University of Alberta

Library Release Form

Name of Author: Kai Tan

Title of Thesis: Pattern-based Parallel Programming in a Distributed Memory Environment

Degree: Master of Science

Year this Degree Granted: 2003

Kai Tan
Apt 103, 10621 80 Avenue
Edmonton, AB
Canada, T6E 1V6

Date: _____

University of Alberta

Pattern-based Parallel Programming in a Distributed Memory Environment

by

Kai Tan

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Pattern-based Parallel Programming in a Distributed Memory Environment** submitted by Kai Tan in partial fulfillment of the requirements for the degree of **Master of Science**.

Duane Szafron
Co-Supervisor

Jonathan Schaeffer
Co-Supervisor

Robert Hayes

Paul Lu

Date: _____

Abstract

Design patterns document general sequential algorithmic solutions to recurring problems; parallel design patterns extend design patterns into the area of parallel programming. CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) uses parallel design pattern templates to support the fast and reliable design of parallel applications in a shared-memory environment.

This dissertation describes enhancements made to CO_2P_3S to use parallel design pattern templates for parallel programming in a distributed-memory environment. Jini was chosen to be the underlying infrastructure for the Distributed CO_2P_3S environment (DCO_2P_3S), and RMI was used as the inter-process communication mechanism. A distributed synchronization mechanism, a process manager and a performance monitor have also been designed for the environment.

A new version of RMI was devised and a new serialization scheme was implemented. They support high performance and low overhead communication in distributed memory parallel computing. The new designs maintain the ease of use of the original RMI and JDK-serialization without putting any additional burdens on the programmer.

This dissertation describes the design and implementation of DCO_2P_3S as a Jini distributed system. The usability and applicability of CO_2P_3S are extended by this work.

Acknowledgements

First of all, I would like to thank my wife, Ji Jia, for her love, care and patience. I also want to thank my parents for all of their support and understanding. My thank you also goes to my supervisor Jonathan Schaeffer and Duane Szafron for their guidance and funding. Thanks also to other members in the CO_2P_3S research group: Steve Macdonald, Steve Bromling and John Anvik. Their research efforts and advices greatly improved my work. I am also grateful to Dr. Paul Lu and Dr. Robert Hayes, who took precious time to read and comment on this dissertation, and to Dr. Herb Yang who chaired my thesis defense.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 The scope of this dissertation . . . 3
- 1.3 Dissertation organization . . . 4

2 CO_2P_3S Introduction **5**
- 2.1 Introduction to CO_2P_3**S** . . . 5
- 2.2 An Introduction to the Parallel Design Pattern Process . . . 7
- 2.3 Design patterns and frameworks . . . 9
  - 2.3.1 Design patterns . . . 9
  - 2.3.2 Frameworks . . . 9
  - 2.3.3 Generative design patterns . . . 10
  - 2.3.4 Parallel design patterns . . . 10
  - 2.3.5 From parallel design patterns to frameworks to programs . . . 11
- 2.4 MetaCO_2P_3S introduction . . . 13

3 Jini overview **20**
- 3.1 Introduction . . . 20
- 3.2 Applications . . . 21
- 3.3 How Jini works . . . 21
  - 3.3.1 The Jini architecture . . . 21
  - 3.3.2 The key concepts of Jini . . . 23
  - 3.3.3 An advanced Jini system architecture . . . 31
  - 3.3.4 Launching and shutting down the Jini system . . . 31
- 3.4 Conclusion . . . 32

4 Remote Method Invocation **33**
- 4.1 Introduction to RMI . . . 33
- 4.2 The RMI programming model . . . 33
  - 4.2.1 The programming API . . . 35
  - 4.2.2 Remote object stubs and skeletons . . . 36
  - 4.2.3 RMI registry . . . 37
  - 4.2.4 RMI activation . . . 37
- 4.3 RMI architecture . . . 38
- 4.4 Conclusion . . . 40

5 Serialization **41**
5.1 The purpose of serialization . . . 41
5.2 The programming API . . . 42
5.3 The serialization process . . . 43
5.4 The de-serialization process . . . 46
5.5 The layout of the serialized data . . . 47
5.6 Discussion . . . 49

6 The Architecture and Infrastructure of Distributed CO_2P_3S **50**
6.1 Introduction . . . 50
6.2 The DCO_2P_3S architecture . . . 51
6.3 Communication scheme . . . 54
6.3.1 Design choices . . . 54
6.4 Distributed synchronization mechanisms . . . 56
6.4.1 Synchronization . . . 57
6.4.2 The distributed synchronization implementation . . . 57
6.4.3 Discussion . . . 60
6.5 Performance monitoring and process management . . . 61
6.6 Distributed parallel design patterns . . . 64
6.7 Conclusion . . . 64

7 RMI and JDK-serialization Modifications **65**
7.1 Motivation . . . 65
7.2 Related work . . . 67
7.3 Design of DCO_2P_3S-serialization . . . 68
7.3.1 The CLASSPATH approach . . . 68
7.3.2 Implementation details . . . 69
7.4 Performance comparison . . . 74
7.5 Conclusion . . . 76

8 Distributed Design Pattern Templates in DCO_2P_3S **77**
8.1 Introduction . . . 77
8.2 The DM_Mesh pattern . . . 77
8.2.1 Intent . . . 77
8.2.2 Motivation . . . 78
8.2.3 Structure . . . 79
8.2.4 Pseudo code . . . 80
8.2.5 DM_Mesh pattern template parameters . . . 80
8.2.6 Use of the template . . . 82
8.2.7 Implementation details . . . 82
8.2.8 An example DM_Mesh application . . . 83
8.2.9 Application performance . . . 86
8.3 The Phases pattern . . . 88
8.3.1 Intent . . . 88
8.3.2 Motivation . . . 88
8.3.3 Using the template . . . 89
8.4 The DM_Distributor pattern . . . 91
8.4.1 Intent . . . 91

8.4.2 Motivation . . . 91
8.4.3 Structure . . . 92
8.4.4 Parameters . . . 92
8.4.5 Use of the template . . . 93
8.4.6 Example application . . . 95
8.5 The DM_Wavefront pattern . . . 98
8.5.1 Intent . . . 98
8.5.2 Motivation . . . 99
8.5.3 Structure . . . 100
8.5.4 Pseudo code . . . 101
8.5.5 Parameters . . . 102
8.5.6 Use of the pattern template . . . 103
8.5.7 Applications . . . 104
8.5.8 Example application . . . 104
8.6 Conclusions . . . 107

9 Summary and Conclusions **108**
9.1 Contributions of this research . . . 108
9.2 Future work . . . 108

Bibliography **110**

List of Tables

6.1 Comparisons between RMI, Java TCP sockets and C sockets (in microseconds) . . . 57

7.1 The RMI time split-up (in microseconds) . . . 66
7.2 Performance comparison for an array of TransportableTree (in microseconds) . . . 74
7.3 Performance comparison for a TransportableTree (in microseconds) . 74
7.4 Performance comparison for an array of TestClass (in microseconds) 74
7.5 Performance comparison for a TestClass (in microseconds) . . . 75
7.6 Performance comparison between Java RMI and CO_2P_3S-RMI (in microseconds) . . . 75

8.1 Hook methods of the DM_Mesh Pattern . . . 85
8.2 The Reaction-Diffusion application performance (in seconds) . . . 86
8.3 The Reaction-Diffusion application performance (in seconds) . . . 87
8.4 The PSRS application performance (in seconds) . . . 99
8.5 The sequence alignment application performance (in milliseconds) . 107

List of Figures

2.1 The template GUI 6
2.2 The framework from the user's point of view 6
2.3 Choosing the right pattern template 11
2.4 Parameter instantiation 12
2.5 Framework generation 12
2.6 Displaying hook methods 13
2.7 Editing hook methods 14
2.8 The MetaCO$_2$P$_3$S GUI 15
2.9 The constants of the DM_Mesh pattern in MetaCO$_2$P$_3$S 16
2.10 The GUI configuration of the DM_Mesh pattern in MetaCO$_2$P$_3$S . . 17
2.11 The structure of the DM_Mesh pattern in MetaCO$_2$P$_3$S 18
2.12 Pattern parameters in MetaCO$_2$P$_3$S 19

3.1 A basic Jini system 22
3.2 The JiniPrinterInterface 23
3.3 Source code for the proxy implementation 25
3.4 Source code for the service backend implementation 26
3.5 Source code for the service backend implementation (continued) . . . 27
3.6 The LUS registration code 28
3.7 The LUS lookup code 29
3.8 An advanced Jini system 30
3.9 Jini launching and shutting down scripts 31

4.1 The RMI programming model 34
4.2 An example of building an RMI server 36
4.3 RMI stub and skeleton 37
4.4 RMI registry 38
4.5 RMI architecture 39

5.1 The serialization interfaces 43
5.2 An example of using serialization 44
5.3 The serialized data of an instance of class Integer 48

6.1 The overall structure of DCO$_2$P$_3$S 52
6.2 Menu options to start up the environment 53
6.3 The window to configure the DCO$_2$P$_3$S environment 53
6.4 The interface of this service 56
6.5 Class hierarchy of distributed synchronization mechanism 58
6.6 The distributed monitor 59

6.7 The distributed barrier 60
6.8 Java Native Interface 62
6.9 The menu item to start up performance monitoring 62
6.10 The CPU usage of the machines in the environment 62
6.11 The memory usage of the machines in the environment 63
6.12 The run dialog of DCO_2P_3S 64

7.1 The stream format of the 1000-element integer array 70
7.2 The enhanced stream format of the `Integer` array using compact reference scheme 72
7.3 The stream format for a final array 73

8.1 Two examples of a general mesh and a rectangular mesh [24] 78
8.2 Neighbour stencils 78
8.3 Boundary exchange scheme 79
8.4 The class diagram of DM_Mesh 79
8.5 The graphical display of the DM_Mesh pattern template 81
8.6 The boundary conditions of DM_Mesh 82
8.7 The zebra strips generated by Reaction-Diffusion 83
8.8 The DM_Mesh template with instantiated parameters 84
8.9 Code fragment of meshMethod 84
8.10 Code fragment of mainMethod 86
8.11 Code fragment of the modified meshMethod 87
8.12 The class diagram of the Phases pattern [23] 88
8.13 The template GUI of the Phases pattern 90
8.14 The code template of the Phases pattern 90
8.15 The object diagram of DM_Distributor 91
8.16 The class diagram of DM_Distributor 92
8.17 Distribution strategies 93
8.18 The template gui of DM_Distributor 94
8.19 The method dialog of DM_Distributor 94
8.20 Editing one parallel method 95
8.21 The four phases of the PSRS algorithm [25] 96
8.22 The PSRSPhases pattern template 97
8.23 The PSRSDistributor pattern template 98
8.24 The MergePhases pattern template 98
8.25 The Wavefront pattern [3] 99
8.26 The DM_Wavefront pattern structure 101
8.27 Three matrix shapes supported in DM_Wavefront 102
8.28 The DM_Wavefront template GUI 103
8.29 The dependency dialog 104
8.30 The code template with all hook methods 105
8.31 The hook method instantiation dialog 106
8.32 The sequence alignment application 106

Glossary

- **API** Application Program Interface.
- **COM** Component Object Model. A Microsoft standard defining the interraction mechanism between components designed by Microsoft development tools.
- **CO_2P_3S** Correct Object-Oriented Pattern-based Parallel Programming System. **CO_2P_3S** is a parallel programming system designed by the systems group of the University of Alberta.
- **CORBA** Common Object Request Broker Architecture. A platform independent infrastructure technology defined by Object Management Group
- **DCO_2P_3S** Distributed CO_2P_3S. **DCO_2P_3S** is the system that was developed in the thesis research.
- **DSM** Distributed Shared Memory. A design approach of distributed systems.
- **GUI** Graphical User Interface.
- **HPC** High Performance Computing.
- **JDK** Java Development Kit.
- **Jini** This is not an acronym. **Jini** is a Java infrastructure technology to build Java distributed systems.
- **JNI** Java Native Interface. **JNI** is a technology that helps Java program to interact with native code.
- **JRMP** Java Remote Method Protocol. **JRMP** is used in RMI.
- **JVM** Java Virtual Machine. **JVM** is an abstract machine that interprets Java bytecodes.
- **LUS** Look Up Service. **LUS** is a registry service provided by Jini.
- **LRS** Lease Renewal Service. **LRS** is a lease renewal service provided by Jini.
- **MPI** Message Passing Interface. **MPI** is a programming interface to implement network communication in distributed applications.
- **RAM** Random Access Memory. A type of computer memory.

- **RPC** Remote Procedure Call. **RPC** defines the protocol of invoking remote methods in distributed computing.
- **RMI** Remote Method Invocation. **RMI** is an Java Object-Oriented version of **RPC**.
- **RRL** Remote Reference Layer. **RRL** is a layer in RMI's three-layered architecture.
- **PDP** Parallel Design Pattern. **PDP** encapsulates parallel designs in code templates; it is used in CO_2P_3S to facilitate parallel application designs.
- **PSRS** Parallel Sorting by Regular Sampling. **PSRS** is a sorting algorithm.
- **PVM** Parallel Virtual Machine. Like **MPI**, **PVM** is another programming inteface used to transfer (data) messages between distributed processes.
- **SOAP** Simple Object Access Protocol. **SOAP** is a platform-independent communication protocol.
- **TCP** Transmission Control Protocol. **TCP** is a connection-oriented communication protocol.
- **UDP** User Datagram Protocol. **UDP** is a connectionless communication protocol.
- **XML** EXtented Marking Language.

Chapter 1

Introduction

1.1 Motivation

Parallel programming offers the potential to significantly improve the performance of computationally-intensive programs. Sequential solutions to such problems may take hours, days or even weeks to finish. In a shared-memory environment, a multithreaded program can create multiple threads of control, with each performing an individual task such as intensive disk I/O, network communication or computation. Threads can be scheduled to run in parallel by controlling the system's resources in different time slices. In a distributed-memory environment, a program can be expanded to incorporate multiple machines to gain more computing power and resources. This has the potential to achieve even greater parallelism and faster execution than a multithreaded program.

As high-speed networks become ubiquitous, computing resources at distributed locations can be federated for large-scale parallel processing. With the advent of high-performance workstations with high-bandwidth low-latency network connections, many attempts [35, 9, 1, 2] have been made to form commercial networks of workstations as supercomputers, at a much lower price than those of normal multiprocessors. This research addresses problems related to parallel programming in the distributed-memory environment.

Despite the appeal of higher performance, parallel programming has coding, performance and correctness issues that are not present in sequential programming. These issues are related to communication overhead, synchronization complexities, data inconsistency, job distribution strategies, load imbalance, deadlock and nondeterministic behavior. In a distributed-memory environment, these problems are even harder to solve than in a shared-memory environment. Faced with the complexities associated with parallel programming, only experienced programmers can make full use of it.

Many efforts have been made to mitigate such programming complexities at different levels. MPI and PVM [18, 16] provide low-level programming interfaces and libraries for explicit and efficient point-to-point communication among distributed processes. Java/DSM, cJVM and Hyperion [5, 44, 12] simulate a shared-memory environment by providing a multithreaded model that overlays a distributed-memory environment. These approaches relieve many of the complexities of developing distributed parallel applications. However, parallel programming using these tools is

still a highly skilled task. A good parallel solution needs not only experience in algorithm design but also a thorough understanding of the underlying architecture. A parallel design can be error-prone and can consume a significant amount of the total development time. As a result, even though such tools may facilitate a parallel design, the quality of the resulting implementation largely depends on the competence and experience of the programmer. Newcomers with little experience in parallel programming cannot make full use of such tools to realize the performance potential of their programs.

To solve this problem, a system called Correct Object-Oriented Pattern-based Parallel Programming System [23](CO_2P_3S, called "cops") has been designed by our research group at the University of Alberta. As its name implies, CO_2P_3S uses software engineering techniques to alleviate the complexities involved in parallel programming. The underlying philosophy of the CO_2P_3S system is based on abstraction and development tools, both of which are widely used in the area of software engineering. Parallel solutions to different problems share a certain number of commonalities that can be captured by abstractions such as object-orientation, design patterns, and frameworks. These abstractions can facilitate both design and code reuse to reduce parallel software development complexities. CO_2P_3S supports the automatic generation of object-oriented frameworks from design patterns. The frameworks which encapsulate the parallel infrastructures, can easily be instantiated into concrete parallel applications.

- **Object-oriented programming** enhances the separation of concerns and code reuse by encapsulating data and operations into different self-contained objects. Java is an object-oriented language that provides complete support for multithreaded programming and facilities—RMI[1] and sockets—for distributed computing. These characteristics, as well as platform independence and code mobility, make Java a good tool for high performance distributed and parallel computing.

- **Design patterns** [15] record general algorithmic solutions to recurring problems in different domains. Software designers can benefit from the experience of experts by adapting one or more design patterns to a specific problem.

- **Frameworks** [15] provide a set of classes that implement a software architecture in a certain context. A framework defines a general structure for a group of applications in one specific domain. A programmer can instantiate frameworks by filling in hook methods to quickly create concrete applications.

The CO_2P_3S parallel programming system combines these three abstraction technologies to create a layered design model which is the basis of the PDP (Parallel Design Pattern) process for correct and rapid development of parallel programs. A new construct, the **design pattern template**, is used in CO_2P_3S to facilitate automatic code generation from design patterns to frameworks. Three layers—**the patterns layer, the intermediate layer and the native code layer**—are involved in this layered design approach. The patterns layer constrains the programmer to focus on only application-specific details for the purpose of guaranteeing the

[1]RMI stands for Remote Method Invocation, a distributed object calling model used in Java-based distributed computing.

structural correctness of an application. The complexities of parallel programming are pushed down to the lower levels, and the user can focus on the sequential parts of the algorithm and ignore the parallelism by working at the higher level. Furthermore, unlike other parallel programming systems that leave the programmer no options for performance tuning. CO_2P_3S gradually exposes the parallel design details to the programmer through the lower two layers to allow users to customize applications to enhance performance.

When this research began, CO_2P_3S supported only parallel program design in a shared-memory environment. However, as low-cost and high-performance networks of workstations become ubiquitous, extending CO_2P_3S to support distributed-memory programs will increase its usability and applicability. This dissertation discusses how a Distributed CO_2P_3S (referred to as DCO_2P_3S) environment was designed and implemented to support parallel computing in a distributed-memory environment.

Jini, RMI and JDK-Serialization are the main Java technologies used to build DCO_2P_3S. Jini is a Java-centric infrastructure technology for building reliable distributed systems. The network is used as a central connecting medium in such systems. Components—hardware or software—can be dynamically federated by the network to offer services or interact with each other regardless of their network location and communication protocol. Therefore, Jini creates a network of services that is flexible, reliable and scalable. RMI provides a convenient way to have distributed Java objects interact with each other. Using RMI, an object can easily invoke methods of another object in a remote JVM in much the same way that local method calls are issued. RMI is used as the communication scheme of entities in the DCO_2P_3S environment. Both Jini and RMI use JDK-Serialization as a vital component for object mobility. JDK-serialization is used to serialize an object into a linear stream of bytes that can be transferred over the network and de-serialized to re-create an identical object in another process. DCO_2P_3S uses parallel design pattern templates to generate object-oriented framework code compatible with the Jini and RMI standard.

1.2 The scope of this dissertation

The goal of the DCO_2P_3S environment is to aid in the design and implementation of high-performance distributed-memory parallel applications. Efforts have been made to address many performance considerations involved in the design of distributed parallel applications, such as communication overheads, synchronization costs, process spawning and killing, and real-time performance monitoring.

This research uncovered problems with RMI and JDK-serialization that seriously impacts the performance of parallel applications in a distributed-memory environment. Although Java RMI is easy to use and it offers many advantages, it has two main defects.

- Firstly, RMI has a TCP-based network subsystem which does not work efficiently and cannot make use of other network technologies (like Myrinet). The standard JDK-serialization, an important component used by RMI, is not suitable for high-performance computing.

- Secondly, although JDK-serialization provides high reliability and security guarantees for distributed computing at the Internet level, it is too slow and storage inefficient for high performance applications in its present form.

These problems directly affect Java's role in the area of high-performance computing. This dissertation analyzes them in detail to find the source in Sun's JVM implementation. It also describes some enhancements to RMI and JDK-serialization to make them more suitable for high-performance computing.

1.3 Dissertation organization

The rest of the dissertation is organized as follows. Chapter 2 introduces the CO_2P_3S parallel programming system, describing its layered design model, the PDP process, its main features and components. Chapter 3 gives an overview of Jini, its motivation, features, and architecture. Chapter 4 presents the RMI (Remote Method Invocation) technology. Its layered architecture and its characteristics are discussed. In Chapter 5, JDK-serialization is explained in detail, including the wire protocol, complete serialization and de-serialization process, its effectiveness in distributed computing at the Internet-level and its weakness in high-performance computing. In Chapter 6, the overall architectural design of DCO_2P_3S is discussed. Three aspects of the DCO_2P_3S environment are explained in detail. First, the considerations for using RMI instead of TCP sockets as the inter-process communication scheme are given along with a performance comparison. Secondly, the distributed synchronization model used in DCO_2P_3S is described. Finally, the process management and performance monitoring mechanism are presented. In Chapter 7, a new-version of RMI and a modified serialization mechanism are introduced to address the performance issues related to DCO_2P_3S. In Chapter 8, three new distributed parallel patterns, DM_Mesh (Distributed Mesh), DM_Distributor (Distributed Distributor) and DM_Wavefront (Distributed Wavefront) are presented in detail. Finally, the conclusions of this dissertation are provided in Chapter 9.

Chapter 2

CO_2P_3S Introduction

2.1 Introduction to CO_2P_3S

Programmers can enjoy significant performance improvements by taking advantage of parallel programming. However, such improvements come at a cost. The programmer has to consider both the sequential application algorithm and the parallel design. Implementing the parallel design can be a difficult and error-prone task that may benefit from a tool such as a parallel programming system.

A parallel programming system, CO_2P_3S, has been designed by our research group at the University of Alberta. The CO_2P_3S system applies abstractions and development tools to the area of parallel programming. Abstractions are widely used in the software engineering discipline for sequential programming. Abstraction techniques such as object-oriented programming, design patterns and frameworks can facilitate automatic code generation and code reuse so that software development complexities can be significantly reduced. We apply these techniques to the parallel programming domain to encapsulate the expertise in parallel design.

The research goal of CO_2P_3S is to allow users to easily parallelize their sequential programs without introducing errors. This is achieved by using parallel design patterns. Parallel design patterns capture commonalities of solutions to different parallel applications. A framework implements a design pattern in a certain context, providing parallel structural designs and hook methods. CO_2P_3S also provides a meta tool to support the addition of new design patterns and the modification of existing ones. Design patterns in CO_2P_3S are generative. Frameworks can be automatically generated from the corresponding pattern(s) based on programmer adaptations. Figure 2.1 shows CO_2P_3S with a new application using one parallel design template, a Mesh.[1] The right pane in the figure provides a graphical user interface to let a programmer specify problem-specific information. After being adapted, the design template can generate a framework which provides all the parallel structures required to parallelize this application. By filling in the hook methods provided in the framework with application-specific code (Figure 2.2), the user can quickly create a concrete parallel application without even knowing much about the concurrency.

