started simultaneously. Under these circumstances the *push()* operations are processed in strict time order as received at the *InvokeStream* object. In this implementation the result is that the *InvokeStream* will ensure that requests coming from the same thread of control are sequenced for a particular object, but these requests may be intermingled with invocation requests from other threads. If some sort of additional sequencing is required between threads using the same *InvokeStream*, then the onus is on the application to perform the coordination.

Using *InvokeStreams* in a role similar to an advisory file lock in other systems makes it possible to provide a mutual exclusion service to an object. Such a facility is useful when multiple threads of control are trying to access an object that was not designed to support concurrent access. To use a programming convention to establish mutual exclusion, an *InvokeStream*
would be created and then all requests to an object would be issued by pushing a companion containing the request(s) into the stream. Multiple references to the stream can be passed around, and the stream will ensure that the invocations targeted at the object will be executed serially regardless of what thread of execution pushes the companion into the stream. Even though the requests will be executed serially, the underlying system achieves some parallelism at the protocol level by allowing the transmission of requests to occur in parallel.

*InvokeStreams* are not one-way operations that sequence only requests to objects; they also sequence the return results. The individual methods within a companion can return results. When a companion is pushed into an *InvokeStream* the return results are sequenced. The programmer can be assured that if the result of a parallel execution request targeted at a particular object has returned, so too have all the other sequenced requests targeted at the object that used the same stream.

*InvokeStreams* are an ideal platform that can be used as the basis for implementing other invocation sequencing operations. For example, *InvokeStream* could be extended to provide ordering amongst different target objects. Further functionality could be introduced by providing a lock method to guarantee absolute access to an *InvokeStream* when more than one reference to it exists. This latter service might be particularly useful when the stream is heavily used and the thread requires exclusive access to the *InvokeStream*. This additional functionality can be provided by creating new subclasses that inherit from *InvokeStream* or by adding methods to the existing *InvokeStream* class. Flexibility like this would have been hard to achieve if support for *InvokeStreams* had been relegated to the compiler.

### 4.4 Properties

The exercise of writing a program has several distinct aspects to it. The programmer decides upon a set of algorithms to use in solving the problem and how they need to be integrated. Once the basic design decisions have been made they are implemented by selecting and defining the data structures that will be used and the functions and procedures to operate on that
data. In the object-oriented domain, data structures and procedures translate into class definitions and their methods; however, defining classes is somewhat different from specifying data structures since the programmer can often use existing class libraries or develop classes that inherit from the predefined classes. The ability to use classes and subclasses allows programmers to benefit from the work of other programmers.

In an object-oriented system, when the conventional notion of system services, like support for parallelism or atomic transactions, is moved into the language, some accommodation is required to make these features easier to use. To illustrate this point, consider an object-oriented language that supports parallelism and provides the class List as part of a class library. There are two distinct environments that the class List could be operating in—the sequential environment, and the parallel environment. These two environments are different and place different constraints on objects and how they are used. In particular, in the concurrent environments, instances of List are expected to handle concurrent accesses in a reasonable manner.

One approach to managing the concurrency problem is to write the List class for the worst case concurrent access scenario. In doing this, system services would be used to restrict concurrency and supply mutual exclusion for critical sections. This, however, introduces unneeded overhead when the object is used in a sequential environment where concurrency control features are not required. Additionally, an object can be used in different ways and in different environments than those anticipated by the designer. To fully support reuse, the programmer must anticipate all the ways that an object of the class will be used, and program the class accordingly. To illustrate the difficulties this presents, suppose that in addition to supporting concurrency control, the system also supplied support for transactions. With the model being described, the programmer would be required to program the object for usage in a concurrent environment, an environment that supports transactions, an environment that supports both transactions and concurrent access, and an environment that uses neither. Additionally, some mechanism would also be required to indicate which level of support to use. The result of this
is that as the number of system services available to the application increases so does the complexity of the programming task.

Requiring the programmer to develop a class that can be used in all the different environments does not make programming a class easier. Code reuse is reduced since the class, if fully programmed, introduces unneeded overhead and complexity. The increased complexity also makes it more difficult to ascertain whether or not a particular class will operate correctly in a selected environment.

Another way to view the problem associated with providing system services to objects is to observe that services like concurrency control can be managed by the system instead of the programmer. To accomplish this, the information that a particular object is going to be used concurrently must be propagated to the system, so that the appropriate actions and controls can be placed on the object.

A way to incrementally build new classes is by starting with a basic implementation and then using class inheritance through subclasses to add the required features [93]. If this were done with the List class, the system might consist of a basic List class designed to operate in a sequential environment with another class designed to provide concurrency control, ConcurrentControl. Using the class inheritance approach, if List were required to operate in a concurrent environment, a new class ConcurrentList, which inherits from List, would be created. The implementation of ConcurrentList would deal with all the issues associated with concurrent access to a list by making invocations on an instance of the concurrency control object. However, as other operating system services are supported they would require their own special classes, and as the number of available operating system services increases the permutations on how the services can be combined increases rapidly and a class explosion results. Multiple inheritance would not solve the class explosion problems. Specific classes still have to be declared for each of the various ways the system services can be combined. With the list example, a ConcurrentList class would still have to be declared. It would inherit from List and Con-
currentControl. This would make the implementation easier than in systems with single inheritance, but it does not solve the class explosion problem.

Another way to address the class explosion problem is to use reflection [61]. In a reflective system an object is essentially split into two components. There is the object the user sees and an associated meta-object that governs the underlying, or system level, semantics of the object. The meta-object is an object in its own right. Reflective systems have particular problems associated with efficient implementations [108][109] since all the meta-level behavior of an object is subject to change. These changes can take place at any time during an object’s execution. The result is that any execution of an object requires constant checking to determine if any of the associated meta-objects have changed. There are also problems relating to determining the scope of any changes to meta-objects. For example, should the specified change apply to just one object, all objects in the system, or simply to objects of a given type? Another important consideration about reflection is the side affects that result when a meta-object is changed. If care is not taken changes to meta-objects could affect the behavior of other meta-objects, thereby making it difficult to reason about the behavior and relationships between objects. Class hierarchical systems and reflective systems do not adequately address the class explosion problem; however, Raven does address the problem through the introduction of properties.

Properties in Raven are designed to curb the class explosion and provide a mechanism to allow programmers to extend a class’s characteristics on an object-by-object basis. The properties added to an object do not affect the outward interface of the object and the way the object is used. A property is a way of changing the runtime support for a particular object from the default support for that class. In essence, a property is a mechanism used to inform the system about the basic services that an object requires.

To illustrate this point, consider the programming of the classes List and ConcurrentList. Except for the case when there is concurrent execution, these classes are identical. They have the same method interface and method description, and the same expected effect during execution. The only difference is that when operating in a concurrent environment the concurrent
version should protect the object instance data from concurrent accesses that could be harmful. By associating a property to an object, and having the property managed by the system, it is possible to program one List class, yet have it operate correctly in both sequential and concurrent environments. This works because once the system knows that an object requires a certain service, that service can be provided for the object automatically at method invocation time.

In Raven, each object is assigned a set of properties that define how the object is managed by the system. A property can be specified as part of a class definition. When a property is specified as part of the class definition, the property is always associated with instances of that class and the property cannot be overridden by either attaching the property to a subclass or explicitly at object creation time. This feature is used in the case when the class has certain expectations about how it is going to be used. For example, a class may be marked as controlled because it is only meaningful to use the class in a multi-threaded environment. If the class definition of an object does not specify a particular property it can be assigned at object creation time.

A second way for an object to acquire a property is to assign the property at object creation time. An object is normally created by invoking new() on the desired class object. When an object is to be created with additional properties, \texttt{pnew()} method is invoked instead. The first parameter of \texttt{pnew()} specifies the set of additional properties the created object is to have. The remainder of the parameters are exactly the same as \texttt{new()}. This feature means that all instances of a class are not required to have the same set of properties.

In deciding upon the original set of properties, the goal was to specify a set of mutually orthogonal properties that could provide a number of operating system services to the application. The initial set of properties dealt with object storage, concurrency control, object security, object recovery, and distribution. Based on this initial set of properties, others working on the Raven system explored the possibility of supporting such services as atomic transactions through properties and the dynamic runtime relationship between objects [41]. The resulting set of properties are:
• **Controlled.** This property protects an object against concurrent accesses by multiple threads which may modify its instance data. Multiple readers, but only a single writer are supported. Threads that cannot be granted the access they require are blocked until the request can be satisfied. The Raven compiler tags each method as either a read or write method based on an analysis of the statements within the method.

• **Immobile.** Objects which implement machine-specific tasks, such as device drivers, or those which need to remain on the machine where they are started, are given the property immobile. This means the object remains on the machine where it was started and is not eligible for migration.

• **Recoverable.** The recoverable property is given to those objects that only want the changes made to an object’s instance data to be maintained if the method terminates normally. If the method terminates abnormally, through the use of Raven’s `abort` statement, the instance variables are reset to the values they had before execution of the method was started.

• **Persistent.** Normally objects only exist in RAM. An object with the persistent property has a copy of that object maintained on disk. Since persistent objects are only written to disk periodically, there are no assurances that the disk copy of an object will match the RAM copy at the time of a system failure, but the stored copy will be a complete consistent copy of the object.

• **Durable.** The Durable property is similar to the persistent property except that control is returned to the invoking object only after the object marked as persistent has been updated on disk.

• **Replicated:** Under certain conditions, for example as a means of improving efficiency, it may be desirable for the system to maintain more than one copy of an object. Objects given the replicated property are candidates for duplication and distribution to multiple machines. Replicated objects are only weakly consistent and the individual replicas may not always have the same state.

• **Immutable:** Objects identified as immutable cannot have their instance data changed. Attempts to invoke a method that modifies an immutable object’s instance data will result in a runtime error.

Each of these properties is supported by the system independently of the others. Any combination of these properties can be selected and assigned to an object of any class. Note, however, that properties are always stated in the affirmative. Unless an object is given a specific property it does not have it, a property once given cannot be taken away. For example, if a class is specified as having the controlled property, then its instances will always have the controlled...
property. There is no way that a subclass can revoke the controlled property, and similarly there is no way for \texttt{pnew()} to take the property away. This approach was taken based on the assumption that if the designer of a class assigned a set of properties to that class, then those properties were required for the class's correct operation. It should be noted that there is no requirement for objects to have properties assigned. Objects without properties are said to be BASIC objects. BASIC objects are managed in the following way:

- The object is maintained only in RAM and therefore does not survive across system failures.
- The object is not controlled. This means that multiple threads of control can access an object simultaneously and that no concurrency control is exercised.
- An object is eligible for migration.
- All changes to an object's instance data happen immediately and the previous state of the object cannot be recovered.

For the majority of programming problems these basic characteristics are acceptable and there is no need to augment objects with additional properties. Several examples of creating objects with and without additional properties are shown in Figure 4.8. In the example, an instance of \texttt{List} is first created with the basic properties. The next line shows the creation of a \texttt{List} with the controlled property, because the object will be accessed concurrently. In the third line a durable \texttt{List} is created, since a copy of the object needs to be stored across runs of the application. The final line demonstrates how an object with the durable, controlled, and recov-

```plaintext
list = List.new();
list = List.pnew(CONTROLLED);
list = List.pnew(DURABLE);
list = List.pnew(DURABLE&CONTROLLED&RECOVERABLE);
```

FIGURE 4.8 Creating objects with different properties.
erable properties can be created. In all these cases, the List class is the same and the user of the resulting instance will manipulate the list in the same way. No new methods result when objects are created with properties. The objects differ only in the types of services the system automatically supplies for each object.

Since most of the objects in a Raven system are basic objects, the system is designed to deal with this case as efficiently as possible. This is accomplished by associating different method lookup and dispatch routines with objects (see Section 3.2.4). Objects with the basic properties are associated with dispatch routines that take advantage of the knowledge that the object is basic. Objects with properties are assigned different dispatch routines. These dispatch routines support properties by executing pre- and post-invocation functions to supply the required service. Having different dispatch routines associated with objects means that only objects with properties pay the extra execution overhead of dealing with the properties.

4.5 Object Concurrency Control

With the Raven facilities described in the previous sections, it becomes readily apparent that a single object can have multiple method invocations targeted at it simultaneously, and that properties provide the mechanism to tell the system to supply concurrency control for an object. The method invocations that result in concurrent access can be derived from other threads of execution on the same machine or from threads of execution on other machines. In dealing with the issues of concurrency control, any solution must consider the following points:

- How can concurrency be maximized?
- How are calls across machine boundaries handled?
- What happens when a controlled object invokes on itself?
- How are deadlocks dealt with?

Figure 4.9 shows two objects concurrently accessing a shared object. In this example both objects A and B are executing in parallel and are performing method invocations on a third
object, C. Depending upon the method invocations and the nature of the object, there are a variety of ways in which the access to the object could be handled.

The simplest solution to avoiding the possibility of object A or B seeing object C in an inconsistent state is for the object to permit only one method to be active within it at a time. This is the approach taken in POOL-T [12][11] and ACTRA [103], two other object-oriented systems which support concurrency. Serializing requests through the object keeps the instance data consistent, but at the price of eliminating concurrency through the object. By forcing an object to be accessed serially, even when it is not required, a needless bottleneck could be introduced into the system. Concurrency through the object can be increased by permitting multiple
readers of object instance but only a single writer of object instance data to be active in an object. This is the approach taken in Raven.

In Raven methods which only read an object’s instance data are classified as read methods, and those which modify an object’s instance data are write methods. At compile time each method is classified as a read or write method based on whether or not it modifies the object’s instance data. This information is then used by the compiler to generate code to acquire the appropriate lock at method invocation time and to release the lock when the method completes. By adopting this approach, Raven ensures that each method acquires and releases its locks at the appropriate time, thus relieving the programmer of performing this task. This simplifies the task of converting sequential programs to ones that may be used in a parallel or distributed environment.

4.5.1 Locking Costs

Although the relatively conservative locking scheme described above does help ensure the integrity of an object’s instance data, it does impose a cost. For the ease of use of Raven’s locking facilities the programmer pays an execution time cost in both the amount of time an object can have multiple methods running in it and in the extra code that must be run to acquire a lock.

In the current implementation of the Raven runtime system, the association of a lock with an object results in a significant overhead in the acquisition and freeing of locks. Most of the cost can be attributed to the effort required to ascertain whether or not the lock can actually be granted and to recording the fact that a lock is granted or must be waited for. When the lock is released, this information must also be recorded and a check done to see if any method invocations waiting for the lock can now be started.

Since locking substantially increases the fixed overhead of performing a method invocation, objects should use locks only when they are needed. For example, applications which are only single-threaded do not need to lock their objects. The majority of the objects in a Raven environment do not require locks. An object has concurrency control applied to it by associat-
ing the \textit{controlled} property with the object. This is either done at class definition time or with \texttt{pnew()} when the object is created. Making an object controlled causes the Raven runtime system to automatically generate a lock to control the access to an object's instance data. When a method is invoked on an object, the object's associated lock is acquired, and upon method termination the lock is released.

An uncontrolled object has no lock associated with it. The expectation is that either concurrent access to the object is not an issue or that the programmer is going to use some other mechanism to maintain the required level of consistency for the object's instance data.

\subsection*{4.5.2 Locking Duration}

A cost of using a lock can be characterized as any activity of the lock management system that increases the execution time. As the previous section described, there is some fixed system administrative overhead associated with using locks. Another cost, the length of time an object lock is held, greatly affects the amount of concurrency that can be extracted from that object. If other threads of control need access to a locked object, their execution is delayed. Additionally, methods hold their locks for the duration of their execution. It is likely that the locks are held longer than actually needed, and to maximize the concurrent accesses to an object, the amount of time locks are held should be minimized. The automatic locking of Raven minimizes the programming effort required to keep the instance data consistent at the expense of object concurrency.

One way to increase concurrency is to use an uncontrolled object and to have the programmer of the class manage the concurrency manually, using, for example, semaphores. This provides the opportunity for the class programmer to minimize the exclusive-use regions in the class. Although such a solution is always available to the programmer, it does not prevent the user of an instance of the class from creating a controlled object. If this happens, the running of the application is slowed because concurrency control is applied twice. Furthermore, there is the increased potential that the two concurrency control methods will conflict and cause a
deadlock. To avoid this problem, the facilities for tailoring concurrency control must communicate with the lock managing subsystem. Two approaches are used to do this. One is to allow the class programmer to specify the exact lock type a method is to use, and the other is to provide locking statements to control the locking from within a method.

In some circumstances, the classification of a method as a read method or a write method may not be the best choice, or the method itself may not need locking. To override the compiler's classification of a method, the class programmer may tag a method as no lock, meaning that the method does not need a lock, or as write lock to indicate that the method should acquire a write lock. With these two method modifiers, the default system locking can be overridden.

All the locking control discussed to this point has been applied for the duration of a method. If finer locking control is desired, then the ability to manage locks within a method is required. In essence what is desired is the ability to manage the locking within a region. Accordingly, the statements lock and unlock combined with their read and write modifiers are available to allow the programmer to specify a locking region within a method. This affords the programmer the opportunity to exercise a fine degree of control over the locking in an effort to maximize parallelism within an object.

In all cases, the lock and unlock statements work only on a controlled object. If the object is uncontrolled these statements have no effect. The lock statements work in the following manner:

- **read lock**: If the object's lock is not held, then a read lock is acquired. If the lock is currently held as a write lock, then it is demoted to a read lock. A read lock will remain as a read lock.

- **write lock**: If the object's lock is not currently held, then a write lock is acquired. In the case when the currently held lock is a read lock, this statement results in the promotion of the read lock to a write lock.

- **unlock**: If a lock has been acquired by the programmer, then it is released. An implicit unlock of user acquired locks is performed when a method completes.
These statements, combined with the ability to specify the locks on a method-by-method basis, provide the programmer with a set of tools that can be used to develop a large range of locking strategies. The programmer is free to manually manage the locking in one method while still letting the system manage the locking in the other methods. However, the implicit locking provided by the system is probably sufficient for the majority of applications. This locking flexibility is possible only because the lock management system is treated as a complete unit and integrated into the Raven system.

4.5.3 Locking problems

Locking of instance data needs to be more than demand-based, since a purely demand-based system does not take into account the various sequences of events that could lead to a lock being requested. For example, it is common for a method to invoke another method on itself. With a strictly demand-based locking scheme this will create problems, since the resulting lock request will be viewed as another request for a lock. Suppose that the initial method was a write method and the self invocation was a read method: by the locking rules outlined previously, the self invocation would block, waiting for the write method to complete before gaining access to the object, and, since the write method isn’t going to release the lock until it is completed, a deadlock occurs. The problem is also exacerbated by the fact that the second invocation on the object may actually come through an intermediate object, potentially resident on another machine. These types of invocations are referred to as recursive invocations, and Figure 4.10 shows such a legitimate Raven execution. In this example execution starts in object A, which then invokes on object B. Object B, which may be on a different machine, then invokes back on A. Assuming that the initial invocation on A was a write method, something must be done when B invokes back on A if a deadlock is to be prevented. One possibility is to require the programmer to explicitly relinquish the object’s lock before invoking on B and then reacquire it when the method on B completes. This has two disadvantages: one is that the programmer must now deal explicitly with the lock management issues, and secondly the programmer must be aware of possible side effects or unexpected method invocations from other objects which
FIGURE 4.10 Object invoking back on self

could modify the instance data while executing in B. The programming burden could be relieved if the compiler automatically released and reacquired the locks, but the problem of dealing with unexpected changes in the instance data would remain.

Since multiple readers and only a single writer are supported in the Raven locking model, a problem with recursive object invocations exists only when a recursive invocations involves a write lock currently held or being requested. To permit this invocation sequence, Raven assigns a globally unique session ID to each thread of execution. The ID is used to track chains of invocations locally and across machine boundaries. An invocation chain is the Raven equivalent of the call stack in programming languages like C, with the addition that an invocation chain may also cross machine boundaries. The purpose of the session ID is to make it possible to determine when a method invocation attempts to acquire a lock on an object locked earlier in the invocation chain, even if that invocation chain crosses machine boundaries. In particular, a write lock will be granted if no other locks for the object are granted or if all the locks granted
belong to the same session as the requestor. Similarly a read lock will be granted with outstanding write locks on the object if all the locks belong to the same session as the read request.

Pictorially, Figure 4.11 shows a sequence of method invocations and the way that the locks are managed. In this figure, each column represents an object and the locks that the object has granted. Locks closest to the object are the most recently granted ones. When a method invocation is made, the method presents its session ID to the object. A check is then made in the list of granted locks to determine whether or not this lock can be granted. If the lock is granted, it is added to this object’s list of granted locks; otherwise it is added to list of requests blocked and waiting for a lock. In the example, Object 2 has blocked Session 2, since Session 1 currently holds a write lock on Object 2. Session 2, however, was able to obtain a read lock on Object 3, since that would not conflict with the read lock previously granted to Session 1.
In the scenario depicted in Figure 4.11, Session 1 starts by requesting a read lock on Object 1, which is granted. The session then requests read locks on Object 2 followed by Object 3, and both are granted. From Object 3, Session 1 requests a write lock on Object 2, and it is granted, since Session 1 is the only holder of locks on Object 2. If other sessions had read locks on Object 2, then Session 1 would have been blocked in its attempt to get the write lock until all other sessions had released their locks on Object 2. From Object 2, Session 1 makes a request for a read lock on Object 1 and it is granted. Session 1 makes a request for a write lock on Object 1, and it is granted, since no other sessions hold locks for the object.

It should also be noted that in this example the invocation requests, and resulting lock requests, span machine boundaries. Using the session ID presented to an object at the time of the method request, it becomes a simple operation to check the object’s list of granted locks to determine whether or not an already granted lock belongs to the session making the new request.

The approach to locking described above provides a mechanism to deal with recursive method invocations and with the tracking of chains of invocations across machine boundaries. The locking scheme, however, needs further refinement to deal with the early and delayed result primitives of Raven. Companions do not require any new support since they are new execution threads with their own unique session ID. This could be illustrated in Figure 4.11, where it is conceivable that Session 2 is a new thread of execution resulting from the execution of a companion in Session 1. Being a new thread, the session has no association with any previous lock acquisitions; therefore it presents no new challenges to the locking scheme.

### 4.5.4 Locking and Early Result

Controlled objects using early or delayed result present a different challenge to locking. When a recursive invocation from a thread performs an early result, a problem occurs if some method in the session chain has a lock on an object while the method doing the early result has a lock on the same object. When the early result is performed, a new thread of execution is started,
and the new thread and the one it is derived from could now be simultaneously accessing the
same object, and they both could be expecting to have a certain level of access to an object (i.e.
read or write access). These access expectations are potentially in conflict. For example, if the
invoking method is a write method, it will expect to be able to update the object with impunity
when it gets control back, since it has a write lock. Similarly, the thread resulting from the early
result could expect the object instance data to be unchanging if it is a read method, or change-
able if it is a write method. This would be in conflict with the access the invoker expects to have
when it gets control back. Given these possible conflicts, an early result must reconcile the
locking desires of the locks held before the method invocation with those of the method doing
the early result.

Fortunately, the semantics of an early result are such that the invoker of a method cannot
tell whether the method runs, returns a result, and continues executing, or runs and returns the
result upon method completion. (Implicit in this is the assumption that the method doing the
early result eventually returns and does not enter an infinite loop.) The Raven system exploits
this property to reconcile the locking issues when an early result is done in a controlled object.

As described previously, an early result could result in two threads of execution both
expecting to modify the same instance data. To eliminate this possible conflict, a result is
returned and a new thread of execution created only if no other locks are held on the object by
the invocation chain, or if all the locks on the object are read locks. When a mixture of read and
write locks are present on the object, no new thread of execution is created and the result of the
early result is returned when the method finishes executing. Although this eliminates the par-
allelism between invoker and invokee, it maintains the proper method invocation semantics.

Raven has adopted the approach that the details associated with the management of par-
allelism should be as transparent as possible. As such, when an invocation is made, the invoker
should not need to know anything about how the method is implemented. In using the above
 technique for early result, it always appears to the invoker that the invoked method has com-
pleted, regardless of whether or not an early result was done and of the type of locks the invoked method acquired.

4.5.5 Locking and Delayed Result

Delayed results also present a challenge to Raven's locking scheme. When the delayed result is done the method terminates, and the instance data lock acquired to enter the method is released. The invoking method, however, remains blocked, holding locks and waiting for another method to complete the delayed result. It is possible that the session could still hold locks on the object the delayed result was done from. Delayed results are typically used within a single object. This means that the method that will ultimately complete the delayed result will need to lock the object. If some other portion of the invocation chain holds a lock on the object, it is likely the lock will conflict with the lock required for completing the delayed result, resulting in deadlock. To avoid this problem, when a delayed result is done, all the locks in the invocation chain associated with the object are temporarily released. This permits invocations from other sessions to acquire locks on the object, with the expectation that one of them will complete the delayed result. When the delayed result is completed, the suspended thread can be restarted, provided all the locks previously held on the object can be reacquired. If the locks cannot be granted the delayed thread will remain suspended until they are.

4.5.6 Deadlock Handling

Deadlock detection, avoidance and prevention are harder to deal with in distributed systems than in single processor systems [100], since resource usage information is scattered over many processors. This scattering of data makes an already hard problem even harder to deal with. The combined execution cost and network overhead associated with running deadlock detection or avoidance algorithms is high. Additionally, if implemented, these schemes would add to the complexity of the Raven system and impose extra overhead on all applications, even those which would not need the facilities to deal with deadlocks. Deadlock detection and avoid-
ance are active research areas beyond the scope of this thesis work they present many interesting problems in their own right [101][21].

It is, however, realized that dealing with deadlocks in a distributed systems is a serious and difficult problem. The implementation of a general solution for dealing with deadlocks would seriously impact the performance of the Raven system. To deal with deadlocks, Raven offers a compromise by providing some tools to help avoid writing programs that have deadlocks in them and by providing a facility to specify how long an application is willing to wait to acquire a lock. Using this approach is not necessarily the best way to handle deadlocks, but it does acknowledge the problem and establish a framework so that Raven can be modified later to use more sophisticated deadlock detection or avoidance algorithms.

The tools Raven provides for dealing with deadlocks rely upon the facilities that make it possible to override the default automatic locking provided by Raven. For example, one way that a deadlock can occur is when method’s acquired is not strict enough. The ability to convert a read lock to a write lock could fend off deadlock. Consider the scenario where a chain of multiple read type invocations on the same object end up doing a write invocation on this object. Since the initial invocations use only read locks, multiple simultaneous threads of execution through an object are possible. Ultimately, if these threads proceed through the object together, a deadlock will occur when the write invoke is done. By explicitly allowing a method to acquire a write lock, even if the current call chain doesn’t immediately need it, it is possible to fend off some inter-thread deadlock scenarios. The other facilities to upgrade locks, release locks and to acquire a lock independent of the automatic locking provided by Raven provides the programmer with the ability to program around deadlock scenarios, although the burden of identifying such scenarios remains the programmer’s responsibility.

Since Raven supports neither deadlock avoidance nor deadlock detection, it is a real possibility that an application using controlled objects could deadlock. From an intuitive perspective, a programmer would probably say that a system is deadlocked if the application has to wait more than some specified time to obtain a resource (lock). How long to wait is application-
specific. For some applications it may be a few seconds, while for others it may be acceptable to wait for several minutes, or even hours. Regardless of the time, the basic premise is that if the lock cannot be acquired in some specified time frame then it is probable that a deadlock has occurred.

Raven, therefore, makes it possible for an application to specify how long it is willing to wait for a lock. When the specified time has elapsed, without the acquisition of the lock, the method `lockAcquisitionTimeout()` is invoked by the system on the object attempting to acquire the lock. The invoker then has the option of determining how the processing should proceed. The application’s action could include an attempt to invoke the method again, backing off and releasing locks, or even aborting execution. Although this approach to dealing with deadlocks is not ideal, it does provide some primitive support to allow the application to do its own deadlock detection.

One particular problem with this approach is the mechanism used to perform the time-out notification. As it is, the object doing the lock acquisition is notified of the lock acquisition failure, and it is not clear whether or not this is the correct object to notify. Perhaps the notification should be delivered to the executing thread instead of the object. The proper way to deal with this is to develop an exception-handling model and mechanism for Raven. Such work is beyond the scope of this thesis.

None of the facilities provided by Raven preclude programmers from implementing their own deadlock avoidance or detection schemes. However, such a system would be operating at the application level and as a result would not have access to the locking information maintained by the system. To avoid problems with the automatic locking provided the system, the user would probably have to disable most of the automatic locking provided by Raven. The locking and deadlock handling facilities of Raven have not been optimized for performance. Therefore, it is expected that by examining the literature on these topics [21][87] substantial improvements in the performance of the locking subsystem could be made.
4.6 System Resource management

One of the jobs of an operating system is to control access to and manage system resources. In a parallel and distributed system, two of the resources that require special attention are memory and processor usage.

4.6.1 Memory

When an object is created in Raven there is an implicit allocation of memory made to hold the object. Once the object is no longer required, this memory should be freed for reuse. One approach to this problem is to equate the creation of an object to that of `malloc()` when programming in UNIX, and then using something akin to `free()` to release the memory. The problem, however, is how to determine when an object is no longer in use so that `free()` can be called. In a system like Raven, that provides support for parallel and distributed computation, the problems are compounded further as there are now multiple threads of control that could be accessing an object, and it becomes difficult for the programmer to reason about when an object can be freed. For example, when a method is invoked through a companion, where does the responsibility for freeing the object lie? The invoker cannot free it because the invokee could be still using it or could have passed it off to another object that is now using it. Likewise, the invokee cannot free the object because it does not know what the invoker did with the object. To properly manage the freeing of an object, all the potential users of the object must coordinate their efforts, and this places an unneeded burden on the programmer. Two well known solutions to this problem are to use reference counting or to use garbage collection. Garbage collection was selected because it required the least amount of new system support. A discussion of garbage collection in Raven can be found in Section 5.2.

4.6.2 Processor Management

Sequential programming languages and environments force the programmer to contort inherently parallel problem solutions into a sequential environment. However, making paralle-
lelism easy to use also introduces problems not present in a sequential environment. One of the problems concerns processor usage. In some circumstances the natural way to extract parallelism would result in unbridled parallelism far beyond the support capabilities of the hardware. To address this problem a throttling mechanism external to the application is introduced to the Raven system.

During the early phases of Raven usage it soon became apparent that certain structures within the sequential environment could be easily parallelized, an example of which is shown in Figure 4.12. In this example, a method invocation inside a \texttt{for} loop is used to update and compute a line on a workstation display. Suppose now that the function to compute and display a line takes a long time, that there is no need to wait for the updating of the display to complete, and that the computations to update and display a line are independent of one another. If these conditions are met then, as shown in Figure 4.13, the conversion of the code to a parallel imple-

\begin{verbatim}
... for (line_no = 1; line_no < MAX_LINE; line_no++)
    workstation1.computeAndDisplayLine(line_no);

FIGURE 4.12 Sequential code to update a display.
\end{verbatim}

\begin{verbatim}
... for (line_no = 1; line_no < MAX_LINE; line_no++)
    !{ workstation1.computeAndDisplayLine(line_no) }.start();

FIGURE 4.13 Conversion of sequential code to parallel version.
\end{verbatim}

mentation is a relatively straightforward and easy exercise. The only problem with this scenario is that it does not take into account the number of processors that can work on a problem. In
many respects that is the way it should be. The application programmer should be free to pro-
gram in a manner that is, as much as possible, independent of the amount of parallelism that
can be supported by the processing environment. Furthermore, in this example, which is a
fairly common way of creating parallelism, making the programmer directly responsible for
constraining the parallelism introduces its own problems. In particular the programmer would
have to explicitly monitor the parallelism with respect to a maximum permissible amount. Hav-
ing to monitor and control the parallelism with respect to this piece of code would greatly
increase its complexity. Additionally, some mechanism would have to be introduced to convey
to the application what the constraints on parallelism are to be. This would further increase the
complexity of the application code.

The approach adopted in Raven is to consider the controlling of parallelism to be a system
configuration issue. The Raven system itself, therefore, needs to have some mechanism to
determine what the acceptable level of parallelism should be and to be able to restrict the par-
allelism to the required level. In essence, Raven needs to be externally tunable, so that various
system resources and limits can be set at execution time when the exact environment is known,
and not when the program is being written and little is known about the execution environment.
Additionally, there could be multiple disparate environments that a single application is going
to run in, and a different version of the application should not be required for each one. For
example, initial testing and debugging of an application might be performed in one environ-
ment before final runs are made in another.

The solution that has been implemented in Raven consists of classifying the various
forms of parallelism possible in a Raven application and then placing limits on them. To
accomplish this, four categories of processes are recognized by the underlying Raven support
system. These categories are:

- System processes (threads) that provide services to the application.
- User level processes (threads) that form the application.
• **Companion threads** spawned by a user process.

• Companion threads spawned by a system process.

The categorization of processes into execution classes is transparent to the programmer. A combination of compiler support and runtime system support automatically performs this task. Although it is conceptually possible to place restrictions on any one of these classes of processes, the initial experiences with Raven indicated that only companion threads spawned by user processes are likely to cause problems. The number of system processes is relatively small and only under rare circumstances do they spawn threads. Likewise, there is usually only one user thread per Raven environment and that is the initial starting thread. If there is more than one user thread, the threads tend to be operating in a client/server type relationship where their number is relatively constant and is not a function of the inputs to the problem being solved. With these observations and experiences, it was decided that, at least initially, limits on the number of execution threads would be applied only to companion threads spawned from a user thread.

In the current implementation, the Raven system reads a value from its environment specifying the maximum number of user companion threads that can exist at once. Just what the value should be is a function of both the application and the execution environment. Among other things, the execution environment includes the number of physical processors and their organization. As an example, in a single processor environment an application might restrict the number of user companion threads to 1, while for the same problem on a 20-processor shared-memory machine it might be 20. Although this example has a one-to-one correspondence between the number of processors and the maximum number of user companion threads, it is not a requirement.

Once the Raven system has determined the maximum number of user companion threads, it uses that value to control the creation of user companion threads. As long as the current number of companion threads stays below or equal to the maximum, everything proceeds normally. If, however, the creation of the thread results in the maximum number of companion threads
being exceeded, then the creation operation is suspended. As user companion threads terminate, the suspended creation operations are allowed to continue. As long as all the running companions do not depend, either directly or indirectly, upon any suspend or unstarted companions for their completion, progress in the computation will be made. Although this scheme is simple, to date it has been a successful approach for applications that derive their parallelism from constructs similar to the approach used in Figure 4.13.

With the introduction of this sort of parallelism control into the system, there is a potential for deadlock that is not present in a sequential system. If one assumes that the application is correctly written, then if there are no restrictions on the number of companion threads the application will not deadlock. A restriction on the amount of parallelism, however, could result in the application deadlocking.

A delayed result is the only way an application thread can suspend its execution indefinitely. For a deadlock to occur because of the restriction on the number of companion threads permitted to start, all the companion threads must be doing a delayed result or suspended trying to create a companion thread. As long as at least one companion thread is executing there is the potential for the delayed results to be completed, or for the executing thread to terminate. If the thread terminates then a new companion thread can be started, and it might be the one to complete the delayed results. Since the restriction on companion threads can only cause a deadlock under these limited circumstances, Raven allows a new companion thread to be started if all the companion threads are suspended doing a delayed result or suspended creating a new companion thread. This companion thread will either allow progress to be made by completing some delayed results, or it will end up suspended too. It is possible that all subsequent companion threads will end up suspended until all Raven’s system resources are consumed. In this case the application prematurely terminates. Prior to this, Raven would have printed warnings indicating that the specified degree of parallelism was being exceeded. It should be noted that the limit is effectively removed when companion threads are created in the situation when all other companion threads are suspended and the limit on the number of companion threads has been
reached. Therefore, if the system deadlocks, it would also have deadlocked if no restrictions had been placed on the number of companion threads that could be started.

4.7 Object Transparency

From a programmer's perspective, the difficulties associated with programming in a shared memory environment are considerably less than those of programming in a distributed memory environment. In a distributed memory environment the programmer must manage many of the tasks done by the memory management system of a shared memory environment. With shared memory the programmer does not have to be concerned about locating data or whether or not particular functions are available at a remote site. In a distributed memory environment, a programmer needs to know how to locate remote data and how to access the functions to manipulate the data. In a distributed environment it is not the case that a simple subroutine call will access the desired information. The problems, however, do not end with locating the data or the functions. In a distributed environment the application has to address the problems related to the differing data representations on different machines, communications failures, the passing of parameters, and the returning of results. Compared to calling a subroutine or accessing data through a pointer, these types of activities are much more complex. To simplify programming in a distributed environment, as many of these impediments as possible should be eliminated, with the goal being transparency.

Transparency is defined by Coulouris [34] as:

The concealment of separation from the user and the application programmer, so that the system is perceived as a whole rather than as a collection of independent components.

When trying to decide if system is transparent, there are multiple forms of transparency that can be considered. Coulouris identifies eight categories of transparency:

1. Access transparency. The procedure for accessing remote and local objects is the same.
2. **Location transparency.** There is no need to know the location of an object to access the object.

3. **Concurrency transparency.** Multiple users can operate on shared data without interfering with each other.

4. **Failure transparency.** The failure of hardware or software is hidden from the user.

5. **Performance transparency.** The system can be reconfigured to adjust to varying loads and if this is done can the user detect it?

6. **Replication transparency.** Multiple copies of objects are maintained to improve performance; this should be transparent to the users.

7. **Migration transparency:** Objects can be moved and relocated within the system, without affecting the execution of applications.

8. **Scaling transparency:** The system and applications can change in size without need to change the system structure and without affecting user applications.

Of these categories of transparency, the first five are the most applicable to Raven, since the remaining three issues are not addressed in the current Raven system. In general terms, if an environment supports transparency, then it should not be possible to determine whether a particular function is being performed remotely or locally. Transparency should extend to both the user and the provider of the service. Some of the issues [101] to consider are:

1. Does the programmer have to know which library procedures are local and which are remote? (This can be either the application programmer or the library programmer.)

2. Can procedures be written without regard to knowledge about whether they will be executed in a local application or by a remote application?

3. Are there constructs that can be used for a local call but are not allowed for a remote interaction? (As an example of this, can pointers or arrays be passed around freely?)

4. Do remote interactions require special constructs that are not used or optional in a local case?

These tests are basically ones of appearance. If a distributed application looks different from the local application then, by implication, there is an acknowledgment that the local and
remote cases are not transparent. However, other tests or events at runtime might also remove
the cloak of transparency in a distributed system. At runtime, a distributed system may fail to
provide performance and failure transparency, thereby revealing the fact that the system is dis-
tributed.

Performance transparency can be tested for by having an application probe and monitor
its environment in an attempt to distinguish a remote interaction from a local one of the same
kind. One test that can be used is the elapsed time for the request. Typically there will be some
sort of measurable difference between the elapsed times. Given the additional processing costs
of preparing to execute a remote call, the delays and latencies associated with operating over a
network, and the extra dispatching overhead at the server site, it is expected that on similarly
configured machines that these extra overheads would manifest themselves through longer exe-
cution times and that the execution times would have greater variances. Certainly the goal of
performance transparency is a difficult, if not impossible, one to achieve. As a result the issue
of performance transparency is only addressed in a superficial fashion, based on whether the
remote operations can be perceived by the end user. If remote requests form only a small part
of an application the fact, that they are remote may not significantly affect the overall perfor-
ance of the application. Raven takes no special measures to supply performance transparency.

Failure transparency is present in a system when system software or hardware compo-
nents can fail, but the application can successfully complete its execution. Classic examples
that illustrate the problem are network partitioning and the failure of the remote site. In Raven,
if a remote site becomes inaccessible, either through a machine or network failure, the system
detects this and initiates error handling to print a message. The application is returned a value
of nil to indicate a problem with the request; however, it does not know whether or not the
request was executed. This behavior is something not expected in the local case and allows a
remote access to be distinguished from a local one. The proper way to deal with failures of this
type is to use an exception handling mechanism designed for the system. Raven does not have
such a mechanism, and as a result the handling of errors in Raven tends to be ad-hoc.
Before examining how Raven implements transparency at the programming level, consider how the well known notion of remote procedure call (RPC) operates. With RPC the approach is to take the well known and understood programming paradigm of the procedure call and extend it to the distributed environment. RPC’s strength lies in its ability to provide a straightforward way for an application programmer to make use of well defined remote services. Based on the criteria given previously, RPC is not overly transparent. The service provider, or library writer, is all too familiar with the fact that the service is going to be provided in a remote environment, as the various data structures to be used for parameter and result passing must be specified and external data representations for them provided. Stub routines for the various functions have to be provided for use by the client, and the server must be structured to handle the service requests directed at it. Care must also be taken to ensure that pointers are not used as parameters, or, if they are, that special processing is performed.

Raven uses global object identifiers to provide invocation transparency. In Raven each object has two identifiers associated with it. One is a \textit{global object identifier (GID)} and the other is the object identifier (OID). The GID is a system-maintained and assigned identifier that uniquely identifies each object in the Raven object space. This identifier is never seen or used by the application programmer. Instead, the Raven programmer sees and uses only OIDs, which uniquely identify objects within a single Raven environment. An object ID is, therefore, only meaningful within a given Raven environment. It is through the OID and the mechanisms that it uses to map OIDs to GIDs that Raven achieves its location transparency. Figure 4.14 provides a pictorial representation of how OIDs and GIDs interact to provide location transparency. Within a Raven system all objects are referenced through an OID, and these object IDs are either part of the instance data of an object or the working variables within a method. An OID references either a real local Raven object or a proxy object [91]. A Raven proxy, which is similar to the proxies of SOS [92], is the local agent for an object resident in some other Raven environment. A proxy object knows the GID of the object it is representing. To the programmer, a proxy object appears identical to a local object. Proxy objects distinguish themselves from local objects by the way they respond to a method invocation.
When a method is sent to a local object, that object looks up the method and runs it. An invocation request on a proxy object results in the proxy object doing some checking to see if the method can be run locally, and if it can the method is run; otherwise the request is packaged up and sent to the machine where the real object resides. As part of the request packaging actions, all OIDs are converted into their GIDs. Only GIDs are passed between Raven environments. The GID for a remote object can be obtained directly from its proxy object, and GIDs for local objects are assigned on an as-needed basis. When the request arrives at the remote machine, the remote site accepts the request and maps each GID to a local OID. For objects resident on the target machine, the OID references the real object, and for remote objects it references the proxy. When a GID for a remote object is received, a proxy object may have to be created if a proxy for that object currently does not exist in the environment. After the conver-
sion of all the GIDs to OIDs, the remote method can be invoked as if all the objects were local, thereby achieving location transparency for objects.

A concrete example of the way in which remote objects are handled in Raven is provided by Figure 4.14. In this figure there are two Raven environments, A and B. Environment A has a proxy to object A which resides in environment B. Environment B has a proxy to object B which resides in environment A. Initially some method has a reference to object A, but because object A is remote the reference identifies the local proxy for object A. In environment A, methods can be invoked on object A as if A were local. An attempt will be made by A's proxy to execute the method locally, and if it cannot be executed locally the request is forwarded to environment B, where A resides. As part of the forwarding, all OIDs are converted to their GIDs. Once in environment B, all GIDs are mapped to their local OIDs for that machine. In environment B that means, with respect to the servicing of this remote request, that object A can now be accessed using its OID. In this example object A has as part of its instance data a reference to object B, but because object B is remote to object A, object B is represented by a proxy object. Any references to B are treated in the same way as are references to A in environment A.

Providing a mechanism like GIDs and OIDs only goes part way to addressing the problem of location transparency. The other fundamental data structures available in the language also play a contributing factor. For example, if pointers were allowed then the Raven system would be unable to provide location transparency. Raven in fact does not have pointers: it conceptually has only objects; therefore a proxy can always be constructed. Some objects, like integers and floating point numbers, are basic machine-level data types. Since these objects are immutable (i.e., the integer object 5 will always be the object 5 and it cannot be changed to the object 17), they can be freely replicated and distributed, thereby providing location transparency for these basic types of objects. The object model itself also contributes to location transparency by providing a strict way to change instance data. Instance data can be accessed only through the invocation of a method; therefore, there is no way for an application to take an OID and attempt to modify or read the object's instance data directly. It is through the combination
of OIDs, GIDs, a pure object model, and instance data access only through methods, that Raven attempts to provide location transparency.

Raven's success at providing location transparency can be evaluated by using the criteria established earlier in this section. The first criterion listed for location transparency addressed the issue of whether or not an application programmer needed to know which library procedures were local and which were remote. The equivalent question in Raven revolves around whether or not the application (client) programmer has to be aware of whether or not an invocation on a server object is local or remote. In Raven the client routines to execute for either a remote object or a local object are exactly the same and no distinction is made.

The second criterion deals with whether or not the library (server) programmer must program differently if the routines are to be used in a local environment versus a remote one. In many respects this issue does not arise in Raven because all method invocations are local. If an invocation is targeted at a remote object, the information about the invocation is bundled up and sent to the machine where the object resides, effectively turning the remote request into a local one. The only issue for the server writer is whether or not any special actions need to be taken for parameters passed in that might reference remote objects. Again, the use of proxies, combined with the strict method interface to access instance data, means that the remote objects can be operated on as if they were local.

The third criterion concerns constructs that can be used for a local call but not for a remote call, with the prime example being pointers. In fact, there are no constructs in Raven that can be used for a local call but not for a remote call. However, the notion of constructs can be extended in the object-oriented environment to ask if the objects themselves place restrictions on their usage based on being local or remote. Care has been taken in the Raven environment to ensure that all special purpose or system type objects are location transparent. A good example of this is the location transparency that has been achieved with instances of the class Thread. A Thread object is the runnable entity associated with the thread of execution. It is the Raven analog to a process ID. Unlike process IDs, however, the OID of a Thread is location transpar-
ent. If a Thread’s OID is passed to another Raven environment, an object in that environment can control the remote thread as if it were on the local machine.

The final criterion concerns whether of not there are any constructs or features required in the remote case that are not needed or optional in the local case. It is on this point that Raven does not meet all the criteria for location transparency. In a distributed environment one of the problems concerns making the initial contact with the other machines and determining what sort of services these other sites are providing. For example, in Figure 4.14, environment A somehow had to acquire a reference to object A. In a local environment there is no need to establish the outside connection since everything is local. This problem, however, is not solely confined to the distributed processing environment but is more general in nature and results any time information must be exchanged across protection boundaries. These boundaries can be the result of such things as the separation of functionality between the system and the application (i.e. file systems), virtual memory protection schemes on a single processor, or protection via physical separation as provided in the distributed memory case. There are a number of techniques available for passing information across a protection boundary, but a typical approach, especially in the distributed memory case, is to create a name server at a well known location (address). The name server then becomes a repository for named items in the system. Objects that want to be known to the rest of the world register themselves with the server, and objects that want to locate other objects query the server to get back information on how to make that contact. The effect of this in Raven is that in the distributed case some setup to register, retrieve or construct the initial contact OIDs is required.

The significance of requiring some mechanism to establish contacts between the protection domains is at least partly related to how often the application must perform special operations to cross these boundaries and how it is done. In a UNIX environment, information about the other domains can be obtained through keyboard or file input, command line arguments, calls to the operating system, or environment variables. Based on the information obtained through these various mechanisms, the application might make a system call to get access to a
service or establish a network connection to a process on another machine. The approach is comprised of three major steps. First some user-level information is used to name a service, that name is converted into an access procedure for the desired service, and finally the desired service is accessed. Similar procedures are required in Raven where first some Raven environment independent name is obtained to identify a service. This name is then converted to an OID, and finally method invocations are made on the OID to actually access the service or information managed by the object. Raven provides several functions that simplify these actions and allow a name server to be constructed. More details on the operation of the name server can be found in Section 5.4.5.

Based on the criteria established for location transparency, Raven has met three of the four criteria. The only one it has not fully met concerns the use of special calls to allow different Raven environments to discover information about the objects that each of them has. This discovery or registration-type requirement is not unique to Raven but exists in any system where information has to be exchanged across a protection boundary. Being object-oriented, Raven has the opportunity to consolidate the various approaches to dealing with cross-boundary communication in a single cohesive fashion. However, Raven does not perform as well when tests for transparency based on runtime information are considered. In fact, Raven makes no attempt to address issues associated with performance transparency, and failure transparency is also poorly supported. The current Raven environment expects to run on a local area network which is highly available. It is expected that as Raven becomes more fully integrated into a system environment (i.e. the equivalent of a file system is added and support for transactions is added) the distinction between the local and remote execution case will become less noticeable. Additionally, Raven needs an exception handling mechanism to deal with failures.

4.8 The System Object

The operating configuration of the Raven system is tailorable by both the end application user and the programmer. The configuration is stored in the predefined object, system, which is a
reserved word in the Raven language. There is only one system object per Raven environment and it is an instance of class System. The system object is instantiated during the start-up phase of Raven.

During the running of the constructor for the system object, environment variables and the host system are checked for configuration parameters. Based on these values, system's constructor sets its instance variables to reflect the resulting configuration. If necessary, the constructor will also make the appropriate calls to communicate this information to the runtime system. Example specifications that can be set in the constructor are the amount of system-permissible parallelism and the amount of memory the system can use.

The system configuration parameters determined during the instantiation of the system are application independent and reflect the operating environment the application is to operate in. The application program can also reconfigure parts of the system configuration. The configuration parameters directly settable by the application control the usage of services dependent upon the nature of the application. Currently, the primary purpose of this capability is to allow the application program to enable special system services, for example the enabling of network based services and the time-out checking for potential deadlocks.

4.9 Failures and Exceptions

Within a system, failures and exceptions will occur. An application might attempt to divide a number by zero, the network might fail during a remote invocation, or a requested service might not be available. Just how to inform the application of these problems and how to specify exception and failure handling within the Raven system is still an open research issue. Part of the problem is that it is not clear how an error notification should be delivered. Furthermore, in some situations the invoking object should deal with the problem, whereas in other situations the thread of control, which is itself an object, might be the appropriate entity to inform.
To illustrate the nature of this problem, consider a method that encounters a divide by zero exception. In this case the method itself may want to deal with the problem. It is easy to imagine that within a method the same sort of error processing of a divide by zero fault would be performed regardless of the application. In other situations, for example a deadlock time-out, such a failure, and the desired corrective action, is more closely associated with the thread of control. It is unreasonable to expect every method to be prepared to deal with a deadlock error when such an event is more likely a function of the total system configuration than of an individual method.

Failures associated with the network, or involving inter-environment interactions, are detected by the Raven runtime system. Upon detecting the error special error handling code is run to log this exception. In the Raven Configuration Management System [32], this information is used to construct a response to the failure. In addition to the special error handling, the nil value is returned to the application.

The last example illustrates the general problem of how to inform the runtime system of any failures or exceptions. Two approaches have been tried within Raven. One approach is to have methods, particularly those that provide services, return nil as a failure indication. This presents a problem when nil is a legitimate return value. It also requires an application to be constantly checking return values for error indications. A second approach is to specify the name of a method to be invoked when a failure is encountered. For example, the class Object defines the doesNotUnderstand() method. This method is invoked when an attempt to invoke a method is made on an object, and the method cannot be found. Since all objects inherit from Object, they automatically have some limited error handling for this case. An object can change its error handling by overriding the doesNotUnderstand() method. This approach has problems because it is not clear which object the error notification should be delivered to (i.e. the invoking object or the thread object). Additionally, the corrective actions and how they are accomplished need special support. At the moment Raven uses a mixture of these two approaches, but there is no clear consensus on the best way to handle these problems,
and they have been left for future work. Before developing a proper error and exception handling mechanism for Raven, the large body of literature in this area [39][86][88] needs to be consulted.

4.10 Unique Raven Language Features

Unlike many systems, the support for parallel and distributed computation forms an integral part of the Raven programming language and was not added once the language was completed. The result is a smooth integration of the support for parallel and distributed computation into the language. Raven's support for parallelism is unique in that it recognizes a single logical program location where parallelism and synchronization between threads can occur: All parallelism and thread synchronization in Raven occurs at the point of method invocation. Although the parallelism occurs at the method boundary, the parallelism is functionally extracted on either side of the boundary. If the method invoker wants parallelism, a companion is used; if the method itself wants parallelism, an early result is used; and, if thread synchronization is needed, a delayed result is used. No other system combines these three related activities to make them a logical part or extension of the sequential method invocation.

Raven’s companions extract parallelism at a higher programming level than other systems by eliminating the notion of explicit process creation. For example, in Distributed Smalltalk [20] and Actra [103] the programmer must explicitly create the process and then program the parallel actions within the process. This approach to creating parallelism is significantly different from Raven’s, where any object, and any of the object’s methods, can be used to generate parallelism at any time. Although CST [37] also uses the method invocation point to extract parallelism, its use is restricted to objects which are aggregates, or groups, of other objects. In much the same way that vector processing is performed, the invoked method is, at least conceptually, simultaneously run on all the objects in the aggregate. Unlike Raven, objects that cannot be represented as aggregates cannot be the subject of parallel invocations.
When a companion is specified, it encapsulates a block of requests that are to be run in parallel. In that context the companion is very much like the `parBegin/end` style delimiters of parallel requests found in SR [9] and Argus [69]. However, it differs from these in that the block of requests comprising the companion are run in parallel with each other and with the thread of control starting the companion. In these other systems the block of requests are run in parallel only with each other. The main thread of control, in Argus's case, waits for all the requests to complete before proceeding. With SR, the main thread of control can wait, or it can continue processing. If it continues processing, the main thread cannot later synchronize with the requests. Raven overcomes this synchronization problem by creating a `Certificate` for each companion. The `Certificate` permits the application to, optionally, wait for the companion to complete.

Raven's `Certificates` are similar to the futures of Multilisp [50] and the promises and claims extensions to Argus. However, all these constructs wait only for a single outstanding request to complete. Raven is the only system that extends the futures idea, through `CertificateGroups`, to allow an application to build collections of futures or promises and then to wait for any one of the requests within the `CertificateGroup` to complete. By being able to wait for any one of the outstanding activities in the `CertificateGroup` to complete, Raven applications can process the asynchronous completion of outstanding requests. The other systems offer no mechanism to perform this type of processing on requests.

Closely associated with companions and `Certificates` are `InvokeStreams`, which provide an ordering service for a sequence of companions. The Raven `InvokeStream` differs from the stream implementations of Gifford [48], and the corresponding extensions to Argus, by allowing one stream to provide ordering to multiple target objects. This approach eliminates the need to create and manage multiple streams at the application level, where one stream would suffice. Raven's `InvokeStream` is also unique in that it can be shared and passed between objects in potentially different threads of control. For example, one object can push a sequence of requests onto an `InvokeStream` and then pass it to another object in another thread of control.
That object can then push its own requests into the stream, and the sets of requests emanating from the different objects will be appropriately sequenced.

Synchronization between independent threads of control in Raven is accomplished through the delayed result construct. The Raven approach differs from other approaches to synchronization by basing synchronization on the notion of returning results, instead of on direct control of the process. For example, in representative systems like SR [10], with guards, or Choices [27], with direct message passing, there are application visible processes, and synchronization is accomplished by suspending the process and then restarting it. When the suspended process is restarted, it is the process's responsibility to decide how to proceed with processing, and in particular the object must be prepared to deal with any changes in object state that occurred while the object was suspended. The delayed result of Raven uses a different perspective and places the programming emphasis on returning a result. Raven's underlying assumption is that a method should run until it can return the result. If the method cannot run that long, it does a leave and the method suspends itself and waits for the returned result to be determined. The result to return is computed by another method, and the result statement used to direct the result to the suspend method for returning. Since the suspended method is only returning the result, the problem with object state changing while the method is suspended is eliminated.

Of the higher level object-oriented languages or systems, of which Distributed Smalltalk and POOL-T [11] are representative, POOL-T is the only system that supports the notion of an early result. In POOL-T each method is specified as two distinct parts. One part computes the value to return, while the other part specifies the code to execute after returning a result. Raven's early result offers much more flexibility then the POOL-T approach because the processing to be performed after the early result immediately follows the early result. This allows the method to perform different types of post-processing depending upon where in the code the result is returned.
Underlying all Raven’s support for specifying parallelism is the need to control concurrent access to an object’s instance data. Existing systems use two approaches to protect instance data. One technique, as exemplified by Distributed Smalltalk and Choices, is to do nothing and make concurrency control the explicit responsibility of the programmer. The other technique, as illustrated by POOL-T and Actra, is to allow only one method of an object to be executed at a time. Some of these approaches also limit which methods can be invoked from outside the execution scope of the object. Locking in Raven differs significantly from all these systems by permitting multiple method invocations to be active in an object and by supplying the locking support automatically. Raven allows multiple read methods or a single write method to be active in an object, thereby increasing the potential for parallelism on an object, while relieving the programmer of the responsibility of explicitly managing the instance data locking. The removal of explicit lock management from the programmer’s responsibility simplifies programming and reduces the chances of errors due to improper use of concurrency control features. Raven’s locking scheme, unlike the other systems, also permits a locked object to recursively invoke back on itself without causing a deadlock. The execution pattern where an object invokes, either directly or indirectly, back on itself is very common in object-oriented programming.

One of the problems with object-oriented systems is finding ways to deliver system services, like concurrency control to individual objects. One approach, as illustrated by Arjuaon [93], is the class-hierarchical approach, where the object inherits from the classes that provide the services of interest. This has two problems. One is that the object still has to invoke the appropriate methods to internally manage the desired services, and the second is the class explosion problem. Reflective systems [108] attempt to solve this problem by allowing entities in the system to redefine system level behaviors. There are, however, problems associated with implementation, with determining the scope of any changes made to the meta-objects providing the system behavior, and with the introduction of side-effects with respect to other system services. Raven’s property mechanisms eliminates all these problems by providing a mechanism to attach operating system services to objects on an instance-by-instance basis. The
orthogonality of properties guarantees that multiple properties can be associated to an object without them interfering with each other, or with other objects, and it also eliminates the class explosion problem.

4.11 Summary

Parallelism is supported through companions and Certificates, along with the constructs for early and delayed result. Together these features allow an application programmer to introduce parallelism from within a method (class-based parallelism), and at the point where the method is invoked (user-based parallelism).

The introduction of parallelism into the Raven system also introduces extra dimensions into the programming problem with respect to concurrency control. Raven introduces properties to provide concurrency control. Properties allow several orthogonal system supplied services to be associated with an object. As part of concurrency control, automatic locking is performed on instance data. To support locking in a distributed environment, session IDs are introduced.

In any system there are issues associated with resource management. Raven uses garbage collection to address issues of memory management. Processor control is provided by limiting the number of companion threads that can be started. InvokeStreams can also be used to extract protocol parallelism from sequential requests.

The final issue addressed by this chapter is location transparency. A scheme using global object IDs, local object IDs and proxy objects is described that allows for Raven to have a substantial degree of location transparency.
This chapter deals with the underlying implementation issues associated with providing the facilities for parallel and distributed computation described in Chapter 4. The chapter starts with a discussion of how objects, and in particular class objects, are implemented in Raven, along with their role in supporting parallel and remote invocation. During the implementation, problems associated with method dispatch and memory management were encountered. These problems severely affected the performance of Raven. The problems are discussed in detail and the resulting solutions described.

5.1 Object Implementation

At the application level, everything in Raven is an object and all objects are semantically identical. An object is accessed through an object identifier (OID), which is also known as a capability. At the implementation level, however, some objects are handled in a special way. These are objects that have a machine level representation can be manipulated directly by the hard-
ware. These objects are called primitive objects. Examples of primitive objects are integers and floating point numbers.

The second type of objects identified by an OID are complex objects. Complex objects are object types that are programmer or system defined, and their component parts consist of any combination of primitive objects and complex object references. A complex object has all the mechanism and organization needed to support the full range of actions expected of an object. All primitive objects have a complex object representation. It is the compiler's responsibility to determine when a primitive object needs to be converted to a complex object or when a complex representation of a primitive object needs to be converted to its primitive form.

The Raven system is organized around the principle that everything is an object. Within an application objects can serve two functions. The one function, filled by class objects, is to provide the basic template or definition for a class so that new instances of the class can be created. The second function, filled by instances of classes, is to provide the actual data structures manipulated to solve a problem.

Since new objects can come into existence only through invocations on a class, special consideration must be given to initializing the basic Raven class objects. During initialization, the runtime system creates the basic classes Object, MetaObject, MetaClass, and MetaMetaClass (see Figure 3.5). Once these classes are created the UndefinedObject class and its sole instance, nil, are created. The remainder of the classes are created and initialized in a breadth first manner based on the inheritance hierarchy constructed by the compiler. Only those classes needed by a particular application are loaded. This class loading is accomplished by having each class record in well-known data structures the name of its parent class and the names of all the classes that it uses. Each class has a well-known system initialization routine that gets called during system start-up. When the initialization phase starts, and the classes Object, MetaObject, MetaClass, MetaMetaClass and UndefinedObject have been initialized, the class Main, which descends directly from Object, is initialized. Class initialization consists of building all the data structures needed to support the class in a running system (see Section 5.1.3). As each class is
initialized, it must ensure that all the classes it uses or inherits from are also scheduled for initialization.

### 5.1.1 Object Organization

There are three broad categories of objects in the Raven system. These are instance objects, class objects, and meta-class objects. All objects have the same structural organization. An object's uniqueness is established by the contents of its instance data and its methods. Figure 5.1 provides a pictorial representation of the component parts of an object as they appear in main memory.