To address correctness and openness[2] in the design of parallel programs, CO_2P_3S

[1] One of the patterns supported in CO_2P_3S

[2] Openness means that the system provides users access to all the code used in architectural

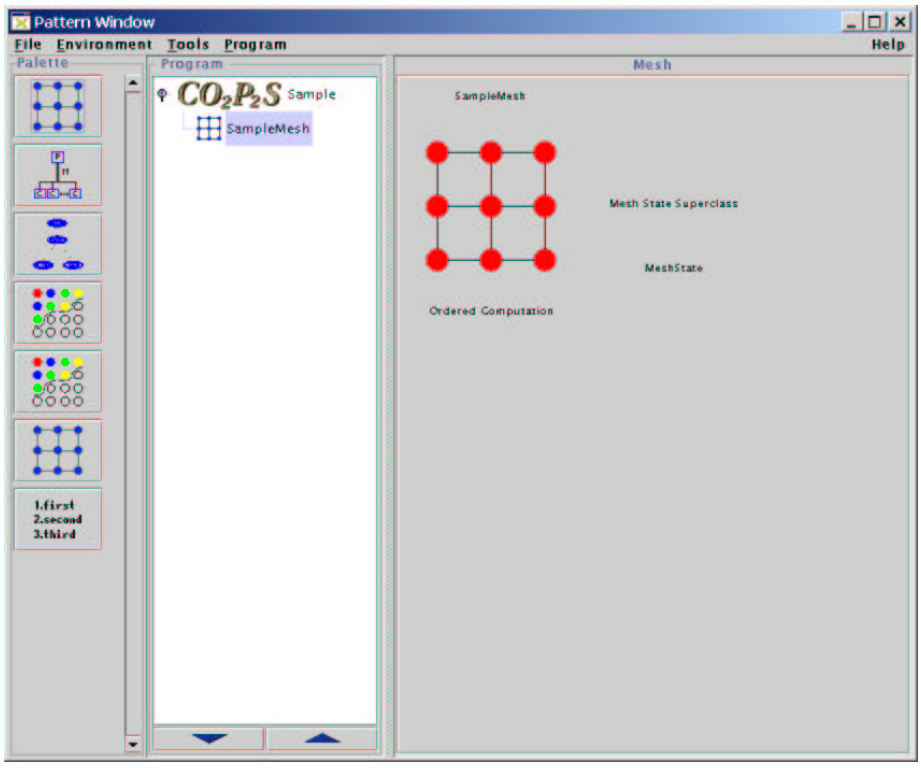


Figure 2.1: The template GUI

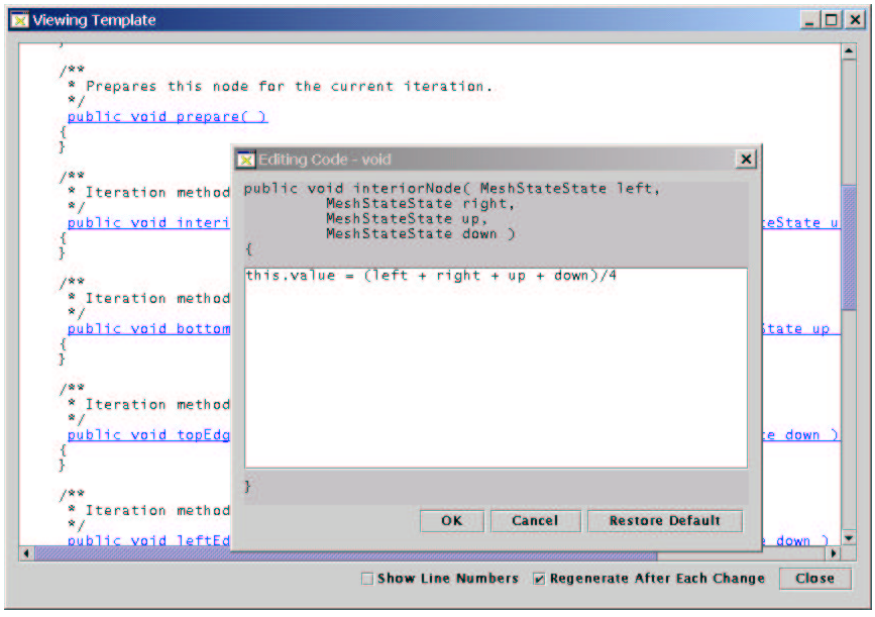


Figure 2.2: The framework from the user's point of view

applies the Parallel Design Pattern (PDP) process. The PDP process combines the three abstraction technologies mentioned previously into a layered design approach, including the Patterns Layer, the Intermediate Code Layer and the Native Code Layer. The user can focus on the sequential components of a parallel program by working at the Patterns Layer. The complexity of parallel structure design is handled by lower layers.

Section 2.2 gives a brief overview of the PDP process, including its three abstraction layers and the steps needed for a user to build a parallel application. Section 2.3 discusses design patterns and frameworks. In CO_2P_3S we apply design patterns in the parallel programming domain to achieve the concept of *Parallel Design Patterns*. We describe how CO_2P_3S supports the adaption of a generic pattern to generate a more specific framework.

At first CO_2P_3S supported only four design patterns: Mesh, Phases, Distributor and Wavefront. The capacity of the pattern library was limited. Such an embarrassing situation exists in many pattern-based programming systems and prohibits them from gaining greater acceptance. CO_2P_3S solved this problem by introducing a new tool called MetaCO_2P_3S [7]. This tool helps the developer generalize, create and modify design patterns in a well-defined manner. It enables the pattern library to be extended incrementally by developers. In fact, MetaCO_2P_3S was used to design and implement all of the distributed pattern templates described in this dissertation. MetaCO_2P_3S will be introduced in Section 2.4.

2.2 An Introduction to the Parallel Design Pattern Process

The PDP process is a pattern-based design approach. It combines object-orientation, design patterns and frameworks into a three-layered design model: the Patterns Layer, the Intermediate Code Layer and the Native Code Layer. This three-layered design model addresses both the correctness and openness issues of a pattern-based parallel program design.

- **The Patterns Layer** generates a structurally correct framework for the user's program. It is the level that the user can directly access. Provided with certain domain-specific information, a pattern can be adapted to a framework which encapsulates the overall control flow and the parallel structures of the user's program. The user only has access to the sequential hook methods provided in the framework to create concrete applications, as depicted in Figure 2.2. In addition, the programmer must provide code (a driver program) to create the application objects instantiated by the framework. Such code is used at execution start time and, if necessary, also at execution termination time. The Patterns Layer includes the generated framework code, the hook method code, the initialization code and the result processing code. To facilitate the adaptation from design patterns to frameworks, we use the *design pattern template*, a parameterized construct that implements a design pattern. Different pattern and parameter combinations result in CO_2P_3S generating different frameworks with varying parallel structures.

code.

- **The Intermediate Code Layer** uses a high-level explicitly parallel programming language to describe the parallel infrastructure, synchronization and communication in the framework code. This high-level language uses parallel primitives defined in the Native Code Layer to express the parallelism. The Intermediate Code Layer addresses the openness issue by making the framework code available to the programmer. Even though a design pattern is general enough to provide solutions to diverse applications, it is still unable to offer the best design for certain situations, which cannot be easily captured in design patterns. The CO_2P_3S system allows the programmer to modify the control flow code and the parallel structure in the framework to better fit specific application requirements. When using this level, the programmer is responsible for maintaining the correctness of the parallel structure of the program.

- **The Native Code Layer** uses a full-fledged object-oriented programming language, as well as certain libraries, to implement the primitives used in the Intermediate Code Layer. Primitives such as the barrier, mutex lock and communication channels can be used to express the parallelism. The implementation of parallel primitives can be tuned based on the architecture of the execution environment.

Based on this three-layer design mode, the PDP process is a five-step methodology for the programmer to create a parallel application in the CO_2P_3S environment:

1. Identify one or more parallel design patterns suitable for a given application and choose the corresponding pattern templates provided in CO_2P_3S.

2. Adapt the chosen pattern templates. A programmer selects application-specific template parameters allowing them to customize the design patterns to their application.

3. A framework is generated with application-dependent sequential hook methods. Fill in the hook methods with application-specific sequential code as well as write initialization code and result processing code to build a complete parallel application. The first three steps focus on the Patterns Layer and they are necessary to create an application.

4. Evaluate the resulting application for initial performance results. If they are not acceptable, inspect the framework code in the Intermediate Code Layer and edit it.

5. Re-evaluate the performance of the modified application. If it is still not acceptable, the programmer can do anything at the Native Code Layer to enhance the performance.

In summary, the five-step PDP process allows the user to quickly create a parallel application. The first three steps are necessary for beginners to create parallel applications from scratch. The last two steps allow more advanced programmers to incorporate their own experiences into a parallel design. The use of the PDP process will be further illustrated in the descriptions of supported distributed pattern templates described in Chapter 8.

2.3 Design patterns and frameworks

2.3.1 Design patterns

A design pattern records general algorithmic solutions to a recurring problem in different contexts [15]. It is widely used in object-oriented software designs. A design pattern captures the knowledge and experience of expert programmers to be reused by others. In general, a design pattern consists of a pattern name, intent, motivation, applicability, structure, participants, collaborations, consequences, implementation and sample code [15].

The most common form of a design pattern is a document, which describes these components in a textual form. A design pattern in this form is more like an instruction sheet instead of a code template that a programmer can directly work on. Such design patterns are easy to read and access. Instead, after identifying a design pattern for a real problem, programmers have to adapt the pattern and convert it to code manually. The qualities of the adaptations and conversions made by different programmers vary drastically because of their different interpretations of the design pattern and their programming proficiency. Experts may reuse their previous design experiences to quickly adapt the design pattern. However, without tool support, it is hard for a novice to create proper class hierarchies from scratch only based on applying pattern documentation to a real context. Thus, even though a design pattern in textual form can be easily accessed and shared by programmers, the real code implementations are not readily available to programmers.

2.3.2 Frameworks

A framework is a set of collaborating classes which implement a software architecture. It can be instantiated to a specific application by sub-classing the framework classes and filling in its hook methods [15]. A framework defines a general design solution to a group of applications. It illustrates the overall architectural design of the solution in this specific domain [15]. Based on a properly constructed framework, a programmer can easily obtain a correct overall architecture and build an application, because only application specific issues need to be focused on. A framework is like a main program which defines an execution sequence that invokes a series of methods. These methods, called hook methods, should be implemented by the user to provide application specific details. In the parallel programming domain, the idea of using frameworks is especially useful. A framework can encapsulate the parallel structure as well as other abstractions. Frameworks allow the user to add application specific code by sub-classing and filling in hook methods.

Although design patterns and frameworks both provide reusable designs for groups of applications, there are several differences between them:

- a design pattern is more generic than a framework;
- a design pattern usually appears in a textual format while a framework is implemented with code;
- a design pattern can only be adapted to a number of frameworks, and a framework can be composed of several different design patterns. There is no one-to-one mapping between design patterns and frameworks.

2.3.3 Generative design patterns

A generative design pattern extends an ordinary design pattern with the capability of automatic code generation. A generative design pattern is a parameterized set of code templates that can be instantiated to select any one of a group of frameworks [4]. It records the descriptive properties [6] of an ordinary pattern in the form of template parameters with domain restrictions. Different combinations of the parameters help generate a diversity of frameworks. There have been several efforts [14, 17, 11, 38] to create generative design pattern. In CO_2P_3S, generative design patterns are implemented as design pattern templates. A design pattern template is a construct that is used to encapsulate a design pattern and its corresponding frameworks so as to facilitate the automatic generation of framework code from the adapted design pattern. One meta tool, called MetaCO_2P_3S, is devised to help create design pattern templates [7].

2.3.4 Parallel design patterns

Parallel programming is facing a similar dilemma as object-oriented programming once did: it promises the potential for significant improvements to programs. However it requires strong skills to realize this potential. After identifying the parallelism in an application, a programmer has to consider many aspects to make a good design of the concurrency, including:

1. the data layout for achieving good cache performance and minimal communication,
2. the load balancing strategies and task granularity,
3. the special characteristics of the target platform, which may affect the application performance,
4. the best communication scheme,
5. efficient coordination and synchronization schemes between processing units, and
6. the design's scalability and portability across different platforms.

In addition to these design complexities, parallel programming is error-prone. A small error or omission may result in non-deterministic behavior. The program is often difficult to debug and trace.

Fortunately, different parallel applications share certain commonalities in the design of their overall parallel structures, synchronization and communication schemes. Such similarities can be abstracted and encapsulated into design patterns. As an innovative application of the general design pattern in parallel programming, the *parallel design pattern* encapsulates both the general design solutions of a recurring problem and the concurrency designs needed to parallelize it. Furthermore, we apply the concept of the design pattern template to the parallel design pattern. Parallel design pattern templates provide correct and readily re-usable concurrency designs for parallel program development.

2.3.5 From parallel design patterns to frameworks to programs

After choosing an appropriate parallel pattern template, the user selects parameters that are used to guide the code generation process. The parameters help refine the pattern's code template and provide domain-specific information to generate a compact and efficient framework.

A code generator is used to facilitate the automatic framework generation. This process is based on the conditional compilation of a pattern template. During the design process of a given pattern template, the designer associates a code fragment with each value of one parameter or a value combination of several parameters. The code fragments reflect design solutions based on different situations as represented by the template parameters. Frameworks are generated by sewing together different code snippets based on the combination of parameters selected.

The following screen shots depict the step-by-step process of generating a framework from a pattern template.

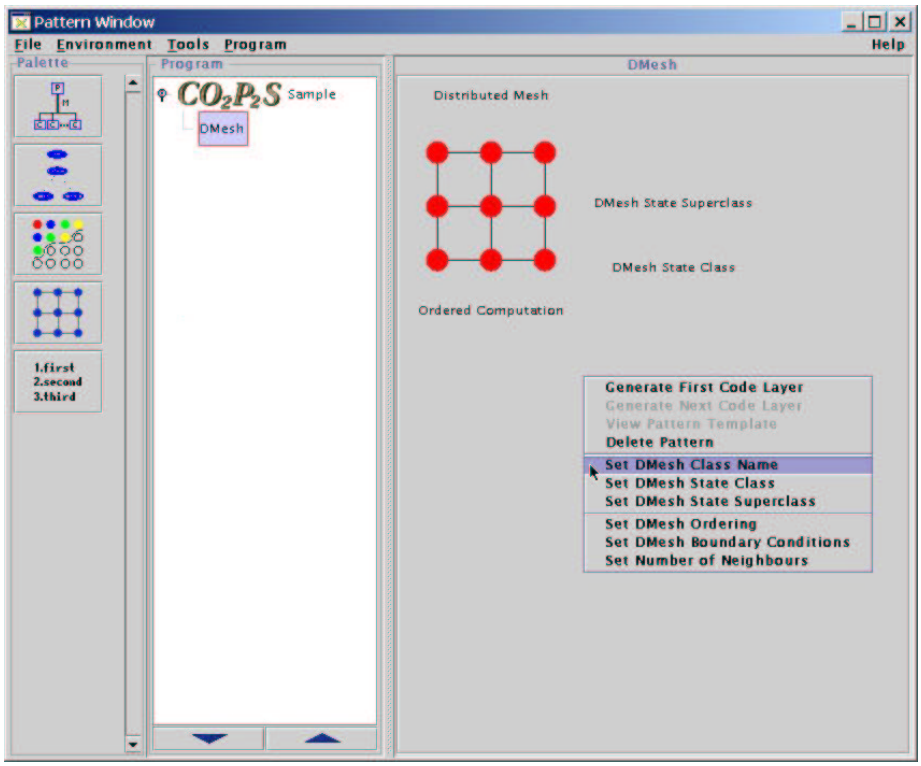


Figure 2.3: Choosing the right pattern template

Figure 2.3 shows that a DM_Mesh pattern[3] is chosen by the programmer. As described in Chapter 8, this pattern has six parameters. At the beginning, all these parameters are set to default values. The programmer needs to instantiate these parameters to provide the template with domain-specific information. A pop-up menu shown in Figure 2.3 provides an interface for the programmer to set all the parameters. The menu contains six menu items, each invokes a different pop-up

[3]Distributed Mesh Template, which will be introduced in Chapter 8.

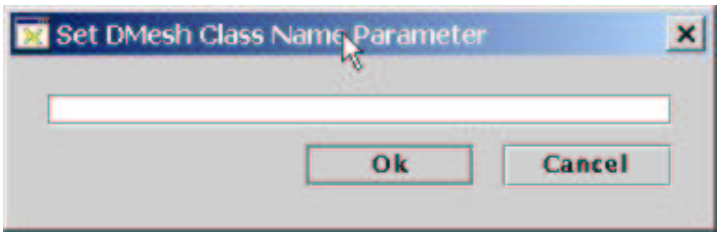


Figure 2.4: Parameter instantiation

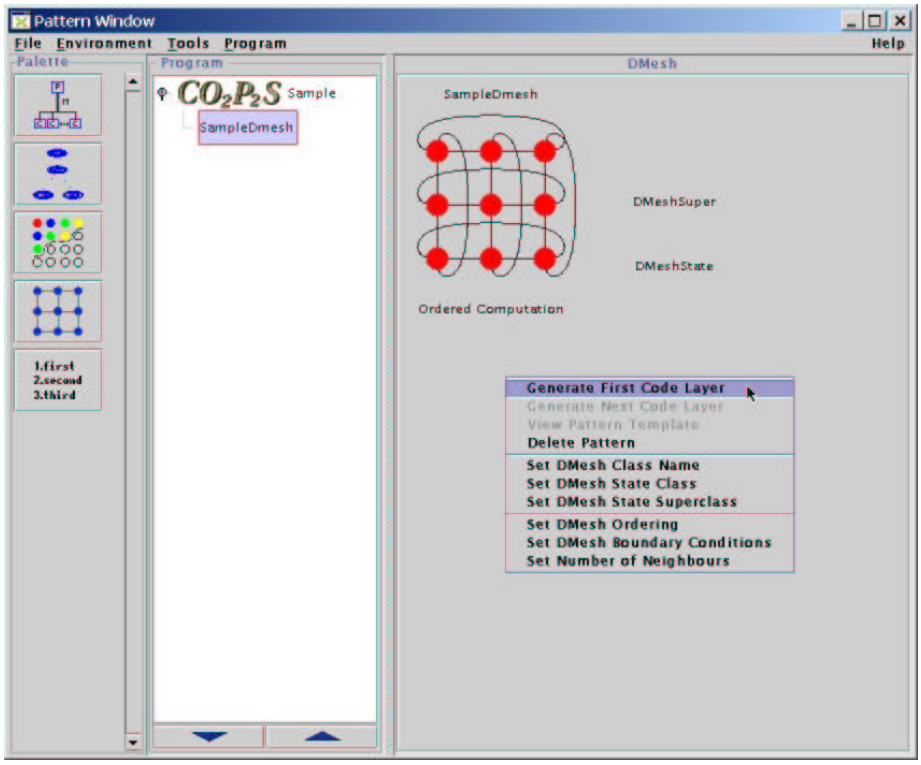


Figure 2.5: Framework generation

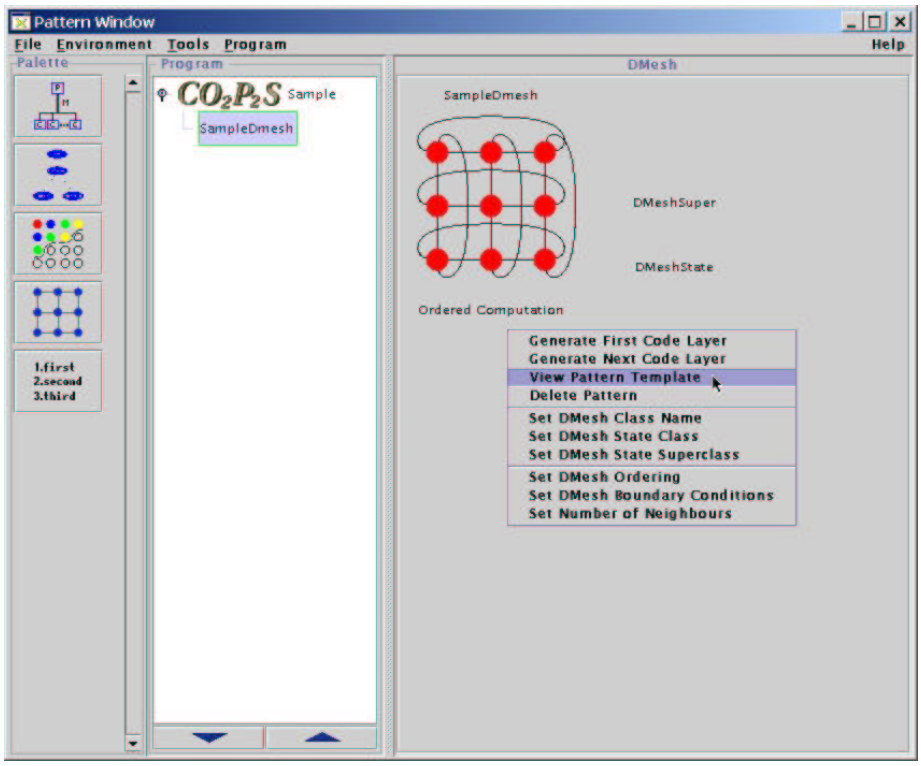


Figure 2.6: Displaying hook methods

window to set one parameter. One such window is depicted in Figure 2.4. Once all the parameters are instantiated, the first menu item in the pop-up menu will be chosen (Figure 2.5) to start the code generation process. The code generator parses the pattern code template and concatenates together different code fragments according to the instantiated parameter values to create the application's code framework.

After all this has been done, the rest of the menu items in the pop-up menu are enabled (Figure 2.6), notifying the programmer that the framework is ready to be instantiated. The programmer is now able to insert application-specific details into the generated framework. After clicking the "View Pattern Templates" menu item (Figure 2.6), the user-accessible part of the framework is shown in a pop-up window (Figure 2.7). The contents are listed in an HTML file. Clicking a hyperlink as shown in Figure 2.7 will launch a new window which allows the programmer to input the sequential application code.

This step-by-step process helps the programmer provide domain-specific information to guide the CO_2P_3S code generator to create efficient framework code for a given application.

2.4 MetaCO_2P_3S introduction

Although design patterns try to be generic to cover as many situations as possible, they are still not comprehensive enough to handle some requirements and ramifications which may appear in future applications. The limited capacity and extensibil-

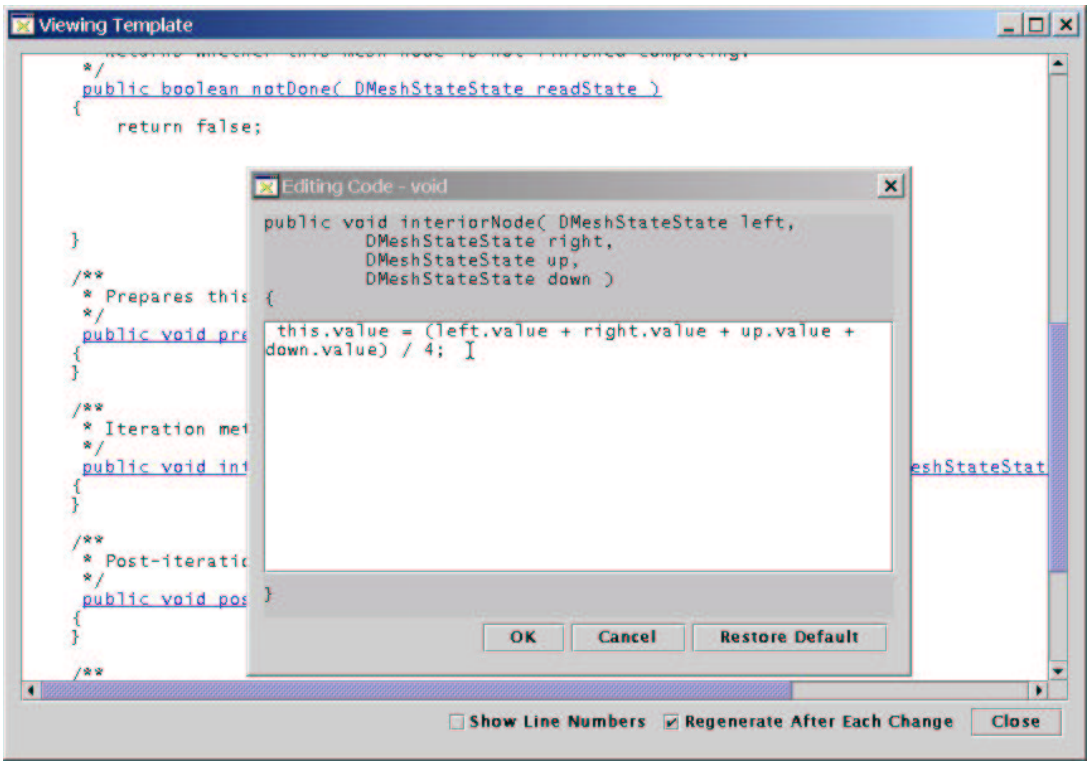


Figure 2.7: Editing hook methods

ity of pattern libraries is one of the major reasons that pattern-based programming systems have not gained greater acceptance.

CO_2P_3S has addressed this problem by introducing a new tool—MetaCO_2P_3S. Using MetaCO_2P_3S, a pattern designer can identify and generate a new design pattern and add it into the CO_2P_3S pattern library.

MetaCO_2P_3S provides a means of constructing generative design patterns (Section 2.3.3) in a structured way. Each generative design pattern constructed using MetaCO_2P_3S contains two parts: the GUI part and the pattern template part. The GUI part allows the user to specify pattern parameters through a graphical user interface. The pattern template part follows the three-layered design model (Section 2.2) to implement a design pattern. All design patterns created in MetaCO_2P_3S are stored as XML and Java files, which are easy to store and share with other systems.

Figure 2.8 shows a graphical display of the DM_Mesh template in MetaCO_2P_3S. The left pane of this figure displays all the components in the template, including constants, classes, parameters and GUI configurations.

The constants and GUI configurations define the GUI part of the pattern template. The constants in Figure 2.9 describes the text labels that will be displayed on the template GUI as in Figure 2.3. The GUI configuration (Figure 2.10) sets up the graphical display, such as the picture which shows the pattern's object structure and the position of the constant labels in the window of the pattern template that will be used by a programmer.

The class names and parameters are the core of each template since they imple-

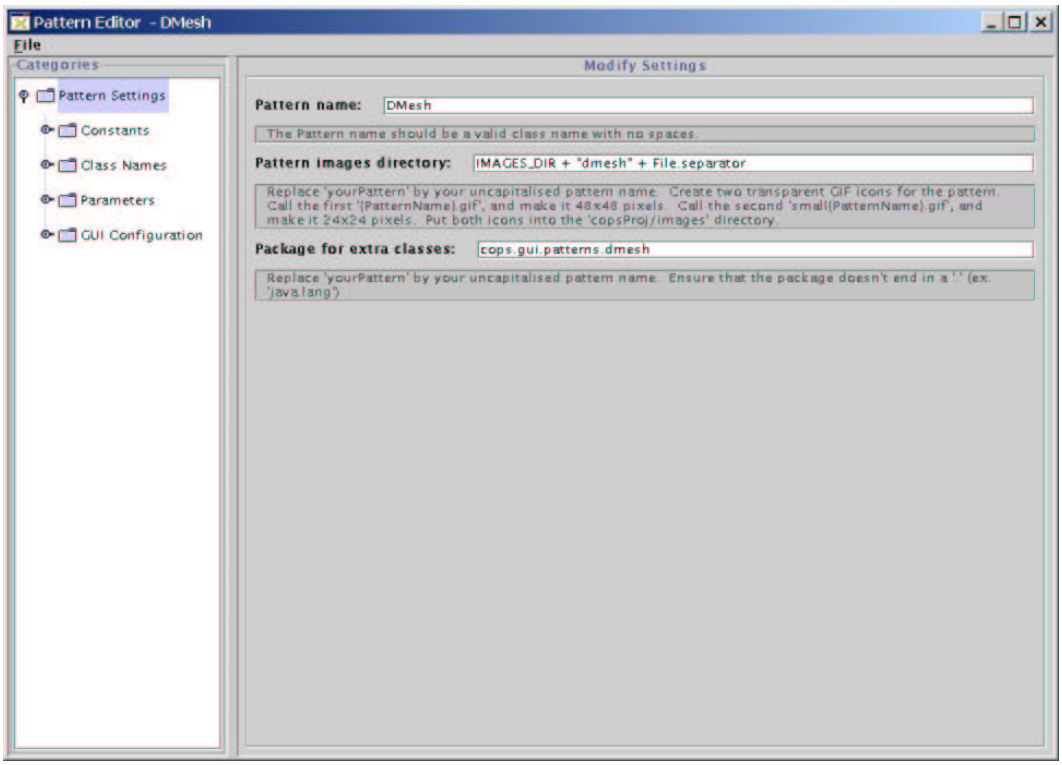


Figure 2.8: The MetaCO$_2$P$_3$S GUI

ment a design pattern using a code template.

The classes consists of all necessary components that contribute to the implementation of a design pattern. Different classes have different attributes, reflecting whether a class contributes to the private parallel structure or offers hook methods for the programmer to input application-specific details. The class named DM_Mesh in Figure 2.11 describes the collector class in the DM_Mesh pattern; the collector class is a framework class that encapsulates the structure information and cannot be accessed by the user.

An important step in creating a design pattern template is to generalize and quantify the descriptive properties of an ordinary design pattern to formal parameters with domain restrictions. Each pattern parameter has a direct effect on the code that is generated by a pattern. After identifying all possible parameters, the pattern designer provides code fragments for each legal combination of parameters. A design pattern thus becomes a template for a group of frameworks. For example, Figure 2.12 lists a parameter of the DM_Mesh pattern called "_neighbors", which specifies how many neighbour nodes (4 or 8) are needed to compute the mesh node.

MetaCO$_2$P$_3$S was a key tool in constructing DCO$_2$P$_3$S to support the generation of parallel design patterns for a distributed-memory environment. It allowed us to focus on designing efficient parallel structures, communication and synchronization mechanisms for new distributed-memory version of patterns.

The next three chapters describe three Java technologies that are used in the implementation of DCO$_2$P$_3$S.

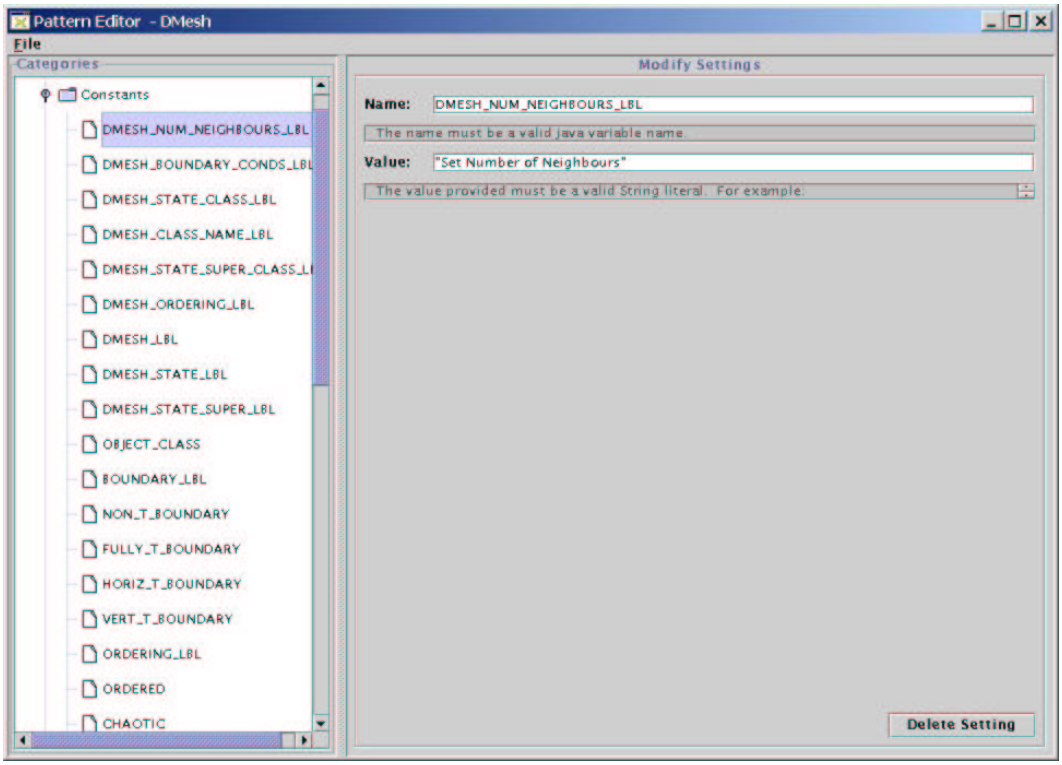


Figure 2.9: The constants of the DM_Mesh pattern in MetaCO_2P_3S

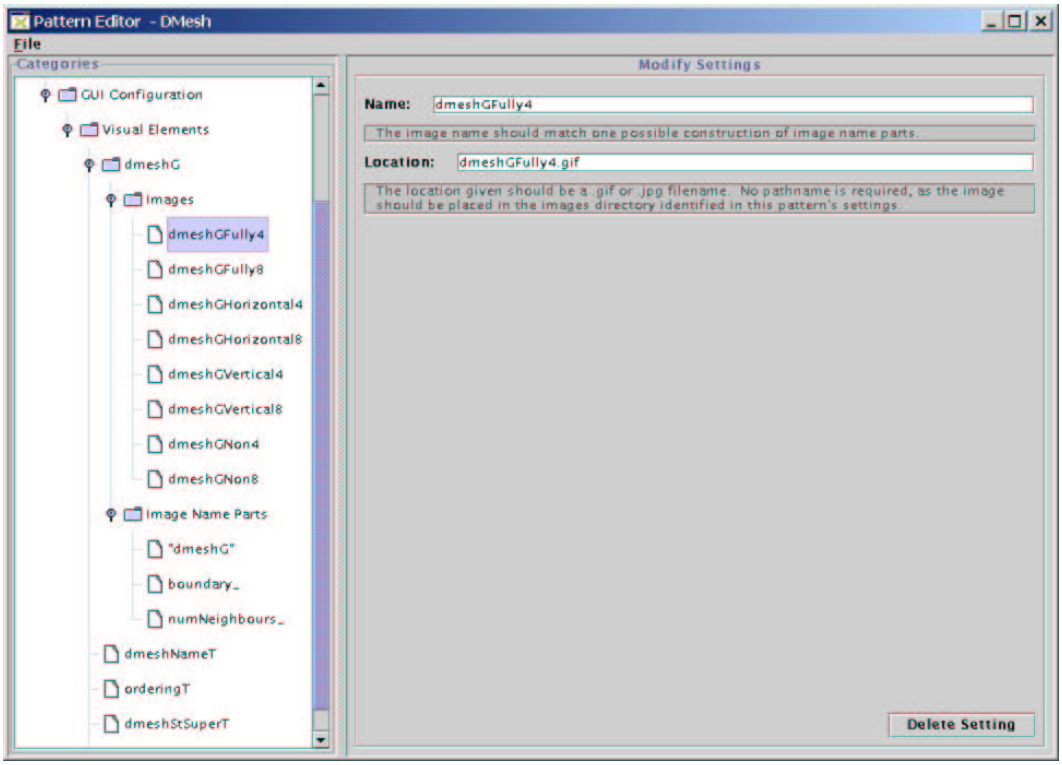


Figure 2.10: The GUI configuration of the DM_Mesh pattern in MetaCO$_2$P$_3$S

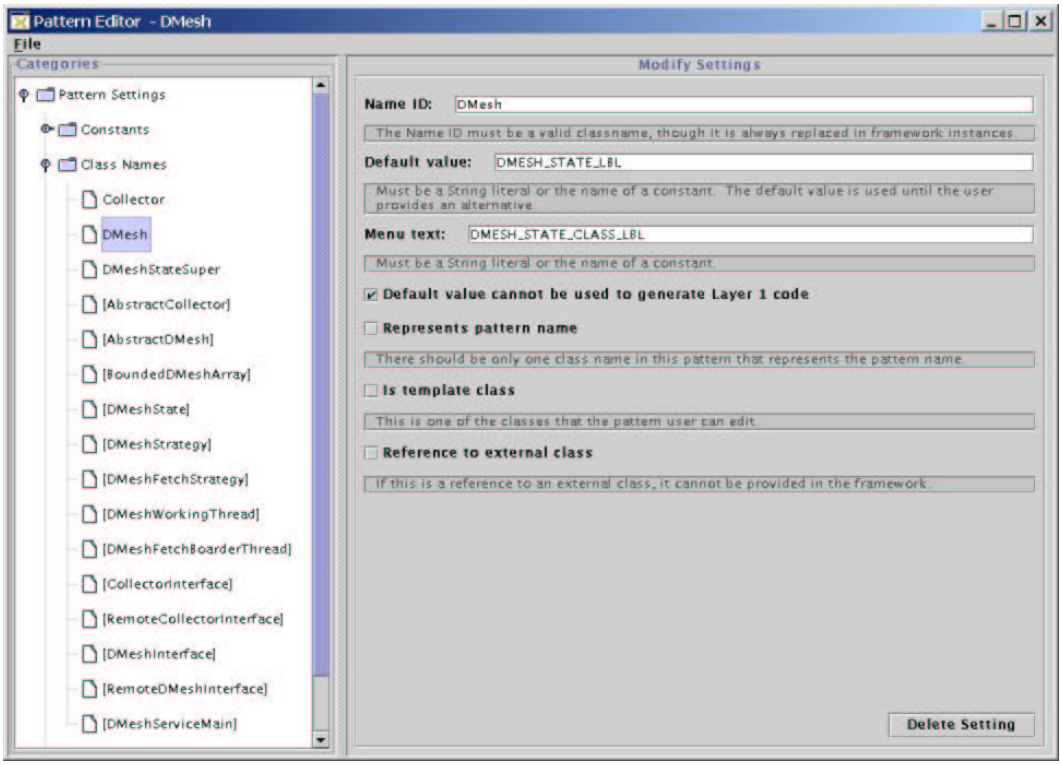


Figure 2.11: The structure of the DM_Mesh pattern in MetaCO_2P_3S

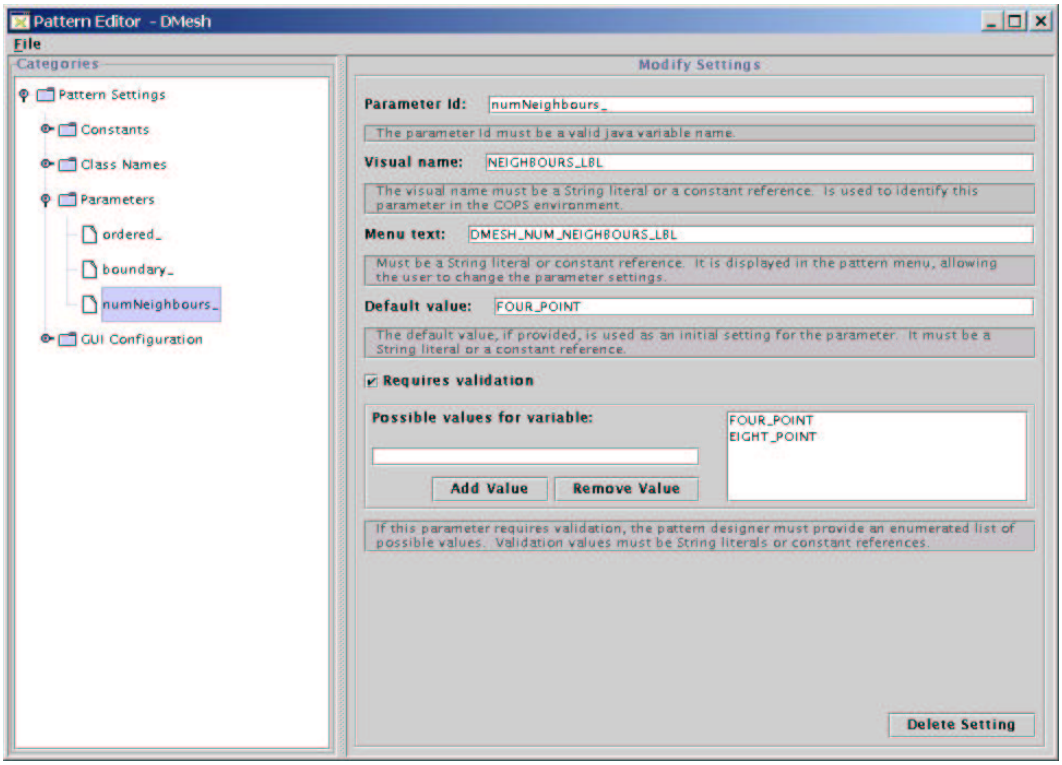


Figure 2.12: Pattern parameters in MetaCO$_2$P$_3$S

Chapter 3

Jini overview

3.1 Introduction

Jini is a newly-emerging Java technology for building distributed computing systems. It offers "network plug and play" [32] for hardware and software components. They can be federated in a Jini-enabled network without being previously configured. Moreover, it is widely claimed that Jini can be used to build highly personalized, cross-network, platform-independent, flexible and intelligent web services [32]. In Jini, the view of a conventional computing system, which consists of hardware (computers, peripheral equipments) and software is changed to one that is composed of services (hardware devices and software components) and clients inter-connected by a central network [41]. In a Jini system, there is no difference between the hardware and the software: hardware devices and software components are treated in the same way. Components over a network can be dynamically federated to provide web services or to interact with each other regardless of their network locations and communication protocols. Jini offers a network of services with **flexibility, reliability and scalability**:

1. A Jini system is **flexible** in terms of the following attributes:

    - A service can dynamically join, leave and access a service group. A central bootstrapping service, the Jini lookup service, takes care of all service registrations and discoveries in a Jini system. It helps a service to dynamically attach to and detach from a Jini system. Thus, a distributed computing system can be constructed by simply plugging in different services.
    - Components can be incrementally and individually updated, i.e, a component can be updated several times without involving other components or shutting down the whole network. This will be explained in Section 3.3.2.
    - The communication scheme between a service and a client is open-ended. A service can define customized communication protocols such as RMI, SOAP, TCP/UDP Sockets and CORBA and their implementation details are totally transparent to the clients. To access a Jini service, a client only needs to know the interface.

2. Combining technologies such as distributed leasing, distributed transactions and distributed events with the Java security model, the Jini infrastructure provides **reliability** for distributed systems. The Java built-in security model ensures the safety of running code downloaded from remote machines; the strong typing of Java helps identify the class of an object created on other machines; the distributed leasing model addresses partial failures inherent in distributed systems; and the distributed transaction model helps distributed services coordinate with each other in mission-critical tasks.

3. Jini can not only group services across different networks together, but also manage different collections into hierarchies to form systems with higher-**scalability**.

3.2 Applications

Jini technology has many applications. Jini has been used to build web servers, information systems, mobile computing applications, mobile appliances and network storage [32]. Here is a simple example of using Jini. A printer joins a Jini system by registering with a central bootstrapping service to announce its presence and properties. Properties may include the printer type (laser or bubble jet, color or gray scale), the printing speed, current states, etc. A client can find this printer by querying the central registry and specifying its requirements. In this way, the client can use this printer without even knowing the printer's exact network location or having to be specially configured for the printer. The client can use well-known APIs for basic functionalities such as printing one page or several pages, or it can do more complex and more specific jobs by understanding the printer service's interface before use. The printer can be upgraded without making the client aware of it. The client can still use the printer in the same way as before. If it needs new functionalities, it can explicitly check the extended service API of the printer. It can use Java Reflection, which offers the capability to discover the structural composition of objects at runtime to obtain the extended interface. Alternatively, the client can browse the graphical user interface provided by the printer service to obtain extended services.

As a more complex example, consider connecting a scanner to the Jini system to become a Jini service. The scanner can actively locate the printer service itself, by searching the central registry for the printer, and then use it to print out scanned documents. Thus two services on the network can coordinate with each other to fulfill one task, or to become a more useful service.

3.3 How Jini works

3.3.1 The Jini architecture

A Jini system consists of three parts:

- **Infrastructure components** help federate services in one or more networks into a distributed system. They provide mechanisms for a service to dynamically join and detach from a distributed system, and to easily discover and

deliver services without making its clients aware of its physical location. The basic infrastructure components are: an http server, an `rmid` (a daemon object of the RMI system, which will be introduced in Chapter 4) and an LUS (Jini Look Up Service, which will be introduced in the next section). An advanced Jini system may also include a transaction server, a Lease Renewal Service (LRS) and a JavaSpaces service (a central object space). These services will be introduced in Section 3.3.3.

- **A programming model** defines standard APIs for devising reliable distributed services.
- **Services** are entities of a Jini distributed system. They provide user-defined functionalities to other members in the system.

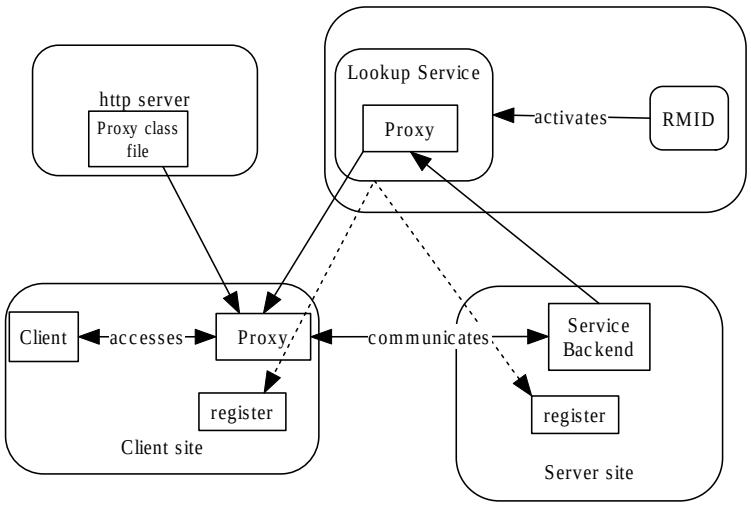


Figure 3.1: A basic Jini system

Figure 3.1 shows a basic Jini system consisting of an http server, an LUS, an `rmid`, one service and one client. The LUS and the `rmid` reside in the same machine because as an *activatable*[1] service the LUS needs an `rmid` to take care of its activation issues. In Figure 3.1, the service first locates the LUS by broadcasting a standard protocol request into the network. The LUS responds by producing a proxy (the register) to the service. The service then sends its own proxy to the LUS which stores the proxy in a registered service database. The service also needs to specify a place (the http server) to store the class file of its proxy.

The client locates the LUS in the same way as the service and uses a register object to send queries to the LUS looking for a service. The LUS will then look up in the service database and return the proxy of the found service. In Java, to send an object across the wire, the sender JVM needs to serialize the object (discussed in Chapter 5) and then send the resulting byte stream across the wire. The object is de-serialized from the byte stream at the receiving end. However, the de-serialization process (discussed in Chapter 5) needs both the class definition and the serialized

[1]Descriptions related to **activatable services** and **RMI Activation** will be given in Chapter 4

data (the flat byte stream). The LUS only sends the serialized data of the proxy, which cannot be de-serialized without its class file. From the http server specified by the service, the client can download the class file from the http server in the network and use it for the de-serialization process.

3.3.2 The key concepts of Jini

Services

The service is the most important concept in the Jini architecture [31]. A Jini system consists of a collection of different services. The Jini architecture provides standard APIs to build and manipulate services in a Jini system. A Jini service consists of two parts: the service implementation (the backend) and the proxy. The service backend contains the concrete implementation of the service behavior, while the proxy describes the service's interface and its communication with the backend. The network communication scheme is totally encapsulated in such a way that the interactions between the client and the service are greatly simplified. Because of the code mobility and reflection that Java provides, deploying a service will not involve all its potential clients having the proxy statically installed in their spaces. On the contrary, the proxy can be dynamically loaded into a client's space at run-time. As a result, a service implementation and its proxy can be updated incrementally without interfering with other entities on the network. A client can always load up-to-date proxies dynamically and use reflection to obtain new functionalities.

Jini is built on top of RMI, which is used for communications in all the Jini infrastructure components. However, third-party services can choose to use RMI or other applicable protocols for their communications.

Consider the printer example again. Suppose the name of the printer service is JiniPrinter. The first step to generate a Jini service is to determine its behavior. In this example we define three methods: `printText, printGraph` and `printPage`. These three methods are declared in `JiniPrinterInterface`, as depicted in Figure 3.2.

```
1: public interface JiniPrinterInterface {
2:       public void printText(String txt);
3:       public void printGraph(Graph graph);
4:       public void printPage(Page[] page);
5: }
```

Figure 3.2: The JiniPrinterInterface

In the design of the proxy and the backend, there are basically two approaches: RMI and customized-communication. In the RMI approach, the proxy and the communication details are generated automatically by the RMI system. Chapter 4 will describe this approach while introducing RMI. In this chapter, we choose the customized-communication approach, which needs the designer to specify the proxy and the backend.

In this example, a Java TCP socket is chosen to set up a communication channel between the proxy and the backend. We also select the wire format for trans-

ferring the method information. The proxy implementation is listed in Figure 3.3. Extending `JiniPrinterInterface` enables instances of the proxy class to accept requests from the printer's clients. `JiniPrinterProxy` also implements `Java.io.Serializable` so it can be transferred between different JVMs. The proxy class contains a port number (static variable PORT) that is set statically at design time (line 3 in Figure 3.3). This static number informs the proxy that requests can be sent through this given port. Furthermore, the argument to the proxy's constructor is a String representing the URL of the backend. Combining the static port number with the dynamic network address of the backend, an instance of `JiniPrinterProxy` can dynamically set up connections with the backend at runtime. Once a client downloads a proxy from a central registry (LUS), it can invoke the startup method, as shown in Figure 3.3, to create a communication channel between the proxy and the service backend. After that, the client can issue requests to the proxy as if it were accessing the printer service locally. An alternative is to make the startup method private to `JiniPrinterProxy` and use a lazy channel-creation scheme. No connection will be set up after the proxy is downloaded to the client's space until the first request is issued. If the proxy notices that the connection is null when a request comes from the client, it will set up the channel and then send the request across. Otherwise it would forward the request directly through the channel that has already been set up. However this approach incurs additional checks per request so it is not suitable for high-performance computing.

The core of `JiniPrinterProxy` is the marshalling of method information. Each method definition in this class takes care of passing the marshalled method information across the wire. The implementations of the three methods listed in Figure 3.2 share a similar structure and can be described in three phases:

1. Send the method name across the wire. As the sender and the receiver both know the method names, identifiers which can be mapped to distinct method names can be sent across instead of the real names to reduce the data to be transferred.

2. Send method arguments to the service backend so that the method can be correctly executed there. We use JDK-serialization (line 36 in Figure 3.3) to convert the arguments to a flat byte stream. Any other schemes that can properly convert and restore the arguments to and from byte streams can also be used.