![Object organization in memory](image)

Each object ID in the system maps to one object in main memory. In the current implementation of Raven, an object ID is simply the 32 bit address of an object block. An object's
representation in main memory is called an object block. An object block has the following fields:

1. Invoke function pointer. This field identifies the function to perform the method lookup when an invocation is performed on this object. This allows different lookup functions to be attached to individual objects.

2. Object ID. This field contains the object ID of this object and allows an object to specify its identity to the rest of the Raven system.

3. Is-a pointer. This field points to the object block of the class this object is a member of.

4. Method type field. This field is used to select the category of methods this object is to use. The category of methods selected depends upon whether this object is a local object or a proxy object. Since proxy objects are agents for a remote object and do not have any instance data, any request to execute a method must be forwarded to the remote site. This field is used to make that distinction.

5. Global ID pointer. This field points to a region of memory that contains the specification for the global ID assigned to the object. Global IDs are assigned on an as-needed basis, and if an object does not have a global ID this pointer is set to NULL. Proxies always have a global ID.

6. Lock pointer. This field points to the lock control data for the object. The lock is used to provide concurrency control on the object.

7. Object properties. This field specifies the properties this object has. Properties are set on an object-by-object basis and assigned at object creation time.

8. Instance data pointer. This field points to the region in main memory where the instance data for the object actually resides. The interpretation of the instance data is a function of the class that the object is created from. Proxies do not have instance data.

With the data provided by an object block, it is possible to discover all needed runtime information about an object. The class hierarchy can be traced, the methods to execute obtained, the instance data can be obtained, and the object properties determined.

5.1.2 Global IDs

A global ID (GID) is the mechanism used by the Raven system to provide a unique identifier for an object over all Raven environments. For the vast majority of objects, a local object
ID is all that is required. Only those objects used in a remote invocation need a GID; therefore, GIDs are assigned on an as-needed basis.

Global IDs are used to identify objects in one Raven environment to other Raven environments. Global IDs are used only at the Raven system level and not by the application. An application always references an object through the local object ID. When an object ID needs to be passed between Raven environments, the object ID is converted to its GID and it is the GID that is passed. At the receiving site the GID will be converted to a local OID, either through a proxy, or, if the object exists on that site, the actual object. Figure 5.2 illustrates the

<table>
<thead>
<tr>
<th>Host ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Raven Env. ID</td>
</tr>
<tr>
<td>Local Object ID</td>
</tr>
<tr>
<td>Class Name</td>
</tr>
</tbody>
</table>

FIGURE 5.2 Format of global ID information.

global ID information transmitted between Raven environments and stored as part of an object. The fields in a GID provide the following information:

- **Host ID.** This field provides the host id of the machine where the object resides. In the current implementation, this is the simply the IP address of the host.

- **Local Raven Environment ID.** Each Raven environment is implemented using a process. Since multiple Raven environments can be running on a single machine, an identifier is required to select the appropriate Raven environment. In the current implementation this is the environment’s UDP port number.

- **Local Object ID.** This field identifies a particular object within a Raven environment.
• Class name. This field is the textual representation of the class name that the object represented by this GID belongs to. The class name is required so that the receiving site can construct a proxy for the object identified by this GID, if a proxy is needed.

When an object ID is passed as a parameter in a remote invocation, a GID needs to be sent. If the object ID does not identify a proxy, then a global ID is assigned to the object and passed to the remote site. Once the global ID is assigned to the object, that association is fixed and the global ID can be obtained from the data structure pointed to from the object block. The information that constitutes a GID is sufficient to allow a proxy to be constructed at the remote site and to allow the remote site to locate the object.

On the remote site, when a global ID is received a proxy object may need to be created to represent the remote object and to provide an object ID for use in the local environment. As part of constructing the proxy, space for a global ID is allocated and the global ID that was passed in is stored: the class name passed as part of the GID is used to set the proxy’s type (is-a field in the object block). A proxy, like all objects, is represented in main memory with an object block, except there is no instance data. A proxy object will have the data fields in an object block (Figure 5.1) set in the following manner:

1. Invoke function pointer. This field will point to a function tailored to perform remote method invocations.
2. Object ID. This is set to be the object ID of the proxy.
3. Is-a pointer. This points to the object block of the class the remote object is a member of. The class name passed in the GID is used to lookup the class.
4. Method type field. This field is set to indicate that the remote version of a method should be invoked.
5. Global ID pointer. This field points to the global ID of the remote object.
6. Lock pointer. This field is NULL. Locking is handled by the site where the object resides.
7. Object properties. This is set to BASIC (Section 4.4). Any special handling of an object because of its properties are handled at the site where the object resides.
8. Instance data pointer. This pointer is NULL.

The global ID information stored with the proxy will be used either when the proxy is passed as a parameter, or when an invocation is performed on the proxy. When a global ID arrives at a site, a proxy is constructed only if a proxy for the object does not exist and if the object identified by the GID does not reside here.

5.1.3 Class Objects

As mentioned earlier, classes in the Raven system are objects. They obtain their uniqueness through their instance data and behaviors. It is through a class object that methods are looked up, and a description of an object’s instance data maintained. Each object block in Raven has an is-a field which references a class. A class object describes the common properties associated with all instances of the class. Figure 5.3 shows the layout of the instance data for a class object. For all but the basic classes the description of the instance data is taken from the class declaration. The format of the class instance data is needed before the system starts so that the basic classes can be created. The layout of the class instance data and methods is, therefore, specified in the runtime system. The fields and the functions of the class instance data are as follows:

1. Parent class object ID. This is the object ID of the parent class and is used to implement inheritance. When a method cannot be found at this level, the parent class is consulted to determine if the method resides there.

2. Class data size. This field indicates how large the class data is. Raven supports the notion of class data that can be associated with a class. This data is set when the class is created and cannot be changed after that.

3. Class data pointer. This field points to the class variables accessible to the programmer.

4. Class ID. This is a unique identifier for class on this machine.

5. Instance data size. This field indicates how much instance data is associated with instances of this class.
6. Max parameters. This field records the maximum number of parameters that any method in this class uses. This is used to decide the default invoke function to assign instances of this class.

7. Default invoke pointer. This field points to the invoke function assigned to basic instances of this class. The invoke function assigned to an object can be changed at object creation time depending upon the program-assigned properties.

8. Method count. This field indicates how many methods are in the method table.

9. Method table pointer. This field points to a table listing all the methods implemented by this class. The method contains: the method ID field to identify the method; the parameter count and type; the lock type (i.e. read, write or nolock); return type; a list of pre-handlers to invoke before running the method; a list of
post-involve handlers to run when the method terminates; the method to run for a local invocation; and the method to run for a remote invocation.

10. Instance variable count. This is the count of the number of instance variables this class has.

11. Types of instance data. This field, which is variable in length, specifies the type of each piece of instance data.

A class object plays two pivotal roles in the Raven system. First, when a new object is created, the class is consulted to construct an instance of the object. This involves allocating space for the instance data and making the appropriate class relationships. Secondly, when a method is invoked, class objects are consulted to determine the method code to execute.

5.1.4 Properties

Each object in Raven can have properties attached to it; and the assigned properties are specified in the property field of the object block. Properties associate particular operating system services with an object, and the object property field indicates what properties are currently in effect for this object. Based on that information, the invoke routine will perform specific processing. When the invoke function is called it checks to see what sort of properties are associated with the object, and executes the pre-involve functions for the properties The method is executed and then any post-involve functions required by the properties are executed. As an example of the processing, consider the actions taken when an object has the controlled property. In this situation the pre-involve function will acquire the appropriate lock for this method; the method will be executed and, when it returns, the lock released using the post-involve function. The data structures to manage the lock are pointed to by the lock pointer in the object block.

5.1.5 Method Invocation and Lookup

To invoke a method, a method invocation function is called. The function takes as parameters the object, some ancillary information useful for looking up methods, and the parameters for the method. The invoke function is responsible for looking up the method to run, making copies
of any parameters marked as copy, performing object specific pre-invoke processing, determining if the method is local or remote, running the method, performing object-specific post-invoke processing, and returning the result, making a copy if necessary. Figure 5.4 provides a pseudo-code description of the steps that need to be performed by the invoke function.

-lookup method to be run
- if (parameters need to be copied) then copy required parameters;
- if (property1 in effect) then execute pre-invoke function for property1;
- if (property2 in effect) then execute pre-invoke function for property2;
- if (property3 in effect) then execute pre-invoke function for property3;
...... -execute other required pre-invoke functions

- if (local method) then return_value = execute_lookup method;
  else return_value = execute_remote_request();

......-execute post-invoke functions
- if (property3 in effect) then execute post-invoke function;
- if (property2 in effect) then execute post-invoke function;
- if (property1 in effect) then execute post-invoke function;
- if (return result needs to be copied) return_value = copy of return value;
- return return_value;

FIGURE 5.4 Pseudo code illustrating steps involved in performing an invoke.

The pre-invoke and post-invoke processing consists of checking the object to see which properties are in effect. However, each one of these checks takes time, and in the vast majority of cases there are no properties in effect, so the tests are wasted. A similar observation was made about remote invocations and the copying of parameters and returned results: again, most objects do not need special processing to handle these cases. Given the large number of conditions the invoke routine must check and the fact that every method must run this code, the invoke code is a potential performance bottleneck. It is therefore imperative that the invoke routine perform the minimum amount of work required to properly execute a method.
One way to reduce the invoke processing overhead is to have multiple invoke routines designed to handle different invocation scenarios. For example, one routine could handle the basic case when no special processing is required, while another could be a generic routine capable of handling all the special cases. Then only those objects needing the extra processing would use the generic routine and pay the execution cost. The difficulty with this approach is determining which invoke routine to run for a particular object. Partly because of properties, each object is different; therefore, the invoke routine needs to be dynamically determined on an object-by-object basis. In Raven the invoke field of an object’s object block specifies the routine to use when processing method requests, and each object can be assigned the invoke routine best suited for handling its invocations. As part of the object creation process, the runtime system examines the properties of the object, checks to see if parameters or results are copied, and checks to see if the object is remote (this is done if a proxy object is created). Based on these checks, the runtime system selects one of several invoke routines for use by this object. An object that has no special processing needs will be assigned the invoke routine which simply looks-up the method and executes it, whereas an object with multiple properties and copy parameters might be assigned the generic invocation routine. The generic routine is basically the one described by Figure 5.4.

By using this approach, the runtime system can provide a large collection of invocation routines specially tailored to the needs of the objects, as opposed to one generic routine that can handle all possible types of objects. The result is that the invoke routines need to execute only the code relevant to the object, so only those objects that require special processing pay the increased execution costs. The invoke routine does not need to be concerned about service requirements that do not apply to the object.

For a method to be executed on an object, the class and object must be co-resident, since the class is the only entity that identifies the methods (code) that can be used to manipulate an object. The association between an object and its class is accomplished through an object’s is-a pointer, which points to the class the object is a member of. Once the object’s class is estab-
lished, the methods runnable on the object can be determined. If the object being invoked on is itself a class then the is-a pointer identifies the meta-class for this object. Meta-classes are automatically constructed by the runtime system when a new class is created and are not user specified. All meta-classes have their is-a pointer directed at the system defined class \texttt{MetaMetaClass}, whose is-a pointer points at itself. Again Figure 3.5 can be consulted to see the is-a relationship between instances, object classes, meta-classes and \texttt{MetaMetaClass}.

In Raven, method lookup is dynamic; therefore, for Raven to work quickly and efficiently it is imperative that as little time as possible is spent determining the method to run. Conceptually, when a method is invoked, the object's is-a pointer is followed to get to the class this object is a member of. The method is then looked for in the method table of the class using the \textit{method ID} as a key. If the method is not found, then the search continues for the method by looking in the parent's methods, and so on until either the method is found or the top of the inheritance hierarchy is reached and a method lookup error signalled.

In the first implementation of Raven, methods were identified and dynamically looked-up at runtime using the method name as the key. Profiling was performed on the code and it was discovered that a substantial amount of time was spent performing the string comparisons in the method lookup. It was obvious that string comparisons needed to be avoided. After some experimenting, a technique using hashing was selected. The approach relies upon replacing, for method lookup purposes, the string representation of the method name with a four byte hash value. For this to be an effective solution there needs to be few hash collisions, and the hash value, for performance reasons, should be computed during compilation.

The problem of hash collisions was solved by selecting a good hash function. A number of hash functions were evaluated and a public domain hash function was taken from sdbm [107], a hashed database system. This function was found to be very good at generating unique hash values. In one test 84,165 strings taken from a list of English words and the symbol tables from various programs were hashed and no collisions occurred. The number of methods in a Raven application is typically an order of magnitude less than this, so the chances of a hash
collision are expected to be small. Of course, hash collisions are of consequence only if the collision occurs for methods in the same class or in the inheritance hierarchy as traced from a leaf class to the root. Still, a hash collision, although highly unlikely, could have devastating effects, and could cause a correct program to fail. This problem is addressed by checking hash values for collisions as the methods are registered with a class during the system initialization phase. If a collision is detected, the Raven program stops and reports the collision. It is then the programmer’s responsibility to change one of the method names so that a collision does not take place. Using this load-time check, the hash is assumed to be perfect. To date, no Raven program has ever reported a method name collision.

To avoid the expense of having to compute the hash value at every invocation, the key for method lookup is the hash value. Within Raven this hash value is called the method ID, since a one-to-one mapping between the method IDs and method names is assumed. When an invocation is performed, the method lookup function takes the method ID to be looked for and checks it against the method IDs stored in the appropriate method table. Since the methods being invoked are known at compile time, the compiler can compute the method ID immediately and use a constant for the method ID instead of calling a runtime routine to compute the ID.

Although the use of method IDs significantly reduced the execution overhead associated with method lookup, a substantial amount of time was still used in looking for the method. It was therefore decided to use a method cache to speed up method lookup. However, it was also important to minimize the overhead associated with determining whether or not a method was in the cache. This requirement effectively ruled out using any caching strategies that involved some organized search of the cache. The solution selected was to use a fixed size hash table to store information about the cached methods. The initial hash function used an exclusive oring of the object ID and the method ID. The method ID could not be used by itself, since methods with the same name, but belonging to different classes, would have the same method ID. It was observed that, although this worked, the hit rate on the cache was low, and the same method
was often used by different objects. These observations resulted in changing the hashing scheme so that the class ID of the object was exclusive or ed with the method ID. Implementing these changes resulted in the cache hit rate with a 1024 slot hash table exceeding 95% for the test applications. Of course, this result was not unexpected, as programs typically exhibit a large amount of execution locality. Once a computation enters one of these phases, the methods currently in use will fit in the cache and a large number of hits will be registered before moving onto another phase of the computation and reloading the cache.

This combination of changes significantly improved Raven's method lookup performance. However, the effect of the changes is dependent upon the application being run. In particular, the access patterns to the cache, and how far up the inheritance hierarchy a method is, will have a significant effect on the observed improvement. For code that had a high cache hit rate and methods that were found in the method tables at the leaves of the class hierarchy, the method invocation time went from over 10 procedure call times to about 2.7 procedure call times. A procedure call time is the amount of time required to execute the null C procedure. What this means is that if the C call takes 1 time unit to execute then the equivalent Raven invocation takes 2.7 time units to execute.

This particular configuration served Raven well until Raven was ported to run on a shared memory multiprocessor. On the first run of an application executing multiple disjoint threads of control and using two processors, there was only a slight performance increase. As the number of processors working on the problem was allowed to increase there was actual performance degradation when compared to the single processor case. The problem was traced to the method hash table. Since there was the potential for the hash table to be modified if there was a method collision, or if a new entry was added, the table had to be locked for exclusive access by the method lookup routine. It was originally assumed that the amount of time the lock on the table would be held would be much shorter than the amount of time between lock requests; therefore, there would not be a contention problem. This assumption was incorrect. There was,
in fact, extremely heavy contention for the hash table, and this accounted for the observed performance degradation.

For Raven to run in a multiprocessor environment, some way of reducing the contention for the hash table during method lookup was required. One solution considered was to change the granularity of the locking on the table so that each entry would be independently locked instead of the whole table. It was not clear if this solution would solve the problem. It is easy to imagine an application with a number of parallel threads all executing the same method in a loop. With this solution, each one of these requests would still block and there would not be any noticeable performance improvement. A modification of this approach, such that read and write locks could be used on entries, was also considered. This approach seemed overly complex and it would certainly require considerably more CPU time to run than the much simpler single lock approach.

Another way to reduce contention is to have multiple caches and use some sort of simple scheme to decide which hash table to use next. Part of the difficulty with this solution is deciding upon how many hash tables to have, and there is still the possibility of hash table contention if applications get in step with one another. The likelihood of this happening is reasonably high, given how often methods are looked up in Raven, and the experience with a single lookup table. The three solutions proposed so far all had the advantage that they could be easily and relatively quickly added to the existing Raven system. Their primary disadvantage was that there was some question as to whether or not the solutions would actually deliver improved performance, and if they did, how well the solutions would scale.

Since it appeared that the proposed solutions were not going to be able to provide the desired level of support, it was decided to use a more complex strategy, based on multiple hash tables. The approach taken is to associate a hash table with each Raven thread. By doing this it is possible to dispense with locks on the hash table, since only one thread of control can ever access a table. To be effective, there must be quick access to the hash table. With the existing organization that was not the case. The hash table exists and is managed completely by the run-
time system; therefore any access to process specific data structures, which might be useful for keeping information about the hash table, are one layer removed from the runtime system and not accessible without a call to the threads package. If possible, function calls to determine the location of the hash table were to be avoided.

With the hash table being so closely bound to a thread, it seemed appropriate to allocate the hash table at the same time the process stack was created. The threads package was modified so that as part of the basic starting of a thread a hash table for the thread would also be allocated. To get quick access to the hash table, it was decided to pass the address of the hash table as a parameter to the invoke routines. Since an invoked method could itself invoke a method, the hash table information needed to be passed into the actual Raven method. This requirement necessitated a change to the compiler so that all Raven methods would be expecting an additional parameter with the hash table address. Additionally, the compiler had to be changed to ensure that all calls to the invoke routine that it emitted also included this new parameter.

By bypassing in the hash table, a mechanism was being developed that allowed thread-specific information to be made easily and quickly available to the runtime system. Once this was realized, it became apparent that other parts of the runtime system and compiler could take advantage of this organization. As such, a new entity, the Raven environment, was specified. It is actually this new Raven environment structure that is passed around. Currently, the per-thread environment regions specify the process ID of the system process running this thread, the Raven object ID for this thread, the hash table, and the end-of-stack location. Some of this information is accessed frequently, and having it in the environment offers performance improvements over function calls to the threads subsystem. Some of the other information is difficult to obtain and modify, so having the information available to the runtime system provides greater flexibility in the type of code generated by the compiler and ease of use by the runtime system.
Once all of these changes were made, the original application program that highlighted the deficiencies of the hash table system was rerun. This time the performance improvements were as expected, and the solution scaled with the number of processors. The hash table was no longer a system bottleneck.

5.2 Memory Allocation and Garbage Collection

Within Raven, the unit of memory allocation is the object. There is no notion of directly allocated memory through a function like malloc() that is common to C programs. Instead, the programmer explicitly creates new objects. The creation of an object results in the Raven runtime system acquiring some memory to hold the object and in the initialization of the object. A reference (OID) to the object is returned to the application program. In the C/Unix environment, once the allocated memory is no longer needed the program frees it. With multiple threads of control, it becomes increasingly difficult to determine when it is feasible to free allocated memory.

The problem is compounded in Raven since capabilities can be freely distributed and duplicated. When an invocation is performed, the parameters to the method reference other objects. When control is returned to the invoker, it is not possible, in general, to determine what was done to object references passed as parameters. The invoked method can make a copy of the object reference, pass the reference on through another invocation, or be finished using the object. If copies were made, then the object cannot be freed. Likewise, the method that makes a copy of a reference cannot, in general, know what other objects already have a reference to an object. Many systems deal with this problem by establishing strict memory allocating and freeing protocols between program modules. The very nature of an object-oriented system and its approach to data hiding and encapsulation discourages such an approach. One possible solution would be to adopt a compiler-based reference counting technique. However, this technique becomes increasingly difficult to implement as local method variables go in and out of scope, capabilities get passed as parameters, returned as results, stored in instance vari-
ables, assigned to method variables, and lost as method and instance variables have new data assigned to them. Instead, the approach adopted in Raven was to use a conservative garbage collector.

The memory allocation and garbage collection system being used is the implementation by Boehm and Demers of the Xerox Corporation and described in Boehm and Weiser 1988 [23]. Modifications to operate in the Raven parallel processing environment were made as part of this thesis work. This collector has the advantage that it can be grafted onto an existing system through the storage management functions of free() and malloc(). To use the collector, the standard C library calls for memory allocation and freeing are replaced with the equivalent ones from the memory management package. The design of the Raven system is such that there are only a few locations in the runtime support code where memory allocation is performed, thereby making the task of changing the memory allocation techniques a trivial one. Raven application programs are not affected by any changes to the interface to the memory allocation routines because Raven applications cannot directly allocate memory. Raven applications acquire new memory to work with by creating objects, and this is done through an interface to the Raven runtime system.

The collector being used is described as a conservative collector, designed to operate in an uncooperative environment. The garbage collector uses a mark and sweep approach consisting of two passes. The first pass marks all the allocated data objects accessible by a program, and the second pass returns the unmarked memory objects to a free memory list. The mark phase treats all data accessible by the program as potential pointers. This includes all the data in registers, on the stack and in static program data areas, as well as the data within an allocated memory object. Data alignment within each of these regions is assumed to be on boundaries dictated by the particular hardware. For the machines Raven currently runs on, 4 byte boundaries are used. Each piece of data in the region is checked to determine whether or not the data could be a reference to memory managed by the allocation system. If it could, then the assumption is made that this data value is indeed a pointer and the memory object is marked as acces-
sible. It is possible that an integer value may map to a valid object, and this will result in unused memory being marked as in use. This will not result in incorrect program execution, only in excess memory consumption. Further details on the implementation of this storage allocation system can by found in Boehm and Weiser [23].

5.2.1 Modifications to the memory allocation system.

The original collector offers some support for a parallel environment by providing hooks to serialize requests for memory. Serialized memory requests severely degrade a parallel application's performance when the application places high demands for memory on the storage allocation system. The initial efforts at using the storage allocator in a parallel environment used the provided facilities. As long as Raven was operated in a single processor environment there were no problems. However, a problem was encountered as soon as the test applications were run on a shared-memory multiprocessor. It was apparent that the memory management system was experiencing problems when high demands were placed on it.

During parallel operation the Raven system placed extremely high demands on the allocator, and the allocator was essentially unable to keep up with requests. The rate of memory requests was so high that the memory allocator became a bottleneck and forced the application to become sequential. Although each application has its own memory use profile, generally, as the number of processes and processors in use increases, the rate of memory requests increases. Once the interval between requests becomes less than the service time for a memory request, the application is effectively serialized while processes wait for memory. As the queues for memory requests grow, the overhead in maintaining the queues also increases and the system gets progressively slower and less and less useful work is done.

Figure 5.5 is a graph of the processor efficiency\(^1\) versus the number of processors and the proportion of memory allocation delays versus processors. The application being run is a Raven program consisting of a loop that creates new objects and then discards them. In this

\(^1\) Efficiency is speedup divided by the number of processors. More formal definitions are provided in Section 6.1.1.
example, the main program creates 200,000 new objects, which translates directly into that number of calls to the memory allocation system. The graph shows that with two processors there is only contention for the memory allocator 5% of the time. As soon as three processors are used, over 99% of the memory requests result in contention and there is a corresponding dramatic drop in efficiency. By this point the application has been effectively serialized, and, as processors are added, there is an increasing amount of overhead associated with managing access to the memory system. In fact, the elapsed times for this application are increasing. Although this example represents a worst-case scenario for memory allocation, it illustrates how sensitive the memory allocator is to the number of simultaneous requests being made of it. In an effort to reduce the contention on the memory allocation system, modifications were made to the allocator.

The modifications consisted of dividing the memory allocation pool into multiple parts. For example, instead of having a single pool of memory to allocate from, this single pool is
divided into several pools. The number of pools to use is a configuration parameter of the Raven system, but typically the number of memory pools equals the number of processors in use.

Figure 5.6 shows the changes in processor efficiency and the proportion of memory allo-

![Graph showing processor efficiency and memory allocation delays vs. Processors.](image)

**FIGURE 5.6** Processor efficiency and memory allocation delays when multiple heaps are used.
It is expected that contention can be reduced by increasing the number of memory pools. This hypothesis was substantiated by increasing the available memory pools to 20 and using 5 processors. In this test, memory request delays were reduced to 2.86%. Although more memory pools decrease contention, they do not come without a cost. Each memory pool has a number of data structures associated with it, and these must be initialized. The result is that as the number of memory pools increases so does the administrative overhead associated with the initial start-up and continued management of the memory allocation system. The initialization sequence is single-threaded, thereby reducing the overall efficiency of the application. For long running jobs, this is not an issue, as the start-up time will be dwarfed by the total running time. With shorter jobs, such as the examples used to check for memory contention, the additional initialization costs may more than offset the potential performance gains. Again, this is a situation where a memory allocation system designed for use in a parallel environment should be able to offer an important performance improvement.

In Figure 5.6, the proportion of memory allocation requests that delay peaks at 4 processors and then drops and never reaches the same level of delays again. Further instrumenting of this application provided additional insight into what was happening. It was determined that the memory allocation system was sensitive to the size of the memory pools. A contributing factor is that the pool size can affect how often garbage collections are performed. For example, if the same application is working with a one-megabyte pool one time and two-megabyte pool another, the amount of memory that can be allocated between collections is greater for the two-megabyte pool. The memory allocation system attempts to limit this problem by dynamically adjusting the size of the memory pools. However, it is not successful under all circumstances.

To illustrate the problem, compare the memory pool from which memory is allocated to a barrel. As memory is allocated, objects, representing the allocated memory, can be dropped into the barrel. When the objects in the barrel reach a certain height, a process comes along to check all the objects to see if they are being used (referenced). Objects which are not referenced are removed, thereby reducing the level of the objects in the barrel. The difference between the
height when the checking is finished and the level where checking is initiated will affect how long the application runs before a collection takes place. If the height after checking does not drop below a specified level, a new larger barrel is acquired and the objects transferred to it. This corresponds to the memory management system expanding the memory allocation pool.

Under certain circumstances this approach results in the system entering a thrashing phase where few memory objects are allocated between requests, and the increased number of garbage collections causes performance problems. To get into this state, the following conditions need to be met:

1. The rate of memory requests needs to be high.
2. Allocated memory soon becomes garbage.
3. The base memory usage is just below the level where a heap expansion would take place.

When these conditions are met the system will spend a great deal of time garbage collecting.

When a memory request is made, a pool is selected to allocate the memory from using a round robin strategy. The pool is then locked while the allocation is made. If a selected pool is currently in use, the requesting thread will block, waiting for the pool to be freed. When a garbage collection takes place, the allocation pool is locked much longer than for a straight allocation, and requests will start to block waiting for this heap to be freed. The problem is most pronounced when the memory request rate is high. Under these conditions, requests tend to bunch up with a resulting drop in parallelism. It is this phenomena that was occurring around processor 4 in Figure 5.6. When an additional processor is used, an extra heap is also added. The extra heap and processor combine to change the allocation patterns and heap sizes so that there are fewer garbage collections. Fewer collections result in fewer memory allocation delays and the amortization of the administrative costs of doing a collection, with a resulting improvement in system performance.
Ideally, heaps should be large so that no garbage collection has to be performed. For most applications that is not possible, and a compromise between the frequency of garbage collections and the duration of the collections needs to be established. For example, large pauses for a huge garbage collection may not be acceptable when the application involves a communications protocol, whereas a large number of smaller collections may be fine. The best pool size to work with varies on an application-by-application basis.

Further work was not done on improving the memory system due to time constraints. By making the modifications outlined here, the parallel constructs of the Raven system can be properly demonstrated. In general, performance issues and quirks associated with garbage collection have been left for future work.

5.3 The Thread Environment

The Raven virtual machine defines the low level operating system interface that the runtime system uses. The runtime system is based on a threads package developed at UBC [80]. The basic threads package supplies a basic set of services for process creation and management, memory management, and message passing. The message passing functions are Send/Receive/Reply. This section describes the modifications and extensions to the Raven runtime system to support parallel and distributed processing. How Raven uses these services to implement companions, early result and delayed result is also described.

In its simplest form, parallelism can be supplied by providing access to system functions to explicitly create processes. The Raven language, instead, provides direct syntactic support for creating parallelism through companions and early result, thereby hiding the system-specific way of creating process from the application. The Raven runtime system, however, needs to have access to the process creation services to support Raven's facilities for parallel programming. Since several of Raven's system classes (e.g. Certificate, CertificateGroup and InvokeStream) need to work with threads of control, and to have as much of the Raven system written in Raven as possible, it was decided to create a Thread class to provide an object inter-
face to the underlying thread creation and management functions. Additionally, an executing thread needs a language-level interface to be able to perform control operations, like `sleep()`, on itself.

### 5.3.1 The Thread Class

The Raven system is implemented using a layered approach. Each layer of software provides a certain functionality and typically makes use of only the layer directly below it. Figure 5.7 shows the layers of the Raven system from the hardware up. The lowest layer depicted is the hardware. On top of the hardware is the threads subsystem. This subsystem provides the lowest level of process (thread) creation, management, and control. Unlike other layers in the Raven system, the runtime layer actually knows something about the Raven class library, which is the layer directly above it. The runtime layer knows about the threads subsystem and how to interact with it, and it knows about system specific objects. As a result, the runtime system, in conjunction with the Raven compiler, provides a “bridge” between the threads subsystem and the higher level Raven abstractions. To accomplish this bridging, the runtime system makes assumptions about the objects that exist in the layers above it and about the methods these objects support. The layers above the runtime system are the class library, which supplies a set of predefined Raven classes, and the actual Raven applications. In addition to providing predefined classes, the class library also implements system classes. System classes are those
classes which provide some sort of basic service or require the runtime system to provide a service being mapped to the Raven object world.

The Thread class provides an object interface to the process creation and management services of the threads subsystem. Within the threads subsystem there are four categories of threads:

1. System processes (threads). These threads typically provide services to the Raven system; the communications manager is an example of a system process.
2. User level processes (threads). These are the processes that form the application.
3. User companion threads (Section 4.6.2). These are threads spawned from a user process using Raven's certificate and companion support.
4. System companion threads. These are the same as user companion threads, except they are started from a system process.

Any threads associated with managing or monitoring the Raven system are system threads. User threads, including companion threads, are threads created by an application. When a Raven environment starts, there is exactly one user thread, the one executing the application. Other user threads are created on demand by the Raven system to handle early reply, companions, or to service remote invocation requests.