3. Force the channel to send out all the buffered data and then clear all used internal data structures for the next use.

To correctly interact with the proxy, the backend also needs to be customized to deal with the specially-encoded method information. The code listed in Figures 3.4 and 3.5 shows the implementation of the JiniPrinter backend. `JiniPrinterBackend` also implements `JiniPrinterInterface` as `JiniPrinterProxy` does. Unlike `JiniPrinterProxy`, it provides concrete implementations for the methods in the interface. An instance of `JiniPrinter-Backend` spawns two threads. One thread waits for any incoming requests from the proxy (line 19 in Figure 3.4), on the same port as the one the proxy has. Once a request arrives, the other thread unmarshalls the received information (from line 61 in Figure 3.4). The method identifier will

```
1:  public class JiniPrinterProxy implements JiniPinterInterface, Serializable {
2:
3:     static public final int PORT = 2981;
4:
5:     protected String host,socket;
6:
7:     protected ObjectOutputStream writer;
8:     protected ObjectInputStream reader;
9:
10:    protected BufferedInputStream in;
11:    protected BufferedOutputStream out;
12:
13:    public HelloServiceProxy(String host) {
14:          this.host = host;
15:    }
16:
17:    public void startup(){
18:    try {
19:       socket = new Socket(host, PORT);
20:    } catch(Exception e) {
21:       System.out.println("Exception "+e.toString()+"\n");
22:    }
23:    try {
24:       out = new BufferedOutputStream (socket.getOutputStream());
25:       writer = new ObjectOutputStream(out);
26:    } catch(IOException e){
27:       System.out.println("IOException "+e.toString()+"\n");
28:    }
29:
30:    }
31:
32:    public void printText(String txt)
33:    {
34:         try {
35:              writer.writeInt(1);
36:              writer.writeObject(txt);
37:              writer.flush();
38:              writer.reset();
39:         } catch(IOException e) {
40:              System.out.println("IOException "+e.toString()+"\n");
41:         }
42:     }
43:
44:    public void printGraph(Graph graph)
45:    {
46:          //omitted
47:    }
48:
49:    public void printPage(Page[] page)
50:    {
51:          //omitted
52:    }
53:  }
```

Figure 3.3: Source code for the proxy implementation

be read first to determine the right method to be invoked, and proper routines are launched to de-serialize the method arguments. The concrete implementation of the appropriate target method is executed once all the arguments are ready.

The Jini Look Up Service (LUS) is a central bootstrapping mechanism of a Jini system. It acts as a main entry point for services and clients to access the whole system. A service needs to locate an LUS on a network in order to join a Jini system. The LUS can be located by using one of two discovery protocols, depending on whether the service knows the LUS's URL ahead of time or not [33]. If the service knows the location of the LUS ahead of time, then it can directly send a request to this location to setup a connection. If the service does not know the URL, then it has to broadcast a request on the network. Any LUS that hears the

```
0:  import java.net.*;
1:  import java.io.*;
2:
3:  public class JiniPrinterBackend extends Thread
4:      implements HelloService {
5:
6:      protected ServerSocket listenSocket;
7:
8:      public JiniPrinterBackend() {
9:          try {
10:             listenSocket = new ServerSocket(JiniPrinterProxy.PORT);
11:         } catch(IOException e) {
12:             e.printStackTrace();
13:         }
14:     }
16:
17:     public void run() {
18:         try {
19:             while(true) {
20:                 Socket clientSocket = listenSocket.accept();
21:                 new Connection(clientSocket).start();
22:             }
23:         }    catch(Exception e) {
24:            e.printStackTrace();
25:         }
26:     }
27:
28:     private int count=0;
29:
30:     public void printText(String txt) {
31:          //print string
32:     }
33:
34:     public void printGraph(Graph graph){
35:         //print graph
36:     }
37:
37:     public void printPage(Page[] page) {
38:         //print pages
39:     }
40:     class Connection extends Thread {
41:
42:        protected Socket client;
43:        ObjectInputStream reader;
44:        ObjectOutputStream writer;
45:
46:        BufferedInputStream in;
47:        BufferedOutputStream out;
48:
49:        String tempStr = null;
50:        Graph tempGraph = null;
51:        Page[] tempPage = null;
52:
53:        public Connection(Socket clientSocket) {
54:
55:            client = clientSocket;
56:
57:            try {
58:                clientSocket.setTcpNoDelay(true);
59:            } catch(Exception e) {
60:                System.out.println("Exception "+e.toString()+"\n");
61:            }
62:
63:            try {
64:                in = new BufferedInputStream (clientSocket.getInputStream());
65:                out = new BufferedOutputStream (clientSocket.getOutputStream());
66:                reader = new ObjectInputStream(in);
67:                writer = new ObjectOutputStream(out);
68:            } catch(IOException e) {
69:                e.printStackTrace();
70:            }
71:        }
```

Figure 3.4: Source code for the service backend implementation

```
72:   public void run() {
73:      int methodType = -1;
74:      int methodArg;
75:      int methodReturn = -1;
76:
77:      while(true)
78:        {
79:           try{
80:             methodType = reader.readInt();
81:             switch (methodType) {
82:             case 1:
83:                {
84:                     try{
85:                         tempStr=(String)reader.readObject();
86:                     } catch (OptionalDataException Oe){
87:                     }
88:                     catch (ClassNotFoundException Ce) {
89:                     }
90:                     catch (IOException Ie){
91:                     }
92                      HelloServiceImpl.this.printHello(tempInt);
93:             }
94:             case 2:
95:                {
96:                       //omitted
97:                }
98:             case 3:
99:                {
100:                      //omitted
101:               }
102:            }
103:          } catch(IOException e) {
104:            //omitted
105:          }
106:      }
107:  }
108: }
109:}
```

Figure 3.5: Source code for the service backend implementation (continued)

request will answer the service. Figure 3.6 depicts the unicast approach. First, the service constructs a locator (line 11 in Figure 3.6) with an argument representing the URL of the LUS. Next, using the URL, the locator can set up a connection between the service and the LUS and fetch the LUS's proxy (called register in this example) for the client. Then, using the register, the service exports its own proxy which contains the interface and communication implementations into the database

of the LUS. After that, the service is available to others on the network.

```
1:      System.setSecurityManager (new RMISecurityManager());
2:      LookupLocator locator = null;
3:      ServiceRegistrar registrar=null;
4:      HelloServiceProxy proxy=null;
5:      String host = null;
6:      try {
7:           new HelloServiceImpl().start();
8:      } catch(Exception e) {
9:      }
10:     try {
11:          locator = new LookupLocator("jini://192.168.23.7");
12:     } catch(java.net.MalformedURLException e){
13:     }
14:     try {
15:          registrar = locator.getRegistrar();
16:     } catch (java.io.IOException e) {
17:     } catch (java.lang.ClassNotFoundException e) {
18:     }
19:     proxy = new HelloServiceProxy(host);
20:     Entry[] entries = new Entry[]{new ServiceType("Proxy")};
21:     ServiceItem item = new ServiceItem(serviceID,proxy,entries);
22:     ServiceRegistration reg=null;
23:     try{
24:          reg = registrar.register(item,Lease.FOREVER);
25:     }catch(java.rmi.RemoteException e) {
26:     }
```

Figure 3.6: The LUS registration code

Clients can locate an LUS in the same way and then search through the LUS's database for required services based on service types, service names and many other attributes [33]. If any service is found, its proxy will be returned to the client to handle the client's requests. After having the LUS's proxy, the client can then send queries through the proxy to ask for the required service. Line 12 in Figure 3.7 constructs a service template to act as a filter in a query to retrieve desired servers. The results will contain the proxies of services that satisfy the client's requirements. After that, the client can use the returned proxies to interact with the services.

The use of the LUS avoids the unfortunate situation in RMI where each client has to know the exact network location of the desired services before they can search for them. The LUS makes a distributed system more flexible and dynamic. The distribution of services in a Jini system does not depend on the real network topology. A service can flow in the network as long as it keeps the information registered on the lookup service up-to-date.

A Jini distributed system can scale up by utilizing several LUSs managed into a hierarchy. Based on this scheme, an LUS can act as a bridge between several different systems. Another advantage of using several cooperating LUSs is that they can avoid the bottleneck caused by single-point access.

```
1:      try {
2:          locator = new LookupLocator("jini://192.168.23.7");
3:      } catch(java.net.MalformedURLException e){
4:      }
5:      try{
6:         registrar = locator.getRegistrar();
7:      } catch (java.io.IOException e) {
8:      } catch (java.lang.ClassNotFoundException e) {
9:      }
10:     Class[] classes = new Class[] {HelloService.class};
11:     Entry[] entries_Proxy = new Entry[] {new ServiceType("Proxy")};
12:     ServiceTemplate template_Proxy=new
                ServiceTemplate(null, classes, entries_Proxy);
13:     try {
14:         hello_Proxy=(HelloService)registrar.lookup(template_Proxy);
15:     } catch(java.rmi.RemoteException e) {
16:
17:     }
```

Figure 3.7: The LUS lookup code

The Leasing Model

Many services provided in a Jini system are granted based on some agreement made between the grantor and the grantee. This agreement is called a lease [31] that defines how long this grant will last. By using the Jini distributed leasing model, a distributed system can allocate resources in a well-managed way. A client has to negotiate a lease with the service, before any resource can be granted to it. The resource will be released eventually unless the client renews the lease before it expires. Once the lease expires, the service will stop serving the client and it will release any resources that have been allocated for it. Each service registration on the Jini LUS is also based on a lease. A service has to keep renewing its registration lease if it wants to be available. Once it stops renewing the lease, either because this service is withdrawn or because it breaks down, the lease will eventually expire and the LUS will remove all information about this service from its database. After that, no one else will find this outdated service any more. Based on this mechanism, the distributed system can deal with partial failures (such as network outages and machine break downs) efficiently. No outdated information will be accumulated, and the system can be reconstructed dynamically based on the real-time conditions. Thus, catastrophes are mitigated when some part of the system fails to work.

Jini supports the RMI **Activation** system (introduced in Chapter 4). The Activation mechanism offers a Jini service the ability to be activated upon request. The service will be swapped out of memory if no one asks for it. This way of managing Jini services uses system resources efficiently. However, this approach has a potential problem. If a Jini *activatable* service is not active when its registration lease expiry event arrives, it is unable to react because it is out of the runtime environment. A Jini service, called the Lease Renewal Service (LRS), can solve this problem. All *activatable* services can register with an LRS at the same time as when they register for an LUS. From then on, the LRS keeps running and interacts with

the LUS to renew the leases of the *activatable* services.

Transactions

When it is important to guarantee the order of a series of operations, or when data consistency must be maintained between participants, this series of operations can be combined into a **transaction** [31]. A transaction will guarantee that the operations will all succeed or all fail. Transactions are very important in database systems and other mission-critical systems. In Jini systems, transactions can be used as another mechanism to deal with partial failures. The Jini architecture provides a distributed transaction manager that can be used as a central monitor for two-phase commits.

Distributed event model

The Jini architecture supports a distributed event model, which supports one service to receive notifications of state changes from other services on the network. This model enables the construction of distributed event-based programs. A service that cares about the status change of another service can register for event notifications from that service. Each event registration is also based on a lease.

The JavaSpaces technology

A JavaSpaces service is an object-oriented implementation of the tuple-space that first appeared in the Linda system [8]. It is a central object space that provides simple and synchronized access to the entries stored in it. A JavaSpaces service is implemented by Sun as a common Jini service for collaborative distributed applications; shared data can be stored in it so that concurrent accesses can be guarded to achieve mutual exclusion and coordination among distributed processes.

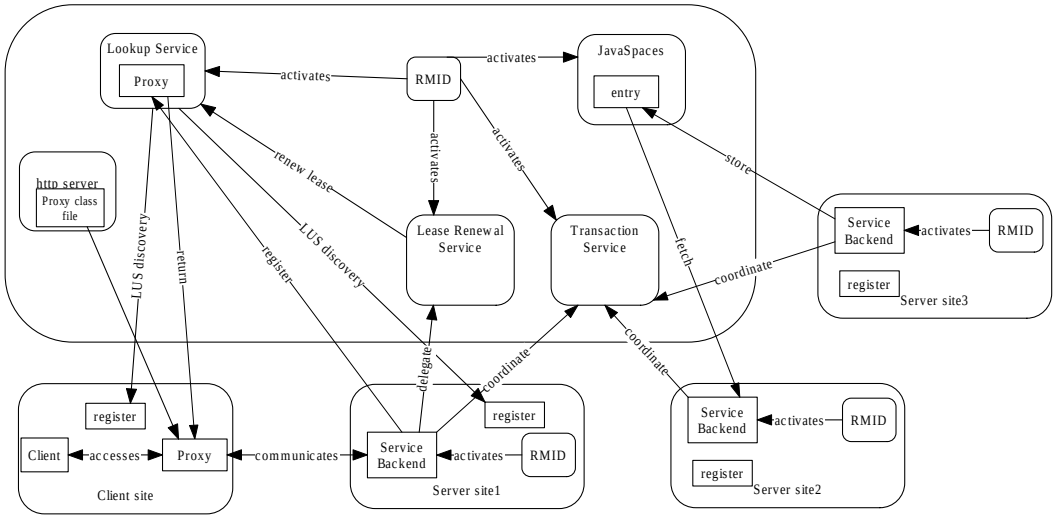


Figure 3.8: An advanced Jini system

3.3.3 An advanced Jini system architecture

Figure 3.8 shows a more complex Jini system than the one depicted in Figure 3.1. In Figure 3.8, three additional common Jini services (an LRS, a Transaction service, and a JavaSpaces service) are added to provide comprehensive support for user-defined services. If the number of services is not large, all common services can run on one machine, as depicted in the figure. Because the LUS, LRS, Transaction service and JavaSpaces service are activatable services, an `rmid` has to run to take care of them. If the system scales up so that putting all common services in one place will degrade the overall performance, we can assign those services to different machines, each of which must also run an `rmid`.

Services can use a JavaSpaces service as a central object space to share information. Operations owned by a collection of services can be wrapped into a transaction in the transaction server to guarantee that all operations will succeed or all will fail.

3.3.4 Launching and shutting down the Jini system

```
1:     java -jar $JINI_HOME/lib/tools.jar -port 8080 -dir $JINI_HOME/lib/&
2:         /*start up the http server*/
3:     rmid -log $HOME/jini_startup/log                          \
4:         -J-Djava.security.policy=none                         \
5:         -J-Dsun.rmi.activation.execPolicy=none                \
6:         -J-Djava.rmi.server.logCalls=false                    \
7:         -J-Dsun.rmi.server.exceptionTrace=true&
8:         /*start up the RMI daemon*/
9:     sleep 2s
10:       /*wait for two seconds*/
11:   java -jar -Djava.security.policy=$JINI_HOME/policy/policy.all      \
12:       -Djava.rmi.server.codebase=http://myhost:8080/                 \
                          $JINI_HOME/lib/reggie.jar                     \
13:       http://myhost:8080/reggie-dl.jar $JINI_HOME/policy/policy.all  \
15:       $HOME/jini_startup/reggie_log public&
16:       /*start up the Jini Look Up Service*/
18:   java  -jar -Djava.security.policy=$JINI_HOME/policy/policy.all     \
19:       -Djava.rmi.server.codebase=http://myhost:8080/outrigger-dl.jar \
20:       -Dcom.sun.jini.outrigger.spaceName=JavaSpaces                 \
21:       $JINI_HOME/lib/transient-outrigger.jar public&                \
22:       /*start up the Transaction Server*/
23:   java  -jar -cp $JINI_HOME/lib/reggie.jar                          \
24:       -Djava.security.policy=$JINI_HOME/policy/policy.all           \
25:       $JINI_HOME/lib/norm.jar http://myhost:8080/norm-dl.jar        \
26:       $JINI_HOME/policy/policy.all $HOME/jini_startup/norm_log
27:       /*start up the JavaSpaces*/
28:
29:
30:   java -Xbootclasspath/p:$HOME/io -jar $JINI_HOME/lib/tools.jar -stop
31:   rmid -J-Xbootclasspath/p:$HOME/io -stop
```

Figure 3.9: Jini launching and shutting down scripts

Figure 3.9 illustrates how to use scripts to launch and shut down all Jini infrastructure components. Lines 1 - 27 start up the components, including an http server, an `rmid`, an LUS, a JavaSpaces service and an LRS. Lines 30 - 31 shut down the

http server and the `rmid`. As the LUS, JavaSpaces service and LRS are activatable services, destroying the `rmid` will result in stopping the execution of all of them.

3.4 Conclusion

Jini technology provides standard APIs to build Java distributed systems. Furthermore, it provides system support to make these distributed systems flexible, reliable and scalable. It changes a conventional computing system to a network of services. In this dissertation, Jini is used to construct the infrastructure of the distributed CO_2P_3S environment. It greatly mitigates the complexity of implementing such a system.

Chapter 4

Remote Method Invocation

4.1 Introduction to RMI

In distributed computing, it is necessary to support inter-process communication for synchronization and coordination purposes. At first, programmers used sockets which were originally UNIX APIs for TCP/UDP communication. Java supports TCP and UDP sockets for low-level communication in Internet-centric client/server applications. TCP sockets provide reliable, two-way communication; UDP sockets offer broadcasting communication in a more efficient and faster way, without providing the data delivery guarantees that TCP sockets provide. In sockets programming, the programmer is fully responsible for network connection management, wire protocol definition and additional checks for reliable transportation.

RPC (Remote Procedure Call) is an alternative to sockets. It is a mechanism that defines a distributed calling model for distributed applications. The traditional way for a program to interact with remote nodes is by explicit message passing. With RPC, a client can access a remote server in much the same way that local method calls are made, except for some data restrictions. In RPC, network complexities are hidden from the programmer. Explicit message passing is replaced by procedure calls to a local stub of the remote server. The underlying RPC runtime, together with the stub, transfers the procedure information to the server to be processed and receives any results. This kind of programming model simplifies distributed computing. The user can focus on devising the application algorithms instead of spending time on communication designs. However, the distributed computing model supported in RPC is not object-oriented. Since JDK 1.1, Java has provided RMI (Remote Method Invocation), a distributed communication mechanism, to support the extended semantics of RPC in the object-oriented world. The key difference between RMI and RPC is that RMI supports object migration, polymorphism and automatic distributed garbage collection [29].

4.2 The RMI programming model

RMI is implemented on top of Java TCP sockets and provides the semantics of RPC with additional support for object orientation. It seamlessly incorporates the distributed object model into the Java language to simplify object-oriented designs in a distributed-memory environment.

Basically, a `Remote` object in the RMI model is an object that provides services to clients on other virtual machines. A key principle in the RMI programming model is that the behavior and the implementation of a remote service are separated. Designing an RMI server involves two steps:

- Determining the server's behavior. The behavior is implemented as an interface, including a set of public methods. Only the public methods defined in the interface are accessible to the clients.
- Providing concrete implementations to those methods in a class.

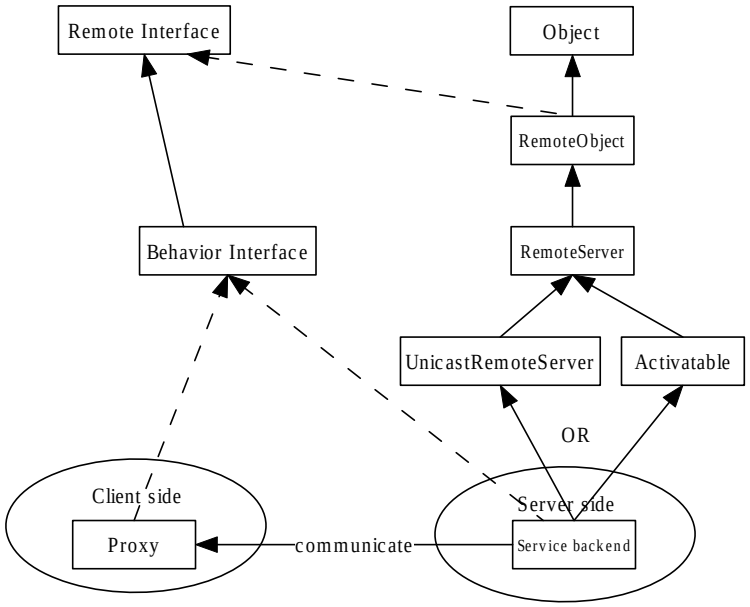


Figure 4.1: The RMI programming model

After these two steps, the interface and the implementation class are provided to the RMI system to create the remote server. Each remote server consists of two major components, one proxy and one backend. In reality, the backend consists of one RMI generated skeleton class and the user class. In the RMI model, the proxy is generated automatically from the interface provided by the designer. Both of these two components implement the same interface (depicted in Figure 4.1) that defines the server's behavior. From the client's point of view, the client accesses the proxy in the same way that it accesses the remote server. Other than providing concrete implementations of the remote behavior as the backend does, the proxy works as a local deputy of the remote RMI server and is much like the local stub in RPC. The network complexities are totally encapsulated in the proxy and the RMI system. This proxy/backend model is the same as the one that Jini supports. However, Jini supports proxies that can be devised by designers as well as be automatically generated in the RMI system.

Using RMI, programmers can design distributed programs in much the same style in which they develop sequential programs in a single-processor machine. A programmer needs to obey the inheritance hierarchy (as described in Section 4.2.1)

and the behavior/implementation scheme to create a remote service. After that, the client program has to follow proper procedures to load the service's proxy into its own memory space and to invoke methods on the proxy to access the remote service.

4.2.1 The programming API

To have remote behaviors, an object has to directly or indirectly implement the `Remote` interface. Consider the printer example described in Chapter 3. In this chapter we choose to implement the printer service as an RMI server. In the RMI implementation, the `JiniPrinter` class (Figure 4.2) implements an interface called `RemoteJiniPrinterInterface`, which extends `Remote` and `JiniPrinterInterface`. The latter defines the remote object's behavior while the former helps the RMI system identify the methods in the behavior interface as remotely accessible.

After the behavior has been specified in the interface, we have to provide concrete implementations for each method in the `JiniPrinter` class. At line 21 in Figure 4.2, the concrete class `JiniPrinter` is defined.

After creating the remote interface and defining the concrete remote class, we can use `javac` (Java compiler) and `rmic` (RMI compiler) to obtain the stub (the proxy) of the JiniPrinter.

As depicted in line 21 of Figure 4.2, `JiniPrinter` extends interface `Java.-rmi.UnicastRemoteObject`, which is one of the two classes that the RMI system provides to facilitate the creation of remote servers. Each remote service should extend `java.rmi.server.UnicastRemoteObject` or `java.rmi.activation.Activatable`. These two classes define the `hashCode(), equals()` and `toString()` functions for remote objects. If a class extends none of the above, it has to provide the correct semantics for these three methods itself. Moreover, these two classes support two different reference semantics for remote servers:

1. `UnicastRemoteObject` [29] supports point-to-point active object references. An instance of such a class does not have multiple identical duplicates on different locations. The communication between each client and the server is one-to-one. Moreover, instances of this class have to be running all the time after being created. However, having multiple active servers running all the time on one machine may take up significant system resources unnecessarily.

2. `Activatable` [29] enables servers to be activated on demand. This approach is especially useful for large-scale distributed systems. Such systems may be on different networks and share computing resources with many other applications. Using the Activation system (introduced later) will reduce unnecessary resource use.

The RMI programming model also extends the Java exception model to handle distributed exceptions during remote method invocations. The distributed exception model includes a series of new exception classes, which are all subclasses of `java.rmi.RemoteException`. As specified in the interface definition that starts from line 1 in Figure 4.2, each remote method declared in the interface must have `java.rmi.RemoteException` in its throw clause.

```
1:    import java.rmi.Remote;
2:    import java.rmi.RemoteException;
3:    public interface JiniPrinterInterface {
4:          public void printText(String text) throws RemoteException;
5:          public void printGraph(Graph graph) throws RemoteException;
6:          public void printPages(Page[] page) throws RemoteException;
7:    }
8:
9:    import java.rmi.Remote;
10:   import java.rmi.RemoteException;
11:   public interface RemoteJiniPrinterInterface extends
12:                       JiniPrinterInterface, Remote
13:   {
14:   }
15:
16:
17:   import java.rmi.Remote;
18:   import java.rmi.RemoteException;
19:   import java.rmi.server.UnicastRemoteObject;
20:
21:   public class JiniPrinter extends UnicastRemoteObject
22:         implements RemoteJiniPrinterInterface{
23:
24:         public JiniPrinter() throws java.rmi.RemoteException
25:         {
26:         }
27:         public String printText(String str){
28:               //print a text;
29:         }
30:         public String printGraph(Graph graph){
31:              //print a graph;
32:         }
33:        public String printPages(Page[] page){
34:              //print pages;
35:         }
36:   }
```

Figure 4.2: An example of building an RMI server

4.2.2 Remote object stubs and skeletons

Each RMI server contains one stub and one skeleton (the latter has become optional since JDK1.2), both of which are generated automatically by the RMI system to encapsulate communication details. These two entities act as two end-points of a communication channel set up between a client and a server. The stub runs at the client side and forwards the client's requests to the skeleton running in the server space; the skeleton analyzes the requests from the stub and dispatches corresponding methods to the server's backend. The communication protocol used between the stub and skeleton is the Java Remote Method Protocol (JRMP). These two entities are generated by compiling the remote object class using `rmic`. Since JDK 1.2, skeletons of all local servers can be replaced by one `rmid`. This daemon communicates with the remote stubs of local RMI servers and dispatches requests to corresponding server backends.