Each thread is managed by the underlying threads package. This means that if a thread needs to use the process management services (e.g. sleep()) this request must be communicated to the threads package, which has a C interface and not a Raven one. This problem is overcome by providing a Raven interface through the system class Thread. Each application-level thread has associated with it an instance of the class Thread. From an application, the following steps need to be performed to create a new thread of control:

1. Instantiate an instance of Thread.
2. Start the Thread executing by invoking the start() method. The start() method takes as parameters an object ID, the name of the method to invoke on the object, and the parameters for the method.
It is only when the `start()` method is called that the underlying threads package is called to create a new process and start it executing. The most commonly used methods of the `Thread` class are:

- `start()`. Start execution of the `Thread` on the provided object with the given methods and arguments.
- `sleep()`. Suspend execution of the `Thread` for the supplied time. One thread of control may not issue the `sleep()` method on another thread of control.
- `suspend()`. Suspend execution of the target `Thread`.
- `resume()`. Resume execution of the target `Thread`.
- `kill()`. Terminate the target `Thread`.

Other methods are supported by the `Thread` class but they are used internally and not generally available to the programmer.

In Raven, it is the language-level features of companions, early result, and delayed result that use the process-creation facilities of the threads package. Processes created by the threads package as part of a Raven application are mapped to `Thread` objects. To perform this mapping, the threads package was modified to allow application-level data to be associated with a thread. The runtime system uses this feature to map system threads to a corresponding Raven `Thread`.

### 5.3.2 Companions

In Raven, a companion is a collection of method invocations to be run in parallel. A companion is identified by a `Certificate`. Each invocation within a companion (Figure 5.8) is allowed to run in parallel with each other and the initiating thread. In Raven this is implemented through a `CompanionThread`. A `CompanionThread` is a subclass of `Thread` and differs from `Thread` only in its added support for `InvokeStreams` (Section 5.4.6). In the threads subsystem there is a corresponding companion thread process type. It is like a regular thread except there are limits to how many companion threads may be active at once (Section 4.6.2). Figure 5.8 shows a sample Raven companion containing two invocations along with a pseudo-code description of the
// Raven code
l{ res1 = workstation1.displayTime,
    res2 = workstation2.displayTime() }l.start();

// Pseudo-code illustrating the underlying use of the Raven system
- create a Certificate
- create a CompanionThread
- add the companion thread to the certificate
- indicate to the companion thread the object, methods, and arguments for the invocation on workstation1
- create a CompanionThread
- add the companion thread to the certificate
- indicate to the companion thread the object, methods, and arguments for the invocation on workstation2
- invoke the start method on the certificate

FIGURE 5.8 Actions undertaken upon companion creation.

actions undertaken by the Raven system to support the companion. Since all companions are identified by a Certificate, the first action in setting up a companion’s execution is the creation of its Certificate. If the Certificate had a tag value assigned to it, it would be set at this point. For each invocation within the companion the following steps are performed:

1. Create a CompanionThread.
2. Add the CompanionThread to the Certificate.
3. Invoke methods on the CompanionThread to indicate the object, methods, and method arguments to perform the invocation with. The location of where to put the returned result, if needed, is also specified.

Once all the statements comprising the companion have been processed, the CompanionThreads can be started. This is done by invoking start() on the Certificate or pushing the Certificate into an InvokeStream, where the start is performed implicitly.
Companions, in Raven, are new threads of control. They are created through the thread package’s process-creation services. Simple process creation can provide most of the functionality needed for companions, but not for companion threads. To support companion threads, a finer degree of control over process creation is required. This control is provided through modifications to the process-creation routines. Companion threads are supported by providing the ability to classify the processes being created, and by creating the process in the initially suspended state. The classification type is needed so that limits on the number of companion threads can be enforced. New threads need to be created in the suspended state to permit Raven level Threads to be attached to the new processes. Once the attachment is made, the controlling of a thread’s execution is managed through Thread objects. When a companion thread is created in Raven, it is initially suspended and the companion thread object ID is returned to the application. The application then issues the start() method on the OID to initiate the thread’s execution.

### 5.3.3 Early Result

An early result differs from a companion in two ways:

1. The new thread needs a copy of the execution state of the method. In particular this means that copies of all local variables, including parameter values, are required.

2. It may need a lock on the object instance data.

When an early result is performed, a new thread of control is introduced where the early result is performed. The initial thread of control returns to the calling object, and a new thread continues execution in the current object. Since the new thread is executing instructions after the result statement, it is expecting to access all the parameters and local variables that were active prior to the early result. To supply the new thread of control with a copy of the execution state, a limited Fork() function was added the threads subsystem. This Fork() function was designed specifically to support early result and takes advantage of the fact that stack frames do not contain pointers to other stack frames, and that only the stack frame of the method doing
the early result needs to be preserved. This makes the restricted `Fork()` simpler to implement and quicker to execute, since it has less work to do than a full-featured `Fork()`.

Figure 5.9 shows a Raven code fragment taken from Figure 4.1, along with some pseudo-

```plaintext
// Example Raven code showing early result
return_code = message.validFromField();
result return_code;

// Pseudo-code to describe actions to perform for early result
if (early result hasn't been done before) then {
    - indicate that early result has been done
    - attempt to acquire any needed locks for the new thread
    if (could not get lock) then
        - compute return result and save it
    else { // Got the needed locks so can do early result
        - perform Fork()
        if (in parent process) then {
            - compute return result of method
            return result
        } else { // In the child process
            - create a new Raven Thread object
            - in user setable data of this process record the
              object ID of the new Thread object
            - change the session ID in any granted locks to be
              this thread's session ID.
        }
    }
}

FIGURE 5.9   Pseudo-code illustrating early result
```

code to outline the steps the system takes when an early result is done. Only one early result can be done during each invocation, so the first thing the code for an early result does is check to see if an early result has already been done. If an early result has already been executed, then execution continues as if the `result` statement was not executed. If an early result has not
been performed, then processing continues in an attempt to start a new thread of control. The next step is to acquire any locks the new thread of control might require. It may be that both the new thread and the existing thread may need locks on the object, and these locks could be in conflict, so a call is made to the lock management routines to determine whether or not locks required by the invoker and the new thread can be held simultaneously. If there is a lock conflict, the return result is computed and saved for the actual return and execution continues to be single threaded. If, however, the locks can be granted, then a new thread of control can be started and a call to Fork() made. When Fork() returns, the parent computes the return result and returns. The child process then needs to create a Raven Thread to associate with its thread in the thread subsystem. After creating the Thread object, a method is invoked to attach the system-level thread to the Raven Thread object. Within this method the process ID of the thread is recorded in the object's instance data and the user setable instance data in the system thread descriptor is set to the object ID of the Raven Thread object. Once this is done, the new thread proceeds to execute the code following the result statement.

5.3.4 Delayed Result

A delayed result is different from a companion or early result in that it does not create a new thread of control. Instead, it assumes that there are already multiple threads of control within the Raven system and that if one thread of control needs to wait for certain conditions to be met then it can do a delayed result, thereby suspending itself. Another thread of control will return a result for it at a later time.

A delayed result has two parts two it to it. One is the method doing the leave and the other is the method that will ultimately complete the delayed result with the result statement. The result statement takes an argument in square brackets that indicates which waiting thread to complete along with the return result. (Without a square bracket argument the thread to reply to is considered to be the calling thread and the result is an early reply.) Using Figure 4.2 as a base, Figure 5.10 excerpts the relevant Raven code and provides a pseudo-code description of the actions the system performs for a delayed result. If multiple threads have out-
standing leaves within the same object, the method(s) performing the completions need to be able to determine the thread of execution to complete. The way an application accesses the underlying threads is through objects of the Thread class. Each system-level thread has associated with it an instance of the Raven class Thread. A method can get the OID of the Thread it is running in through the reserved word me. Once a method has its thread OID, the OID can be stored and a leave performed. The first action undertaken as part of the leave statement is the releasing of any locks this Thread is holding on the object. This ensures that another thread can get any locks it needs on this object so that the completion of the delayed result can be accomplished. Once the locks are released, a call to Receive() is performed to wait for the result. Receive() is a blocking operation, and when it completes the receiver will have the result to return as the delayed result.

The method doing the completion uses the stored Thread OID to specify the underlying thread to send the result to. There is the slight possibility that the thread being replied to has not performed its receive yet. This is not a problem as the sender will block, waiting for the receiver to become ready.
The threads package is a realization of a virtual machine description that the Raven runtime system is targeted to. When Raven is ported between different architectures, only the threads package needs to be modified, since it is the only piece of Raven software that interacts directly with the host system. The only part of the threads package important to the runtime system is the interface; therefore, considerable flexibility is available when a port is done. This flexibility was demonstrated in the port of the threads package to Mach 3.0 [70]. Mach’s support for thread creation and management is very similar in functionality to the services provided by Raven’s thread package. Consequently, instead of porting the threads package directly, the interface of the threads package was mapped to the Mach system calls through wrapper functions.

5.4 Remote Invocation

In the previous chapter the general design approach used to achieve transparent method invocation was described. This section describes Raven’s support for remote invocations that helps realize this transparency. To actually accomplish a remote method invocation, consideration was given to the following issues:

1. What relationship has to exist between class hierarchies on different machines?
2. How are problems associated with heterogeneous machine environments addressed?
3. How are the remote objects located (found)?
4. What mechanism needs to be used to exchange data between Raven environments?

Before elaborating on these points consider Figure 5.11, which details the steps of a remote invocation. In this figure rectangles represent objects and circles represent processes. The remote invocation is started by an initiating object in environment $A$ performing an invocation on a proxy. The method lookup routine for the proxy then calls a remote request handler routine. This routine marshals the request, contacts the local communications manager process, and instructs the communications manager to send the data comprising the request to the envi-
environment where the target object resides. The communications manager contacts the remote communications manager process and together they ensure the reliable delivery of the remote request handler's data. Once the remote communications manager begins receiving the data, it starts the remote worker process.

The remote worker is responsible for examining the incoming data stream to determine what should be done with the request. In the current implementation, the only request expected is a remote invocation request. Any other request type is considered an error and ignored.

The remote worker continues to accept data until a completed request is received. A request is completed when the target object has been identified, the method to execute specified, and all the arguments received. The de-marshalling of arguments might require proxies to be created or objects to be copied. The remote worker, which is operating as part of the runtime
system, assembles the incoming data into the format of an invocation request and calls a method invocation routine to lookup and execute the method. When the method completes, a result is returned to the remote worker and the information concerning the returned result is propagated back to the initiating object.

5.4.1 Class relationships

Each Raven environment has within it a class hierarchy. The class hierarchy is accessed for all the local actions associated with objects, and it plays a special role when a remote request is received at a Raven environment. As part of a method invocation, an invoker may pass as parameters references to other objects in the system, or an actual copy of the object. A copy is passed if the copy option was specified for a parameter during the method declaration. When two Raven environments interact there is the potential for the class hierarchies to be in conflict. A conflict results when a class does not exist, classes have the same name but do not describe the same type of object, or if the method signatures for a class are not identical.

The problem can be highlighted by considering what happens when a global ID is received in a Raven environment. This GID needs to have a proxy constructed for it, and the proxy must be associated with a class so that its is-a relationship can be established. The class name of the object is sent as part of the GID, so that the is-a relationship can be established. Copied objects also rely on the remote site class information to provide a proper description of the object's instance for reconstruction purposes and to provide the actual method code that can be run on the copied objects. In constructing either a copy or a proxy, the class is first looked up. If the class is not present on this machine then a problem exists.

One solution to a missing class is to ask the class hierarchy where the original object resides for the needed information. It might be possible to splice this information into the class hierarchy if only proxies existed; however, in the case of a copy, the method code would also have to be transferred. This presents numerous problems with respect to locating and transferring the actual object code for the methods and then splicing it into the target system. Addi-
tional problems would also be expected if this were done in an heterogenous environment with different machine architectures. Simply passing enough of the class description to allow the site to perform method dispatch also would present a problem. An object could be asked for its class and then a new( ) operation performed on the class. In this case, the method's code would be needed, so passing a method signature is not an acceptable solution. Numerous variations on this approach to solving the problem are possible. For example, one could have a central class hierarchy that all environments refer to. This solution is best suited to a homogenous environment connected via a local area network. The solution becomes less feasible when the machines are heterogenous and possibly physically separated, such that updates to the class hierarchy from a central repository would be impractical. Additionally, the simple grafting of a class into an existing class hierarchy will not ensure that the parent classes are compatible.

One way to deal with this problem would be to have all environments have the same class hierarchies. This would ensure that copies could always be made and proxies constructed, but it means that the resulting environment is the union of all the Raven environments involved in the distributed computation. The problem with this approach is that it places the onus on the programmer to make sure that the Raven environment being constructed has all the needed classes. It also makes it difficult for environments not originally intended to be involved in the computation to successfully become part of the distributed computation.

A slight modification to this last idea is the one used in Raven. Raven encourages typing, both for instance data and for parameters. This greatly reduces the probability that a GID for an object will enter an environment where its class is not known. This means that it is not necessary for every environment to know about all the classes, only the ones that it will use directly. It is also assumed that classes with the same name have the same organization of instance data and the same method signatures.

This does introduce the small possibility that a GID will be delivered to an environment that does not know how to construct a proxy for it. Since the assumption being used with this approach is that each environment is self contained with respect to its class hierarchy, then it is
not an option to obtain a class description from elsewhere. One way to deal with this problem is to create the proxy as an instance of class Object with the expectation that no methods that the class Object does not know will be used on the proxy. Although this may be acceptable for object references, it is not an acceptable approach for copy parameters. In the copy scenario, information about the copied object is lost. In particular, the information about what class the object is a member of is missing. This missing information could be detrimental to the correct operating of the object, especially if the object copies itself or is passed as a parameter to another Raven environment. Given the nature of the problems associated with making a proxy or copy be a member of Object, the decision was made to return an error indication to the invoker. The fact that a class does not exist will be discovered while the remote worker process is decoding the request and before an attempt is made to execute the request. Consequently, no changes to the normal method dispatching software are required. With Raven’s strong typing, the arrival of an unknown object type at a Raven environment probably indicates some sort of programming error. In an environment where distributed applications are directly derived from programs that run in a single environment and have as one of their aims the attempt to harness the processing power of distributed idle machines, this is a reasonable assumption.

However, as applications become less tightly coupled, and larger both in scale and physical dimensions, the assumption that the receiving Raven environment should know all about the classes associated with the objects delivered to it is not reasonable. In a client/server model, for example, the server could be providing an OID storage service for clients on the networks. A mail service is a prime example of this. Both the sender of a message and the receiver of a message may know how to deal with objects forming the body of the message. There is no need for the intermediate message transfer agents to know anything about the objects forming the body. Addressing the problems associated with this scenario have been left for future work.

The three remaining issues that have to be addressed in order to support distributed computations are: dealing with heterogenous machines, locating objects, and transferring data. In broad terms these issues are all related and each forms a small part in the mechanism used to
establish a communications path between Raven environments. Within the Raven environment the communications managers (CMs) are responsible for managing network access. The CM accepts packets for transmission and sends them to the destination machine. On the remote machine the CM accepts the packets from the network and then, based on a protocol type field in the packet header, passes it to a routine that knows how to deal with packets of this type. The protocol handler examines other fields in the packet and determines what Raven process this packet is destined for and arranges for delivery of the packet to that process. In Figure 5.11, the remote worker and the remote request handler are responsible for managing the stream of messages between the two Raven environments. These two processes manage a reliable streamed request/response protocol between each other by breaking a stream of data into packets that the communications managers transfer. The result is an invocation protocol with at-most-once semantics. The resulting protocol is functionality similar to other remote procedure call protocols [98][56]. If subsequent performance measurements of the Raven system identify problems with remote interaction, a high performance remote procedure call facility could be adapted for use in Raven.

Figure 5.12 shows the format and fields transferred as part of an invocation request. The data can be broken into three parts: information associated with invoke streams, information about the actual request, including the target object, and information associated with the parameters. All the fields of this request are encoded/decoded using XDR [98].

The first part of the request deals is used to support the InvokeStream class. The invoke stream data is always present and associates this request with a particular invocation stream. Special values are assigned to the invoke stream fields if this request is not part of an invoke stream. Further details on the implementation of invoke streams can be found in Section 5.4.6.

After the invoke stream information, data about the actual remote request is presented. The fields associated with managing the request are:
• Target Object GID. This is the GID of the object the method is to be invoked on. Since the target object is on the recipient machine, the GID will specify the local object that is the target of this request.

• Method Name. This text field contains the name of the method to be invoked. In the local case method names are represented by a hash value, but in the remote case the actual method name is sent. This allows different Raven environments to use different hash functions to compute the method ID, or to use an entirely different technique for dealing with method dispatching.

• Session ID. Session IDs are used in the support recursive method invocations that cross machine boundaries (Section 4.5).

• Return Type. This is the return type of this method invocation. The type is not a type as defined in the Raven language, but a low-level type that indicates the format of the data being returned. The possible return types are integer, copy, and capability. If the type is integer, then a primitive integer is being returned. If it is copy then a copy of the object is to be returned, and if it is capability the GID of the object is returned.

**FIGURE 5.12** Example of the data fields for an invocation request.
• Parameter Count. This is the number of parameters being sent. This allows the application to determine when all the data for this request has arrived.

• Parameter Classification. This field is similar to the return type field. Each passed parameter is classified as to format, using the same classifications as Return Type, with some minor extensions. (These extensions are described later.) This field is always paired with the next field, and together they provide all the information needed to decode a parameter. These two fields are repeated in the remainder of the request until Parameter-Count parameters have been decoded.

• Parameter classification specific data. This is the actual parameter data to decode.

Some of the information passed in the request could be derived by looking up the method and examining the appropriate data fields, but that is not done. It was decided to encode the information directly into the request to avoid duplicating the method lookup steps. Additionally, not all Raven method invocations have a fixed number of parameters. When a method with a variable number of parameters is invoked, the receiving site must be prepared to decode all of the parameters and make them ready for use. In this case the required information cannot be obtained from the method description and has to be contained in the remote request. In the interests of keeping the remote request processing as uniform as possible, parameter typing information is always passed in the request.

Each passed parameter is sent in two parts with the first part being the parameter classification field described previously. The first part of the parameter specification is the type of parameter being passed. This indicates whether the parameter is an integer, a floating point number, an object ID in the form of a GID, or a copy. Floating point numbers and integers are decoded appropriately, and global IDs are converted to a local reference either by matching the object directly or through a proxy. When an object is passed as a copy parameter, the object and its instance data are recursively copied until all the instance data for the object has been sent. The copy operation is a deep copy that will cross machine boundaries if needed. Copying an object consists primarily of identifying the class the object is from, specifying the properties of the object, and encoding the object’s instance data. With this information, an object of the appropriate class can be constructed and the passed instance data to placed into the new object.
Once all the parameters have been decoded the remote site has the following information:

- The local object ID of the target.
- The hash value of the method to invoke as derived from the method name that was passed.
- The local object IDs for all the parameters. (Some of the local IDs may reference proxies.)
- The session ID.
- The format of the return type.

To properly handle locking, the session ID of the thread that has decoded this request, and will ultimately execute the request, is set to the passed session ID. Once the session ID has been changed, an invocation request is constructed and a method invocation routine called to execute the request. When the request completes, the return type is used to format the response and the result is returned to the initiating site.

### 5.4.2 Avoiding Copying Loops

When an object is being copied, special attention must be given to proxy objects and to dealing with circular references. An example of a circular reference occurs when an object’s instance data contains a reference to itself. Attempting to perform a recursive copy operation in this situation would have disastrous results, since the copy operation would not terminate. As part of the copy operation, the instance data of the object must be marshalled. For a proxy there is no instance data held locally so the data cannot be marshalled. To get the data for marshalling, the site where the proxy resides needs to be consulted. This will result in a copy from the real object site to the proxy site, with the proxy site then copying this information to the site of the method invocation. Such an approach incurs needless copying. Instead, a proxy targeted for copying is tagged for remote copying within the request, and the GID for the proxy is transmitted. The receiving site detects this, and if the object is resident at the receiving machine, the copy can be done locally. In the worst case the receiving site simply asks for a copy from the site where
the real object exists, thereby saving a copy and network access compared to having the initiating site perform the copy.

When an object is being copied, it may contain references to either itself or to a sequence of objects that eventually reference itself, thereby resulting in a cycle of object references. Blindly recursively encoding these references will result in an infinite loop. To avoid this problem, some mechanism is required to detect the cycle and then communicate this information to the remote site.

A two-level approach is used to avoid cycles. Each copied object is transmitted in potentially two parts. The first part is the object's global ID, followed, if needed, by the information necessary to duplicate the object. On the sending side, before the copying of the top-level object is started, a data structure keyed on the object ID is built. Before an object is encoded, a check of the data structure is made to see if this object has already been sent. If the object has not been sent then the encoder places the object's GID in the data stream and registers the object in the data structure. The GID is then followed by the information needed to copy the object. If the object has been seen before then only the GID is placed in the stream.

On the receiving side, a data structure similar to the one used by the sending side is created at the start of a top-level decoding. This structure is keyed on the sent GID and has associated with it the object ID that resulted when the object was reconstructed. When an object is being decoded, the GID is the first item encountered. This value is looked up in the data structure to see if the object has been encountered before. If the object has been seen, then the local object ID is retrieved and used as the object ID. If the search reveals nothing, then the GID is added to the lookup structure, and a new object is constructed based on the data in the stream following the object ID. After the object has been fully constructed, the new local ID is associated with the GID used in the lookup.

By using this approach, cycles are avoided even if they cross machine boundaries. For a cycle to cross a machine boundary, a piece of copied instance data must be marked as a remote
copy. Upon receiving this data the decoder first checks, using the GID, to see if the object has been copied. If not, it will ask the remote site for a copy, which the remote site will provide. In that copy there could also be a piece of data marked as a remote copy. If that object has already been copied, then when the decoder encounters this item it will find it in its list of copied objects and not make a request of the remote site for a copy. If that object was a part of a cycle, then the cycle will be broken. In addition, any object referenced multiple times throughout one encoding operation will only be reconstructed once at the destination site. In this way, the same relationship between the original objects and the copied objects will be maintained.

5.4.3 Object Encoding

In the discussion of the copying of objects the implied assumption has been that the two sides engaged in the processing of the remote invocation agree upon the storage format of the data being exchanged. In a homogenous environment the possibility of a data format mismatch is often overlooked. However, being designed to operate in a heterogenous environment, Raven has, from the outset, taken these issues into consideration. Some form of external data representation has always been considered. The primary candidates for the external data representation were ASN.1 [59], Sun Microsystems’ XDR [98], and a customized exchange format.

Using a customized format was quickly ruled out as being unacceptable from the perspective of development time and completeness. The ASN.1 and XDR systems available provided support to ease the job of specifying the encoding rules to use at the application level. An important consideration in the selection of the encoding format involved being able to encode into a fixed region (transmission buffer), and then when the buffer was filled notifying the application. The application could then take the buffer and arrange for it to be transmitted to the remote site. Once arrangements for the sending are made, the encoder is allowed to continue its encoding into a new buffer. With this approach, the encoder essentially produces a stream of packets filled with the encoded data. The stream of packets is then turned into real network packets for transmission. On the receiving side, the decoder needs to be able to consume packets in a similar manner. XDR, with some minor modifications was able to provide
this service, while the ASN.1 packages available at the time could not. XDR routines are also more generally available than ASN.1 routines, so issues of portability also helped in selecting XDR as the external data representation system for Raven.

### 5.4.4 Locating Objects

Taken together, the previously described services provide a mechanism to initiate a remote invocation. The only remaining issue concerns actually locating objects at runtime. The key to locating an object is its GID. When an invocation is determined to be remote, the invocation software gets the GID that the proxy is associated with. Since Raven currently runs as a process in a variety of Unix environments, communication between Raven environments is accomplished using UDP. Raven’s own higher-level protocol, implemented using UDP, is designed to support remote invocations. To operate the protocol correctly, the GID needs to provide information on how to contact the Raven environment where the object resides, and, once contact is made, how to identify the object on the target machine. The result is that the GID contains four pieces of information:

1. The host ID, in IP form, of the machine where the target Raven environment resides.
2. The local ID, or port number, of the process that the Raven environment is running in.
3. The actual object ID on the remote machine of the target object.
4. The class name of the object. (This is required so that the receiving site can construct a proxy for the remote object.)

Combined with the class description maintained on the local machine, this information is all that is needed to construct and deliver a remote execution request. As long as an object does not move, which is the case since the current Raven implementation does not support object migration, this scheme will work. As soon as object migration is allowed, some mechanism will be required to permit an object’s proxies to be updated. This might take the approach of a forwarding mechanism or perhaps a more general lookup procedure that allows a site to locate an object when the location specified by the GID does not work.
5.4.5 Name Server

When Raven is being used in a distributed environment, a mechanism is required to allow the disjoint Raven environments to establish communication with each other. Communication at the application level is not established directly by the application, but implicitly by performing method invocations on remote objects. To start this interaction, Raven provides two methods that make it possible for disjoint Raven environments to locate objects in each other’s space.

Within each Raven environment one well known GID is maintained. This GID is reserved and allows the system to provide a mapping from the well known GID to a designated local object. All objects respond to the method `makeWorldVisible()`. When this method is invoked on an object, it causes a well known GID to be constructed for that object and registered within the runtime system. When a remote request is received at a Raven environment, the incoming GIDs are first checked against the well known GID to determine whether the GID matches the well known GID.

For a system to be able to access objects that have been made world visible, a way to create a proxy object with the GID of the remote object is needed. An underlying assumption to this interaction is that the application wanting to communicate with the remote object knows its class. This is not an unreasonable assumption, since the need to access this object is probably based on the desire to use some service, and to use the service knowledge of the class will be required. To create the proxy object the application invokes the method `configureAsRemote()` on the class object corresponding to the remote type. The method will create a proxy for an object of this type using its method arguments to construct the appropriate GID. To construct a GID the following information is needed: an ID that identifies the target host the Raven environment is running on, an ID that identifies the particular Raven environment on the target machine, and an ID that can be mapped to the local object. When `configureAsRemote()` has been invoked, it returns an object ID that identifies the proxy to the remote object.
With `configureAsRemote()` and `makeWorldVisible()` Raven environments can make objects accessible between Raven environments. Each Raven environment is restricted to supporting only a single mapping of a well known GID to a local ID, so this cannot be used as a general name server. By using `configureAsRemote()` and `makeworldVisible()` as a foundation, a more general-purpose name server can be constructed.

The name server in Raven works by designating one Raven environment as the host of the name server. The designated environment is responsible for the creation of an instance of class `NameServer`. The class `NameServer` defines the method interface to the name server object. The provided `NameServer` class supports methods to register a name along with an associated object identifier, to lookup an object based on its name, and to remove a name to object mapping. A Raven environment that wants to use the name server object creates a proxy for the name server object by invoking `configureAsRemote()` on the class `NameServer`. The arguments to `configureAsRemote()` provide the host and environment identifiers needed to construct the proxy. Figure 5.13 shows the steps required on the client and server to create

```java
// Code on server side
name_server = NameServer.new();
name_server.becomeWorldVisible(456);

// Code on a client side
name_server = NameServer.configureAsRemote(REMOTE_HOST, LOCAL_ID, 456);

// The name server is now ready to use.

FIGURE 5.13 Creating a name server.
```

and access the `NameServer`. Once the proxy has been created the application can then register objects it wants other environments to use. The application can also lookup objects that it wants
to use. Queries to the name server simply return object IDs in the form of GIDs. The runtime system will convert the GID to a local ID for use in the local environment.

Currently, the application is responsible for either creating the name server object or the proxy to it. This approach was taken since not all applications will make use of a name server. The determination of the name server location and the creation of a proxy to it are system configuration issues. As Raven matures as a system, the functions associated with the creation of the name server and references to it should be made part of the start-up processing for the system.

5.4.6 InvokeStreams

InvokeStreams are used to sequence requests targeted at the same object when the requests cannot be executed in parallel with each other, but when they can be executed in parallel with other requests (Section 4.3.2). An example of such a sequence of requests is shown in Figure 5.14.

```java
stream.push(!{workstation.positionCursor(cursor_position)});
stream.push(!{workstation.displayData(datal, amount)});
```

**FIGURE 5.14 Example of stream usage.**

In this example, two requests to manipulate a display are executed. These requests can be executed in parallel with the initiating thread, but must be executed serially with respect to each other; therefore, they are pushed into an InvokeStream object. The stream object performs an implicit start on the CompanionThreads making up the companion and ensures that the requests are executed serially relative to each other.

There are two components central to the implementation of InvokeStreams. One is the class InvokeStream and the other is the class Sequencer. InvokeStream provides the application-level definition of the invoke stream service, and the Sequencer object implements the service at the
Raven system level. Within an *InvokeStream*, each unique invocation target object is associated with its own *substream ID*. This ID identifies a stream of requests all targeted at the same object. Individual method requests are assigned a sequence number within the appropriate sub-stream. The sequence number specifies when a method should execute relative to the other requests on this sub-stream. When all the method requests on a sub-stream have completed, that sub-stream is discarded. A new substream ID is used if the target object of the discarded sub-stream reappears. Discarding substream IDs conserves memory and reduces execution time by maintaining only the information currently needed to manage the outstanding streams. When a companion is pushed into an *InvokeStream* the following actions are performed by the `push()` method for each constituent *CompanionThread*:

- Determine the substream ID for the request. A new substream ID may need to be assigned if there are no outstanding requests for the target object.
- Assign a sequence number for the method request.
- Associate the sequence number, *InvokeStream* ID, and substream ID with the *CompanionThread*.
- Issue a method to start the *CompanionThread* executing.
- When the method completes, check to see if the substream ID can be abandoned.

All the actions taken by the *InvokeStream* object simply establish where in the *InvokeStream* sequence a method should execute. The Raven runtime system is responsible for ensuring the proper sequencing of the method invocations; it is the responsibility of the *Sequencer* object to actually sequence the requests.

There is one instance of the class *Sequencer* per Raven environment, and it is created at system start time. The *Sequencer* maintains a list of all *InvokeStream* objects that have requests active in this Raven environment. When a method invocation is to be sequenced, a special invoke routine is used. This routine performs special pre- and post-stream processing. Before the method is executed the invoke routine performs an invoke on the *Sequencer* object to have this request sequenced. The parameters to the request are the *InvokeStream* object ID, the sub-
stream ID, and the sequence number. Control does not return to the invoke routine until it is this method’s turn to execute. When control returns, the requested method is executed and then the post-stream processing is performed before returning the result. The post-stream processing informs the Sequencer object that the method has completed, and that the next method on this substream can execute.

The Sequencer object is implemented entirely in Raven and the stream invoke routine uses the method sequenceRequest() to request that an invocation be sequenced. The method requestCompleted() is invoked by the stream invoke routine once its invocation has finished executing. When the SequenceRequest() method is invoked it checks to see if this request can be executed, and if it can, it returns immediately. Otherwise the Thread object ID of the current thread is stored and a leave is performed. When the requestCompleted() method is executed it checks to see if there is a request that can now be executed. If there is, a delayed result is performed and the waiting request can now execute.

The sequencing of requests is always performed at the site where the target object exists. This means that if a remote invocation is sequenced, the information necessary for sequencing must be passed to the remote site. From Figure 5.12 it can be seen that the first three fields of a remote invocation request contain the substream ID, substream sequence number, and initiating InvokeStream global object ID. If this request does not involve a substream, then the substream ID will be 0. If it does involve a substream, then the other two fields of the request are meaningful and they are decoded appropriately. To have the remote request properly sequenced, the worker process responsible for performing the invocation at the target site uses the special stream-invokeroutine instead of the generic invoke routine.

5.4.7 Lock Management

One of problems of operating in a distributed environment when using locking is dealing with recursive locking invocations that cross machine boundaries (Section 4.5). The general problem of supporting locking is addressed by associating a lock control data structure (Figure 5.1)
with each controlled object. The lock control structure maintains a list of all the threads currently holding the lock, and all the threads waiting to be granted a lock. When a request to acquire a lock on an object is made, the checks are made of the current lock holders to see if the lock can be granted. If the lock cannot be granted the thread is blocked until it can get the lock.

One of the problems identified earlier was dealing with locking when recursive invocations are made. The key to supporting locking with recursive invocations is being able to identify the thread of control a request is coming from. To do this each thread has a globally unique session ID associated with it. Although the thread’s Thread global object ID could serve this purpose, it was decided to generate a different globally unique ID to keep session IDs and GIDs separate. This reduces dependencies between unrelated parts of the Raven code and simplifies future changes if either the GID or session ID formats need to be modified.

A session ID is not confined to a single machine. When a remote invocation is performed, the session ID is transmitted as part of the request and assigned to the worker processes performing the remote invocation. This means that all threads of control, in all Raven environments, that have the same session ID are in the same execution chain. The ability to determine if the locks are being held by the same execution chain, even when the execution chain crosses machine boundaries, is crucial to the management of locks associated with recursive method invocations.

### 5.4.8 Remote Invocation Failures

Section 4.9 discussed the difficulties associated with handling failures and exceptions in Raven. When performing a remote invocation there are two broad classes of problems that can occur. One class of problems can be described as communication problems. This type of problem results from events like network partitioning, machine crashes, or protocol errors. The second class of problems are those associated with actually executing the method. These are problems like being unable to locate the target object, being unable to decode a parameter or
complete a copy operation, or perhaps the invocation itself failing because of a bug in the method code.

If any of these errors occurs, it is detected by the runtime system software managing the remote execution and waiting for the result. The software will detect the error and proceed to execute failure handling code. Currently, the failure handling code does little other than log the error. This is because it is not clear where notification of the failure should be delivered. Additionally, the runtime system signals the application of an error by returning `nil`. In many cases applications will not be checking for this failure so it is not an effective general-purpose way of handling remote invocation failures.

### 5.5 Summary

This chapter has discussed the issues, implementation details, and programming problems that had to be addressed to effectively implement support for parallel and distributed computation in Raven. To do this the object and class organization in the Raven system were discussed, including details of how objects are organized in memory, how classes are organized and how methods are dispatched. A method ID hashing scheme was developed to speed-up dynamic method lookup and to eliminate conflicts during lookup. The scheme to handle remote method invocation was described, with attention being paid to the implementation of proxies and to the copying of objects. The changes made to the threads package to provide support for the Raven runtime system were also described.

The distributed and parallel nature of Raven makes it difficult to determine when objects are no longer in use. A conservative garbage collector from Xerox is used to reclaim storage. This collector was modified to permit multiple memory requests to be granted simultaneously. For the purposes of testing, the allocation system performs adequately; however, under high demand its performance degrades rapidly. The degradation is attributable to the garbage collector, which has a large sequential component and can act as a barrier to other requests.
This chapter provides information about the performance of the Raven system. The chapter starts with a general discussion of the performance metrics used and how the measurements were performed on Raven. The performance results for several problems running on a shared memory multiprocessor are then reported. This is followed by descriptions of a distributed mail application and other miscellaneous applications. The chapter closes with an overview of what the performance results have taught us about the Raven system and a summary of the chapter.

6.1 Performance Metrics

In many parallel programming environments the raw performance of the system, when compared to the best existing benchmark for the same problem in a similar hardware and system software environment, is all that matters. If the new system cannot offer performance comparable to existing systems then it is a failure. Other environments, like Raven, however, emphasize the writing of system level programs where increasing the raw performance of the system
is not as important as it is in computationally intensive scientific applications. What is important is the ability to harness at least some of the extra processing power available in parallel and distributed systems. It is therefore crucial that the overhead of using the system's parallel constructs be small. The cost of using the parallel constructs can be determined by measuring the performance of an application using different numbers of processors relative to the sequential version the parallel application was derived from. By performing measurements in this way, the system overhead associated with using the parallel constructs can be determined, any problems with the scaling of the support software can be revealed, and insight into any system-level bottlenecks obtained. Although these measurements can provide some information on the costs of using parallelism, each application will have its own performance profile, and it is only by developing a large body of experience that the programmer can make an informed decision concerning how to employ Raven's parallel constructs.

To allow the performance of parallel programs to be assessed, some performance metrics need to be defined. This will allow comparisons to be made between various runs of the same problem using different numbers of processors or different algorithms [63]. The metric being used is the running time of the program, which is also referred to as wall clock or elapsed time. This particular metric was selected over others since it captures any inefficiencies or performance bottlenecks present in a program that are not captured in a metric like CPU time usage. The running time also captures any delays, like communications overhead, present in distributed applications but not in sequential single processor applications. The efficiency and speedup of a parallel application can be determined when the running time is known.

As an example of why the running time is a more interesting metric than CPU time usage, consider a sequential program that uses $X$ seconds of CPU time. Assuming there is no overhead for parallelizing the application then the total amount of CPU time used when running in the parallel environment would remain at $X$ seconds, regardless of the number of processors used on the problem. The elapsed time for completing the problem, however, will vary depending upon how successful the parallelization of the problem has been. In the most successful case
the elapsed time will be reduced to \( X/P \) where \( P \) is the number of processors being used on the problem. In practice, this is seldom the case, since some portion of the problem may be inherently sequential, and in general it is not possible to keep all the processors busy 100% of the time. A worst case scenario would have the elapsed time for the \( P \) processor solution be greater than the elapsed time for the same application running on a single processor. If the amount of CPU time used was the metric then the discrepancies between the various scenarios could be absent. The running time of an application is also probably the most meaningful metric, since it tells the users exactly how long they will have to wait to get results. More formally, the running time for a program is given as \( rt(f, m, p) \), where \( f \) is the program being run, \( m \) is the machine, and \( p \) is the number of processors being used.

### 6.1.1 Speedup and Efficiency

Although the elapsed time is interesting, it does not directly provide any information about how well one implementation compares to another. A more useful metric for comparison is speedup [63]. Speedup captures how much faster one program runs relative to another. More formally the speedup of program \( f_1 \) running on \( p_1 \) processors relative to program \( f_2 \) on \( p_2 \) processors on machine \( m \):

\[
speedup(f_1, p_1, f_2, p_2, m) = \frac{rt(f_2, m, p_2)}{rt(f_1, m, p_1)}
\]

To be of any use, the programs being compared should solve the same problem. The most common use of speedup is to compare a sequential implementation of a program with a parallel version (i.e. \( p_2 = 1 \)). Knowing the performance of a parallel program relative to a sequential one is important, because few people will write a parallel program when a sequential one performs as well or better than a parallel one. In addition to providing information about speedup, the comparison between a single processor implementation and a parallel implementation on one processor provides insight into the overhead introduced by the parallel constructs.
The efficiency of a program is a relative measure of performance that compares a program to a specified standard. Efficiency is defined as speedup, relative to one processor, divided by the number of processors. The efficiency, \( \text{eff} \), of a program, \( f_1 \), running on machine \( m \) using \( p \) processors relative to a program, \( f_2 \), using one processor on machine \( m \) is:

\[
\text{eff}(f_1, m, f_2, p) = \frac{\text{speedup}(f_1, p, f_2, 1, m)}{p}
\]

Program \( f_2 \) represents the standard against which the first program is being compared. Depending upon the information desired, the standard may vary. In some situations the standard may be the best sequential implementation, while in others it might be the single processor running of a parallel implementation. In this last case, the metric captures the overhead associated with running the same program using different numbers of processors. An ideal parallel implementation will experience no additional program execution overhead, with the result that the program's efficiency will be 1. Synchronization, communication and shared resource contention typically prevent applications from achieving this, with the result that the efficiency for more than one processor is less than one.

The measurements that will be presented in this thesis are efficiency and speedup, and are calculated using the formulas just introduced. In both formulas, the ratios for speedup and efficiency will be between a sequential implementation and a corresponding parallel implementation using varying numbers of processors. Since one of the goals of this work is to make it easy to convert sequential programs to parallel programs, the parallel programs are derived directly from their sequential ones. For ease of presentation, the speedup and efficiency information is presented graphically.

All the measurements reflect the running time for the true working part of the problem. Being a large system, the Raven environment goes through a number of phases before the actual user part of the application is started. The main phases of interest are:
• The start-up phase.
• The application execution phase.
• Garbage collection phases.
• The shutdown phase.

To provide a realistic measure of the efficiency of the Raven parallel constructs, the running time reported corresponds to the application execution phase. This number is arrived at by measuring the total elapsed time and subtracting the times for the start-up, garbage collection and shutdown phases. The start-up phase involves the setting up of the memory allocation environment and the loading and initializing of the Raven class library. The start-up code is executed before any application code is run. Removing this time from the measurements is reasonable because this start-up time varies from program to program and the code is highly sequential and memory intensive. All of these factors contribute to an increase in running time that does not accurately reflect performance changes due to the use of the Raven parallel constructs. Since the start-up code is executed before the application begins, any changes in the application to support the various degrees of parallelism are properly captured.

Practical constraints also place limits on how long it is feasible to run an application program for testing. Some of the timings for a single processor run take over two hours, while the equivalent multiple processor run takes only a few minutes. In these latter situations the running time is dominated by the start-up time, and, as a result, does not provide an accurate measure of the costs of working in the parallel environment. Ideally, the runs taking several processors should take hours, with the result that the start-up costs would become insignificant. If that were done, however, timing experiments using a single processor would take days to complete, and this was impractical.

The second running time component subtracted from the reported times is the time spent in the garbage collector. Garbage collection occurs when a request for storage cannot be granted without first reclaiming unused memory. When this happens some processing is done while the other threads continue to execute. But at some point all the other processes have to
be stopped so that the marking and sweeping of their data spaces can be done. This stop-the-world approach to garbage collection is not well suited for operations in a parallel environment. A garbage collector designed for use in a parallel environment would reduce or eliminate the need to stop other executing processes for garbage collection related activities, and this would increase the overall efficiency of the application. Subtracting the time that processes are stopped for garbage collection gives a more accurate measure of the performance of the parallel constructs. The performance should also be more representative of the efficiency and speedup expected with a parallel memory allocation and garbage collector system installed.

The performance numbers in Section 5.2 demonstrate that even with a predominately serial memory allocator it is possible to reduce contention within the allocator. Although the existing allocation code does not address the problem of performing storage reclamation in parallel, it does address the issue of acquiring storage in parallel. Improvements in performance similar to those obtained for allocation should be possible for the garbage collection phase, which remains a serial operation in this implementation. The garbage collection time, which includes a large sequential component, is not included in the elapsed time for an application's execution. By Amdahl's law [7] the speedup of a program is constrained by the sequential parts of the program.

An additional problem with garbage collection involves the starting and stopping of other threads during collection. During the stopping phase, parallelism is reduced as active threads are suspended. Likewise during the restarting phase, parallelism gradually increases as the suspended threads are restarted. Other factors to consider concern the fact that as processes are suspended they cannot free resources that other threads might need. This results in some processes waiting for resources and the parallelism being reduced. When processes are restarted, the parallelism gradually increases, but again it is possible that a restarting thread will be immediately blocked on a resource one of the suspended threads is holding. Additionally, while a collection is in progress, that target memory pool is locked. This locking prevents other threads from using the pool and causes threads to block waiting for access to the pool.
CHAPTER 6: Performance

The third phase of an application’s execution that was removed form the timing is the shutdown phase. This shutdown code is executed once the user-written code has terminated. The amount of code executed depends upon the postprocessing the system is performing. In general this overhead is negligible. This is not the case in an instrumented system like Raven. The post processing in Raven consists of extracting system statistics and timings and displaying the information. Some of these operations are quite time consuming and are therefore excluded from the timings.

For the purposes of these measurements, an application performs two types of work. One is the useful work of the application; the other is the overhead, or administrative work, associated with servicing the application’s requests. The ratio of administrative work to real work can be quite different between tests using different numbers of processors. This depends upon such factors as the application’s sensitivity to the memory pool size. In one situation the application might run the garbage collector several times, while in another not at all. The addition of another processor can change memory allocations so that garbage collections are more frequent. Consequently, one application can pay a much heavier administrative cost for memory allocation than the another. Again, changes to the memory allocation system can eliminate this problem.

6.2 Example Problems

To demonstrate the usefulness and practicality of the Raven support for parallel and distributed computing, several example applications were programed. There are two major operating environments that Raven runs in. One is a collection of Sun SPARCstation class machines and MIPS machines connected via an Ethernet [74]; the other environment is a 20-processor Sequent Symmetry 81, courtesy of the Department of Computer Science and Engineering at the University of Washington in Seattle, Washington. The Sequent is a shared-memory machine that utilizes 80386 microprocessors running at 20MHz, and it is running Mach 3.0
with local modifications to support the shared-memory multiprocessor environment of the
Sequent.

Each of these environments presents different properties to the application and as a result
each highlights something about the Raven environment. The Sun machines are distributed
around the department and suffer the vagaries of being on the network. The machines are not
isolated enough to be able to provide a consistent environment from one run to the next. People
can login to machines, unsolicited requests can be made of a machine, and the traffic load on
the network can undergo dramatic changes depending upon what other machines on the net-
work are doing. Compared to the immediacy of local invocations, the latency for remote object
invocations can be substantial. The distributed-memory environment does have the advantage
that the contention for storage requests is greatly reduced in the situation where the applications
running on the nodes are essentially single threaded. However, if the amount of concurrency
on a single node is increased, similar problems with memory contention will be introduced.

The shared-memory environment of the Sequent is more controlled, and as a result pro-
vides a more consistent environment to conduct measurements associated with how well the
parallel constructs within Raven are capable of scaling. It is for this reason the only reported
results are for a shared-memory environment. As pointed out earlier, this is not without its own
problems, as the system resources often do not scale as processors are added. For example, the
system being used has 32 megabytes of main memory, and that remains fixed. With a single
processor this amount of memory may be perfectly adequate for some applications. However,
when 20 processors are used the amount of main memory per processor is grossly reduced. The
resulting amount of memory per- processor may not be enough to execute the application pro-
cess in a reasonable manner. Consider that if the application is memory-request intensive it
would be using up memory twenty times faster than a single processor, yet the system config-
uration has not changed.
6.2.1 Mandelbrot Set

The first example selected is a Mandelbrot set computation. It is an easy to problem to parallelize and it supports a high degree of parallelism. By choosing a straightforward problem to parallelize, the goal was to immediately highlight any fundamental problems in the Raven subsystem and runtime system support for parallelism. Since the application is method and computation intensive, any problems with the method dispatch system or with the thread scheduling procedure would be quickly exposed. In essence, the Mandelbrot set computation provides a measurement of the basic level of overhead that the Raven system imposes when managing multiple threads of control on multiple processors. Since this problem has minor memory requirements, any potential contention areas associated with the memory management system are avoided. Although this problem may seem unduly simple, it has the same form and structure as such problems as ray tracing in graphics.

6.2.1.1 Problem Description

A simple description of the Mandelbrot computation is that it is a computation that applies a function to a number in the complex plane. A region of the plane is divided into a grid and the function is applied to each grid point. Each computation is independent of the other computations; consequently, it is possible for all the computations to be performed in parallel. This computation is an easy problem to parallelize and therefore useful for developing experience on working with Raven in a parallel environment.

The programming approach for this problem decomposes the computation into three major object components. One object is responsible for displaying or recording the data, one or more objects are responsible for the computation, and one object decides what points are to be computed. A subset of the Raven code used to perform the Mandelbrot computation on the Sequent multiprocessor is shown in Figure 6.1. Two versions of the class Main are provided. The first is a sequential version of the Mandelbrot computation. The second is the parallel version.
When the Mandelbrot application starts, it first creates an instance of the `DisplayManager` object. The `DisplayManager` object encapsulates information about the region that the Mandelbrot set is being computed over, and it is responsible for displaying the computational results. The actual computation is performed one line at a time through a `Worker` object. When the `Worker` object has the `execute()` method invoked, it first contacts the `DisplayManager` to get display information. This information consists of data describing the spacing of the lines and grid points over the computation region and the number of colours to use. Next, the worker cre-
ates a Mandel object to perform the detailed Mandelbrot computation of a line. The worker object then continually queries the DisplayManager object for a line, computes the results for the line and returns the results to the DisplayManager. This continues until there is no more work to be done.

In a sequential computation all the start() method does is instantiate a DisplayManager and Worker object. Invoking the execute() method on worker is sufficient to compute the Mandelbrot set over the region defined in DisplayManager. A version of the code to perform the computation in parallel is shown in the second implementation of the class Main in Figure 6.1.

Parallelism in this application is achieved by creating multiple Worker objects and performing the invocation of the execute() method inside a companion. The creation of the companion and the check with the DisplayManager, along with a way to specify the number of companions to use distinguishes the parallel computation from the sequential computation. In this example, the application relies upon Raven’s automatic control of process creation (Section 4.6.2) to restrict parallelism to the number of available processors. This, however, requires the addition of a call to the DisplayManager to see if there is more work to perform. Another approach would have been to pass, as a command line argument, the number of worker processes the application was to use and then having the application create only that number of workers.

6.2.1.2 Performance Results

Figure 6.2 provides the graphical representation of the performance results for the running of the Mandelbrot set computation on from 1 to 20 processors. On the horizontal axis is the number of processors used by the computation and the vertical axis has the speedup factor. The dotted line represents perfect (maximum) speedup and the solid line is the measured speedup. For this particular example the speedup is quite good with only a slight decrease in efficiency as extra processors are added. The gap between the measured speedup and perfect speedup is the overhead associated with running this problem. The efficiency of the solution ranges from a maximum of 0.98 to a minimum of 0.93 with, in general, the efficiency decreasing as the num-
ber of processors working on the problem increases. A drop in efficiency is expected since the administrative overhead associated with adding and managing processors will increase as more threads of control compete for access to common system management resources, such as the scheduling queues and message passing services.

### 6.2.2 Bayesian Search

Another problem that demonstrates the use of Raven’s parallel constructs is an application based on the work of Poole [84]. The particular problem example consists of performing fault diagnosis on a multiple-bit adder constructed from a sequence of 1-bit adder units.
6.2.2.1 Problem Description

A subsection of a multiple-bit adder circuit (also known as a ripple-carry adder) is shown in Figure 6.3. This figure shows two connected 1-bit adders and each adder consists of four gates, each with two inputs and one output. Each gate in this circuit can fail and the purpose of this application is, given a known input and an incorrect output, to determine the most likely failure scenario for this circuit that produced the observed results.

Each gate in the circuit has a probability of functioning in a particular way. The operational states that a gate can be in are:
1. The gate works correctly with probability $P$.
2. The gate is stuck and always returns a 1 with probability $N$.
3. The gate is stuck and always returns a 0 with probability $N$.
4. The gate fails in an unknown way and returns a 0 with probability $U/2$.
5. The gate fails in an unknown way and returns a 1 with probability $U/2$.

Given these possible operational modes the task is to determine the most likely failure scenario given a particular set of inputs and an observed error output. The total number of possible failure scenarios is defined by:

$$\sum_{i=1}^{C} \frac{C!}{i! (C - i)!} F^i$$

where $C$ is the total number of components of interest and $F$ is the number of ways a component can fail. In this example all the components have the same number and type of failure modes. Each element of the summation represents the number of failures for a particular grouping. For example when $i$ is 1 the number of 1 component failures is determined, when $i$ is 2 it is the number of two component failures and so on. As is evident from the formula, the number of failure scenarios grows quickly as a function of the number of components. For a 128 bit adder with 640 components and 4 failure scenarios per component the total number of failures that need to be checked is greater than $2.9 \times 10^{89}$. Obviously, it is not feasible to check all these possibilities, and some strategy is required to test the most likely candidates.

Since it is impossible to check all the failure scenarios, a central server approach is used to direct the search into the areas determined to be the most likely to produce useful results. Basically, the central server is responsible for determining the next failure scenario to test based on criteria selected by the programmer. The result is that the implemented solution has two major components to it. One is the search space manager and the other is the worker object that
does the actual testing of a failure scenario. This approach also has the advantage that it is easy to modify the search strategy by simply changing the search space generation object. This approach also lends itself well to parallelization, as multiple error testing objects can examine different error scenarios concurrently.

Additionally, this solution relies heavily upon the object-oriented approach of using building blocks to construct applications. In the implemented solution, a ripple-carry adder is composed of a collection of identical 1-bit adder circuits connected to form a complete adder unit. Figure 6.3 contains two 1-bit adder circuits. In the Raven implementation each gate within the unit is also an object. A circuit is then constructed by creating gate objects; the gate objects have methods that are called to indicate what objects the outputs need to be directed to. To simplify the construction of the adder circuit, a further grouping of gates into 1-bit adder objects is performed and these adder units are finally connected together to form the completed adder circuit. Each Worker object creates an instance of the adder circuit that needs testing. The worker has complete control over the circuit and can specify the failure behavior of each gate.

Figure 6.4 shows the Raven code that forms the main body of a Worker object. The execute() method uses the instance data iwords1 and iwords2, which are the two input numbers to be added, with owords being the output number being looked for. Within the execute() method, the Worker object first creates an Array of AdderStates. The information in this array specifies the operational state of each gate in the adder circuit. The Worker then gets its first failure scenario and while there are failure scenarios the following actions are performed:

1. Set the operational status of the gates to those required to check this failure scenario.
2. Add the two input numbers together, getting a result.
3. Compute the probability of this failure scenario.
4. Check the returned result to see if the result indicates that this failure scenario could have produced the desired results.
behavior execute {
    var fail : Array[AdderState]; // State each 1-bit adder unit is in
    var res : Array[Int];       // The result
    var prob : Float;          // The probability of the result

    fail = Array[AdderState].new(O);
    fail = state.getFailureScenario(fail);
    while (fail.atGet(O).unit < size) { // Unit is a complete adder unit
        unit.setOperationalStatus(fail);
        res = unit.add(iwords1, iwords2); // iwords{12}, instance data
        prob = unit.computeProbability();
        if (compare(res, owords) == EQUAL) {
            state.reportMatch(fail, prob);
        } else {
            state.reportNoMatch(fail, prob);
        }
        fail = state.getFailureScenario(fail);
    }
}

FIGURE 6.4 The main body of the execute method in the Worker class.

5. Report the outcome of the scenario test, along with the probability of the outcome.

6. Get a new failure scenario,

The actual implementation consists of 1 or more Worker objects querying a central object to get new adder unit states to test. Figure 6.5 shows the implementation code for the main body of the sequential and parallel implementations of this application. Both implementations require code to instantiate the StateGenerator object, which determines the failure scenarios to examine. The parallel implementation requires extra code to specify the companions and their number. The StateGenerator object also needs to be created with the controlled property to ensure the consistency of the object in case multiple workers attempt to access it simultaneously. The parallel implementation also demonstrates how an application can explicitly control parallelism by explicitly starting the number of desired worker processes.
CHAPTER 6: Performance

6.2.2.2 Performance Results

Figure 6.6 is a graph of the speedup obtained for examining part of the error space for the 128-bit adder described earlier. The speedup remains close to ideal until about 11 processors and then begins to tail off. The efficiency of the processors remains above 0.85 for the first 15 processors and then begins to level out and even drop off. In particular, an unexpected large dip in the speedup occurs in the 16 to 19 processor range.

In many respects, this problem has structure similar to the Mandelbrot set problem. A speedup graph similar to that one would, therefore, be expected. A comparison of Figure 6.6 to Figure 6.2 indicates that the graphs are not similar and start to diverge at around 9 processors. A further examination of the implementation and the Raven system was undertaken to try to explain this discrepancy.

Since the checking of a failure scenario is a faster operation than the corresponding computation in the Mandelbrot set, it was decided that perhaps the object providing the failure scenarios was being overwhelmed and could not respond quickly enough. If that were the case,
then increasing the length of time required to check a failure scenario would drive the speedup curve considerably closer to the perfect speedup line. This possibility was checked for by artificially doubling the failure scenario checking time. The resulting measurements showed no significant change in the reported speedups. This implied that the failure scenario server was not being overwhelmed. This conclusion was not entirely unexpected, since there was a dramatic drop-off in speedup as opposed to a gradual leveling out of the speedup.

Another possibility was that some of the worker objects were checking significantly more failure scenarios than other objects. In particular, maybe all the checking was being done by the first workers started and none of the work was being done by the last workers started. Unlike the Mandelbrot set computation, each worker object has a lengthy initialization phase.
to go through before the failure checking phase can start. This results because each worker object creates a model of the adder circuit to work with. The construction of this circuit object is memory intensive, so any unfairness in the memory allocation scheme could favour some objects. This would result in some objects starting work on failure scenarios sooner. To check this, the worker objects were instrumented to count how many failure scenarios they checked. Although it was true that the work load was not perfectly balanced, the discrepancy was insufficient to account for the dramatic dropoff in speedup.

The results of the two previous experiments indicated that the drop in speedup was not related to the checking of the failure scenarios, but to the initial construction of the adder circuit. The building of the adder circuit involves the creation of many objects, thereby putting a heavy strain on the memory management system. The elapsed time for the purposes of measuring speedup spans the time from when all worker processes are created until they have completed executing. This time includes the time to construct the circuit object. If there was contention in the memory allocation during the initialization phase, then as more processors were used the contention would become more pronounced.

To test this hypothesis, the worker objects were modified to first initialize themselves and then to synchronize with one another before proceeding. Figure 6.7 is the code for the **Barrier** class that was used to perform the synchronization, and it demonstrates the usefulness of delayed result. The **Barrier** class is an application level controlled class with a constructor and two methods, and its designed to perform a multi-way synchronization between threads of control. The constructor takes as its argument the number of threads to be synchronized. To use this class, the main body of the application creates the **Barrier** object and passes the object ID of the **Barrier** object to each worker. Each worker, which is a thread, invokes the **synchronize()** method, once it has created its circuit object and is ready to start checking failure scenarios. The **synchronize** method determines whether or not this invocation is from the last worker. If it is not, then the **me** value for the thread is stored in an array, and a delayed result is initiated by executing the **leave** statement. This results in the threads execution being sus-
class Barrier controlled {
    waitfors, index, done_thr: Int;
    blkcd_thr : Array[Thread];
    behavior synchronize();
    behavior finished();
}

constructor {
    blkcdThr = Array[Thread].new(0);
    index = 0; waitfors = 0; done_thr = 0;
}

behavior finished {
    if (++finished_procs == waitfors) {
        ... code to print elapsed time
    }
}

behavior synchronize {
    var thread : Thread;
    if (index == waitfors - 1) {
        //unblock the threads
        ... code to get start time ...
        for (index--; index >= 0; index--) {
            thread = blkcd_thr.atGet(index);
            result[thread] ;
        }
    } else { /* need to wait */
        blkcd_thr.atPut(index++, me);
        leave;
    }
}

FIGURE 6.7  Barrier class to synchronize and time a collection of Threads.

pended. When the last worker invokes synchronize, all the suspended threads synchronizing on this Barrier are released through calls to result. When a worker has finished all its tasks, it invokes finish() to indicated it has completed. When all the workers have finished the elapsed time since the workers were synchronized is computed and displayed.

The measured speedup that results when all the worker processes synchronize is shown in Figure 6.8. In this figure the new speedup values are superimposed on the old values. The speedup has improved significantly and is consistent with the expected speedups. With the modified measurement technique the efficiency of the computation ranges from 0.92 to 0.99 with the exception of the case when 20 processors are used. With 20 processors, efficiency drops to 0.84. The drop in efficiency is not unexpected, since at 20 processors the host system is starting to become over-committed. This overcommitment occurs both at the application level and at the total system level. At the application level, 20 object worker processes plus any
Raven system management processes are running. The result is that more application threads could be ready to run than there are processors. When this happens, the speedup will drop. Conceptually, a worker object must share a processor with the administrative threads instead of having the whole processor to itself. Given the structure of the program, more administrative overhead is present in this problem than in the Mandelbrot set. In addition, when the application is using all 20 processors, there are no other processors available to do processing for the underlying host operating system. This extraneous processing includes other users logging in, servicing network traffic or the managing of memory and disk requests. Servicing these other requests decreases the efficiency of this application.
The new measurements support the hypothesis that the dropoff in speedup is related to the initialization phase of the objects. A further examination of the figures reveals that in the 10 to 20 processor range the initialization time of the worker object represents between 15% and 30% of the elapsed time. Almost all the memory allocation requests were made during the construction of the adder object. This suggests that some aspect of the memory management system is causing a bottleneck. This point will be elaborated on further in Section 6.3.

6.2.3 Prime Number Generation

Another application programmed in Raven is a search for prime numbers. Since restricting the search for prime numbers to the word size of the machine is not particularly computationally intensive, a new class, LongInteger, was programmed. This class manipulates integers too large to be represented in the natural word size of the machine. The class provides the basic arithmetic integer functions such as: add(), subtract(), multiply(), divide(), less_than(), greater_than(), equal(), and negate(). These methods are supplemented with the isPrime() method to test the primality of a number.

6.2.3.1 Problem Description

The basic problem is to start from a specific number and then search the next $X$ numbers looking for primes. A simple algorithm is used to test a number, $N$, for primality. Essentially, all the numbers from 2 to the square root of $N$ are tested to see if they will factor into $N$. Only minor optimizations to the checking routine, such as only checking to see if the odd numbers are factors, are made. With a method to check for primeness, a simple way to look for prime numbers is to loop generating new numbers. The new numbers can then be checked for primality. Some example code to do this is shown in Figure 6.5. The left column provides a sequential implementation and the right column a parallel implementation. Extending the sequential version to operate in a parallel environment is a simple matter of turning the checkPrime() invocation into a companion and invoking the start method on the companion.

---

1. The algorithms for these functions are taken from Knuth [62] with Newton's method, which is used in isPrime(), from [24].
Both solutions perform the following steps:

1. Use the command line argument to determine how many numbers to check (NextVarArg).

2. Start the search at 100,001.

3. The variable range is set to indicate the increment to use when computing the next candidate number to pass to checkPrime(). The method checkPrime() checks the numbers from \( n \) to \( (n + \text{range} - 1) \) for primality and reports the prime numbers.