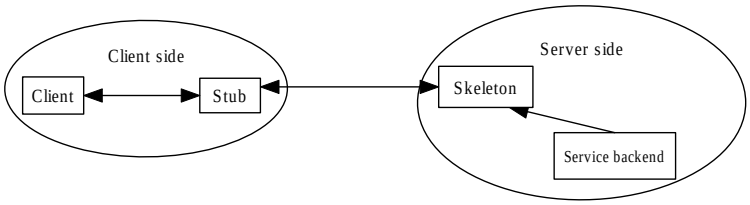


Figure 4.3: RMI stub and skeleton

4.2.3 RMI registry

The RMI system provides a simple naming service, called `rmiregistry`, to help a client locate required services. The rmiregistry is a bootstrapping service that takes care of the registrations and discoveries of the servers in the local machine. Servers that want to announce their existence on the network will register with the rmiregistry using some textual description such as the servers' names and capabilities. Clients make queries into the rmiregistry database based on the textual information for desired servers. The rmiregistry has two apparent weaknesses:

- the registration information is too simple to record some useful well-formatted attributes of the servers, and
- one `rmiregistry` manages servers only on a local machine; a client has to explicitly know the network address of the required server before it can locate a `rmiregistry` at the same location and look for the server.

The `rmiregistry` is not used in our work because of its over-simplification; we rely on the more powerful Jini LUS for the service lookup and registration.

4.2.4 RMI activation

Originally, all remote servers in an RMI system had to be running all the time after being launched, but keeping multiple servers running consumes a lot of system resources. Competitors of RMI, such as DCOM [26] and CORBA [19], can activate objects upon request and deactivate them once the request has been processed. This RMI weakness could seriously strain the performance of its applications. Since Java 2 SDK, the RMI activation framework has been introduced to meet the challenges of its competitors. The basic idea behind RMI Activation is as follows [29]. At first, an `Activatable` remote object will be up and running for a short while after it finishes its registration with the rmiregistry and then quit. All the incoming requests from remote clients to this service will be taken care of by an `rmid`. The local `rmid` peeks at a certain port for incoming requests to all the registered local `Activatable` services. If one request is for a service that is currently active, the `rmid` directly dispatches the request to the service. Otherwise, the `rmid` will activate the service and then dispatch the request to it. Remote objects no longer need to stay active waiting for requests and taking up resources unnecessarily. Instead, they can stay out of the memory and be activated upon request. When a request has been served, the server can deactivate itself to release the allocated resources.

The RMI Activation framework introduces two main concepts: the `Activatable` class and the `rmid` (the RMI Daemon). Each *activatable* remote object should

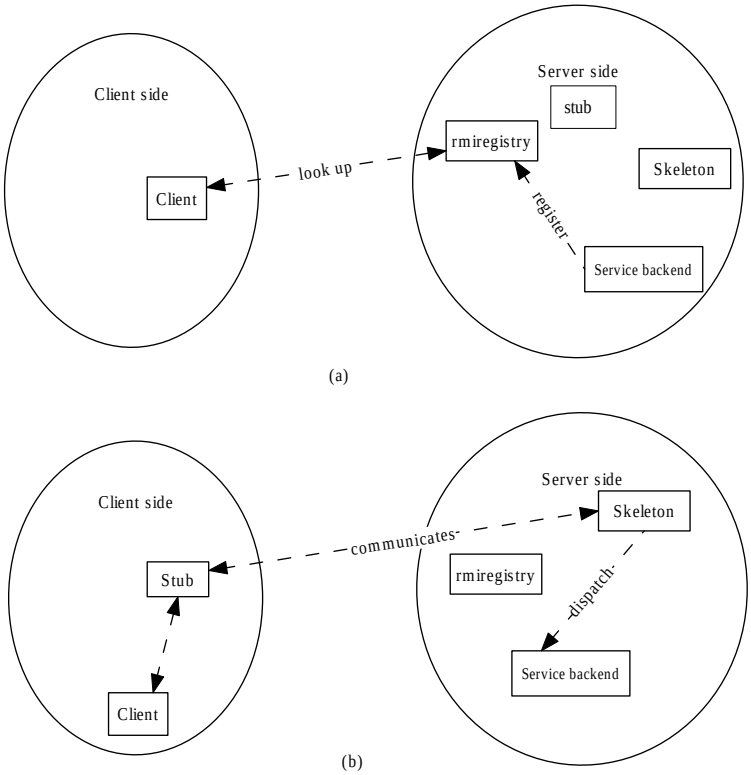


Figure 4.4: RMI registry

extend the `java.rmi.activation.Activatable` class. Moreover, one `rmid` is up and running all the time at each host for all local activatable servers. The `rmid` takes the place of both the rmiregistry and the skeleton.

4.3 RMI architecture

As depicted in Figure 4.5, the RMI system consists of three layers [29]:

- the stub/skeleton layer,
- the remote reference layer, and
- the transport layer.

These three layers work together to transfer the method invocation information between a client and a server. Each layer has its own distinct functionality and interacts with the layer below. At the client side, during one RMI call, the stub encodes the method information, together with all marshalled arguments (Sun calls the marshalling process serialization). This is passed to the lower layer, the remote reference layer, which directs the transport layer to forward the information to the destination. At the server side, the remote reference layer receives the information from the lower transport layer and passes it to the skeleton. The skeleton decodes the method information and unmarshalls all the method arguments before dispatching

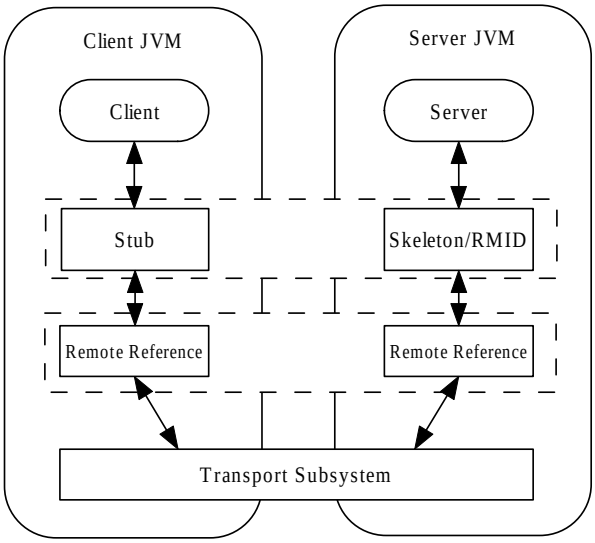


Figure 4.5: RMI architecture

the method to the local service backend. The following is a detailed description of these three layers:

- The stub/skeleton layer uses the JDK-serialization routine (Chapter 5) to marshall and unmarshall method arguments. Since JDK1.2, the skeletons of remote servers on the same machine can be replaced by one `rmid` which then takes care of incoming requests for all registered servers.

- The Remote Reference Layer (RRL) encapsulates different invocation semantics and different reference semantics. The invocation semantics help identify whether a server has only one instance or multiple duplicates at several locations, in order to determine whether a unicast or multicast communication scheme is needed. The reference semantics supports persistent references to both regular remote severs that will be active all the time and activatable remote servers that will only be activated upon request. All these details are transparent to the other layers. Currently, RMI supports only one set of invocation semantics: the server is a single object, not having multiple replicas at different sites.

- After the invocation semantics and reference semantics have been determined by the RRL, the Transportation Layer deals with the communication channel setup and management, remote object tracking and message dispatching to remote objects.

The layered architecture facilitates the separation of concerns related to the complexities of the RMI mechanism. Each layer has clearly defined interfaces and protocols in such a way that it can incorporate different implementations as long as the interfaces and protocols are kept intact. However, the current implementation of RMI is not open-ended; components cannot be easily plugged in and out. For instance, the JDK-Serialization routine, which is implemented in the `ObjectInputStream`

class, is utilized in RMI for argument marshalling. The current RMI implementation refers to it statically instead of binding an instance of `ObjectInputStream` to an instance field, whose static type is a superclass of `ObjectInputStream` (such as `ObjectInput`), at runtime. The transport subsystem of RMI has the same problem. As a result, RMI cannot incorporate optimized serialization and transport components without modifications. Chapter 7 will introduce CO_2P_3S-RMI and CO_2P_3S-serialization, both of which have more efficient implementations to address high-performance computing.

4.4 Conclusion

RMI is a predecessor of Jini. It is also an infrastructure technology to facilitate building Java-centric distributed systems. RMI provides a very easy programming model to create remote servers. However, it is not as powerful and flexible as Jini. In distributed CO_2P_3S, we mainly use RMI as a communication mechanism between Jini services because of its simplicity. We rely on Jini to make the system flexible.

Chapter 5

Serialization

5.1 The purpose of serialization

In procedural programming, most data layout (values, arrays, records) is simple and almost linear in memory so that the elements can be transferred and stored easily. However, in the object-oriented world, the memory layout of objects is much more complex, with many references from one object to another. In other words, linked data structures are much more common. Data and operations are encapsulated into an object to restrict the accesses to the data. Without language support, the programmer cannot easily save objects' states or send objects between different address spaces.

Fortunately, JDK-serialization provides a generic way of linearizing storage for Java objects [28]. Except for primitive data types, all other data are objects in Java. The main idea of JDK-serialization is to use Java's reflection capability to access the internal structure of an object so that all its internal state and class information can be stored into a linear byte stream. This stream of bytes can thus be easily stored to a disk or transferred over a network to be constructed into an identical object somewhere else. The conversion and reconstruction processes can be done automatically by JDK-serialization and these processes are transparent to the programmer. This mechanism greatly simplifies Java distributed computing, where objects often need to be passed across different JVMs. During the serialization and de-serialization processes, JDK-serialization also helps maintain Java security requirements and class version consistency. The purpose is to prohibit malicious revelation of an object's internal structure and to avoid the situation that an object is passed between JVMs which have different versions of class definitions. To use JDK-serialization, the programmer does not need to provide any class-specific methods (unless customization is desired).

JDK-serialization is a vital component of RMI. Methods can be invoked on remote objects with given objects as arguments. To transfer an object to a remote site and hide the complexity and communication details from the programmer, RMI adopts JDK-serialization.

5.2 The programming API

To be serializable, objects should either implement the `java.io.Serializable` or `java.io.Externalizable` interface; the latter is a sub-class of the former. The `java.io.Serializable` interface helps a JVM identify whether an object can be serialized or not. In JDK 1.0, all objects in Java were serializable by default. However, this scheme was not very secure because anyone could easily serialize any object into a byte stream to discover its internal data contents. Since JDK 1.2, an object has to implement one of these two interfaces (Figure 5.1) to be serializable. By this means, the programmer can decide on a class-by-class basis which classes should have exposed contents. Furthermore, using customized serialization, an object can even choose to expose only a part of its state.

The `java.io.Serializable` interface is an empty interface. The programmer is not required to supply any code. However, the designer can provide customized serialization routines by writing two methods: `writeObject` and `readObject`. The former allows an object to write some of its fields into the serialized data, while the latter defines how to parse the customized serialized data. If a class only implements the `Serializable` interface, the JDK-serialization routine will serialize all the fields of an instance of the class by default. However, JDK-serialization will not access any fields of an object that are not serializable. Subclasses of classes which are not serializable can be serialized if the subclasses still implement the `Serializable` interface and their non-serializable superclasses provide a non-argument constructor to let the superclasses' fields be initialized [28].

The `Externalizable` interface is a subclass of `Serializable`. It defines two methods, `readExternal` and `writeExternal`, both of which have to be implemented by its subclasses. The latter allows the class to be fully responsible for the format of the serialized form, while the former defines the reverse.

At first sight, it is hard to tell the difference between the functionality of these two interfaces, since both of them provide a mechanism for customized serialization. There is a slight difference between them: subclasses of the `Externalizable` interface use a more powerful customized scheme. The `writeExternal` method has full control of the layout of the serialized stream. When the serialization routine encounters an object with a `writeExternal` method, it will invoke this method and then return. However, the `writeObject` method only takes care of the contents of the current object defined in the class where it appears. After a `writeObject` method is executed, the serialization code scans the object's superclasses to write their instance fields. The `readExternal` and `readObject` methods have the same difference.

Besides implementing the `(read)writeExternal` or `(read)writeObject` methods, an object can use another form of customized serialization. The object can assign another object to replace itself to be written into the stream. For example, by defining a `writeReplace` method, a Car object can assign a CarAgent object to be serialized. If the `writeReplace` is implemented, another method called `readResolve` has to be implemented to notify the de-serialization routine how to deal with the input object. This will be discussed in the following section.

In addition to implementing the `java.io.Serializable` or `java.io.Externalizable` interface, each class has to provide an empty constructor so the Java

runtime can create an empty instance with initialized fields for the de-serialization process. The code fragments in Figure 5.2 show how an object implements one of these two interfaces and how to serialize an object in Java.

```
public Interface Serializable {
}

public Interface Externalizable {
    public void writeExternal();
    public void readExternal();
}
```

Figure 5.1: The serialization interfaces

5.3 The serialization process

Class `ObjectOutputStream` provides a series of methods to serialize contents into a stream. For primitive data types, it provides `writeByte, writeBytes, writeBoolean, writeShort, writeInt, writeFloat, writeDouble, writeLong`, `writeChar,` and `writeChars`. For object types, one method, `writeObject`, is provided. One object type—`String`—is treated specially: the programmer can use either `writeObject` or `writeString` to serialize a String object.

Likewise, class `ObjectInputStream` provides a set of methods to restore information from a stream. For primitive data types, it provides `readByte, readBytes, readBoolean, readShort, readInt, readFloat, readDouble, readLong, readChar,` and `readChars`. For object types, it provides one method–`readObject`. For `String` objects, the programmer can use either `readObject` or `readString`.

The main entry point into the Object Serialization is the `writeObject` method, the argument of which is the object to be serialized. Object serialization is a recursive process. To serialize one object, the Java runtime recursively writes the whole object graph, including the class information and the states of all the referenced objects into a stream. The following is a detailed description of the serialization process [28]:

1. If a subclass of class `ObjectOutputStream` is enabled to override this serialization process, let an instance of the subclass handle the serialization of this object.

2. If the object is null, a TC_NULL marker is written into the output stream and the process ends.

3. Check if the object has been replaced by another object previously. If the replacement is found, the process writes the handle[1] of the replacement to the stream and ends.

[1] An internal hash table is constructed during a serialization process to store distinct objects that have been serialized. Each time a new object is written into the stream, it is inserted into the hash

```
1:    public Class Serialization_Process {
2:
3:        public Serialization_Process() {
4:        }

5:        public static void main(String[] argv) {
6:
7:            FileOutputStream ostream = new FileOutputStream("t.tmp");
8:            ObjectOutputStream pos = new ObjectOutputStream(ostream);
9:
10:           pos.writeInt(12345);
11:           pos.writeObject("Today");
12:           pos.writeObject(new Date());
13:           pos.flush();
15:           ostream.close();
16:
17:           FileInputStream istream = new FileInputStream("t.tmp");
18:           ObjectInputStream pin = new ObjectInputStream(ostream);
19:
20:           int tempInt = pin.readInt();
21:           String tempString = (String)pin.readObject();
22:           String tempDate = (Date)pin.readObject();
23:           istream.close();
24:       }
25:   }
```

Figure 5.2: An example of using serialization

4. If the object has already been serialized into the stream before, the handle which points to the serialized data in the stream is written and the process ends.

5. If the object is a Class object, its associated `ObjectStreamClass` is written into the stream and the process ends.

6. If the object is an `ObjectStreamClass` object, information about the corresponding class to be serialized including the class name, the Serial Version Unique Identifier, and all the field information (name and type) is written and the process ends.

7. Check to see if the object has not been replaced by another object before. If not, the class of the object and/or a subclass of class `ObjectInputStream` (if exists) will be given a chance to designate another object to replace this object into the stream.

   - As described in Section 5.2, the programmer can define a `writeReplace` method for a class so that an instance of the class can assign a replacement object to be serialized into a stream.
   - The subclass of the `ObjectInputStream` also has the capability to replace an object with another one as long as it calls the `enableReplaceObject` method to enable itself to invoke the `replaceObject` method in `ObjectInputStream`.

   If a replacement is returned after the first step, the subclass of class `ObjectInputStream` will work on the delegate instead of the original object.

   If a replacement is returned after these two steps, the mapping from the original object to the replacement is recorded and the replacement is serialized into the stream.

8. The contents of the object is written into the stream:

   - If the object is a `java.lang.String`, it is written in the Universal Transfer Format (UTF) format.
   - If the object is an array, the `ObjectStreamClass` of the array is written first, which is followed by the length of the array. Then all the elements of the array are written to the stream. If the elements are primitives, they can be directly written into the stream by calling one of the write methods for primitive data types. Otherwise, if the elements are objects, they will be written one by one by recursively calling the `writeObject` method.
   - For a regular object, the `ObjectStreamClass` objects, including the class of the object and all super classes, except for `Object`, will be written. Then, all fields of this object will be written one by one. If the field

table and assigned a unique handle. If later the same object is serialized again, only the handle stored in the hash table will be written instead of the whole object. Based on this approach, if the same object is written twice in one serialization process, the resulting stream will only contain one serialized copy of it and a pointer which refers to the serialized copy.

has a primitive type, the data and type information are written directly by the write methods for primitive data types. If the field has a non-primitive type, the `writeObject` method is invoked recursively to write this field. By this means, the whole object graph will be fully recorded in the stream.

5.4 The de-serialization process

Object de-serialization is the reverse process of serialization. The programmer can call the `readObject` method of an `ObjectInputStream` instance to restore the objects from the stream in the same sequence as they were serialized. The `readObject` method returns an object referred to by an `Object`-type reference, which needs to be cast to its original type.

Object de-serialization is also a recursive process. To de-serialize one object from a byte stream, the routine recursively restores the whole object graph described in the stream. The following is a detailed description of the de-serialization process [28]:

1. If a subclass is enabled to override the `ObjectInputStream`, let the subclass handle the de-serialization of this object.
2. If a TC_NULL marker is read, a null object is returned.
3. If a handle is read, the object pointed to by this handle is returned.
4. Check if the object can been replaced by another object. If the replacement is found, it is returned and the process ends. Corresponding to the serialization process, the class of the object can define a `readResolve` method and the subclass of class `ObjectInputStream` can use a `resloveObject` method to find a proper replacement.
5. If the object is a Class object, its `ObjectStreamClass` is read from the stream and it is inserted into a hash table and assigned a handle.[2]
6. If the object is an `ObjectStreamClass` object, the information of the corresponding class including the class name, the Serial Version Unique Identifier, and all the fields information (name and type) are read to construct an `ObjectStreamClass` object. The `resolveClass` method is invoked to load the class for this descriptor; a subclass of the `ObjectInputStream` is given a chance to load the class specially, e.g. from a remote place. If the class cannot be loaded correctly, a class not found exception will be thrown.
7. The contents of the object are read from the stream:
   - If the object is a string, the UTF format data is read.

[2] In the de-serialization process, a hash table, which is identical to the one that is used in the serialization process is constructed according to the serialized stream. This hash table helps decoding the handles in the stream. Each time an object is read from the stream, it is inserted into a hash table and assigned a unique handle. Later, if a handle is read from the stream, the routine will refer to the hash table by the handle to return an object.

- If the object in the stream is an array, its objectStreamClass and length are read first. A new empty array is allocated and all elements are read using the corresponding read method based on their types and are assigned to the array. If the elements are objects, they will be read one by one by recursively calling the `readObject` method.
- For a regular object, an empty instance of the class is created by invoking the non-argument constructor for the first non-serializable super class. The fields are restored by the following:
  (a) If the object implements the `Serializable` interface, the fields are restored by calling `readObject` methods defined in the class and its super classes. Otherwise, `defaultReadObject` methods are invoked to read in all the serialized fields if `readObject` methods are not defined. During this process, the version of the class specified in the stream is compared with the local version. If they are the same, the field restoration can continue safely. If they are different, fields have to be restored carefully.
  (b) For `Externalizable` objects, the `readExternal` method is called to restore the contents of the object.

5.5 The layout of the serialized data

In this section we will go through an example to provide a clearer idea of the serialization process. Figure 5.3 is the serialized format of an `Integer` object, whose `int` value is 20. The left column shows the hexadecimal numbers while the right column shows the corresponding ascii characters.

The beginning four bytes—"ac ed 00 05"—is simply the stream header marking the start of this stream. After the stream header is the serialized form of the `Integer` object, whose leading byte is 0x73 representing a stream marker TC_OBJECT (listed in Figure 5.3). TC_OBJECT denotes that the following contents form an object. After the TC_OBJECT marker is a TC_CLASSDEC (0x72) marker representing that the following contents is the `ObjectStreamClass` instance of class `Integer`, including the class name, a stream unique identifier (UID) defining the class version, and all field names and types. The 19 bytes following the TC_CLASSDESC marker is the UTF coding of the `Integer` class name. After the class name is an 8-byte UID, then is the SC_SERIALIZABLE (0x02) marker denoting that the `Integer` class is serializable. Then the following 2 bytes denote that there is one instance field, whose type is `int` (0x49); the following 7 bytes are the UTF coding of the field's name, which is "value". After the field name is the TC_ENDBLOCKDATA marker denoting the end of the `ObjectStreamClass` information.

Because the `Integer` class is a subclass of class `Number`, `Number`'s class information should also be written. The objectStreamClass of the `Number` class is written to the stream following the `Integer`'s objectStreamClass in the same format.

After the class description of the `Integer` class is written, the serialization process continues to write all the fields of this `Integer` object. The `Integer` class has one `int` field, so at the end of the stream four bytes - '00 00 00 14' representing the `int` value 20 are written.

```
ac ed 00 05 73 72 00 11 6a 61 76 61 2e 6c 61 6e 67 2e ....sr..java.lang.
49 6e 74 65 67 65 72 12 e2 a0 a4 f7 81 87 38 02 00 01 Integer.......8...
49 00 05 76 61 6c 75 65 78 72 00 10 6a 61 76 61 2e 6c I..valuexr..java.l
61 6e 67 2e 4e 75 6d 62 65 72 86 ac 95 1d 0b 94 e0 8b ang.Number........
02 00 00 78 70 00 00 00 14                          ...xp....


final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
final static byte TC_NULL = (byte)0x70;
final static byte TC_REFERENCE = (byte)0x71;
final static byte TC_CLASSDESC = (byte)0x72;
final static byte TC_OBJECT = (byte)0x73;
final static byte TC_STRING = (byte)0x74;
final static byte TC_ARRAY = (byte)0x75;
final static byte TC_CLASS = (byte)0x76;
final static byte TC_BLOCKDATA = (byte)0x77;
final static byte TC_ENDBLOCKDATA = (byte)0x78;
final static byte TC_RESET = (byte)0x79;
final static byte TC_BLOCKDATALONG = (byte)0x7A;
final static byte TC_EXCEPTION = (byte)0x7B;
final static byte TC_LONGSTRING = (byte) 0x7C;
final static byte TC_PROXYCLASSDESC = (byte) 0x7D;
final static int baseWireHandle = 0x7E0000;
```

Figure 5.3: The serialized data of an instance of class Integer

5.6 Discussion

JDK-serialization provides a linear storage technique for objects that supports a convenient object transferring mechanism for Java distributed computing. It also offers strong security and class version consistency control for transferring Objects between different JVMs.

However, the complexity and high overhead reduce its suitability for high performance distributed and parallel computing. In the HPC[3] area, all participants know exactly what classes are needed and trust the data sent between them. The computation will last only a limited period of time, during which versions of classes are not likely to change. In this case, the security and class version consistency control become an obstacle for gaining high performance. In RMI, one third of the time of a normal RMI call is spent in serialization and de-serialization, which is quite significant.

As can be seen from Figure 5.3, the serialized data contains much redundant information. In the detailed class information, except the class's full-qualified name, all other information such as the serial unique identifier and all field types and names are unnecessary because different places will have the same version of the class and it won't change in one run. In the optimal situation, each object only needs its unique class name and its internal data contents to be written to a stream. There are many other kinds of redundancies that occur in the serialization process and they should be reduced as much as possible to support high performance computing.

In Chapter 7, an enhanced implementation of the Java Object Serialization will be introduced. It is used by our modified RMI. However, the programmer can program with RMI and use our new serialization mechanism without being aware of the modifications.

[3]Stands for High Performance Computing.

Chapter 6

The Architecture and Infrastructure of Distributed CO_2P_3S

6.1 Introduction

The existing version of CO_2P_3S supports only shared-memory parallel programming. Using high-performance machines and commercial network connections, distributed-memory systems are cheaper and easier to build and more readily accessible than shared-memory machines. They can provide better scalability for some applications.

This dissertation describes how CO_2P_3S was extended to support programming for a distributed system (DCO_2P_3S) on a network of workstations. The original design goal was still maintained. This goal is to abstract parallel complexity into parallel design patterns for the easy design of correct parallel programs with reasonable performance. I built a runtime environment to support the execution of distributed programs. The programmer can use distributed versions of pattern templates to generate code that runs in this environment.

In DCO_2P_3S, almost all the details about distributed parallel computing are hidden from the user. Unlike multithreaded programming, which is fully supported in Java, distributed multiprocess programming has little support in Java, except for RMI and Sockets. Thus, to make DCO_2P_3S fully functional, we need to provide the following facilities:

- an infrastructure to construct the whole distributed system,
- a communication subsystem,
- a synchronization mechanism,
- real-time performance monitoring and process management tools,
- and new versions of design patterns that generate distributed-memory code.

The rest of this chapter will give detailed descriptions of the first four components; the fifth will be discussed in Chapter 8.

6.2 The DCO_2P_3S architecture

The first step in constructing DCO_2P_3S is to design a proper structure for the system. As the existing CO_2P_3S system is implemented in Java, I can use two Java-centric infrastructure technologies—RMI and Jini—to design the system architecture.

DCO_2P_3S could be built on top of RMI in conjunction with additional central control services devised by us. However, based on the differences between RMI and Jini given in Chapter 3 and 4, RMI itself is not powerful enough to build a flexible distributed system.

I decided to use Jini to construct DCO_2P_3S. In addition to a standard API, Jini provides extensive support for Java-based distributed computing. Jini can be used to build a distributed system with a scalable and dynamically-configurable architecture. It also provides efficient process coordination mechanisms, a central object space (a JavaSpaces service) for global synchronization and data sharing, and a customizable communication scheme. All of these facilities offered by Jini greatly alleviate the complexities of constructing DCO_2P_3S. More efforts could be focused on pattern designs and performance enhancements.

DCO_2P_3S is implemented as a Jini system. All system control tasks such as process spawning and killing, synchronization and communication, and real-time performance monitoring are implemented using Jini technology. Applications generated using DCO_2P_3S contain multiple processes, each of which is a Jini service running on a distributed machine. All Jini services locate each other through the Jini lookup service (LUS) and coordinate with each other directly or through the Jini Transaction Server and the JavaSpaces service.

Figure 6.1 illustrates the overall structure of the DCO_2P_3S environment. The central control machine provides a graphical user interface for the programmer to access the environment. The GUI extends the one in existing CO_2P_3S system with control over the whole distributed runtime environment. The rectangles with thick borders represent participating machines in the environment. Each machine contains a set of Jini services that interact with others at remote sites. Through the user interface, the programmer can easily configure and launch the whole environment.

The DCO_2P_3S Tools menu in Figure 6.2 shows three options—Distributed Environment, Jini Start Up and Jini Shut Down—that control the DCO_2P_3S environment. The second option launches all Jini infrastructure components on one or more machines. The third option destroys all Jini components in order to shut down the environment. The first option, launches a configuration window, as shown in Figure 6.3. The left list box in the window includes all machines that comprise the system. By clicking the Add, Delete and Delete All buttons, the user can add or remove the machines in the system. The Launch button triggers all listed machines to launch a series of daemon processes, which will be manipulated by the central control to provide execution support for distributed programs. The right list box in the window allows the user to choose one or more machines from the left one as participants for a specific application. This approach gives the user an option to apply static load balancing to the application. By monitoring the real-time performance of all machines in the environment (described later in Section 6.5), the user can choose the most idle ones to use.

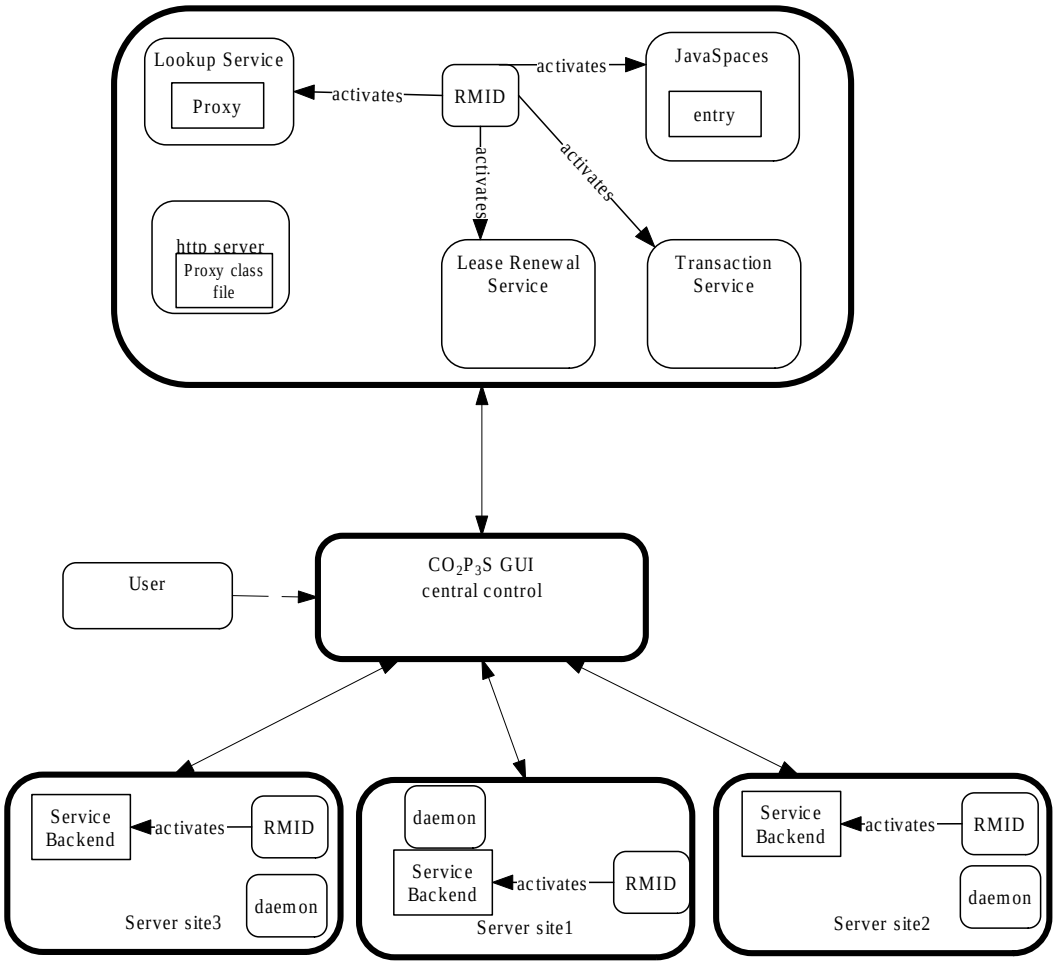


Figure 6.1: The overall structure of DCO$_2$P$_3$S

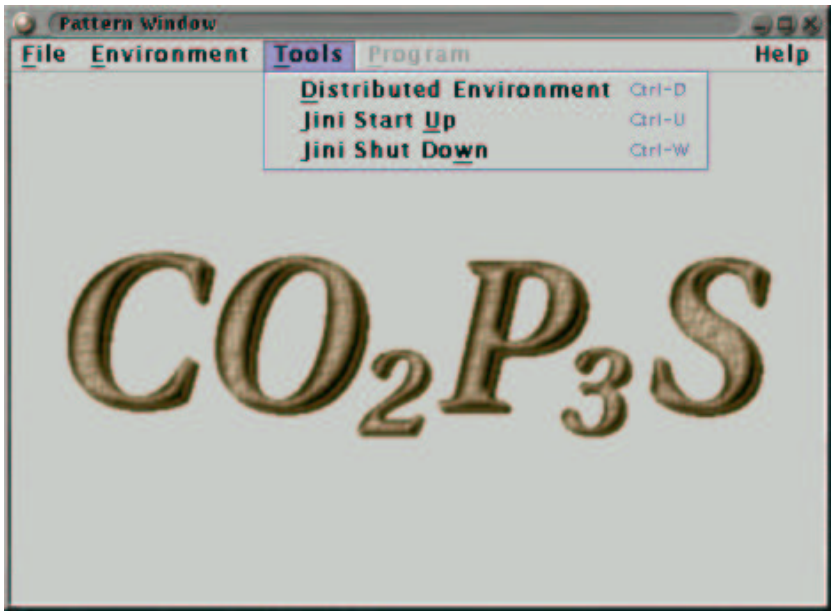


Figure 6.2: Menu options to start up the environment

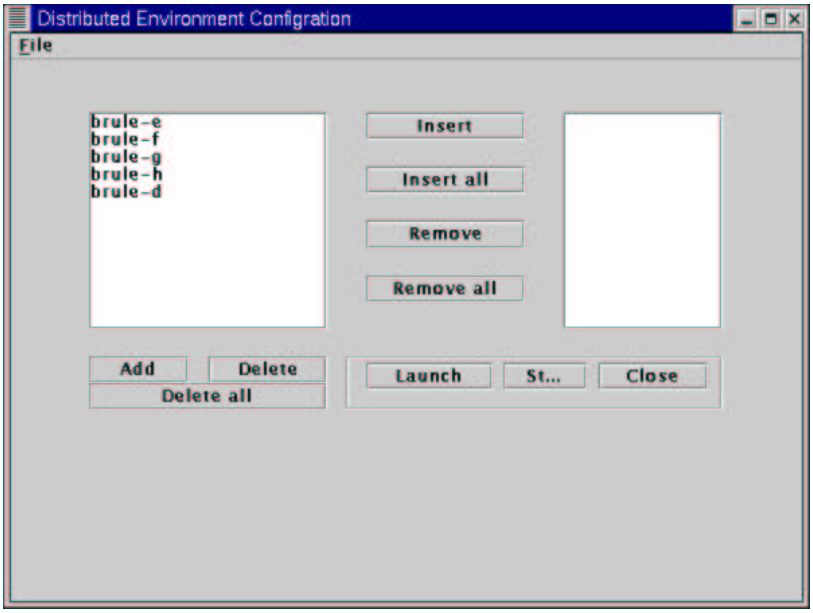


Figure 6.3: The window to configure the DCO_2P_3S environment

After the environment is launched, all the underlying details about the distributed parallelism can be ignored by the programmer, who can use DCO_2P_3S the same as CO_2P_3S. The PDP process still applies. Pattern selection and adaptation, code generation and application instantiation are done using the central control. No runtime environment will be involved until the generated application is actually executed.

In DCO_2P_3S, a generated application consists of a main program and a series of distributed slave processes. The main program is executed on the central control machine. Each process is implemented as a Jini `Activatable` service (described in Chapter 3) running on a distinct machine in the environment. Each slave process contains a proxy and a backend.

Each distributed machine, except the central control, in the DCO_2P_3S environment runs an `rmid` to take care of the launching and shutting down of `Activatable` processes. DCO_2P_3S employs the Java **Activation** system (Chapter 4) to make efficient use of resources on distributed machines.

One LUS is setup in DCO_2P_3S to control the registration and discovery. By using the LUS, the main program can obtain proxies to all distributed processes to launch the computation and keep them coordinated. For instance, a DM_Mesh application (introduced in Chapter 8) has a main program and a set of distributed processes. The main program creates a mesh data structure and divides it into smaller blocks. Through the LUS, the main program finds all registered processes that are able to process the blocks. Using proxies of these processes, the main program can send one mesh block to each process and let them work on the blocks in parallel.

This Jini-centric infrastructure allowed us to easily incorporate proper components to support complex behaviors of distributed applications.

6.3 Communication scheme

After setting up the system architecture, we should decide an efficient and easy-to-use communication mechanism for interactions between distributed processes. This section will describe the design of the communication scheme in detail.

6.3.1 Design choices

A key feature of the Jini technology is that it supports customizable communication scheme. A service designer can choose whatever applicable protocol for communications between one service and its clients. Furthermore, the design choice for the service is totally transparent to the service's clients, who only need to load the proxy at runtime to discover the service's interface for further interactions.

The default communication scheme used by Jini services is RMI, which is also used in all Jini infrastructure components such as the Lookup Service, the JavaSpaces service, the transaction manager and the Lease Renewal Service.

If RMI is used as the communication scheme in a user service, the proxy is simply a stub that is automatically generated by the RMI compiler (`rmic`). Using RMI is advantageous since it greatly simplifies the communication design of distributed applications. However, since RMI is not open-ended and it hides almost all the

communication details from the user, RMI is not flexible or customizable. It leaves programmers little opportunity for fine tuning in high-performance computing applications.

Java sockets are an alternative to RMI. Using sockets, the user is responsible for the correctness and efficiency of communication. As shown in the `JiniPrinter` example in Chapter 3, the designer implements communication details for the proxy and the backend. However, sockets have more flexibility than RMI and may perform better since:

- more efficient wire protocols can be adopted,
- the designer can choose to implement a fat proxy or a thin proxy [33]:
  - a fat proxy can process some or all of a client's requests locally without sending them to the backend. This approach is effective if a request can be efficiently processed locally instead of involving network communications. Consider a request to ask for the remote server to display real-time graphics based on the client's input. It may be better if the proxy processes the client's input directly and then draws the graphics.
  - a thin proxy simply forward the request to the backend and waits for a reply.

Communication is often a performance bottleneck in distributed computing. Customized communication may achieve higher performance by allowing the pattern designer to reduce communication overhead for specific patterns.

DCO_2P_3S aims not only to reduce the complexities of parallel programming in a distributed environment but also to support the design of new distributed parallel patterns. The conflict between requirements for efficiency in communication and broad abstractions to assist pattern designers suggests that a compromise is necessary. RMI was selected to design all required Jini services because of its simplicity. However, the existing RMI implementation is modified to diminish the performance gap between it and Java sockets. Based on this approach, I have created a system that uses RMI for design simplicities but achieves performance similar to that of TCP Sockets. In particular, I have created a modified version of RMI (DCO_2P_3S-RMI) that uses a more compact and efficient serialization scheme (DCO_2P_3S-serialization) designed for high-performance computing. Using the modified RMI provides generality to DCO_2P_3S programs, since it reduces the need for end-user involvement for performance tuning of the communication protocol. This helps to maintain the CO_2P_3S goal of hiding parallelism from the end users. Moreover, it also reduces the complexity of the design of pattern templates. The DCO_2P_3S-RMI and DCO_2P_3S-serialization are discussed in Chapter 7. Experiments were conducted to assess the performance differences between three kinds of communication schemes: RMI, Java TCP sockets and C TCP sockets. The results are listed in Table 6.1. To conduct the performance test, I implement a Jini service both using the RMI and Java TCP socket. These two implementations follow the examples described in Chapters 3 and 4. A C client/server program was also developed to show the C socket performance.

The test program contains three major participants: two services and a client, all of which run in the Jini environment. The schematics of this program are defined

as follows:

1. The two services register with the Jini LUS from two different machines and wait for requests.

2. The client locates the Jini LUS to find these registered services and retrieve their proxies.

3. The client invokes methods on the proxies. Performance results of the method invocations on these two services are recorded.

Both services implement the same interface that contains three methods, each of which has an array argument. The method signatures are shown in Figure 6.4. The TestClass is a user-defined class with 6 instance variables: one is int and the other five are Integers.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloService {
    public void handleObjArr(TestClass[] objArr)
        throws RemoteException;
    public void handleStrArr(String[] strArr)
        throws RemoteException;
    public void handleIntArr(Integer[] intArr)
        throws RemoteException;
}
```

Figure 6.4: The interface of this service

The performance comparison focuses on the difference between the pure overhead involved in the coding and decoding of method invocation information of the RMI implementation and the TCP Sockets implementation. The real semantics of these three methods are not important here. The test program was run on a pc Cluster with 19 nodes connected with both the Myrinet and 100Mb Ethernet connection. Each node has dual Athlon MP 1800+ cpus and 1.5GB of RAM. The operating system kernel is Linux 2.4.18-pfctr and the JDK version is Java HotSpot VM 1.3.1. Because RMI does not make use of non-TCP networks, I ran all performance benchmarking with the Ethernet connection. The virtual machine was started with a 256MB heap space. Table 6.1 records the execution time (in microseconds) of these three methods in the three implementations. The time is the average of 100 executions. Clearly the performance of the C Sockets is the best. And, as excepted, sockets are a bit faster than RMI.

6.4 Distributed synchronization mechanisms

After creating the system architecture and choosing the communication scheme, I need to devise a distributed synchronization mechanism. A proper synchronization package is indispensable for parallel applications. This section will introduce the CO_2P_3S-synchronization primitives that can be used to describe the parallelism in design patterns.

Implementation	handleObjArr (TestClass[1000] objArr)	handleStrArr (String[1000] strArr)	handleIntArr (Integer[1000] intArr)
RMI	36500	4600	6300
Java TCP Sockets	26600	4100	3500
C Sockets	200	170	58

Table 6.1: Comparisons between RMI, Java TCP sockets and C sockets (in microseconds)

6.4.1 Synchronization

Parallel applications use synchronization mechanisms to keep shared data consistent and processes coordinated. Synchronization techniques include Mutex, Semaphore, Barrier, Monitor, global clock and spin lock. The Pthreads, PVM and MPI libraries synthesize complex synchronization mechanisms from simple atomic hardware primitives. Java provides Monitors to fully support thread synchronization at the language level. Such synchronization operations can also be implemented totally in hardware for fine-grained parallelism.

One of the strengths of Java is that it supports multithreaded programming at the language level. Thread synchronization, the major component in the Java multithreaded model, is supported by Java Monitors [39]. The Java virtual machine associates a monitor with each object and provides two opcodes—`monitorenter` and `monitorexit`—to access the monitor lock. In Java, it is convenient to implement complex thread-level synchronization semantics based on Monitors because in shared-memory systems, the heap space and the method area are shared among all threads.

6.4.2 The distributed synchronization implementation

In Java distributed computing, process synchronization has no direct support from RMI and has to be implemented by the programmer. A distributed synchronization mechanism is devised from scratch in DCO_2P_3S to simulate the Java Monitor in a distributed-memory environment. It can be used as a parallel primitive for expressing high-level parallelism in the Intermediate Code Layer of CO_2P_3S pattern templates, or it can be used directly in distributed programming. Pattern designers with experience using Java Monitors can easily acquaint themselves with DCO_2P_3S-synchronization as they share similar syntax.

In our original design of DCO_2P_3S-synchronization, message passing was used. The resulting implementation was self-contained and easy to use. However, because of the lack of lower level communication support, setting up an all-to-all TCP connection between N processes was too expensive. Using UDP does not help either. Since UDP is not able to guarantee the arrival of data packets and their arrival order, it is difficult to use UDP to implement correct coordination semantics, unless additional checks are added at the application level.

Therefore, I chose to rely on the JavaSpaces technology (introduced in Chapter 3). The basic idea is to simulate the Java Monitor in a distributed environment. A JavaSpaces service is setup in the DCO_2P_3S environment; it stores variables that are shared among a collection of processes. The JavaSpaces service provides mutually

exclusive access to the shared variables. By this means, the basic mutex lock can be implemented.

The class structure of the synchronization subsystem is described in Figure 6.5. The whole synchronization package includes the following classes: `Barrier`, `Monitor`, `Mutex`, `MutexEntry`, `ReadyQueue`, `ConditionQueue`.

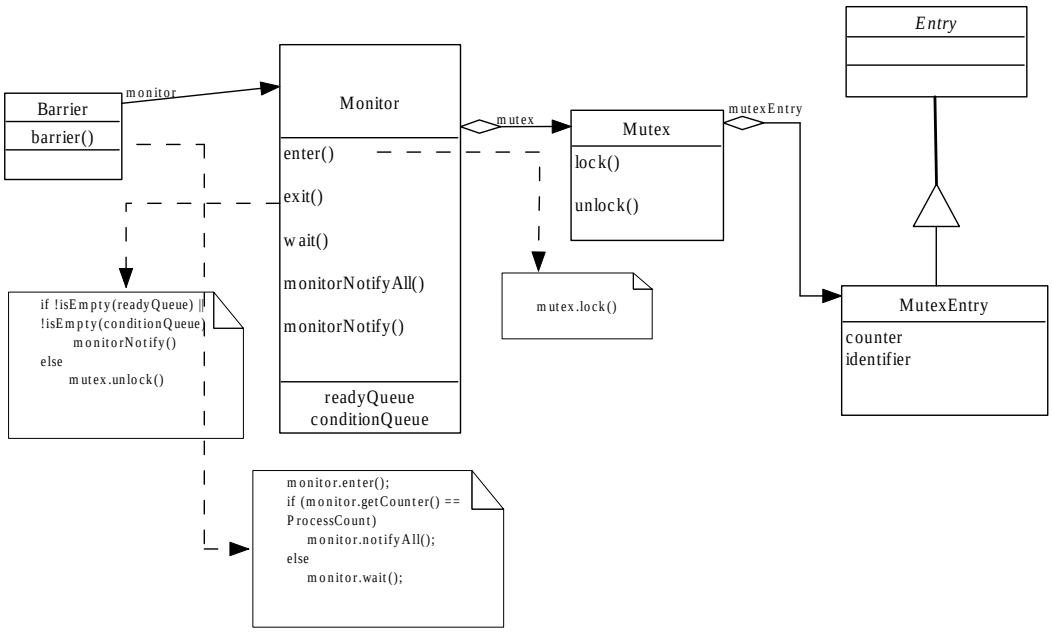


Figure 6.5: Class hierarchy of distributed synchronization mechanism

The `MutexEntry` extends the `net.jini.core.entry.Entry` interface, the superclass of all objects that can be stored in a JavaSpaces service, with two extra fields: a counter to be used in the `Barrier` to count the number of processes that arrive and a string value acting as a unique key identifying each distinct `MutexEntry` object in a JavaSpaces service.

Each instance of class `Mutex` contains a `MutexEntry` and provides mutually exclusive methods such as lock and unlock to access the `MutexEntry` instance. These two methods are implemented based on two blocking methods—`take` and `write`—in the JavaSpaces API. The `take` and `write` methods provide processes synchronized accesses to entries stored in a JavaSpaces service. Each time an instance of class `Mutex` is created, a distinct `MutexEntry` instance is stored in the JavaSpaces service. Processes coordinated by the mutex invoke its lock method to acquire the mutexEntry. One process will eventually succeed and remove the mutexEntry from the JavaSpaces service, while the others will be blocked until the mutexEntry is available again. The mutex lock can be released by the owner process (the mutexEntry is written back to the JavaSpaces service) so that others can compete for it.

I built a DCO_2P_3S-`Monitor` based on the simple `Mutex` for more complex functions. A group of processes can wait on some conditions for coordination, as supported by the Java Monitor. A monitor has one mutex and two queues that store blocked processes. In Java, the JVM associates one monitor for each object auto-

matically, while in this case a monitor has to be generated explicitly for each use. The following is the behavioral description of the DCO_2P_3S-`Monitor`:

- Instead of using the "`synchronized(object)`" syntax to guard a code block, we use `Monitor.enter()` and `Monitor.exit()` to signify the entry and exit of a code block guarded by the monitor. A monitor contains two queues [13] to store waiting processes:
  - A ready queue stores processes that are ready to continue their computations.
  - Processes that are waiting for certain conditions to be satisfied are stored in a condition queue. If a process successfully obtains the mutex lock to continue into the code block, but later it finds that a certain condition is not yet satisfied, the process simply invokes `Monitor.monitorWait()`, which stores the process into the condition queue and releases the mutex lock (Figure 6.6). This is analogous to calling the `wait()` method in a standard shared-memory Java program that uses threads.