4. All the numbers from 100,001 to \( (100,001 + n) \) are checked for primality.

This solution is not a general purpose in that it always starts looking for prime numbers from the same integer. The application was coded in this fashion since the goal was to concentrate on demonstrating Raven’s parallel constructs and not on providing a general purpose prime number tester and generator. The checkPrime() method is somewhat unusual in that it takes a range parameter. In the initial implementation checkPrime() tested only one number. The modification to checkPrime() was undertaken to improve the performance of the parallel implementation with the goal being to provide each companion with more work so that
the cost of creating a companion would be amortized over more tests for primeness. As the numbers being tested get larger, the need to increase the work load of a companion decreases; just performing the basic tests will be an adequate work load. This last approach would have been preferred; however, verifying the results of the application is difficult and reasonable running times for measurement purposes are difficult to achieve.

6.2.3.2 Performance Results

Figure 6.10 shows the speedup achieved when searching for prime numbers. It can be seen that this implementation of the prime number searcher does not achieve the sorts of speedups obtained by the Mandelbrot computation and the Bayesian search routine. At 20 processors the speedup is 7.25 and the marginal performance increase in performance is small. In going from
18 to 20 processors the speedup only increases by 0.25. In the both the Mandelbrot computation and Bayesian computation the efficiency remains high and generally above 0.90. In this prime number searching example, that is not the case. The efficiency for two processors has already dropped to 0.83, well below that of the other two examples. Figure 6.11 is a graph of efficiency vs. processors. The efficiency in this application drops rapidly and is on a constant downward trend. When 20 processors are in use the efficiency has dropped to 0.36. Both Figures 6.10 and 6.11 indicate that the incremental performance increase, especially after 6 processors, is very small. This result is somewhat unexpected, as the problem has all the properties that indicate it should parallelize well.
Further analysis of the performance results, and additional experiments and instrumenting of the Raven system, indicated that the problem was with the memory management system. In performing the arithmetic operations, new objects representing the current and intermediate results of a calculation are constantly being created. These objects are discarded almost immediately, once the next step in the computation is performed. With this rapid consumption of memory, the free memory in the allocation pools is quickly used up, thereby forcing garbage collections. Each time a processor is added, the memory allocation rate of the system goes up. When one processor is in use there are approximately 1,500 memory allocations/second, and when 20 processors are active this has risen to 10,900 allocations/second. The allocation rate when normalized by the number of active threads has dropped from 1,500 allocations/second to slightly less than 550 allocations/second. Since each one of the threads is independent, the expectation is that each thread would like perform about 1,500 allocations/second.

This allocation rate consumes memory extremely quickly, and causes the system to garbage collect frequently. Although the garbage collection times have been subtracted from the elapsed times for the purpose of computing the speedup, it is still instructive to analyze what is happening. The amount of time spent garbage collecting increases from about 10% to nearly 50% of the elapsed time by the time 20 processors are being used. With this much time spent garbage collecting, it is clear that garbage collection is a bottleneck restricting parallelism. Adding more heaps to reduce the frequency of collections is not a practical solution because of the limited amount of memory available to the system. The proper solution is change the memory management system to operate in a high request and reclaim environment.

6.2.4 A Distributed Version of the Mandelbrot Computation

A parallel version of the Mandelbrot computation was initially developed in a distributed processing environment. This allowed the DisplayManager and Worker sides of the program to be tested and debugged separately. The main body of the code for the distributed- and shared-memory versions of the implementation are identical, except for less than 10 lines of well defined code needed by the distributed implementation to get the system configuration. As
Raven matures and configuration information becomes more fully integrated into Raven, these extra lines of code will disappear.

To provide some indication of the types of speedup possible in a distributed environment the Mandelbrot computation was run on a Sun ELC to establish a baseline. Several other runs were then performed using a variety of different machines. Table 6.1 enumerates the types of machines used in the distributed computation and attempts to provide some indication of the relative performance of the machines by reporting the processor speed and various SPEC [106]

<table>
<thead>
<tr>
<th>Machine</th>
<th>MHz</th>
<th>SPEC92 Integer</th>
<th>SPEC92 Floating pt</th>
<th>SPECMark 89</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun SPARCstation LX</td>
<td>50</td>
<td>26.4</td>
<td>21</td>
<td>-</td>
</tr>
<tr>
<td>Sun Sparcstation ELC</td>
<td>33</td>
<td>18.2</td>
<td>17.9</td>
<td>20.3</td>
</tr>
<tr>
<td>Sun Sparcstation IPC</td>
<td>25</td>
<td>13.8</td>
<td>11.1</td>
<td>13.5</td>
</tr>
<tr>
<td>Sun Sparcstation 2</td>
<td>40</td>
<td>21.8</td>
<td>22.8</td>
<td>25.0</td>
</tr>
<tr>
<td>Sun Sparcstation SLC</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>8.6</td>
</tr>
<tr>
<td>Sun Sparcstation 1</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>10.0</td>
</tr>
<tr>
<td>Sun SS10/41</td>
<td>40</td>
<td>53.2</td>
<td>67.8</td>
<td>71.2</td>
</tr>
<tr>
<td>Sun SS10/51</td>
<td>50</td>
<td>65.2</td>
<td>83.0</td>
<td>-</td>
</tr>
<tr>
<td>MIPS 3260</td>
<td>25</td>
<td>-</td>
<td>-</td>
<td>17.3</td>
</tr>
</tbody>
</table>

The measured elapsed times for the Mandelbrot computation are shown in Table 6.2. Making direct comparisons is difficult given the wide variety of machines used for these tests and the inability to control individual machine and network loading. However, the results demonstrate that significant speedup is possible for a distributed computation. For the 1, 5, 10, and 20 processor runs, the typical processor utilization of a client was 95% to 99%. For the run with 29 processors, several of the machines were in general use, with the result that some of the client processes received only 30% of a processor. It should be noted that when 29 processors were in use, the application was operating in a heterogenous machine environment utilizing Sun and MIPS processors. An indication of the discrepancy between the
TABLE 6.2 Elapsed time and speedup for distributed Mandelbrot computation.

<table>
<thead>
<tr>
<th>Processors used</th>
<th>Elapsed time (seconds)</th>
<th>Speedup</th>
<th>Processor types used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2590</td>
<td>-</td>
<td>1 ELC</td>
</tr>
<tr>
<td>5</td>
<td>525</td>
<td>4.93</td>
<td>5 ELCs</td>
</tr>
<tr>
<td>10</td>
<td>280</td>
<td>9.25</td>
<td>7 ELCs, 2 IPCs, 1 SS2</td>
</tr>
<tr>
<td>20</td>
<td>160</td>
<td>16.19</td>
<td>7 ELCs, 4 IPCs, 5 SS2, 2 SLCs</td>
</tr>
<tr>
<td>29</td>
<td>115</td>
<td>22.59</td>
<td>7 ELCs, 4 IPCs, 5 SS2, 4 SLCs, 2 LXs, 3 SS1, 1 SS10/51, 1 SS10/41, 2 MIPS 3260s</td>
</tr>
</tbody>
</table>

Processor speeds can obtained by examining Figure 6.12. This figure shows a partially completed Mandelbrot set calculation when 29 processors were in use. The clear areas are regions where no results have been returned. How long a particular calculation will take depends upon the processor speed, processor loading, and the difficulty of the calculation. Generally, the more black in a line the more difficult the computation, and the longer it takes. If the processor speeds and loading were identical, there would not be uncompleted gaps between large completed areas.
6.2.5 A Distributed Mail Application

As a further demonstration of Raven's ability to support programming in a distributed environment, a distributed mail application was programmed by a colleague. This application consists of two major classes:

- A server object that accepts messages and allows client objects to retrieve messages.
- A client object which acts as a user agent and allows the user to perform administration functions and to compose, send, accept and display messages.

Figure 6.13 depicts the message system and some of the interactions between the various components. In this figure, the round-cornered rectangles are client objects; the rectangles represent mail messages; the oval represents the mail server, and arrows indicate a message transfer and its direction. The client objects and server object each exist in their own Raven environment, thereby making this a distributed application. The mail server object plays a central role in this application by acting as a repository for message exchanges between clients. To illustrate the mail server application, consider the following user-initiated actions and the resulting interactions within the mail system:

- A user composes a message. This is done within the client object and does not involve the mail server.
- The user requests the composed message be sent. This results in the client object invoking the postMsg() method. The actual message is an argument to the method.
- The mail server object accepts the message and stores it. The method returns to the client and the client object can process its next request.
- A user issues the accept messages command to the client object. The client object invokes the retrieveMsg() method on the server object.
- The server object checks to determine whether or not there are any messages waiting for this user. If there are, the messages are returned as the result. The server also deletes its references to the returned mail objects.
- The client object receives the messages as the result of its invocation and the user can then issue further commands to display or delete the messages.
The programming of this application used several features of the Raven programming environment. The mailserver made use of a name server. When the server object starts, it registers itself with the name server. When clients start, they use the name server to obtain the object ID of the mail server. The mail server is a central server and multiple clients can invoke on it simultaneously. Extra programming to deal with concurrent accesses to the server is avoided by creating the mail server as a controlled object. The store and forward nature of a mail system is captured by designating message parameters and returned results to be copied. For example, when a message is sent to the mail server, the complete message is copied and
stored by the server. When messages are retrieved from the server, the returned messages are all copied to the receiving object.

The mail server is a good example of code that makes use of early result. When both the `postMsg()` and `retrieveMsg()` methods are invoked on the mail server, there is the opportunity to perform an early result. An abbreviated version of the message posting and retrieving code is shown in Figure 6.14. In the `postMsg()` method, an early result is done as soon as it has been verified that the invoker is allowed to post a message. The `postMsg()` method can then proceed to check the remainder of the message request and start the sequence of steps involved in delivery of a message to a user. If the `postMsg()` method detects any errors, such as an invalid recipient address, it composes a message detailing the error and sends it to the user. When messages are retrieved, the mail server object similarly uses an early result once the appropriate response has been determined. The mail server is then free to perform

```
// Method to post a message

behavior postMsg {
  ... local variable declarations
  ... and initial message processing code

  // check if sender allowed to post messages
  if (!regdb.isRegistered(user))
    return Result.new(RC_NOT_REG);

  // Early result
  result Result.new(RC_NOERROR);

  ... Code to go through each intended
  ... recipient and perform needed actions to
  ... deliver the message and perform any
  ... error notification to sender
}

// Method to retrieve messages

behavior retrieveMsgs {
  ... local variable declarations and code
  ... to check for messages and construct
  ... a response.

  // Early result of responses
  result rc;

  ... code to perform cleanup processing
  ... message delivery notification etc...
}
```

FIGURE 6.14 Example early result usage in distributed mail application.
post-message-retrieval processing, which includes actions such as message confirmation processing. In both of these methods, the early result minimizes the amount of time the client is waiting for the result from the mail server. Additionally, there is substantial opportunity for computational overlap between the client and server once the early result is performed.

6.2.6 Miscellaneous Applications

The applications that have been presented to this point have all concentrated on using the parallel constructs in Raven to make an application run faster. However, the constructs for supporting parallel and distributed computation are not always used in situations where an increase in performance is the sole goal. A problem may be coded in a parallel way because that is its most natural representation, or an application may be made distributed because the resources the application uses are physically distributed.

A colleague used Raven to build a simulation of an ATM (asynchronous transfer mode) network. The ATM simulation had several components running in parallel. Raven's companions greatly simplified the task of specifying this parallelism. Again, the controlled property played a significant role in simplifying the programming of the simulation. With Raven's built-in support for object access control, the application programmer did not have to develop and use a set of access control primitives to manage access to shared objects.

Raven's delayed result has proven to be very useful in the construction of synchronization objects. The InvokeStream class, used to provide Raven streams, and the synchronization technique used in determining what the problem was in the Bayesian search space program of Section 6.2.2, both use delayed result. In both cases threads perform an invocation to an object that will decide whether or not execution should continue. If it should, the method returns, and nothing else is done. If the thread should not continue, then a delayed result is done. When conditions change to allow the execution to continue, the thread detecting this issues the appropriate result statement, thereby starting the suspended thread. The controlled property is also used on these synchronization objects.
Due to the difficulty of getting a consistent work environment with respect to the number and type of processors, only one set of performance results have been reported for applications running in a distributed environment. However, both the Mandelbrot set and the prime number searching application were initially programmed and tested in a distributed environment. These problems were later converted to run in a shared-memory environment by simply merging the client and server so that they were compiled as one program instead of two.

6.3 Performance Analysis Overview

The dominant conclusion from this section is that memory management is an important issue. Note that there is only so much memory within a computer system, and, as more processors are added to work on the problem, the machine becomes effectively faster and it uses more memory and other resources, like backplane bandwidth. One must be aware of the issues associated with the scaling of system resource availability relative to the number of processors in use.

During the execution of the user-specified part of an application, the amount of parallelism varies. At some points the application will be sequential, while at others it will be fully parallel. To get the maximum speedup, the fully parallel portions of the application need to be maximized, and their computational component should dominate the elapsed time. In keeping with the approach that changes to convert a sequential program to a parallel one should be minimized, the majority of the effort will go to those parts of the application that benefit the most. Typically this means concentrating the effort on the main part of the application and not on the initialization phases. By doing this, the start-up part of the application will remain sequential even though there is some potential for parallelism. During the transition between the start-up code and the main body of the application responsible for the work, the constructs for parallelism are used. The result is that the speedup for an application ramps up to its maximum amount of parallelism, and on completion there is a similar ramping down as the parallel components terminate. Depending upon the nature of the computation, some execution streams will terminate while others continue, and this lowers the speedup. For a long-running application, small
regions that lack maximum parallelism are not significant to the elapsed time of the overall computation, but in short computations they become noticeable.

Although the garbage collection stop times have been factored out, the garbage collection pauses still have a detrimental effect on performance. Each memory pool has a data lock associated with it. When a garbage collection takes place the lock is held on that pool much longer than if a straight memory allocation were done. The memory pools are used in a round-robin fashion on a first-come first-served basis. The garbage collection routine essentially acts as a barrier that results in memory requests stacking up at that pool. As soon as the garbage collection starts acting as a barrier, the parallelism in the application decreases because of the stacking effect. This was observed in the prime number example. With this garbage collector, the problem can be partly addressed by having more stacks, but this carries with it the increased administrative costs of managing more memory pools. Again, a parallel memory allocation environment should alleviate these sorts of problems.

6.4 Summary

To demonstrate Raven’s parallel constructs, several parallel applications were programmed. For the Mandelbrot set and Bayesian search space problems the efficiency was consistently over 0.90. The application that searched for prime numbers was less efficient, but this is attributable to the large number of garbage collections caused by the application’s high rate of memory consumption. Several other problems of a more distributed nature were also described.
The work presented in this thesis revolves around the Raven system and language, and the modifications made to this environment to support parallel and distributed processing. This chapter will conclude the thesis by presenting a summary of the major results and suggesting some future research directions.

### 7.1 Summary of Results

This thesis has two major components to it. One component is concerned with the support for parallel and distributed programming and how it manifests itself at the language level. The second component of the thesis is concerned with the underlying system services required to effectively support applications in a parallel and distributed environment.
7.1.1 Parallelism at the Language Level

This thesis describes parallel and distributed aspects of the Raven language and system. Work from a variety of areas has been brought together, extended and augmented with new features, to provide a programming system that supports the development of parallel and distributed applications.

It was recognized that to support parallel and distributed programming there needed to be a convenient way to express parallelism, issues associated with providing concurrency control needed to be addressed, and some level of object transparency was required. Raven addressed the problem of expressing parallelism by introducing support for class-based and user-based parallelism.

The resulting model of parallelism is based on the observation that objects, with their encapsulated data and well-defined method interfaces, are ideal candidates for parallelism. In conventional systems, when writing a parallel program, the programmer must decompose the application into processes as well as functions and procedures, thereby complicating the task of writing the program. Raven reduces this problem by identifying the method invocation point as the location where parallelism occurs. Parallelism at a method invocation can result in two ways. In one way, the method being invoked can return a result while continuing to execute: this is class-based parallelism. The second way to achieve parallelism is to have the invoking object indicate that the invocation is to proceed in parallel with the initiating thread of execution: this is called user-based parallelism.

Class-based and user-based parallelism form the foundation of Raven's support for parallelism. Certificates and companions, combined with their associated control methods, are the language level-objects that specify and manage user-based parallelism. Class-based parallelism is supported through the constructs of early and delayed result. These constructs were developed by observing that the return statement performs two functions: it returns a result to the calling function, and it terminates execution flow in the current function. Early result permits
concurrency between the invoker and invokee, whereas delayed result allows a method to suspend execution and have a result returned later.

With the introduction of parallelism, objects can be concurrently accessed, with the possibility that multiple methods may try to change an object's instance data simultaneously. Other implementations solve this problem by allowing only one method to execute at a time, or by making concurrency control the programmer's responsibility. This problem is addressed in Raven by concurrency control provided through the controlled property. Properties offer a new way to associate operating system services with objects on an instance-by-instance basis. In doing this, properties eliminate the class explosion problem encountered when the same class is needed, but with different underlying support properties. Properties offer additional advantages in that only the objects using a property incur the extra execution overhead of managing the property; also, the user does not have to explicitly make function calls to access the desired service. Finally, properties provide a uniform way of supplying operating system services independent of how those services are realized in the host environment.

Raven's locking support, which is supplied through the controlled property, supports multiple readers or a single writer in an effort to maximize concurrency. To be effective, the controlled property must not restrict the way an object can be used. In particular, objects must still be able to invoke on themselves, and proper locking must be maintained across machine boundaries. These problems are addressed through the introduction of session IDs, which can be used to track invocation sequences both locally and remotely.

By making parallelism easy to use, it is easily possible to create more parallelism than the execution environment can handle. Other systems often handle this problem by making the control of parallelism the responsibility of the programmer, or by providing information at compile time that specifies the amount of parallelism supported by the target environment. Raven considers the controlling of parallelism to be a system configuration issue. At runtime the system is configured to specify the level of parallelism supported. This approach allows the same application to be run on machines that support different numbers of processors.
To support distributed programming, a certain level of object transparency is required. It should be possible for an application to treat all objects the same, regardless of whether they are local or remote. Raven's global object ID, combined with proxy objects to map a local ID to a global ID, provides this capability. Although InvokeStreams were introduced into Raven to allow invocation sequences consisting of both serial and parallel components to extract some parallelism, they also provide a demonstration of object transparency. A reference to an InvokeStream can be passed around to different processors, and the behavior is identical to that seen as if all the objects existed in the same environment.

### 7.1.2 System support

Implementing Raven in both a shared-memory multiprocessor environment and a distributed environment made it possible to identify some of the key components of the system that affect the performance of Raven.

This thesis provided a detailed description of how objects in Raven are organized and manipulated. Particular details on how remote invocations are performed, methods looked up, and objects and classes represented were provided. As well, the experiments run on method lookup, the analysis of the results, and the resulting changes to the Raven system were also described.

With multiple threads of control it becomes increasingly difficult, at the application level, to track memory usage and to determine when it is possible to free the memory. To deal with this problem, the decision was made to use a memory management system based on garbage collection. The initial experiences with a conservative garbage collector in a uniprocessor environment were encouraging; unfortunately, in a multiprocessor environment, even a light memory request load produced unacceptable performance degradation. After some experimentation, the memory management system was modified to allocate memory from multiple memory pools. Under light to moderate loading this configuration performs adequately,
but it still suffers from poor performance under heavy load. The problem area is the actual garbage collection, which remains single threaded, and not the allocation.

To demonstrate the viability of the Raven system and its parallel constructs, several applications were first coded sequentially and then converted to run in a parallel environment. Those applications which made little use of the memory allocation system were able to achieve substantial speedup. This demonstrated that the basic system support for parallelism was sound and that Raven’s parallel constructs could actually be used to write parallel programs. Other applications demonstrated that Raven could function in a distributed environment.

### 7.1.3 Conclusions

The following conclusions can be drawn from this thesis work:

- A model for parallelism in object-oriented systems was developed. This model exploits the strong data encapsulation of methods and the method invocation interface to specify parallelism. The parallelism can be specified either by the user of a method (user-based parallelism), or by the implementor of a method (class-based parallelism).

- Class-based and user-based parallelism can be realized through the new constructs of early and delayed result, and certificates and companions, respectively.

- Invocation streams can be used to provide third-party sequencing of invocation requests targeted at the same object.

- The concept of properties provides a convenient way to provide operating system services to objects on an instance-by-instance basis, while avoiding the class explosion problem.

- Systems like Raven can be made portable by specifying a virtual machine and building a more complete runtime system on top of this virtual machine. Portability is
then achieved by efficiently mapping the virtual machine services to the facilities provided in the native system, as opposed to porting the virtual machine code.

- The provision of system-level support for automatic concurrency control relieved the programmer of the task of managing concurrent access to objects. This was demonstrated by developing single-threaded applications in a uniprocessor environment and converting them to run in parallel on a shared-memory multiprocessors, without adding additional code to manage concurrency.

- The specification of a uniform programming interface allows applications to be programmed independent of the knowledge about the underlying hardware, be it a shared-memory or distributed-memory environment.

- An implementation of the proposed system was done to illustrate the viability of the features required to support parallel and distributed programming.

- Several examples were programmed to demonstrate the usability of the system for writing parallel and distributed applications in both shared-memory and distributed-memory environments.

### 7.2 Future Work

In this section some possible areas of future work resulting from this thesis are considered:

- Improvements to the structure and organization of the running Raven system. Refinements in this area would have as their goal the improving of Raven’s execution time and system robustness. Example areas of possible investigation include method dispatch and lookup, lock management, system initialization and parameter passing. Improved support for debugging would also be useful.
• Support for memory allocation and its associated garbage collection scheme in a parallel environment needs to be improved. One approach would be to make modifications to the existing collector based on Boehm's recent work [22]. Other approaches could involve using the compiler to provide some help for memory allocation or changing the way objects and object IDs are related. With any of these approaches, it is imperative that they address the problems associated with allocating and freeing memory at high rates.

• Extending garbage collection to operate in a distributed environment would also be interesting.

• Raven needs a comprehensive way of providing support for error and exception handling. The handling of errors and exceptions needs to take into account that there are two objects that are potential candidates to attach the exception handling mechanism to. Under certain circumstances, it is the object doing the method invocation that might be interested in processing an exception. For example, if an arithmetic exception occurs, or a method cannot be found, the invoking object might want to deal with it. In other situations, the object representing the current thread of execution might be the entity that should perform the exception processing. As an example of this, consider how an exception for the detection of a potential deadlock should be handled. For one thread of control the appropriate action might be to do nothing, while for another thread it might be to terminate the thread. In this situation the actions to execute when an exception occurs are not attached to the object doing the invocation, but to the object representing the executing thread.

At the language level, an exception handling package will need a way to associate an exception handling action to a specific exception. Such an association might be part of the specification of an object, just as properties can be, while under other circumstances the exception action may be more dynamic and be required to change during method execution. At the system level, the types of exceptions need to be
identified and a mechanism for delivering exceptions established. Also, special attention will need to be focused on delivering exceptions when Raven is operating in a distributed environment.

- Potentially, one of the most interesting areas of future work involves the extension of properties. In Raven, properties provide a way of associating system services with objects. Currently, the only services that can be associated with an object are those provided by the system. Work in making properties user providable or tailorable would be interesting. It also appears that properties could be used to build operating systems that provide a fixed set of services yet deliver those services in different ways. For example, the property, controlled, could come in two different forms: in one package, an optimistic concurrency control strategy might be used, while in another a conservative approach might be adopted. The types of applications that a system is expected to run would dictate which concurrency control package to use.

In general, a particular operating environment could then be constructed by choosing the types of property modules needed to build that environment.

Altogether, these are a few of the areas where future work in Raven can be done. It includes basic improvements to system organization and performance, addressing problems associated with memory management in parallel and distributed environments, exception handling, and ways of exploiting properties to build configurable systems.
Bibliography


224


Glossary

behavior
A function or procedure associated with an object. Only behaviors may manipulate the instance data of an object.

capability
A reference to an object.

Certificate
The object returned by the creation of a companion.

CertificateGroup
A Raven defined class that can be used to collect Certificates. Of particular not is the ability to iterate on a CertificateGroup. When being iterated on, certificates are returned for those companions which have completed execution.

companion
A delimited set of method invocations that are to execute in parallel with each other and with the other threads of execution in the system.

companion thread
Each method in a companion is executed by its own thread of control. That thread of control is called a companion thread. A Raven environment can place restrictions on the number of companion threads that can be active at once.

complex object
All the objects in the Raven system that are not primitive objects. Complex objects are composed of a collection of object references and methods. The object references are the
instance data for the object and the methods are the functions to manipulate the instance data.

delayed result
On occasion a method may not be able to complete its execution. It may need to wait for certain conditions to be met elsewhere in the system before a result can be returned. Under these circumstances the method does a leave, which effectively suspends the method. Another method will complete the method later, when a return value for the method can be determined. The action of suspending a method and returning a result later is called a delayed result.

early result
An early result occurs when a method returns a result to its invoking object, and continues executing after the result has been returned.

GID
See global identifier.

global object identifier
A unique identifier assigned to object that can be used to identify an object amongst all Raven environments. References to objects between Raven environments are always passed as global identifiers.

InvokeStream
A system class that can have certificates pushed into. The methods identified by the pushed certificate and targeted at the same objects, are executed in sequence.

me
A predefined system variable that is the OID of the Thread object that is currently executing.

method
See behavior.
method Id
A unique identifier within a Raven environment that can be used to identify a method. Method Ids are typically generated by the compiler at compile time.

object block
The structure in main memory that implements an instance of object.

object identifier
A reference to an object.

OID
See object identifier.

primitive object
One of the basic objects that are recognized and manipulated directly by the Raven compiler. Integers and floating point numbers are the only two primitive objects.

property
A special system supported capability that can be attached to an object.

proxy
A proxy is the local object that represents an object resident on another machine. When dealing with objects, an application cannot distinguish a proxy object from a local object.

substream ID
An identifier used in an InvokeStream that identifies a stream of invocation requests all targeted at the same object. It is called a substream because it is a subcomponent of the larger InvokeStream object.

read method
A method that reads, but does not modify the instance data of an object.
recursive invocation

A method invocation when an object invokes back on itself, either directly or indirectly through another object or objects.

write method

A method that, potentially, changes the instance data of an object. A write method is also allowed to read the instance data of an object.
Appendix A Table of Contents

Appendix A The Raven Language ................................................................. 237
A.1 Introduction ....................................................................................... 237
A.2 BNF Syntactic Notation ................................................................. 238
A.3 Syntax of the Raven Language ...................................................... 239
   A.3.1 Basic Building Blocks .............................................................. 239
       A.3.1.1 Comments .................................................................. 239
       A.3.1.2 Strings ....................................................................... 239
       A.3.1.3 Numbers ..................................................................... 239
       A.3.1.4 Reserved Words .......................................................... 240
       A.3.1.5 Punctuation ................................................................. 243
       A.3.1.6 Identifiers and Types ................................................... 244
A.3.2 Raven BNF .................................................................................. 244
A.3.3 A Raven Program ....................................................................... 246
A.3.4 Declaring Classes and Variables ............................................. 246
   A.3.4.1 Parameter and Variable Declarations ............................... 246
   A.3.4.2 Type Definitions and Declarations .................................. 248
   A.3.4.3 Self, Me, and Super ....................................................... 248
   A.3.4.4 Behavior Declaration .................................................... 249
   A.3.4.5 Class Declaration ........................................................ 250
   A.3.4.6 Parameterized Types ..................................................... 253
A.3.5 Control Statements ..................................................................... 254
A.3.6 Defining Behaviors and Manipulating Objects ......................... 254
   A.3.6.1 Behavior Definition ...................................................... 254
   A.3.6.2 Behaviors with a Variable Number of Arguments ............ 255
   A.3.6.3 Method Invocation ....................................................... 256
   A.3.6.4 Creating Objects .......................................................... 257
   A.3.6.5 Casting ......................................................................... 258
A.3.7 Support for Parallelism .............................................................. 259
   A.3.7.1 Early and Delayed Result .............................................. 259
   A.3.7.2 Certificates and Companions ......................................... 261
A.3.8 Type Checking ............................................................................. 262
A.3.9 Locking ......................................................................................... 263
A.3.10 Properties .................................................................................... 265
A.4 Support Classes .............................................................................. 267
   A.4.1 Object .................................................................................. 268
   A.4.2 Class ................................................................................... 269
   A.4.3 Certificate ............................................................................ 270
   A.4.4 CertificateGroup ................................................................. 272
   A.4.5 Thread .................................................................................. 272
   A.4.6 CompanionThread .............................................................. 274
   A.4.7 InvokeStream ....................................................................... 275
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4.8 System</td>
<td>276</td>
</tr>
<tr>
<td>A.4.9 NameServer</td>
<td>277</td>
</tr>
<tr>
<td>A.4.10 Miscellaneous Classes</td>
<td>277</td>
</tr>
<tr>
<td>A.5 The Raven Compiler</td>
<td>278</td>
</tr>
</tbody>
</table>
Appendix A

The Raven Language

A.1 Introduction

Raven is an imperative, strongly-typed, object-oriented language with stylistic influences from C. The basic unit of computation is the object, and in Raven each object is an instance of a particular class. A class is the basic unit of programming and consists of a collection of behaviors, or methods, which define the interaction points to the class. The class also defines instance variables which capture the state of an object. All entities in Raven are objects and can be manipulated as such. An object is referenced through a capability. A capability is a reference to an object and it may also contain information specifying the access rights to an object.

The behavior invocation is the mechanism that causes action in a Raven program. A behaviors is a procedure defined for a particular class and a method invocation is the process of running this procedure on a particular object. Syntactically, an invocation consists of an expression identifying the object being invoked upon, the name of the behavior being invoked, and a list of expressions which constitute the arguments provided to the behavior. During exe-
cution, the behavior has access to the state (instance variables) of the object it is being invoked upon. The execution of a behavior may result in the values stored in instance variables being changed.

A.2 BNF Syntactic Notation

The grammar for Raven is expressed using an extended-BNF notation. To illustrate the notation consider the following production for the definition of an instance variable.

\[
\text{instance-var} \Rightarrow \text{var-name} \{, \text{var-name}\} : \text{type-ref} [\text{public}] [\text{class}] [\text{indirect}]
\]

This notation consists of the symbols \( [], \{ \}, \|, (), \text{ and } \Rightarrow \). These symbols are not part of the language being defined and form part of the mechanism to define the language and are called meta-symbols. The symbols and their meanings are:

- \([x]\) zero or one occurrences of \(x\)
- \(\{x\}\) zero or more occurrences of \(x\)
- \(x\|y\) \(x\) or \(y\)
- \((x)\) \(x\) with parenthesis used for grouping
- \(x \Rightarrow y\) \(x\) is defined as \(y\)

The words \text{public}, \text{class}, \text{indirect}, and the punctuation marks comma and colon, are required to appear in the production at the indicated location. Within a production these symbols are shown in a typewriter font and they are called terminal symbols. Italicized words, like \text{instance-var}, are meta-variables called non-terminals. Non-terminals are used to denote other sequences of words or symbols. The left-hand side of \(\Rightarrow\), \text{instance-var}, is the non-terminal being defined by this production, and the right-hand side of \(\Rightarrow\) is the definition. In this case, the production references the other non-terminal symbols \text{var-name} and \text{type-ref}, which will be defined elsewhere in the language.