```
monitor.enter();

if isTrue(condition)
    monitor.monitorNotifyAll();
}
else {
    monitor.monitorWait();
}

monitor.exit();
```

Figure 6.6: The distributed monitor

No queue is used to store the processes waiting for the mutex lock as such a function is provided implicitly by the blocking mechanism of the take method implementation of the JavaSpaces service. If the mutex lock is unavailable when a set of processes are trying to acquire it by invoking the take method, all the processes will be blocked. Once the mutex lock is available in the JavaSpaces service, one take method will return, resulting in waking up one process. The specific order in which the processes are chosen depends on the implementation of the JavaSpaces service.

- If the condition becomes true when one process enters into the code block, the process can invoke `Monitor.monitorNotify()` or `Monitor.monitorNotifyAll()` to wake up one or all the processes waiting for it. Awakened processes are transferred from the condition queue to the ready queue. Again, this is analogous to the shared-memory Java threads approach.

- Upon finishing the code block, a process must call `Monitor.monitorExit()`, causing the monitor to pick a process in the readyQueue to be activated first. If this queue is empty, the monitor will release the mutex lock to let processes waiting outside the code block compete for it.

These behaviors of a distributed monitor are very similar to the Java Monitor used for threads and are simple to use.

A barrier defines a synchronization point in a program that must be reached by a group of processes before any of them can continue. With the support of DCO_2P_3S `Monitor`, its quite easy to build a distributed barrier. A DCO_2P_3S barrier uses a monitor to guard concurrent accesses to the barrier counter in the monitor's `MutexEntry` instance. Figure 6.7 is the main method in the distributed Barrier class; it can be seen that the programming syntax is very similar to that of a Java Monitor.

```
public void barrier()
{
    monitor.enter();
    counter = this.getBarrierCount() + 1;
    if (counter == procGroup.length) {
        monitor.monitorNotifyAll();
    }
    else {
        this.setBarrierCount(counter);
        monitor.monitorWait();
    }
    this.setBarrierCount(0);
    monitor.exit();
}
```

Figure 6.7: The distributed barrier

6.4.3 Discussion

In the DCO_2P_3S-Synchronization implementation, a JavaSpaces service stores all the shared data and acts as a medium for indirect message passing. This design is valid and provides reasonable performance for medium-scale parallel processing. However, if there are a large number of participants, concurrent accesses to shared data will result in a serious performance problem. There are two possible ways of solving this problem:

- remove the JavaSpaces service and use explicit message passing for exchanging information and pass the mutex lock as a token. A tree algorithm or butterfly algorithm [43] can be used to reduce the number of messages exchanged. JNI (Java Native Interface [30]) can also be used to improve communication performance.

- use a distributed-memory version of the JavaSpaces service. Currently, there is one called GigaSpaces [37] which can act as a shared-memory layer for distributed systems and which has the same interface as a JavaSpaces service. Our implementation can be ported to the Giga-Spaces with only minor changes for better performance for large-scale distributed computing.

6.5 Performance monitoring and process management

In a distributed system, having central control over distributed processes and remote machines helps the user easily monitor and dynamically configure the system. We want to provide such a function in DCO_2P_3S too.

For this purpose, we need to implement the following:

1. a real-time performance monitor which gives graphical run-time information (CPU and memory load) for all participating machines;
2. a daemon that takes care of launching, shutting down and lease renewal (Chapter 3) of remote processes;
3. a central panel displaying the output of remote processes.

The performance monitor

A graphical display of the runtime information of distributed machines helps DCO_2P_3S users monitor the environment to make full use of idle machines and avoid highly loaded ones. Dynamic load balancing is not currently supported in DCO_2P_3S as this requires extensive modifications to the Java virtual machine. Rather, by the use of visual information, for most applications the user can easily improve performance using static load balancing.

Performance monitoring needs to gather low-level runtime environment information from distributed machines. However, as Java programs are interpreted by the JVM instead of being executed on the hardware directly, the programs have access only to the runtime information of the JVM and not to the physical machine. The `java.lang.Runtime` class provides methods—`freeMemory` and `totalMemory`—to return the free memory and total memory in the current JVM. The total memory can be set at the launch of the JVM to designate a certain size of memory to execute Java programs, while the free memory shows the currently unused amount of the heap space. Such information reflects the real-time status of the Java virtual machine. However, it does not represent the overall load in the machine that the JVM is running on.

To access low level hardware information in Java, I rely on the Java Native Interface (JNI), which is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications [30]. JNI enables a Java program to interact with programs that are designed in other languages such as C and C++. By this means, a Java program can cross the JVM boundary and perform many low level operations.

LibGTop [21] is a C library that can fetch a system's runtime information such as the memory and cpu usage. I implemented a performance monitor using a combination of JNI and LibGTop. A C program that uses the LibGTop library to retrieve the required information from the system is manipulated by a Java program through JNI (Figure 6.8). The "system information" menu item of Figure 6.9 is used to launch graphical displays[1] (Figure 6.10 and 6.11). They show the runtime information of the environment.

[1]The graphical display stems from a memory monitor in a Java 2D demo canned in the JDK1.2.2 distribution.

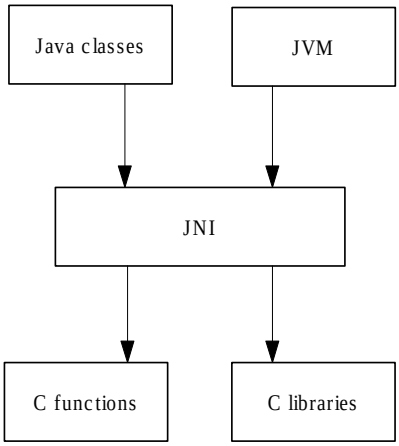


Figure 6.8: Java Native Interface

Figure 6.9: The menu item to start up performance monitoring

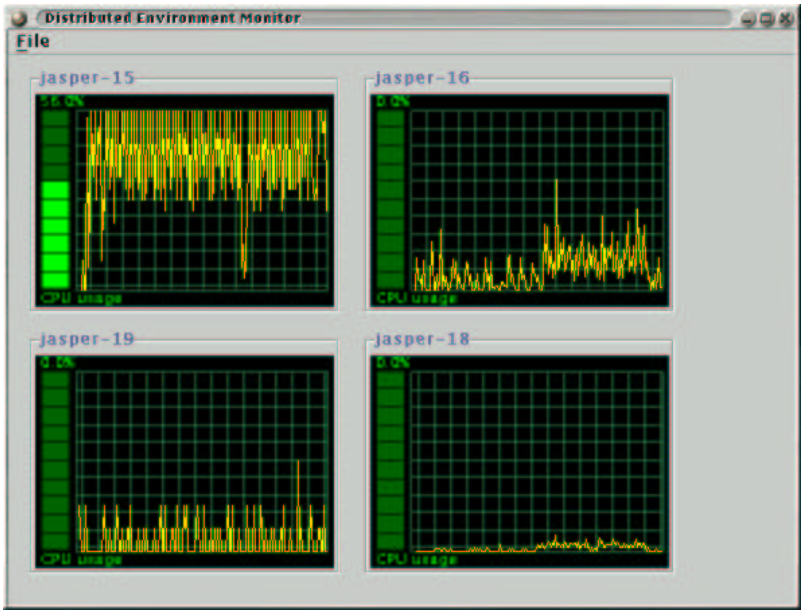


Figure 6.10: The CPU usage of the machines in the environment

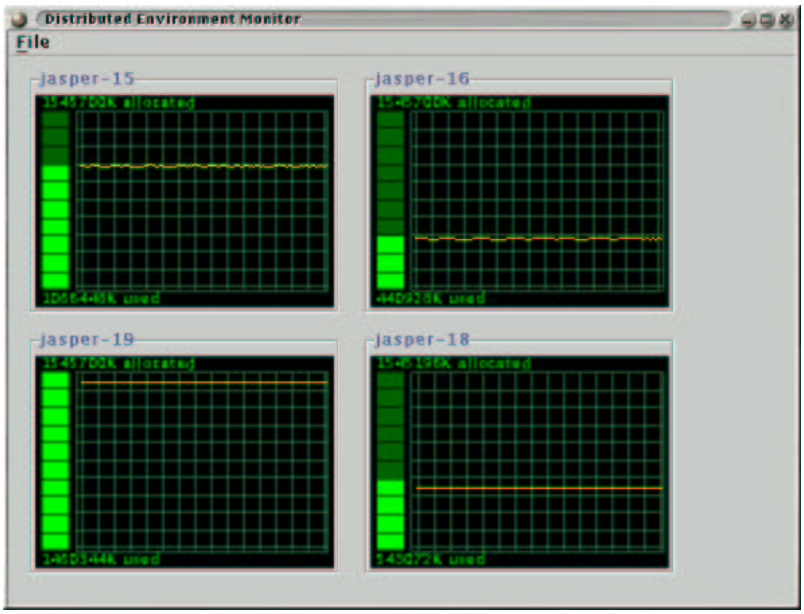


Figure 6.11: The memory usage of the machines in the environment

The process manager

A daemon running at each participating machine is responsible for remote process management. A central manager controls these daemons by sending various commands. These daemons are Jini activatable services so installing an additional daemon on each machine does not require many resources that would degrade the application's performance. Each daemon controls all other local processes required by the DCO_2P_3S environment. The daemon can launch and destroy processes, manage their leases with the Jini LUS, and collect their output.

Remote display

Remote display is always a difficult problem in distributed systems because of a lack of low-level control over distributed processes. However, I have solved this problem by using the daemons. Each daemon gathers the output of all local processes and sends the output to the central manager.

Figure 6.12 shows the run dialog used to execute generated applications. The user must clarify whether an application is intended for a shared-memory environment or a distributed-memory environment. By clicking the radio box labeled "Distributed Memory", the DCO_2P_3S environment is chosen for runtime. The user can also specify the maximum and minimum heap size of JVMs that will be launched on the central control and distributed machines. In the text field labeled "Main class and command line parameters", the user specifies the Java class that launches the instantiated framework and provides necessary command line parameters. After all necessary information has been specified, the user can click the Execute button to run the application. Application output, including the results from distributed machines, is displayed in the large text box at the top of the window.

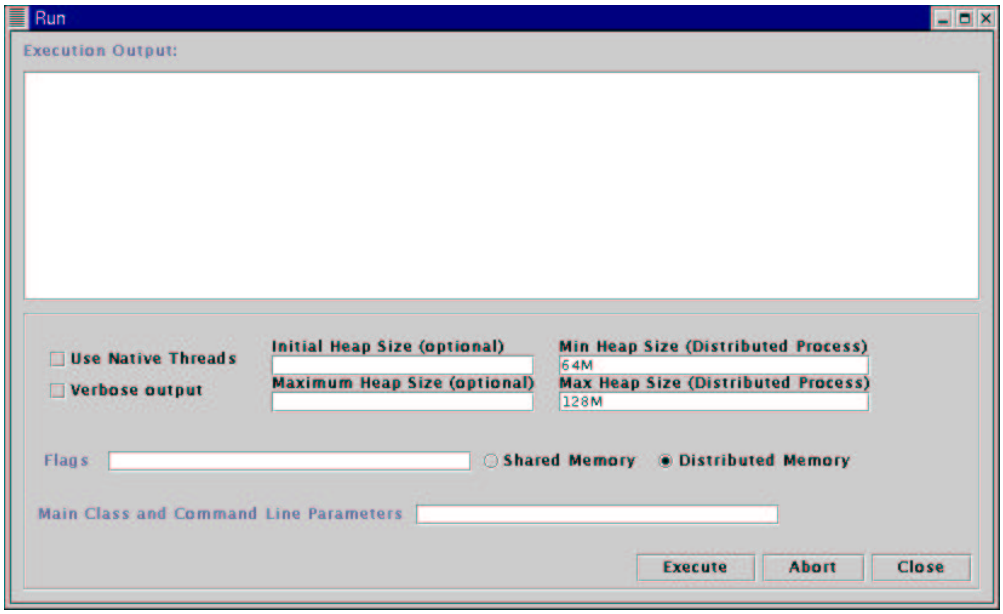


Figure 6.12: The run dialog of DCO_2P_3S

6.6 Distributed parallel design patterns

After designing and implementing the DCO_2P_3S environment, distributed versions of parallel design patterns were required to make it fully functional. The existing parallel design patterns in CO_2P_3S were Mesh, Distributor and WaveFront. Accordingly, three distributed patterns—DM_Mesh, DM_Distributor and DM_WaveFront—were added to the pattern library to support distributed programming in the DCO_2-P_3S environment.

Intuitively speaking, distributed versions of parallel design patterns can be created by replacing the shared-memory code templates in the existing patterns with distributed-memory ones. Threads are replaced by processes, which are implemented as Jini services. All the Jini-specific code is hidden in the framework. The details of each DCO_2P_3S pattern will be discussed in Chapter 8.

6.7 Conclusion

In the DCO_2P_3S implementation, Jini has been chosen to create the overall architecture to make the system flexible and reliable. DCO_2P_3S-RMI, a modified version of RMI, was employed to facilitate designers to generate efficient communication schemes without much difficulty. Besides, we also developed our own synchronization mechanisms and environment management tools to provide complete runtime support for distributed applications. This runtime environment, as well as new distributed design patterns, contributes to a fully functional DCO_2P_3S system.

Chapter 7

RMI and JDK-serialization Modifications

7.1 Motivation

This chapter focuses on enhancing RMI's performance by employing a more efficient and faster serialization routine. Since JDK1.2, RMI has been introduced into Java to seamlessly incorporate the distributed object model. Combined with Java's dynamic class loading, object mobility, an extensive security model and platform-independence, RMI provides a convenient way of building Internet-centric client/server applications. However, RMI cannot offer high performance for applications on high-speed networks. The reason lies mainly in two aspects: an inefficient transport subsystem and a slow object serialization routine. Although using technologies such as the HotSpot adaptive compiler, JIT compiler and Java native compiler can improve an RMI application's execution performance somewhat, the time spent doing object serialization and communication still occupies a significant amount of the total execution time of one RMI call.

- Since it is implemented almost entirely in Java and TCP sockets with only sparse uses of JNI to access low-level buffers, RMI is unable to fully exploit the network hardware resources. RMI cannot even gain much performance by running on Myrinet or some other fast user-level networks.

- JDK-serialization is a key component in RMI to implement the argument passing semantics of remote method invocations. However, as described in Chapter 4, JDK-serialization does a lot of work that is redundant in the context of high performance computing.

Table 7.1 shows the result of an experiment that identifies where the time (in microseconds) of an RMI call is spent. The first row lists the time it takes to finish three RMI calls; each with a different kind of argument. The second to fourth rows are time break-downs for all these calls. Times of serialization, de-serialization and network transportation are recorded in these three rows. Class `TestClass`, `TestClass1` and `TestClass2` were used in the test. `TestClass` contains one `int` field and five `Integer` fields; `TestClass1` class consists of 3 `int` fields; `TestClass2` also contains 3 `int` fields as well as two null `TestClass2`-type references.

The test program was run on a PC Cluster with 19 nodes connected with both the Myrinet and 100Mb Ethernet connections. Each node has dual Athlon MP 1800+ cpus and 1.5GB of RAM. The operating system kernel is Linux 2.4.18-pfctr and the JDK version is Java HotSpot VM 1.3.1. Because RMI does not make use of non-TCP networks, the Ethernet connection is used in all performance benchmarking. The virtual machines were started with a 256MB heap space.

It is important to note that serialization and de-serialization consumes most of the total time of an RMI call, and so does the transportation time. For the data in the second column, the total of the serialization time and de-serialization time exceeds the time of the RMI call. This seems wrong at first sight. However, the serialization process and the de-serialization process are not executed strictly in a sequential order. These two processes are executed on two different JVMs connected by a channel which in this case is a network connection. The serialization, transportation and de-serialization are like three stages of a pipeline that works on the arguments. For an RMI call, once the argument serialization is started, the first available data packet will be transfered through the network connection to the destination JVM, where the de-serialization will start immediately after the first data packet arrives. These three processes will run in parallel after the serialization process runs for a short while. Thus it is possible for the sum of the serialization and de-serialization time to exceed the time of the RMI call.

	TestClass[1000] objArr	**TestClass1 obj1**	**TestClass2 obj2**
RMI	36500	680	700
Serialization Time	22690	60	60
De-serialization Time	26570	160	180
Java TCP Sockets	1160	20	20

Table 7.1: The RMI time split-up (in microseconds)

As described in Chapter 4, the JDK-serialization process addresses the reliability and correctness of transporting objects among different virtual machines. JDK-serialization not only uses many verifications to avoid class version inconsistencies but also writes much extra information into the serialized data to correctly record object graphs. In the high performance computing domain, such overhead incurred is quite significant. The additional checks slow down the process time, while the extra data increases the network transportation overhead.

In the HPC area, I can safely assume that all participants of one parallel application share the same group of class definitions and trust the objects transferred between each other. Based on this assumption, I can remove many checks to reduce much of the overhead. Furthermore, with a thorough analysis of the JDK standard serialization wire protocol, I can possibly compress the serialized data to shorten the network transfer time.

As introduced in Chapter 5, RMI was used to describe communication schemes in design pattern templates. In order to make RMI suited better for high-performance computing as required by DCO_2P_3S, and to maintain the portability and compatibility of existing Java programs, I modified the JDK-serialization implementation. The RMI implementation is also modified slightly to make use of the new scheme.

By this means, our CO_2P_3S-RMI combines the existing RMI's ease of use with higher efficiency and lower overhead. Section 7.2 introduces related research involved in enhancing the RMI and JDK-serialization performance. In Section 7.3 the design of CO_2P_3S-serialization and modifications to existing RMI are discussed in detail. Section 7.4 includes a comparison between three versions of Java object serialization: JDK -serialization, DCO_2P_3S-serialization and UKA-serialization [34].

7.2 Related work

This section will introduce several endeavors that have been conducted to improve JDK-serialization and RMI.

- **Michael Philippsen et al.** re-implemented JDK-serialization in pure Java. Their new serialization mechanism, UKA-serialization, is used by karRMI [34], which is also a re-implementation of Java RMI. The adopted techniques in the UKA-serialization are the following:

  1. **Generated per class marshalling methods**: In JDK-serialization, if an object to be serialized has no customized marshalling routine, the runtime system will use Java reflection to retrieve the class information and all the fields of the object to write to the output stream. UKA-serialization uses a preprocessor to pre-compile each *ukaSerializable* [1] class to generate explicit marshalling and de-marshalling routines, which saves a lot of run-time overhead compared to using type reflections.
  2. **Slim encoding of type information**: Instead of writing the complete description of each class, only the fully qualified name is written into the stream.
  3. **Efficient reuse of type information**: As described in Chapter 5, a hash table is constructed in JDK-serialization to solve aliasing problems. The hash table helps avoid serializing the same object multiple times (it is possible that one object is referred by several other objects to be serialized). However, JDK-serialization refreshes the hash table before processing each new RMI call. The contents of the hash table for one call cannot be shared by other calls. Uka-serialization makes full use of the hash table by clearing it only once for each connection, which may contain multiple RMI calls. Thus different RMI calls can share one hash table.
  4. **Better buffering**: They make some private buffers in JDK-serialization classes public, so that they can be directly accessed and resized.

  UKA-serialization is fully compatible with JDK-serialization. For existing *serializable* objects it will delegate them to an instance of class `ObjectInputStream`. The enhanced serialization scheme will take effect only for those classes that explicitly implement the `UKATransportable` interface and per-class marshalling and de-marshalling methods. Unfortunately, because of the modified

[1] A *ukaSerializable* class can be recognized and serialized by UKA-serialization, just like a *serializable* class to JDK-serialization.

programming API, existing RMI cannot make use of UKA-serialization directly.

- **Henri Bal's group** at the Vrije University in Amsterdam built a system called Manta [20] to support Java-centric high-performance computing. The purpose of Manta is to remove existing Java runtime overhead (such as the bytecode interpretation and runtime reflection) as much as possible at compile time. The whole Manta system is implemented in C, including re-implementations of Sun RMI and JDK-serialization. Similar to UKA-serialization, Manta-serialization applies a new wire format and requires explicit serialization and de-serialization methods for each class. A native compiler compiles all Java class files into native code, including the marshalling and de-marshalling methods. The native code of these methods are invoked once an instance of their associated class is serialized. Based on explicit marshalling and demarshalling routines and native compilation, Manta-serialization is much faster than JDK-serialization. In addition, Manta uses a mechanism for a user to explicitly replicate a remote server at multiple places to reduce the RMI overhead even more.

- **Matt Welsh et al.** developed Jaguar [42] for efficient communication and I/O in Java. Jaguar uses an alternative to JDK-serialization called Pre-Serialized Objects (PSO). A PSO can be thought of as a Java object whose memory layout is already in a serialized form. Associating a PSO with each object can eliminate most of the overhead involved in official serialization and de-serialization. In Jaguar, the first time an object is accessed a PSO is created and associated with it. After that, all reads and writes to the object are mapped to its PSO. During an RMI call, the PSOs are sent across the network.

- There are some other research efforts (**Hyperion [12], Java/DSM [44], cJVM [5], and JESSICA [22]**) on simulating a distributed-shared-memory layer on clusters. The existing multithreaded programming model can be used without change and RMI is avoided.

7.3 Design of DCO_2P_3S-serialization

An important characteristic of CO_2P_3S-serialization is that it keeps the same programming interface as JDK-serialization. The user can program using DCO_2P_3S-serialization exactly the same way as using JDK-serialization. Existing Java distributed programs can enjoy the performance gains provided by DCO_2P_3S-serialization for free.

7.3.1 The CLASSPATH approach

To achieve transparency, portability and compatibility, I chose to implement DCO_2P_3S-serialization as a drop-in replacement for JDK-serialization. Existing JDK classes that implement the serialization process (`ObjectInputStream`) and de-serialization process (`ObjectOutputStream`) can be masked by DCO_2P_3S-serialization classes specified in a classpath. In JDK 1.1, a system's classpath can be easily changed by setting the CLASSPATH environment variable or using the '-classpath'

Java command line option. By this means, requests to JDK-serialization routines can be re-directed to our CO_2P_3S-serialization classes.

However, the class loading mechanism has changed since JDK 1.2 [27]. The JDK-serialization classes are now in another place which cannot be overridden by original "-classpath" option any more. When the Java runtime needs to load a new class for an application, it searches through a series of locations in the following order:

1. **Bootstrap classpath**: this path points to two jar files: rt.jar which includes all runtime system classes and i18n.jar which includes all internationalization classes.

2. **Installed extensions path**: this path includes jar files in the lib/ext directory of the JRE directory.

3. **classpath**: this path includes third-party classes specified by the system property java.class.path, which can be set by the CLASSPATH environment variable or by -classpath/-cp command line option at run time. Since JDK 1.2, java.class.path no longer covers the bootstrap class path and installed extensions as JDK 1.1 does.

Since the separation of class search paths in JDK1.3 disables the simple approach that works for JDK1.1, I resort to the Java command line options to override the bootstrap classpath. Using the non-standard option "Xbootclasspath", I can add our own classes to the existing bootstrapping class search path to realize the redirection. Fortunately, all the execution commands are encapsulated in the scripts launched by the DCO_2P_3S environment, so it is not necessary for the user to know anything about the classloading mechanism.