The production shown above defines an instance variable. An instance variable is defined as a comma separated list of \text{var-names} with at least one member followed by a colon. The
Appendix A: The Raven Language

A.3 Syntax of the Raven Language

This section of the appendix provides details on the syntax of the Raven language. The syntax is first presented in BNF form. After the syntax has been presented, more detail are provided on the various language constructs and how they are used.

A.3.1 Basic Building Blocks

A.3.1.1 Comments

Comments are used to include non-program text within a program. Raven supports two comment forms:

- /* */: all characters between /* and */ are ignored.
- //: all characters from // to the end of the line are ignore.

A.3.1.2 Strings

The following productions define strings in Raven.

\[
\begin{align*}
\textit{any_character} & \Rightarrow \textit{any character in the character set being used} \\
\textit{string_constant} & \Rightarrow "\{\textit{any_character}\}" \\
\textit{letter} & \Rightarrow a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z \\
& \quad | A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
\end{align*}
\]

A.3.1.3 Numbers

Numbers in Raven are defined by the following productions.

\[
\begin{align*}
\textit{digit} & \Rightarrow 0|1|2|3|4|5|6|7|8|9 \\
\textit{octal_digit} & \Rightarrow 0|1|2|3|4|5|6|7 \\
\textit{decimal_integer} & \Rightarrow \textit{digit}\{\textit{digit}\} \\
\textit{float} & \Rightarrow \textit{digit}\{\textit{digit}\}.\textit{digit}\{\textit{digit}\}
\end{align*}
\]
A.3.1.4 Reserved Words

A word, or symbol, in Raven is reserved for one of two reasons. Either the word has a direct meaning to the compiler, or else the word is used in the predefined class library. The following words are reserved in Raven and cannot be used as programmer defined names:

ArgCount - This is used in the manipulation of methods that take a variable number of arguments. ArgCount is the number of arguments present.

Int - The predefined class of integers.

Float - The predefined class of floating point numbers.

NULL - Raven compiles to the C language, and the C code generated by Raven uses the symbol NULL in a special way. To prevent a Raven application programming from encountering a possible conflict in definitions, NULL’s usage is not allowed.

NextVarArg - This keyword is used in the manipulation of behaviors with variable numbers of arguments. Use of this keyword causes an object reference to the next argument in the parameter list to be returned.

Object - The predefined class Object which is at the top of the Raven class hierarchy.

as - This is part of the cast operation.

basic - This is part of a property specification.

behev, behavior, behaviour - These keywords are used as part of the behavior definition or declaration process.

break - A control statement with usage similar to that of C’s.

case - This is part of the switch statement.

cast - This is part of the cast operation.

class - This is used as part of the class declaration or to associate certain actions with classes instead of object instances.

constructor - This is used as part of an object constructor’s declaration or definition.

continue - This is a control statement with usage similar to that of C’s.

controlled - This is a property specifier.
copy - This is used to indicate whether deep copies of objects need to be made in certain circumstances.

default- This is part of the switch statement.
do - This is part of the do/while statement.
durable - This is a property specifier.
else - This is part of the if/then/else statement.
for - This introduces a for loop.
if - This introduces an if statement.
immobile - This is a property specifier.
immutable - This is a property specifier.
leave - This statement is used as part of a delayed result.
local - This is a property specifier.
lock - This is used in locking operations or as a lock specifier.
migrate - This is an obsolete property specifier. A warning is printed if this property is used.
no - This is part of a lock specification.
nolock - This is used in locking operations or as a lock specifier.
persistent - This is a property specifier.
private - This is used to restrict the visibility of methods.
public - This is used to indicate whether classes, instance variables, or methods are publicly visible.
read - This is used in locking operations or as a lock specifier.
readlock - This is used in locking operations or as a lock specifier.
recoverable - This is a property specifier.
replicate - This is an obsolete property specifier. A warning is printed if this property is used.
replicated - This is a property specifier.
restore - This statement restores the instance data to the state it had before this method began executing. Execution continues immediately after this statement.

restricted - This is an obsolete property specifier. A warning is printed if this property is used.

result - This statement returns a result to the invoking routine.

return - This statement returns control, and an optional result, to the invoking routine.

send - Raven compiles to the C language and the C code generated by Raven uses the symbol send in a special way. To prevent a Raven Application programming from encountering a possible conflict in definitions, the usage of send is not allowed.

strong - This is an obsolete property specifier. A warning is printed if this property is used.

strongreplicate - This is an obsolete property specifier. A warning is printed if this property is used.

super - Within a method, this can be used to direct a search for a method to the parent class of where this method is found.

switch - This is part of the switch statement which is similar to C’s.

uncontrolled - This is an obsolete property specifier. A warning is printed if this property is used.

unlock - This is used in locking operations or as a lock specifier

unrestricted - This is an obsolete property specifier. A warning is printed if this property is used.

var - This is used to introduce a variable declaration in a compound statement.

volatile - This is an obsolete property specifier. A warning is printed if this property is used.

while - This is part of a while or do/while statement.

write - This is part locking operations or is used as a lock specifier

writelock - This is part locking operations or is used as a lock specifier

The following is the list of the reserved words (keywords) in Raven:

<table>
<thead>
<tr>
<th>ArgCount</th>
<th>Int</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NextVarArg</td>
<td>as</td>
</tr>
<tr>
<td>basic</td>
<td>behav</td>
<td>behavior</td>
</tr>
</tbody>
</table>
Appendix A: The Raven Language

iour  break  case
cast  class  constructor  continue
continue  controlled  copy
default  do  durable
else  for  if
bile  immutable  indirect  inheritance
its  leave  link
local  lock  migrate
no  nolock  persistent  private
vate  public  read  read-
lock  recoverable  replicate
replicated  restore  restricted
result  return  send
strong  strongreplicatesuper
switch  uncontrolled  unlock  unrestricted
stricted  var  volatile
while  write  writelock

A.3.1.5 Punctuation

The punctuation symbols in Raven separate various units of the Raven language. White-space has no significance in the language and may be used freely to ease reading. Punctuation strings made of multiple characters may not have blanks in them. When the interpretation of a sequence of characters is ambiguous, the interpretation selected is the one which results in the longest string of characters being recognized. The punctuation symbols in Raven are:

(  )  {  }  [  ]  !{  }!  $  
++  --  ...  .*  .  :  ==  +=  -=
/=  %=  *=  &=  |=  ^=  >>=  <<=  <<=  ==
&&  ||  &  |  ^=  -  +  *  /
%  >>  <<  >=  <=  -=  >=  <  <  !=
!  ~  ;  ,  ?  @  "  \

A.3.1.6 Identifiers and Types

Syntactically, identifiers and type names are identical and are defined as follows:

\[
\text{symbol} \Rightarrow (_)\text{letter}(_\text{letter}|\text{digit}) \\
\text{identifier} \Rightarrow \text{symbol} \\
\text{type}\_\text{name} \Rightarrow \text{symbol}
\]

Although an identifier and type name are syntactically identical, semantically they are different. A type name, is a symbol that refers to a previously defined class, and an identifier is a symbol that is not a type name. In the remainder of the BNF description of Raven, this distinction has been maintained to aid interpretation of the grammar. It should also be noted that any white-space or punctuation terminates the definition of a symbol.

A.3.2 Raven BNF

The BNF description for Raven follows. After the full BNF the various components comprising the Raven language are discussed in detail and examples provided.

1. program $$\Rightarrow$$ \{class\_declaration\}
2. class\_declaration $$\Rightarrow$$ class\_header \{behavior\_defn\}
3. class\_header $$\Rightarrow$$ ([public] class\_type\_defn [<- type\_decl][basic]{property})
   (class\_header\_list)
   (class\_repeated\_defn (class\_header\_list))
4. class\_header\_list $$\Rightarrow$$ \{((instance\_variable\_decl) | (behavior\_defn))
5. behavior\_defn $$\Rightarrow$$ ([copy] (behaviour|behavior|behav) identifier (parameter\_list)
   [: type\_decl][lock][private][class])
   (constructor (parameter\_list)[class]):
6. parameter\_list $$\Rightarrow$$ [... | (parameter\_decl {, parameter\_decl})]
7. lock $$\Rightarrow$$ nolock | no lock | readlock | read lock | writelock | write lock
8. behavior\_defn $$\Rightarrow$$ (behavior|behaviour|behav) identifier [class] compound\_statement)
   constructor [class] compound\_statement
9. instance\_variable\_decl $$\Rightarrow$$ variable\_name {, variable\_name}: type\_decl {variable\_attribute};
10. parameter\_decl $$\Rightarrow$$ variable\_name {, variable\_name}: type\_decl
11. variable\_decl $$\Rightarrow$$ var variable\_name {, variable\_name}: type\_decl [= expr];
12. variable\_name $$\Rightarrow$$ [copy][$$]$$ identifier
(13) $\text{variable\_attribute} \Rightarrow \text{public} | \text{class}$

(14) $\text{type\_decl} \Rightarrow ([*]\text{name}) | ([\text{name}] [\text{type\_decl} [\text{type\_decl}]])$

(15) $\text{type\_defn} \Rightarrow \text{identifier} [\text{identifier}]$

(16) $\text{repeated\_defn} \Rightarrow \text{type}\_name [\text{identifier} [\text{identifier}]]$

(17) $\text{compound\_statement} \Rightarrow \{\text{variable\_decl} \{\text{statement}\}\}$

(18) $\text{statement} \Rightarrow ;$
   
   (a) $\text{compound\_statement}$
   (b) $\text{expr}$
   (c) $\text{return} [\text{expr}]$
   (d) $\text{result} ([\text{expr}] [[\text{expr}] [\text{expr}]])$
   (e) $\text{leave};$
   (f) $\text{break};$
   (g) $\text{restore};$
   (h) $\text{readlock} [\text{read\ lock}]$
   (i) $\text{writelock} [\text{write\ lock}]$
   (j) $\text{unlock};$
   (k) $\text{continue};$
   (l) $\text{if} (\text{expr}) \text{statement} [\text{else\ statement}]$
   (m) $\text{while} (\text{expr}) \text{statement}$
   (n) $\text{do} \text{statement} \text{while} (\text{expr})$
   (o) $\text{for} ([\text{expr}]; [\text{expr}]; [\text{expr}]) \text{statement}$
   (p) $\text{switch} (\text{expr}) \{\text{case} [\text{default\ statement}]\}$

(19) $\text{case} \Rightarrow \text{case\ expr} : \{\text{statement}\}$

(20) $\text{expr} \Rightarrow \text{expr} \? \text{expr} : \text{expr}$
   
   (a) $\text{expr} [+|-|*/|\%|>>|<<|&|\^]= \text{expr}$
   (b) $\text{expr} [-|=|>|>=|<|<]= \text{expr}$
   (c) $\text{expr} [|||&|||^&]= \text{expr}$
   (d) $\text{expr} [*|/|\%|>>|<<]+|=|+-\text{expr}$
   (e) $\text{unary}$

(21) $\text{unary} \Rightarrow [-|+|!|\text{-}] \text{unary}$
   
   (a) $\text{unary}[++|--]$
   (b) $[[++|--] \text{unary}]$
   (c) $\text{cast\ unary\ as\ type\_decl}$
   (d) $\text{unary\_ident}$
Appendix A: The Raven Language

(e) \texttt{unary} \left[ \texttt{ident} \left[ \texttt{expr} \left[ \right], \texttt{expr} \right] \right]$

(f) \texttt{super} \left( \texttt{constructor|ident} \left[ \texttt{expr} \left[ \right], \texttt{expr} \right] \right)$

(g) \texttt{unary} \texttt{.ident} \left( \texttt{property_spec|, expr} \left[ \right], \texttt{expr} \right)\left[ \right]$

(h) \texttt{factor}$

(22) \texttt{factor} \Rightarrow \left( \texttt{expr} \right)$

(a) \texttt{integer|string}$

(b) \texttt{ident} \left[ \left[ \texttt{expr} \left[ \right], \texttt{expr} \right] \right]$

(c) \texttt{type\_decl}$

(d) \texttt{!\left[\texttt{expr} \left[ \right], \texttt{expr} \right]} \texttt{!\left[\texttt{expr}\right]}$

(e) \texttt{NextVarArg|ArgCount}$

(23) \texttt{property} \Rightarrow \texttt{immobile|controlled|recoverable|persistent|durable|replicated|immutable}$

(24) \texttt{property\_spec} \Rightarrow \texttt{basic|\left(\texttt{property\{&property\}\}}$

A.3.3 A Raven Program

A Raven program is defined to be a possibly empty sequence of class declarations.

(1) \texttt{program} \Rightarrow \left\{ \texttt{class\_declaration} \right\}$

An empty file is, therefore, a complete Raven program. For execution purposes, every Raven program must define the class \texttt{Main} which has the method \texttt{start()}\). When Raven starts executing, it creates an instance of \texttt{Main}, and begins the execution by invoking the \texttt{start()}\) method on this object.

A.3.4 Declaring Classes and Variables

This section describes how the various entities within the Raven system are declared.

A.3.4.1 Parameter and Variable Declarations

Variable and parameter declarations in Raven generally take the form of a list of variable names followed by a colon and a type declaration. A variable name is defined in the following way:

(12) \texttt{variable\_name} \Rightarrow \left[ \texttt{copy} \right]\left[ \$ \right] \texttt{identifier}$
Some examples of legal variable names are: `copy _fool`, `foo`, and `$foo`. The `copy` keyword is only meaningful for a parameter declaration. `Copy` means that a reference to a deep copy of the object is passed in, instead of a reference to the actual object. When a dollar sign ($) precedes a variable name, it means that static invocations will be used for invocations on this object. Static invocation may significantly decrease the method dispatch time for an invocation, but at the expense of property checking and dynamic method lookup. Great care should be exercised when tagging a variable for static method lookup.

There are three types of variable declarations in Raven. These are instance variable declarations, which form part of the basic class definition, parameter declarations, which declare the parameters for a method, and variable declarations, which define a compound statement’s local variables.

(9) \[
\text{instance\_variable\_decl} \Rightarrow \text{variable\_name}\{,\ \text{variable\_name}\}:\ \text{type\_decl}\{\text{variable\_attribute}\};
\]
(10) \[
\text{parameter\_decl} \Rightarrow \text{variable\_name}\{,\ \text{variable\_name}\}:\ \text{type\_decl}
\]
(11) \[
\text{variable\_decl} \Rightarrow \text{var}\ \text{variable\_name\_list}:\ \text{type\_decl}[=\text{expr}];
\]
(13) \[
\text{variable\_attribute} \Rightarrow \text{public}|\text{class}
\]

The variable attributes of `public` and `class` can be attached to instance variable declarations. `Public` means that that piece of instance data is directly accessible by methods that are not part of this object. The attribute `class` means that the data is class data and is not part of the instance data for objects of this class, but of the actual class. Some example declarations are:

```
\begin{verbatim}
a, b : \text{Int class}; (class instance variable declaration)
a, b : \text{List public}; (public instance variable declaration)
copy a, b : \text{List} (parameter declaration)
var a, b : \text{List}; (variable declaration)
var copy $ foo : \text{List}; (variable declaration)
\end{verbatim}
```

In the example of the parameter declaration, the `copy` keyword indicates that a deep copy of the parameter `a` is to be made. In the variable declaration, the `copy` keyword has no effect, but the dollar sign indicates that all methods invoked on `foo` will be invoked statically.
A.3.4.2 Type Definitions and Declarations

Every object in Raven is an instance of some class or type and every class $c$ is an instance of its metaclass, which is referenced as ${^*}c$. Metaclasses are themselves instances of the class metaMetaClass which is referenced as **Class. **Class is an instance of itself. A a type definition ($\text{type\_def}$) is used in a class header ($\text{class\_header}$) to define the type name for a class and names to use for any parameterized types. A type declaration ($\text{type\_decl}$) is used where entities are being declared to be of a particular type. When a type declaration is encountered, the type must have been seen by the compiler. Raven does not support forward type declarations.

\begin{align*}
(15) & \quad \text{type\_def} \Rightarrow \text{identifier}\{[\text{identifier}, \text{identifier}]\} \\
(14) & \quad \text{type\_decl} \Rightarrow ([^*]\text{type\_name}) | (\text{type\_name} [\text{type\_decl} \{\text{type\_decl}\}]) \\
(16) & \quad \text{repeated\_defn} \Rightarrow \text{type\_name}\{[\text{identifier}, \text{identifier}]\}
\end{align*}

A repeated definition ($\text{repeated\_defn}$) occurs when the compiler encounters a class header that uses a type definition that was previously encountered. A repeated definition allows a class's definition to be reopened so that more instance variables and behaviors can be added. This is useful when the implementation definition of class contains additional behaviors or instance variables that are not needed for the exported public definition of the class. Some examples of type declarations are:

- $\text{Long\_Integer}$ (A simple type declaration)
- $\text{Stack}[\text{Int}]$ (A stack of integers, a parameterized type)
- $^*\text{Long\_Integer}$ (The meta class of the class Long\_Integer)

A.3.4.3 Self, Me, and Super

Raven has three special identifiers $\text{self}$, $\text{me}$ and $\text{super}$. $\text{Self}$ is a reference to the current object, $\text{me}$ is a reference to the current thread of execution, and $\text{super}$ means perform this invocation on self, but start looking for the method in this type's parent class. $\text{Super}$ can only be used as the target of a method invocation. $\text{Me}$ is typically used in conjunction with delayed result.
A.3.4.4 Behavior Declaration

A behavior declaration (behavior_decl) provides the signature of a behavior. It declares the behavior's name, parameters, return type, and class. A behavior declaration starts with the optional copy keyword. If the copy keyword is present, then the object ID returned by this behavior references a private deep copy of the result and not the original result. This is followed by a keyword to introduce the behavior and its parameters. If the method returns a result, the type immediately follows the parameter declarations. Any special instance data locking requirements of this behavior follow the type; this will override any automatic locking provided by the system. If the private keyword is present, this indicates that the method is not publicly visible, while the class keyword means this is a class method that only operates on the class. A parameter list can consist of nothing, a comma separated list of parameter declarations, or three periods (...) indicating that this method takes a variable number of arguments. One special behavior exists, and that is the one defined by constructor(). The constructor() method is called when an instance of the class the constructor() belongs to is instantiated. If the keyword class follows a constructor, it indicates that this constructor is to be invoked when the class is created.

(5) behavior_decl ⇒ ([copy] (behaviour|behavior|behav) identifier (parameter_list)
[:type_dec][lock][private][class]) |
(constructor(parameter_list)[class]);

(6) parameter_list ⇒ [,...| (parameter_decl {, parameter_decl})]

If present, the lock specification (lock) overrides any of the default method locking that might be provided when the object is created with the controlled property. If nolock or no lock is specified then this method will not require a lock when it begins to execute, readlock and read lock indicate that a lock permitting reading of instance data is to be acquired; and writelock and write lock indicate that this method is to acquire a lock to permit both reading and writing of the object's instance data.

(7) lock ⇒ nolock|no lock|readlock|read lock|writelock|write lock
Some examples of behavior declarations are:

```
behav push(item :Object);
behavior pushMany(...);
copy behavior pop() : Object;
```

The above is a simple example of two method declaration that might be found in a Stack class. The first method, `push()`, takes a single argument of type `Object` and is expected to push it onto the stack. The second method, `pushMany()`, demonstrates the declaration of a method that takes a variable number of arguments. The third method pops an item from the stack, makes a deep copy of the object, and returns a reference to the copy. For the example below, assume that a `Clock` class is being defined and that the class `DateTime` already exists.

```
constructor(hours, minutes: Int, seconds: Int);
behaviour currentDateAndTime(): DateTime no lock;
behav convertSecondsToDate() DateTime writelock private;
behavior getClockVersion(): Int class;
```

The first line provides the declaration of a constructor for this class. The constructor takes three arguments to specify the time the clock is to start with. The parameters to the constructor are all integers. The second line declares a behavior that returns the current date and time. Locking of instance data for this method is explicitly disabled. It will be the programmer's responsibility to manage concurrent access to the instance data, if this is a programming problem. The behavior `convertSecondsToDate()`, is a method internal to this class, and when it is invoked it will acquire a write lock, even if a write lock is not needed by the method. The final method is a class behavior, and it must be invoked on this class and not instances of this class. In this example, the method `getClockVersion()` returns an integer representing the implementation version of this class.

### A.3.4.5 Class Declaration

A Raven class declaration is composed of two parts, one is the class header (`class_header`) and the other is zero or more behavior definitions (`behavior_defn`). The class header defines the basic characteristics of the class, such as the visibility of instance data, the parent class, default
properties, and the methods associated with the class. The behavior definitions define the code associated with a behavior.

\[(2) \text{class\_declaration} \Rightarrow \text{class\_header} \{\text{behavior\_defn}\}\]

The class header consists of the optional keyword `public`, which indicates whether or not the instance variables for this class are publicly accessible and can be read or written by methods not associated with the object. This is followed by the keyword `class` and the type definition. The optional “inherits from (<-)” field identifies the parent class of this class. If this field is absent, the parent class is set to be the system defined class `Object`. The next field is the optional properties field. Properties associate system provided services with instances of this class. When present, either a list of properties or the keyword `basic` is used. Properties assigned at the class level cannot be overridden by a subclass. A class defined as having the `basic` property differs from a class with no properties in that a subclass may not add a property that would override one of the basic properties. The class header list (`class\_header\_list`) enumerates the instance variables and behaviors that are associated with a class. The class header basically provides a synopsis of the class. A class definition can be reopened through a repeated class definition, in which case the entries comprising the class header list are merged with any items defined in previous class header lists. Repeated definitions are often used when a class has a public and private definition. For example, the public definition might not specify all the instance variables or the methods of a class because the user does not need that information. In the actual implementation, a repeated definition will be used to reopen the class and add instance variables and methods to be used internally to the class.

\[(3) \text{class\_header} \Rightarrow ([\text{public}]\text{class type\_defn \[-> type\_decl][basic]{property}}] \{\text{class\_header\_list}\}) | \text{class repeated\_defn (class\_header\_list)}\]

Some example class definitions are:

```
class Stack{class_header_list}
public class RestrictedStack <- Stack controlled persistent {class_header_list}
class RestrictedStack {class_header_list}
```
The first example defines the class `Stack`. The second example defines the class `RestrictedStack` which inherits from `Stack`. It is a controlled, persistent class and its instance variables are publicly accessible. The third example is a class redeclaration and it reopens the class `RestrictedStack`, possibly for the specification of some local instance variables or methods.

Taken together, the class header lists (class_header_list) for the initial and reopened class definitions declare all the instance data and behaviors that a class has.

\[\text{class_header_list} \Rightarrow \{(\text{instance_variable_decl}) \mid (\text{behavior_decl})\}\]

Example declarations for the class `Stack` and `RestrictedStack` are shown below:

```java
public class RestrictedStack <- Stack controlled persistent {
    min_size : Int;
    behav pop() : Object;
    constructor(max_size, min_size : Int);
}
```

```java
class RestrictedStack {
    behavior duplicateStack() : RestrictedStack private;
}
```

The first class that is declared is the class `Stack` and it has two methods, a constructor() and three pieces of instance data. `RestrictedStack` is a controlled persistent class that inherits from `Stack` and has one publicly visible instance variable, `min_size`. A new constructor() is also defined for `RestrictedClass`. The final example demonstrates the reopening of a class and the addition of a new method. This method is marked private to ensure that it can only be invoked locally. In many situations, the first definition of `RestrictedClass` would be the one providing a public description of the class implementation. When the class is actually written, the public definition could have additional internal variables and methods added by reopening the class.
A.3.4.6  Parameterized Types

In Raven a type definition (type_defn) is given by the following production:

\[(15)\quad \text{type_defn} \Rightarrow \text{identifier}[\text{identifier}\{, \text{identifier}\}]\]

The identifier in this case can be supplemented with a square-bracket-enclosed optional list of identifiers. These identifiers are place holders for types, and a class declared in this way is a parameterized class. Some example parameterized type definitions are:

- \(\text{List}[X]\)
- \(\text{SymbolTable}[\text{KeyType}, \text{ValueType}]\)

The first example can be read as type \(\text{List}\) of \(X\), and the second can be read as a \(\text{SymbolTable}\) using \(\text{KeyType}\) and \(\text{ValueType}\).

To develop a more intuitive understanding of what a parameterized type is and how a parameterized type is used, consider a \(\text{List}\) object. Within a program, one might have a list of employees and a list of buildings. These two lists will be identical except for the types of entities that they contain. In an object-oriented system like Raven, the programmer could write a generic list class that could be used in both these situations. Although this allows code reuse, only very limited type checking can be performed on the class’s usage, with the result that an employee could be added to the list of buildings, for example. Parameterized classes eliminate this problem by allowing a programmer to specify the class to be associated with this generic implementation. Consider the following partial specification for the parameterized class \(\text{List}\).

```java
class List[X] {
    length : Int;
    behavior append(item : X);
    behavior insertAtFront(item : X);
}
```

This class header defines a class \(\text{List}\) that manipulates items of type \(X\). A user could then declare a reference to a list of employee and instantiate it using the following code fragments:

```java
employees : List[Employee];
employees = List[Employee].new();
```
The first line declares a reference to a list of employees, and the second actually creates the initial list.

A.3.5 Control Statements

Most of the common control statements found in the C language are available in Raven, with the syntax being the same as that in C. The statements supported include: if, switch, for, do, while, continue, break and the null statement (;). The if statement is supported with and without the else clause. Compound statements are also supported. The only statement that is not supported is the goto. The switch statement is also more restrictive in that control statements cannot cross case boundaries. C's general expression format of expr,expr is also not supported.

A.3.6 Defining Behaviors and Manipulating Objects

This section describes the way that behaviors for a class are defined and how objects are created and manipulated to perform a computation.

A.3.6.1 Behavior Definition

The actual code for a method is supplied through a behavior definition (behavior_defn). Behavior definitions have two forms, one for constructors, and one for regular behaviors. Behavior and constructor definitions follow immediately after the class header they are associated with.

\[
(8) \text{behavior_defn } \Rightarrow (\text{behavior}|\text{behaviour}|\text{behav}) \text{ identifier[ class] compound_statement } | \\
\text{constructor[ class] compound_statement }
\]

A behavior definition starts with one of the keywords denoting a behavior, it is then followed by an identifier corresponding to the name of a method that was declared in the class header. If this behavior was declared as a class behavior, then the keyword class follows.
Finally, the definition is completed with a compound statement. The compound statement forms the executable body of the behavior. An abbreviated example follows:

```csharp
class Stack[X] {
    stack_top : int;
    behavior push(item : X);
    behavior pop() : X;
    constructor();
}

behavior push { body of method }
behavior pop { body of method }
constructor { stack_top = 0; super.constructor(); }
```

This example includes a partial class header declaration for a parameterized Stack class. It has two behaviors and a constructor(). The behaviors and the constructor() are defined immediately following the class header. In the example definitions, the statements defining the body of the behaviors have been omitted. The constructor(), however, includes a statement to initialize the Stack's instance variable to zero and to invoke the constructor() method of the parent class.

### A.3.6.2 Behaviors with a Variable Number of Arguments

Behaviors in Raven can be declared to have a variable number of parameters. Within a behavior they can then be accessed using the keywords NextVarArg and ArgCount. ArgCount returns, as an integer, the actual number of arguments passed to the method. NextVarArg returns the object reference to the next parameter. The production for these variable argument manipulation routines is:

\[(21.e) \quad \text{NextVarArg|ArgCount}\]

Some example code illustrating the usage of the Raven's variable argument facilities follow:

```csharp
behavior appendToList {
    var item : cap;
    var count : int = VarArgCount;
    while (count-- > 0) list.append(NextVarArg);
}
```
In this behavior the number of parameters is first determined, and then the parameters are extracted one-by-one and appended to a list.

**A.3.6.3 Method Invocation**

A method invocation in Raven takes the general form of an object identifier, followed by a period (.) and then the name of the behavior to invoke. This general form is shown in the following production.

\[(21.e) \text{unary\(\star\)}\text{ident([expr \{, expr\}]\}}\]

For the method to be invoked, the unary must evaluate to an object, and the identifier must be the name of a valid behavior for the type of object being invoked on. The behavior may also have a comma separated list of arguments. The star (*) operator provides a mechanism for dynamic method specification. When a star is present, the identifier must evaluate to a string.

Some simple invocations are:

```raven
screen.displayString("It's quitting time!");
res = number.complexAdd(number1, number2);
```

A code fragment illustrating the technique for dynamic method specification is:

```raven
var method_name:String = "displayString";
screen.*method_name("It's quitting time!");
```

In this example, a variable of type String is created and assigned a value. When method_name is evaluated because of the star (*) operator the method `displayString()` will be invoked. Because the name of the method is not know at compile time, no checking of the arguments or method name for validity can be done during compilation.

The remaining forms of method invocations, shown below, just deal with special cases of the more general method invocation.

\[(21.d) \text{unary\ident}\]

\[(21.f) \text{super\(.)\text{constructor\ident\([expr \{, expr\}]\}}\]

\[(22.b) \text{ident\([\[expr \{, expr\}]\}]\]

The production (21.d) is used to access instance variables of an object. Every instance variable has special `getValue_variable_name()` and `putValue_variable_name()` methods created for it. This production is a short form for invoking these methods. The unary corresponds to the object identifier and the identifier is the name of the variable. The existence of these methods does not imply that all instance variables are readable and writable; only the instance variables that are public can be accessed this way.

Production (21.f) is a way for a method to invoke a method on itself, but have the method search start in the parent class of this method. This is used primarily to permit a subclass to intercept a method, perform some execution, and then call the original method. In addition to regular methods, the `constructor()` method can also be executed from within a `constructor()` method. This is required so that a subclass can ensure that initialization of the instance data associated with the parent classes have also been initialized before returning.

Finally, production (22.b) is the short form for when a method wants to invoke on the object it is in. This is equivalent to a method invocation on self.

**A.3.6.4 Creating Objects**

Objects in Raven are created by invoking the `new()` or `pnew()` methods on a class. The syntax is captured by the following two productions:

```
(21.e) unary. [*] ident ([ expr {, expr} ])
(22.g) unary. ident (property_spec[, expr {, expr}])
```

Production (21.e) defines a generic invocation, and creating a new object is simply an invocation of the method `new()` on a class. Production (22.g) defines an invocation when programmer defined properties are to be associated with an instance. In this case the properties to be associated with the object are the first argument, and any arguments for the constructor follow.
Assuming that the class *List* and *FixedArray* are defined, then the following code fragments illustrate how new objects are created:

```ruby
class Stack
  def new()
  end
end

class FixedArray
  def new(size)
  end
end
```

```ruby
stack1 = Stack.new();
stack2 = Stack.pnew(controlled & durable);
array1 = FixedArray.new(20);
array2 = FixedArray.pnew(controlled, 15);
```

The first line of this example creates a new *Stack*, and the second line creates a new *Stack* that is controlled and durable. The third line creates a *FixedArray* of size 20, and the fourth line creates a *FixedArray* of size 15 with the controlled property. When the `new()` method is run on a class, it first allocates storage for the new object and initializes it. Once the object has been created, the constructor() method for the class is run. Any arguments to the `new()` method, or following the property specification in `pnew()`, are passed as the arguments to the constructor. The property specification enumerates the additional properties that this object is to be created with. Because properties in Raven are always stated in the affirmative, any default properties associated with the class cannot be overridden. The properties of an object can only be added to.

Although objects are explicitly created, they cannot be explicitly destroyed. The space used by an object is reclaimed by the garbage collector when there are no longer any references to the object.

**A.3.6.5 Casting**

Raven is a strongly typed language and under certain circumstances it may be necessary to treat an instance of an object as a different type. For example, an application may use a predefined class that stores objects as capabilities, and therefore ignores the type of the object. When an object is retrieved, as far as the compiler is concerned it is classless. Casting can then be used to change the type of an object. This temporary type change only occurs at the point where the cast is done. It does not affect the type of the object anywhere else. The syntax for a cast is:

```
(21.c)cast unary as type decl
```
The cast operation is introduced with the `cast` keyword, which is followed by the object being cast, the keyword `as`, and finally the type the object is being cast to. Assume that the class `List` exists and that it stores objects in a generic form. The following code fragment then illustrates the use of the cast operator:

```java
employee = cast genericList.getLast() as Employee;
```

This code retrieves the last element of the list and casts it to the type `Employee` so that the assignment to the `employee` object, which is also of type `Employee`, can be performed and properly typed checked. In this example, if the item retrieved from the list were not of type `Employee` the assignment would still proceed, however, subsequent attempts to invoke methods on `employee` could fail, because the assigned object does not support them. The `cast` operator is passive and does not change the underlying representation of the object being cast in any way.

### A.3.7 Support for Parallelism

This section provides an overview of Raven’s support for parallel and distributed computation. It covers early result, delayed result, `Certificates`, companions and locking.

#### A.3.7.1 Early and Delayed Result

In C the return statement conceptually serves two functions; one, it returns execution control to the caller, and two, it optionally returns a result. In Raven these two functions can be separated. By separating these two functions it is possible for an object to return a result, thus unblocking the invoking object, while maintaining its thread of control until performing a return. A `return` statement (18.c) has the same semantics as in C. To perform an early result, execution control is transferred to the invoking object as expected, and, after some execution, the invoked object returns a result. In returning this result the invoking object is unblocked for execution and, in addition, the originally invoked object continues its execution. This results in both the invoked and invoking object executing in parallel. To issue an early result, the first form of the result statement defined by production (18.d) is used.
(18.c) return [expr];
(18.d) result ([expr]([expr] [expr]));
(18.e) leave;

After executing result, the object continues to execute and keeps any locks that it had. The value of the me variable changes in the invoked object after a result statement is executed to reflect the fact that a new thread of execution has commenced. To terminate this new thread of execution in the invoked object, the object must either execute a specific return, or an implicit return by running off the end of the code. Once a result statement has been executed, the value associated with any return statement is ignored.

The complement to early result is delayed result. With a delayed result the method releases all its locks and terminates without issuing a response. It is expected that another method within the object will perform the result. To exit a method with the intention of performing a delayed result, the leave statement (18.e) is performed.

Leave will cause the method to release any locks it has acquired, and the method to terminate. Execution control, however, is not returned to the initiating object and its execution remains suspended until some other thread performs the result that the invoking object is waiting for. Since there may be multiple threads of control waiting on a delayed result, a way to identify each of these delayed result threads is required. Each thread has a me variable associated with it, me which is a capability identifying the thread of execution. To accomplish a delayed result the method must save the value of the me variable before doing the leave so that the suspended thread can be identified. Assuming that the me variable has been appropriately saved the actual delayed result is accomplished in another thread. Two behaviors illustrating the two sides of a delayed result are shown below:

```plaintext
behavior getResource {
    if (resource_state == FREE) {
        resource_state = IN_USE;
        return OK;
    }
    waiting_threads.add(me);
}

behavior freeResource {
    next_user: cap;
    if (waiting_threads.state != EMPTY) {
        next_user = waiting_threads.nextItem();
        result [next_user] OK;
    } else resource_state = FREE;
}```
In this example, waiting_threads is a Queue of Threads waiting for the resource. To acquire the resource getResource() is called. In the case when the resource is not available, the me keyword is used to put an identifier for the current thread of control onto a queue. A leave is then performed to exit the method and suspend the thread at the caller's invocation point until the delayed result is completed. In the freeResource() behavior, the queue is checked to see if there are any threads waiting for this resource. If they are, then the result statement is used to return a value to thread doing the delayed result. In this situation the result statement consists of three components: the keyword result, a square bracket enclosed reference to the thread the result is being returned to, and an optional return value.

A.3.7.2 Certificates and Companions

In sequential object-oriented languages, an object making a method invocation must wait for the invocation to finish before continuing. One way to extract parallelism is to use an early result (A.3.7.1). A companion is a mechanism that specifies a list of method invocations to be run in parallel with each other and with the invoking thread. Each companion has associated with it a Certificate object that can be used to monitor and wait for the completion of a companion. Each method invocation in a companion is referred to as a companion thread. The syntax for the creation of a companion is:

\( \{ \text{expr} \}_{1}^n \) !{expr[\{, expr\}]} ![(expr)]

A companion is delimited by !{ and }!, and may be followed by an optional curly bracket enclosed expression called a tag. Some example method invocations are:

\( \{ \text{ret\_value1} = \text{screen1.displayTime()} \text{, ret\_value2 = screen2.displayTime()} \} !\{ \text{screen1.displayTime()} \} \{ \text{screen1} \}.\text{start()} \);

.... some other code ... \text{certificate.wait()};

In the first example the displayTime() method is run on two objects, and the return values assigned to the appropriate variables. Since these methods are going to be run in parallel with the executing thread, the return values will not be meaningful until the companion has
completed. Lines two and three illustrate the use of companions with the Certificate object that is returned. The execution of a companion expression always returns a Certificate, but it may be ignored. However, when a companion is created, the companion threads are not started until the start() method is invoked on the Certificate associated with the companion. In the second example, the companion is created, a tag (screen1) is associated with the Certificate, and the start() method invoked on the Certificate all in one statement. Once the start() method is executed all the threads are running in parallel. A thread can wait for a companion's completion by executing the wait() method on the associated Certificate.

Within a companion, the same object can be invoked on. There is no special sequencing performed on the invocations, and there are no guarantees as to which method will execute on the target object first. The class InvokeStream addresses this issue. The expressions making up a companion must be simple in the sense that the variable being assigned to, and the object being invoked on, are identifiers. Also, as in C, the order of parameter evaluation for the methods of a companion is undefined. For an overview of the operations available on Certificate objects see A.4.3.

A.3.8 Type Checking

Raven supports static and dynamic type checking. In order to support static type checking, all variables, parameters, and return values must have a type. A type is simply the name of a class. A variable, return value, or parameter may only reference objects that are of the designated type or subtype. Although inheritance ensures that a subclass is a subtype of a super class, Raven supports sub-typing through contravariant type-checking, which is a conservative type equivalency policy. Ravens’s type checking rules are defined as follows:

A variable of type T can reference any object of class S, if S is a subtype of T. S is a subtype of T if S is identical to T of if the following conditions hold:

1. S provides at least the behaviors of T.
2. For every behavior in S that has a corresponding behavior in T, the corresponding behavior has the same number of parameters, with the same names and positions. The only exception to this rule is the constructor() method.

3. The type of result returned by S's behavior is a subtype of the result returned by T's behavior.

4. The parameters of T's behaviors are subtypes of the parameters to the corresponding arguments of S's behavior.

5. S contains at least the public instance variables of T. Public instance variables are instance variables that are visible outside the class.

6. The corresponding public variables in S and T have identical types.

Type compatibility for parameterized types is performed by carrying out the specified type bindings to create a virtual interface declaration, and then applying the above rules. A reference to an object of an arbitrary type is accomplished with the type cap. The type cap, which stands for capability, is compatible with all other types, and all other types are compatible with it. Invocations on variables of type cap cause a runtime type checking mechanism to be used.

A.3.9 Locking

Locking in Raven is done at the behavior level. Each method in Raven is classified by the compiler as to whether it is a read method or a write method. A write method is one in which the object instance data is modified, everything else is a read method. When a method is invoked on a controlled object, the appropriate lock type is acquired before the method is allowed to execute. The locking system supports a single writer and multiple readers. The programmer may override the automatic lock classification for a behavior when it is declared (A.3.4.4).

The statements defined by the productions (18.h), (18.i), and (18.j) can be used within the application to change the current lock, if the object is controlled. If the object is uncontrolled these statements have no effect.

\[
(18.h) \quad \text{readlock}(\text{read lock}); \\
(18.i) \quad \text{writelock}(\text{write lock});
\]
The statements work in the following manner:

- **read lock.** If the object's lock is not held then a read lock is acquired. If the lock is currently held as a write lock, it is demoted to a read lock. A read lock will remain a read lock.

- **write lock.** If the object's lock is not held, then a write lock is acquired. In the case when the currently held lock is a read lock, the read lock is promoted to a write lock. A write lock remains a write lock. If a read lock is held and a write lock cannot be granted immediately, then execution will block waiting for the write lock.

- **unlock.** Any lock currently held on the current object is released. An implicit unlock of user acquired locks is performed when a method completes.

In all cases when a lock cannot be acquired immediately the thread of execution is suspended until the lock can be granted. A thread of execution does not release any locks it holds while waiting for a lock. Within a method, locks do not stack or from blocking regions like a blocking construct in a programming language does. A method holds only one lock at a time.

The facilities for parallelism combined with the automatic locking described produce some interesting problem scenarios. The three problem areas are:

- An object does an invocation on itself.
- An object holding a lock does an invocation on itself, and the invoked method does an early result.
- An object holding a lock invokes on itself and the invoked method does a delayed result.

Recursive method invocations on an object only present a problem when a write lock is already held or is requested. If only read locks are involved no problem exists. To minimize the effects of recursive invocations, each execution thread is assigned a globally unique ID that can be used to track chains of invocations. The main purpose of this ID is to make it possible to determine when a method invocation attempts to acquire a lock on an object that is already locked within the invocation chain. When a recursive invocation involves a write lock, the lock
will be granted if the releasing of all the locks held on this object by the invocation chain would
permit the lock to be granted.

The problems are more complicated when a recursive invocation involves an early result. Since an early result could result in two threads of execution, unexpected changes to the
instance data through these recursive invocations are a possibility. To eliminate this for early result, the result is only returned and a separate thread created if no other locks are held on this
object in the invocation chain, or all the locks are read locks. When there are a mixture of read
and write locks held on the object by the invocation chain, the result of the early result is
delayed until the method with the early result in it does its return. This eliminates the parallel-
ism between the invoker and invokee that early result provides, but maintains the proper invo-
cation semantics.

A similar problem exists when a recursive invocation has a delayed result. When the
leave statement is executed the method terminates and the instance data lock acquired to
enter the method is released. The execution of the invoking method blocks waiting for the
leave to be completed. However, locks on this object could be held by other methods in the
invocation chain, and this could prevent another invocation chain from invoking the method
that would result in the completion of the delayed result. To avoid this, when the leave is
done, all the locks in the invocation chain that are associated with the object are temporarily
released. This permits other methods to acquire locks on the object with the expectation that
one of them will complete the delayed result. When the delayed result is completed the sus-
pended thread can be restarted provided all the locks it held on the object can be granted. If the
locks can’t be re-granted the “delayed” thread will remain suspended until they can.

A.3.10 Properties

Properties in Raven associate system level services with instances of an object. Properties are
assigned to an object when the object is created based on the mandatory properties, if any, that
were specified as part of the class, or as the first argument to the new() method. Each prop-
Appendix A: The Raven Language

Property is supported independently by the system, and any combination of properties can be assigned to an object of any class. By default, objects are created without any specified properties. Objects without explicit properties are handled in the following way:

- The object exists only in RAM. Therefore, the object does not survive system reboots.
- The system does not exercise any concurrency control support to prevent multiple methods from executing on the same object simultaneously.
- The object can migrate from machine to machine.
- Any changes to instance data are done immediately and are not recoverable.

The properties supported by Raven are given by production (23) and a property specification (property_spec) is given by the keyword basic or a collection of properties connected using an ampersand (&). Sections A.3.4.5 and A.3.6.4 illustrate how properties are associated with an object.

\[
\text{(23)} \quad \text{property } \Rightarrow \text{immobile}|\text{controlled}|\text{recoverable}|\text{persistent}|\text{durable}|
\]
\[
\text{replicated}|\text{immutable}
\]

\[
\text{(24)} \quad \text{property} \_\text{spec } \Rightarrow \text{basic}|(\text{property}\{&\text{property}\})
\]

Properties have the following semantics:

**Immobile:** An object with the immobile property cannot be migrated between machines. This is useful for objects that supply machine specific services.

**Controlled:** An object with the controlled property is protected against multiple simultaneous access by different threads of control. The compiler classifies a method as a read method or a write method based on whether it just reads the values of an object’s instance data, or whether it writes to them. Multiple reader methods, but only a single writer method are permitted access to a controlled object.

**Recoverable:** An object with the recoverable property has an “all or nothing characteristic.” It is only when the method terminates that all the changes made to an object’s instance data are kept. If the method terminates abnormally, or if the method executes the restore statement (18.g), then the object’s instance variables are reset to the values they had before the method started to execute.
Persistent: An object with the persistent property will have a copy placed in non-volatile storage. The in-RAM copy will be marked to be written to storage whenever the instance data is modified; however, for performance reasons the writing of the object may be delayed.

Durable: An object with the durable property is similar to an object with the persistent property, except that when a method returns from an invocation on a durable object, the changes are guaranteed to have been written to non-volatile storage.

Replicated: An object with the replicated property can be replicated on different machines. Replication is at the discretion of the runtime system and replicas are weakly consistent: that is the system does not ensure that all copies of an object are always in the same state.

Immutable: An object with the immutable property cannot have its instance data changed. An attempt to invoke a write method on an immutable object will result in a runtime error. Object properties do not take effect until after the object has been created and the object's constructor called. This makes it possible to assign values to the instance data of an immutable at creation time.

In addition to these properties, an object may be given the basic set of properties. The basic set of properties are those properties that an object is most likely to have. An object with the basic properties is immobile, uncontrolled, unrecoverable, not persistent, not durable, not replicated, and not immutable. If a class is declared with a property, that property cannot be removed in a subclass. Note, that properties are always specified in the affirmative, and that there is no way to remove a property specification. A compilation error is generated if the specification of the basic properties would conflict with the mandatory properties specified in any parent class.

A.4 Support Classes

Much of the functionality of the Raven system is provided through the standard class library provided with the Raven system. The next sections introduce some of the major classes that are crucial to the Raven implementation. The class hierarchy in Raven has the class Object at the top, and all other objects inherit from the class Object. All classes described below have the
behavior constructor(). If the constructor() takes no arguments, the description of
the constructor has been omitted.

A.4.1 Object

The class object is the top most class in the inheritance hierarchy. Methods that all objects need
to respond to are defined here.

Class header for object ⇒ class Object {}

    behav becomeWorldVisible(generic_cap: Int);
    "Make this object visible to the external Raven environment. Use the value specified
    by generic_cap as the object ID that can be used by other environments. (See
    configureAsRemote() in Class.)"

    behav className(): String;
    "Return the name of the class this object is a member of as a String object."

    behav deepCopy(): Cap;
    "Make a deep copy of this object and return the reference to the new object. This copy
    will cross machine boundaries if needed."

    behav doesNotUnderstand(methodID: Int);
    "This method is invoked on an object, by the system, when a method lookup on an
    object fails. The application programmer may override this method on a class by class
    basis. The passed in argument is the method ID that the system uses to identify the
    method being invoked. The method ID is computed using the method name as the
    input to a hash function."

    behav dumpCore();
    "This method displays a traceback of the current Raven invocation chain, and then
    causes the Raven environment to terminate. The invocation traceback does not cross
    machines."

    behav hash(): Int;
    "This is an internal Raven method. This method returns an integer hash value for this
    object."
Appendix A: The Raven Language

behav `instanceOf()`: *Class;

This method returns the object ID of the class this object is a member of.

behav `isIdentical(arg: Cap): Int;`

This method compares the target object and the object referenced by arg for exact identity.

behav `isMemberOf(classCap: *Class): Int;`

This method checks to see if this class is a member of the class referenced by classCap and returns TRUE if it does and FALSE if it does not. Note, this method does not determine if the target object is a member of a subclass of classCap.

behav `isNotImpl(message: String);`

Some classes act as abstract classes and do not implement all the methods of the class. The expectation is that the unimplemented methods will be implemented in any subclasses. To ensure that a proper method interface exists, the abstract class implements the deferred methods as a simple method that invokes the method `isNotImpl()` with an argument string indicating the nature of the problem.

copy behav `migrateCopy() : Cap;`

This is an internal Raven method that is used to make a copy of an object and migrate it to the machine the invocation is being done on.

behav `parmCountMismatch(meth:String, expect:Int, actual:Int);`

The runtime system invokes this method when there is a parameter count mismatch is detected during an invocation. This method takes the name of the method at fault and two integers that represent the number of expected arguments and the number of actual arguments. This is only invoked on those methods invocations that are dynamically type checked.

A.4.2 Class

All classes in Raven are instances of the class `Class`. When a new class is created, methods are invoked on `Class` to create the new class and to add methods to it. Class also implements the methods that all classes are expected to respond to.

Class header for Class ⇒ `class Class {}`
**Appendix A: The Raven Language**

behavior `canUnderstand(meth: String): Int class;`

This method returns TRUE if the target class can invoke the method named by the argument `meth`, otherwise FALSE is returned.

behavior `configureAsRemote(hid, lid, r_cap: Int): Cap class;`

This method is used to create a proxy for a remote object of the type that this class is. To build the proxy the values for the remote host ID (`hid`), the local Raven environment ID (`lid`), and the remote capability for the object (`r_cap`) must be supplied. The returned value is a proxy to the remote object.

behavior `createNewClass(parent: Class, Name: String, instances: Int, properties: Int): Class class;`

This method is used to create a new class and is internal to the Raven system. It takes as arguments the parent class (`parent`), the name of the Class (`Name`), the size of the instance data in bytes (`instanceSize`), and the default properties for the class (`properties`). The method returns the object ID of the newly created class.

behavior `name(): String class;`

This method returns, as a `String`, the name of the class.

behavior `new(...): Cap class;`

This method creates an instance of the target class. Since most classes do not redefine the `new()` behavior, this method is defined to take a variable number of arguments.

behavior `pnew(...): Cap class;`

This method is just like `new()`, except the first argument must be a property specification.

**A.4.3 Certificate**

User based parallelism is accomplished by creating companions. Companions are identified by an associated instance of `Certificate`. Every companion creation results in the generation of a `Certificate`. The methods on the this class allow for the companions to be monitored and controlled. Note, that when a companion is created, the companion threads do not start executing until the `start()` method has been invoked on the associated `Certificate`.

Class header for `Class` ⇒ `class Class {}`
Appendix A: The Raven Language

behavior addThread(thread: CompanionThread);

This is an internal Raven method that adds the CompanionThread (thread) to the collection of threads that are part of this Certificate.

behavior equals(other_cap: cap): Int;

Returns TRUE if other_cap has the same object ID as the target object, otherwise FALSE is returned.

behavior getIterator(): Iterator[CompanionThread];

This is an internal Raven method that is used to iterate over the CompanionThreads that are part of this Certificate.

behavior getTag(): cap;

This method returns the tag associated with this Certificate. If no tag is present nil is returned.

behavior setNotify(cert: CertificateGroup): Int;

This is an internal Raven method. A Certificate can be a member of multiple CertificateGroups. When a Certificate is added to a CertificateGroup, this method is invoked on the Certificate to register the CertificateGroup to be notified when the Certificate completes.

behavior setTag(value: cap): cap;

Set the tag value for this Certificate to value, and return the tag value.

behavior start(): Certificate;

When a companion is created the CompanionThreads composing the companion are initially in a suspended state. To start the threads executing the start() method must be invoked on the Certificate associated with the companion. The start() method returns the Certificate the invocation is being done on.

behavior threadFinished();

This is an internal Raven method. When a CompanionThread completes its execution, it invokes the method threadFinished() method on the Certificate the CompanionThread is a member of.

behavior wait();

This method is used to wait for a Certificate to finish. When all the CompanionThreads of this Certificate have finished, this method returns. If all the CompanionThreads have completed, the call returns immediately.
A.4.4 CertificateGroup

A CertificateGroup is a class that collects Certificates.

Class header for CertificateGroup ⇒ class CertificateGroup <- Object controlled{}

behavior add(new_certificate: Certificate);

This method is used to add a Certificate to a CertificateGroup.

behavior delete(cert: Certificate);

This method removes the supplied Certificate from the CertificateGroup, if the Certificate is is present.

behavior notify(cert: Certificate);

This is an internal Raven method. When a companion finishes executing, it notifies any CertificateGroups it is a member of that it has completed executing by invoking this method on the CertificateGroups it belongs to. The argument, a Certificate, indicates which companion has completed executing.

behavior setWait();

When a CertificateGroup is to be iterated on this method is invoked to initialize the iteration. This permits the same CertificateGroup to be iterated on multiple times.

behavior waitForNextCertificate(): Certificate;

Once the setWait() method has been invoked, it is possible to wait for individual members of a CertificateGroup to complete execution. This method returns the Certificate of the finished companion During an iteration each Certificate will be returned only once, execution will block waiting for a companion to complete, and control will return as soon as a completed companion exists. Only one iteration sequence at a time is permitted.

A.4.5 Thread

The Raven runtime system supplied multiple threads of control. The class Thread, maps the system notion of a thread into a corresponding Raven Thread.

Class header for Thread ⇒ class Thread {"}
behavior attach(): Int;

This is an internal Raven method that associates the underlying system thread responsible for the actual thread execution with the Raven language level Thread. This method is used when an early result is performed.

behavior getPid(): Int;

This is an internal Raven method. It returns the underlying system thread process ID of the target Thread.

behavior kill(): Int;

This method is intended primarily for system use. The kill() method immediately terminates the target Thread. No attempt is made to cleanup after the Thread’s termination, in particular any locks held by the Thread are not released.

behavior resume();

Resume execution of the suspended Thread.

behavior sleep(duration: Int);

Put the target Thread to sleep for duration clock ticks. (Ideally, duration should be changed to put the Thread to sleep for an absolute time, like some number of milliseconds.) This method can only be invoked on the me variable.

behavior stackSize(): Int;

This is an internal Raven method that returns the size, in bytes, of the target Thread.

behavior start(...);

This is an internal Raven method that creates a Thread, associates a system thread with it and starts the indicated method running on the target object. The first argument to this routine is the object ID of the target object. The method name, as a String, is the second argument. The remainder of the arguments are the arguments required by the method being executed. The action taken by the system is to invoke the named method on the target object with the supplied arguments within the object this method is being run in. Note, system level threads in Raven are classified as to type. The types are: user level thread, user level companion thread, Raven system thread, and Raven system companion thread.

behavior startAsSystemProc(...);

This is an internal Raven method that is exactly the same as start(), except the system level thread is classified as a Raven system thread.
behavior suspend();

This is an internal Raven method that suspends the execution of a Thread. Currently, a Thread may only suspend itself.

constructor(pprio:Int);

This is an internal Raven method. This method initializes a Thread and takes as an argument the priority that the created process is to run at.

A.4.6 CompanionThread

A CompanionThread is a subclass of Thread. The individual threads of control within a companion are run in CompanionThreads. This class supports special methods monitor and control CompanionThreads, however, most of the methods are internal Raven methods.

Class header for CompanionThread ⇒ class CompanionThread <- Thread{}

behavior companionDelayedStart(...) nolock;

This is an internal Raven method used in the creation of a companion thread. It takes the same arguments as the start() method of Thread. When the companion is created, the companion is created in the suspended state and does not begin immediate execution.

behavior companionStart(...) nolock;

This is an internal Raven method that is the same as companionDelayedStart(); except, the created companion thread begins executing immediately.

behavior getSequenceNo(): Int;

If a CompanionThread is used in an InvokeStream, each CompanionThread will have a sequence number attached to it. This method returns that sequence number.

behavior getTargetObject(): Cap;

This is an internal Raven method used in the implementation of InvokeStream. To sequence the requests, the InvokeStream needs to know the target object. This method returns that information.
Appendix A: The Raven Language


This is an internal Raven method used in the implementation of InvokeStream. This method is called by setInvokeStreamData() to save the reference to the InvokeStream, the stream id this CompanionThread is to use, its sequence number.

behavior setSequenceNo(num: Int): Int;

This is an internal Raven method that sets the sequence number that a CompanionThread pushed onto an InvokeStream has.

behavior startExecution();

This is an internal Raven method used in the implementation of InvokeStream. This method is invoked when the CompanionThread is to start executing.

A.4.7 InvokeStream

The class InvokeStream provides a mechanism to sequence the method invocations within single and multiple companions.

Class header for NameServer ⇒ class InvokeStream controlled {}

behavior push(certificate: Certificate): Certificate writelock;

This method pushes the given Certificate into the InvokeStream. An implicit start() is done on the CompanionThreads associated with the Certificate. The execution of the CompanionThreads is sequenced with respect to common target objects. (Sequencing is relative to the target object. Requests to different target objects are not sequenced with respect to one another. However, all requests at a particular target object are sequenced with respect to each other.) Return results are also sequenced based on the order the requests are pushed into the InvokeStream. That is, if two companions are targeted at the same object and pushed into an InvokeStream, then the results from the execution of the companion that was pushed first will be available before the ones pushed after it. A Certificate may only be a member of one InvokeStream, and it must not have had the start() method executed on it.

behavior registerCompletion(target_object: cap, str_id: Int, seq_no: Int) writelock;

This is an internal Raven method. When a CompanionThread finishes executing, it informs the InvokeStream that it has completed. To enforce sequencing, this method may be suspended pending the completion registration of some other methods. The
Appendix A: The Raven Language

A.4.8 System

There is only one instance of the System class in a Raven environment. This instance is responsible for recording information about the general state of the system and responding to methods that affect the behavior of the system.

Class header for Class ⇒ class System {

behavior getDefaultStackSize(): Int;

This is an internal Raven method and it returns the default stack size, in bytes, to be used when a new thread is created.

behavior getHID(): Int;

This method returns the host ID of the machine this Raven environment is running on.

behavior getLID(): Int;

This method gets the local Id that represents the actual Raven environment on a particular machine.

behavior getPermissibleParallelism(): Int;

This is an internal Raven method and it returns the maximum number of Companion Threads that may be running at once.

behavior initializeInvokeStreams();

This is an internal Raven method that is used to initialize the InvokeStream service.

behavior monitorLocks();

This method turns on lock monitoring for the system. Lock monitoring is part of Raven’s deadlock detection approach. If a lock cannot be granted within the system supplied length of time, the method LockAcquisitionTimeout() is invoked on the invoking object. The amount of time to wait for a lock is a configuration parameter.

behavior setDefaultStackSize(sz: Int): Int;

This method sets the default stack creation size to be sz bytes. It returns this size.
Appendix A: The Raven Language

behavior setPermissibleParallelism(par: Int): Int;

This method is invoked to change the maximum number of CompanionThreads that may be active at once.

behavior startNetwork();

This method is invoked if access to remote Raven environments is desired. Currently, the programmer must execute this method explicitly, but it could easily be done in the system start-up code.

A.4.9 NameServer

This is a class that implements a simple name server. It’s primary purpose it to provide a simple name lookup service in a distributed environment.

Class header for NameServer ⇒ class NameServer controlled{}

behavior addName(copy name: String, id: cap);  

This method takes the a name that is to be associated with a particular object (id) and stores the key value pair in the name server.

behavior findName(copy name: String): cap;

This method looks up the given name and returns the capability of the object it is associated with. If the name cannot be found nil is returned.

behavior present(copy name: String): Int;

This method verifies whether or not the supplied name is in the name server. If it is TRUE 1 is returned., otherwise FALSE 0 is returned.

behavior removeName(copy name: String);

This method removes, if it is present, the indicated name from the nameserver.

A.4.10 Miscellaneous Classes

In addition to classes described earlier, the Raven class library supplies other classes that are of use to the programmer. Some of these classes are also used extensively by the Raven system. Some of the more commonly used classes are: Array, List, Queue, Semaphore, Set, Stack, String and SymbolTable.
A.5 The Raven Compiler

The current version of Raven is compiled with the Raven compiler which is called rvc. The compiler and the associated class libraries are compiled to run on Sparc, Mips, and the Intel 386/486 architectures. The operating systems used are the ones supplied by the respective vendors, except for the Intel 386/486 machines which are running Mach 3.0. The format of the command to invoke the compiler is:

```
    rvc [options] [files]
```

Where the options are:

- `v` Print the version number of the compiler and exit.
- `p` Run the pre-processor on the files and exit.
- `r` Compile to a `.c` file and exit. Do not link.
- `c` Compile to a `.o` file and exit. Do not link.
- `k` Do not delete any intermediate compilation files.
- `-I path_name` Use the supplied path to search for include files.
- `-L path_name` Use the supplied path to search for libraries to use during linking.
- `-l lib_name` Include the supplied library during the linking phase.
- `g` Generate output that can be used for debugging using gdb. This option implies the `k` option.
- `-o filename` Use the supplied name as the name of the output file.
- `-t` Be verbose while compiling, and print the full commands used during the compiling process.

The file extension used by Raven are:

- `.r` - The Raven source code files.
- `.c` - The intermediate C files.
- `.o` - The object files.

The Raven compiler accepts files with the above extensions. The `.r` files are compiled by the Raven compiler. The `.c` files are passed to the C compiler and `.o` files are used in the linking process. The default output file from the compiler is named `rv.out`.

The program rvc is really just a driver program that parses the command line arguments and executes other programs to do the work. Raven programs (files with the `.r` extension) are
first compiled to C. Any C source code, including the output from the first pass of the compiler, are passed to the GNU C compiler, gcc, for compilation. The object files are then linked, using gcc, to produce the output file rv.out.

To produce an actual executable file, the programmer must define the a class *Main* with the method `start()`. This is analogous to the `main()` function in a C program. Execution of the user's program begins by creating an instance of the object *Main*, and then invoking `start()` on it.