7.3.2 Implementation details

JDK-serialization emphasizes the correctness of shipping object graphs in Internet-centric applications with little considerations for high-performance computing in wide-bandwidth low-latency networks. In DCO_2P_3S-serialization the requirements of high-performance applications can be addressed by removing redundant checking (introduced in Chapter 4) and data compressing. I argue that my approach is valid because in these applications all objects are created and destroyed during a single run of an application, even though this single run is distributed over many processors. During one execution, all participants share the same group of class definitions and the class versions are unlikely to change. Based on this observation, I can save many checks and eliminate redundant information currently being transferred across the network.

The techniques applied in DCO_2P_3S-serialization are the following:

Compact class information

The idea of using compact class information, which has been applied in several other research projects [34] [20] records only the fully qualified name for each different class. As described in Chapter 5, the official serialization process is a recursive

process which saves the complete object graph into a byte stream. In the graph, the root node is the object that is serialized first, and all the other nodes are objects that are referenced directly or indirectly by the root. Each object is associated with one `ObjectStreamClass` object (also called the class descriptor) which contains complete class information per class. The class information includes:

- a fully-qualified class name, e.g. `java.io.ObjectInputStream`,
- an identifier which is unique per class,
- a URL (optional) specifying the location of the class file,
- all the description of all of its super classes except `Object`, and
- all the descriptions of instance fields including primitive and non-primitive ones (note that this may incurs recursive call of the serialization).

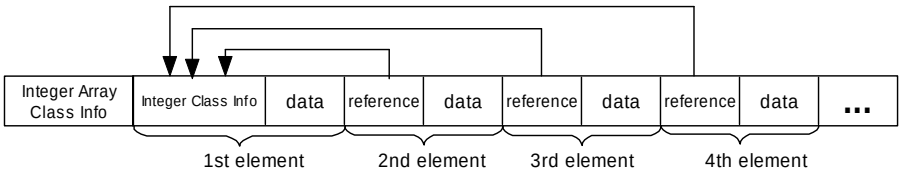


Figure 7.1: The stream format of the 1000-element integer array

In the resulting stream, instances of the same class share one class descriptor. As discussed before, JDK-serialization uses a strategy to solve aliasing problems. For example, during the serialization process of an array of 1000 `Integer` objects, an instance of `ObjectStreamClass` representing the `Integer` class will only be serialized once when the first array element is scanned. The serializations of the other elements will put references (4 bytes each) to the `Integer` class descriptor into the stream. By this means, the resulting object graph can be compacted, as shown in Figure 7.1. However, if the stream is a heterogeneous collection of objects, then detailed per class information still occupy significant space because a single application can use many different classes.

The complete class information can be reduced to a simple fully qualified class name (e.g. `Java.io.ObjectInputStream`) because all the participants in one parallel application have the same version of a class definition. I assume that the class definitions of the serialized data are locally available. This simplification not only saves the size of the serialized data but also removes a lot of reads and writes during the serialization and de-serialization processes.

Remove security checks

In a high-performance computing environment, participants of an application have mutual trust in each other so that objects can flow freely between them. The Java security control can be eliminated under this circumstance. All the security checks in our serialization and de-serialization processes are removed.

Compact object references

Consider the `Integer` array example again. The serialized data stream will include 999 references that point to the same instance of class `ObjectStreamClass` (Figure 7.1), which contains the information for the `Integer` class. JDK-serialization uses four bytes to represent one reference. Thus, each element has a redundant 4-byte reference to the same `Integer` class information. For the `Integer` array, each array element uses four bytes to store an `int` value. So the real data contents to be transferred are 4,000 bytes. However, the serialized data will be at least 8,000 bytes if the references are included. This is a significant amount of redundant space.

The identical object references can be treated in the same way the object aliasing problem is solved. Distinct object references are stored into a hash table. The index length of the hash table can be set to one or two bytes. Based on this scheme, a four-byte long reference can be replaced by an index of one or two bytes long and still supports 256 or 65535 different classes.

Each entry in the hash table has two attributes: one is a sequence number that reflects the order that the distinct reference was stored in the hash table. The other is the content of the reference. Figure 7.2 illustrates the approach. The first time a reference is met by the serialization process, it is written to the stream in full, headed by a short `int` with value -1. The new reference will also be stored into the hash table with a sequence number. In the rest of the serialization process, if an identical reference is to be written, only the sequence number of this reference in the hash table will be actually stored into the stream. During the de-serialization process, if the runtime encounters a -1, it reads in the following 4 bytes which is the content of this reference. New references are appended to the end of a reference array in the read-in order. If a positive number (a sequence number) is read, the process will return the entry in the reference array indexed by this number. Figure 7.2 illustrates the enhanced stream format of the serialized `Integer` array after using the compact reference scheme.

If the number of different references is less than 256, the sequence number can be just one byte long. Thus the 4-byte reference can be compressed to 1 byte. However, if the number exceeds 256, the hash table can grow in size and so does the sequence number; so the resulting compression rate will be reduced from 4 to 2. The sequence number can grow to 3 bytes too. Currently, the case of referencing more than 64k different classes in one application is ignored.

The price of adding an intermediate layer (the hash table) is adding one extra memory access to process each object reference and using more memory storage. However the reduced transfer time outweighs this increased overhead.

Homogeneous array serialization enhancement

In addition to reference compression, I applied another even more aggressive serialization technique for the special case of homogeneous arrays. Like many other object-oriented languages, Java supports polymorphism. Each element of an array with static type classA[ ] can be an instance of any subclass of classA at runtime unless classA is final. Therefore during the serialization process, the runtime type of each element must be computed and serialized even if all the elements end up having the same type. The dynamic type computation of array elements is expensive both

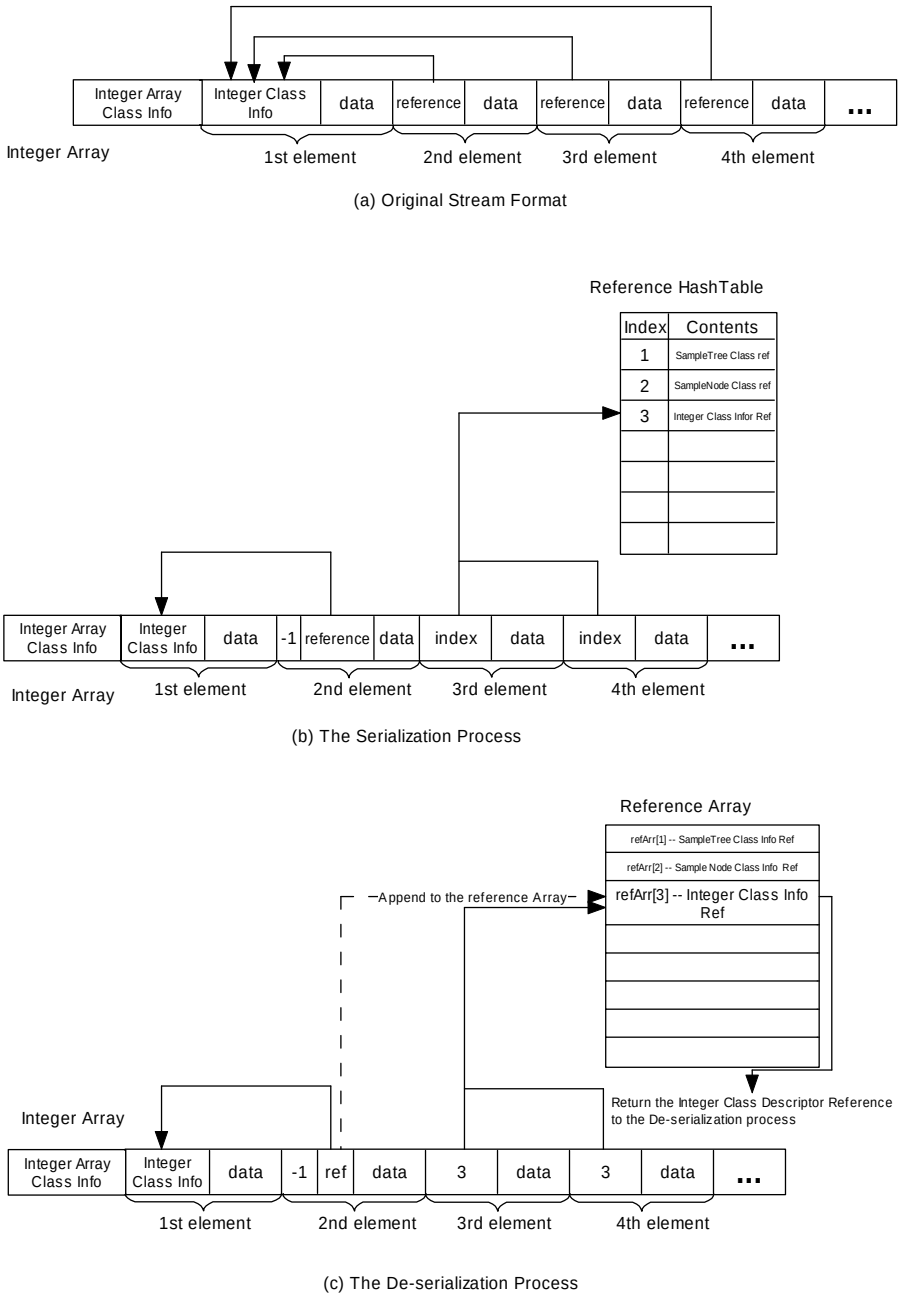


Figure 7.2: The enhanced stream format of the `Integer` array using compact reference scheme

in time and space, especially when the array size is large. If the array elements are heterogeneous at runtime, such computation is necessary. However, if we know the array elements are homogeneous ahead of time, can we do anything to save the overhead? The answer is yes. In Java, a class marked as final can never be subclassed. Therefore, an array whose component class is final must be homogeneous. we call such an array, a `final array`. Unlike DCO_2P_3S-serialization, JDK-serialization cannot make use of this final class information. In DCO_2P_3S-serialization, the wire format of a serialized final array deviates from a non-final array. As illustrated in Figure 7.3 for final arrays, the serialized data consists of two parts: a header and a body. The header simply contains the array class name (e.g. ``[Integer'' [2]) and the length. The body part contains no class information for each element, because such information can be fully inferred from the array class name. In the `Integer` array example, the de-serialization routine can safely infer that the elements are all Integers by the array class name. In Java all the primitive wrapper classes like `Integer` and `Float` are final classes so in practice the opportunity for savings is large.

In the case where an array is not a `final array`, a reference for each element class (dynamic) must be stored. However, if this element has fields (instance variables) that are primitive or instances of final classes, this approach is applied to these fields. The element class information, which includes the class name and the fields information (final or not final) can be cached in the memory. By applying the information to each array element, the serialization cost can be reduced by eliminating redundant class information and unnecessary tests.

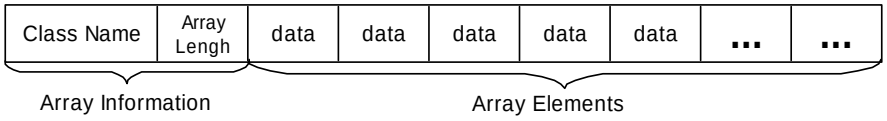


Figure 7.3: The stream format for a final array

The CO_2P_3S-RMI implementation

RMI uses two classes—`MarshalInputStream` and `MarshalOutputStream`—to process remote objects as well as ordinary serializable objects. `MarshalInputStream` subclasses `ObjectInputStream` and `MarshalOutputStream` inherits `ObjectOutputStream`. These two classes treat remote objects specially. For the definition of a remote object, please refer to Chapter 4. Consider an argument of an RMI call is a remote object, the RMI system will not pass the remote object to the callee. Instead, a proxy of it (described in Chapter 4) will be serialized into the stream at the call site. Thus the callee will use the proxy while executing the corresponding method. The mechanism of replacing one object with another to serialize it into a stream was introduced in Chapter 5. As the wire protocol has been changed in `ObjectInputStream` and `ObjectOutputStream`, `MarshalOutputStream`

[2]The name of the `Integer` array class is represented as ``[Integer''

and `MarshalInputStream` also have to be modified. The modification only involves reversing the order of several statements and this is the only change that has been made to the existing RMI.

7.4 Performance comparison

In this section, the performance of three serialization schemes—JDK-serialization, UKA-serialization and CO_2P_3S-serialization—are compared. Tables 7.2–7.5 lists the performance of these three schemes. Four kinds of data were used in the test: an array with 1000 `TransportableTree` elements, one `TransportableTree` object, an array with 1000 `TestClass` elements and one `TestClass` object. For one combination of each data type and each serialization scheme, three times (in microseconds)—the serialization time, the de-serialization time and the sum of these two— are recorded, as well as the length (in bytes) of the serialized data. The hardware and runtime configuration for running the experiment is the same as I described in Chapter 6.

Transportable Tree[1000]	Serialization time	De-serialization time	Total time	Length of the serialized data
JDK	28460	35800	63260	106123
UKA	25070	33030	58100	106084
CO_2P_3S	26430	36110	62540	71033

Table 7.2: Performance comparison for an array of TransportableTree (in microseconds)

Transportable Tree	Serialization time	De-serialization time	Total time	Length of the serialized data
JDK	440	290	730	188
UKA	320	310	630	140
CO_2P_3S	430	180	610	112

Table 7.3: Performance comparison for a TransportableTree (in microseconds)

TestClass[1000]	Serialization time	De-serialization time	Total time	Length of the serialized data
JDK	22690	26570	49260	10061
UKA	26890	26670	53560	11064
CO_2P_3S	19590	26330	45920	5025

Table 7.4: Performance comparison for an array of TestClass (in microseconds)

Class TransportableTree[3] implements the `uka.transport.Transportable` interface. Like `java.io.Serializable`, `uka.transport.Transportable` is the identifying interface for UKA-serialization to recognize *ukaSerializable* objects. The TransportbleTree has two `int` fields and two `TransportableTree`-type fields. This

[3]It is a tree structure designed in [34].

TestClass	Serialization time	De-serialization time	Total time	Length of the serialized data
JDK	360	400	760	38
UKA	720	550	1270	41
CO_2P_3S	350	180	530	22

Table 7.5: Performance comparison for a TestClass (in microseconds)

	HandleObjArr (TestClass[1000] objArr)	HandleStrArr (String[1000] strArr)	HandleIntArr (Integer[1000] intArr)
RMI	36500	4600	6300
CO_2P_3S-RMI	31380 15%	4220 10%	5600 12%

Table 7.6: Performance comparison between Java RMI and CO_2P_3S-RMI (in microseconds)

class also contains a set of methods which are generated automatically by a preprocessor. These methods will be invoked by the UKA-serialization routine to reduce runtime type checking. These methods are transparent to JDK-serialization and DCO_2P_3S-serialization. The level of a binary tree is two, i.e., an instance of TransportableTree in the test program contains 7 nodes (the first level is 0).

Class `TestClass` contains one `int` field and 5 `Integer` fields and only implements the `java.io.Serializable` interface. UKA-serialization will not recognize instances of this class and will just pass them to the standard JDK-serialization routine. Both the `TransportableTree` and `TestClass` class are final in order to use the aggressive array compression scheme of CO_2P_3S-serialization (Section 7.3.2).

As can be seen from the tables, although DCO_2P_3S-serialization is not the fastest in all conditions, the total time is always faster than standard JDK and is very competitive with or beats UKA while being plug-compatible with standard JDK. It is also the technique that provides the shortest serialized data in all cases. As described in Section 7.3.2, the compact stream generated in the serialization process deviates from the standard wire protocol. In the de-serialization process, additional tests have to be made to differentiate the specific meaning of the data in the stream. Extra data structures have to be created too. This extra limits the improvement of the de-serialization process compared to JDK-serialization.

In Table 7.6 I compare the performance of RMI and CO_2P_3S-RMI. KarRMI which uses UKA-serialization in the JavaParty project is not included. Unlike RMI and CO_2P_3S-RMI, KaRMI is re-implemented to support non-TCP/IP networks. It can easily achieve much better performance than existing RMI as the special properties of the network (such as Myrinet) can be exploited. As CO_2P_3S-RMI is still a pure-Java approach based on TCP sockets, I only compare it with JDK-RMI. The hardware and runtime configuration is the same as described in Chapter 6. As I can see from the table, CO_2P_3S-RMI gains up to a 15 percent performance improvement over current RMI. Furthermore, we can expect better performance as the argument size goes up.

7.5 Conclusion

In this chapter, I described the DCO_2P_3S-serialization scheme, including its design, implementation and performance. As a drop-in replacement for JDK-serialization, CO_2P_3S-serialization inherits the programming syntax of JDK-serialization. The two kinds of techniques have been used: removing security checks and employing higher data compression scheme. By applying these two techniques we not only decrease the communication latency (the object serialization and de-serialization time) but also shorten the length of transmitted data. Among these two techniques, the latter contributes more to the performance improvement. The wire protocol in the new scheme is based on the existing scheme, but obtains improvements for certain situations. The DCO_2P_3S-serialization scheme not only provides faster serialization and de-serialization than the existing scheme but also produces much shorter serialized data, which in turn reduces the network transportation overhead. As a result, I realized my goal of facilitating the pattern designer in designing efficient patterns without being involved in the communication design complexities.

Chapter 8

Distributed Design Pattern Templates in DCO_2P_3S

8.1 Introduction

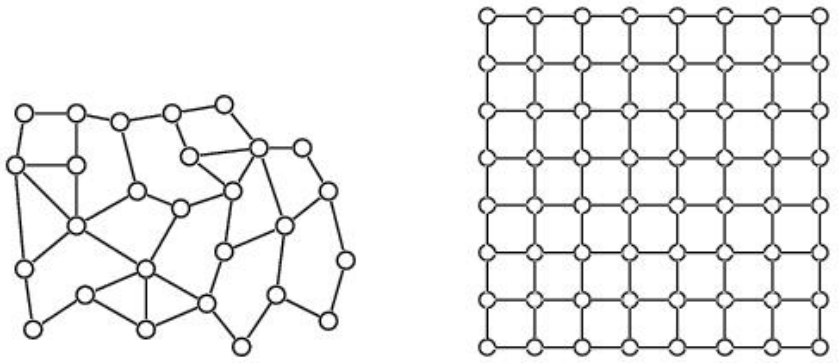
To make DCO_2P_3S fully functional, distributed-memory design pattern templates are necessary for the execution of distributed programs. With the help of MetaCO_2-P_3S, I created three new pattern templates: DM_Mesh, DM_Distributor and DM_Wavefr-ont. Each corresponds to one of the existing shared-memory templates (Mesh, Distributor and Wavefront) supported in CO_2P_3S. These new pattern templates encapsulate concurrency designs supported in the DCO_2P_3S environment. The user uses them in much the same way as their shared-memory counterparts.

8.2 The DM_Mesh pattern

This section introduces the two-dimensional distributed-memory mesh design pattern template (also referred to as DM_Mesh). The description format follows that in [15]. Here, we focus more on describing how to use the DM_Mesh template in DCO_2P_3S to generate framework code and how to use the framework to create a concrete application. Performance considerations are also discussed about how to incorporate efficient customized communication and synchronization implementations into the framework code.

8.2.1 Intent

DM_Mesh supports mesh computation, which iteratively computes new states for all elements of a two-dimensional data structure. Figure 8.1 [24] depicts two typical mesh surfaces; one is an irregular shape and the other is rectangular. The new state of each mesh element depends on the existing states of itself and its adjacent neighbours, which may be four nodes or eight (Figure 8.2). The iteration repeats until the states of all mesh nodes do not change by an amount greater than a predefined threshold [23].



(a) An example of a general mesh. (b) An example of a regular mesh.

Figure 8.1: Two examples of a general mesh and a rectangular mesh [24]

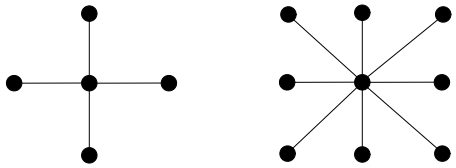


Figure 8.2: Neighbour stencils

8.2.2 Motivation

The mesh computation is commonly used in many computationally intensive applications such as computer animation and simulation. Sequential mesh solutions may take hours, weeks or even months to finish. Better performance can be achieved by exploiting parallelism. The two-dimensional mesh surface can be divided into multiple data blocks; each is assigned a computing unit (a thread or a process) for processing. These computing units run in parallel and coordinate with each other before each iteration. Each block cannot be processed until the boundaries of all neighbouring blocks and itself are exchanged (as depicted in Figure 8.3), else the new values of its boundary mesh nodes will be incorrect in the next iteration.

The existing Mesh Pattern template supported in CO_2P_3S generates multi-threaded Java framework code. A distributed-memory version of the Mesh pattern template for DCO_2P_3S was added into the pattern library.

The DM_Mesh template encapsulates the parallelism in the same way that the Mesh template does, except that it deals with multi-process programming in a distributed-memory environment. The framework code generated from the DM_Mesh template encapsulates distributed process creation, process discovery, process synchronization and network communication. Almost all the details of the distributed computation are hidden from the programmer. The DM_Mesh can be used in the same way as a shared-memory mesh, by simply instantiating the pattern parameters to generate the framework and filling in hook methods which contain sequential code.

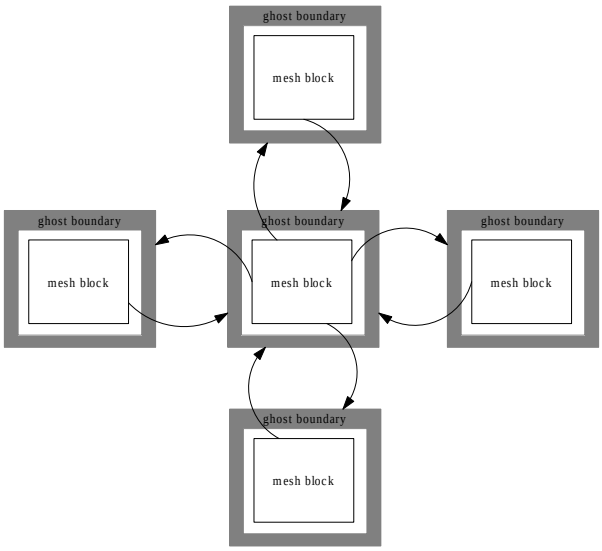


Figure 8.3: Boundary exchange scheme

8.2.3 Structure

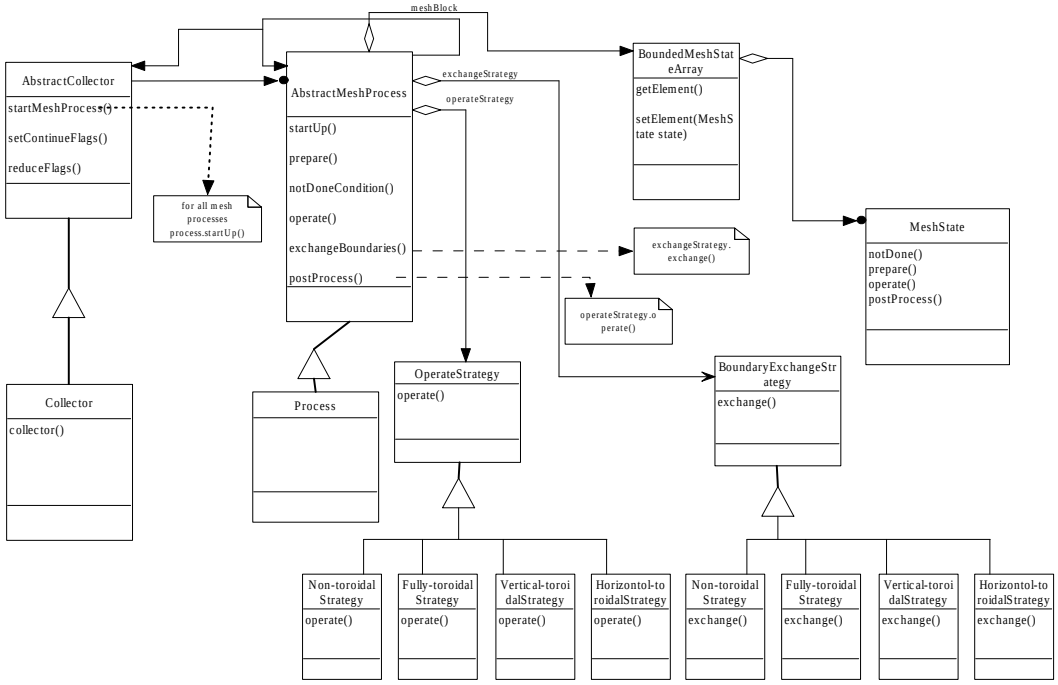


Figure 8.4: The class diagram of DM_Mesh

Figure 8.4 depicts the class diagram of the DM_Mesh pattern, including two major participants—the mesh collector and the mesh process. A mesh collector refers to a set of mesh processes which are computing units that work on mesh blocks (represented by the `BoundedMeshStateArray` class). The collector starts the mesh computation by distributing mesh blocks to processes and starting them up.

DM_Mesh processes coordinate with each other for the computation over their mesh blocks. This pattern also uses a *strategy* pattern [15]. Based on different boundary and neighbouring conditions, different strategies are chosen to iterate through the mesh blocks and exchange boundaries of neighbouring mesh blocks.

8.2.4 Pseudo code

The pseudo code for a DM_Mesh application is:

1. Create an instance of the DM_Mesh class, divide the mesh surface into a number of mesh blocks (according to the number of available machines) and send each block to a DCO_2P_3S process.

2. After the mesh blocks are distributed, the collector starts the execution of all the remote processes. For each process, the following is done in a loop:

```
While (computation not done)
        Exchange boundaries with processes which
                 have neighbouring mesh blocks.
        Compute the new states for all mesh nodes
                 in its own mesh block.
Endwhile
PostProcess
Send Back its final mesh block to the collector.
```

8.2.5 DM_Mesh pattern template parameters

To construct an application that uses a DM_Mesh, the user first selects the DM_Mesh pattern in the CO_2P_3S GUI, as depicted in the right pane in Figure 8.5. The user has to specify the template parameters to allow a specialized framework to be generated. The pop-up menu in the right part of Figure 8.5 displays the associated template parameters, which reflect design choices for this pattern. Different framework code is generated according to different combinations of template parameters. The following is a detailed description of the parameters with allowable values:

- **The DM_Mesh class name**: This is the name of the class that represents the whole mesh structure. This is the collector class. The user can create an instance of this class to start a distributed mesh computation.

- **The DM_Mesh state class name**: This is the name of the class whose instances represent the mesh nodes, which populate the two-dimensional data structure.

- **The DM_Mesh state super class name**: This is the name of the super class of the mesh state class. The user can easily incorporate application specific classes into the framework by fitting the mesh state class into a certain class hierarchy. As the template is implemented in Java, the default value of this parameter is `Object`.

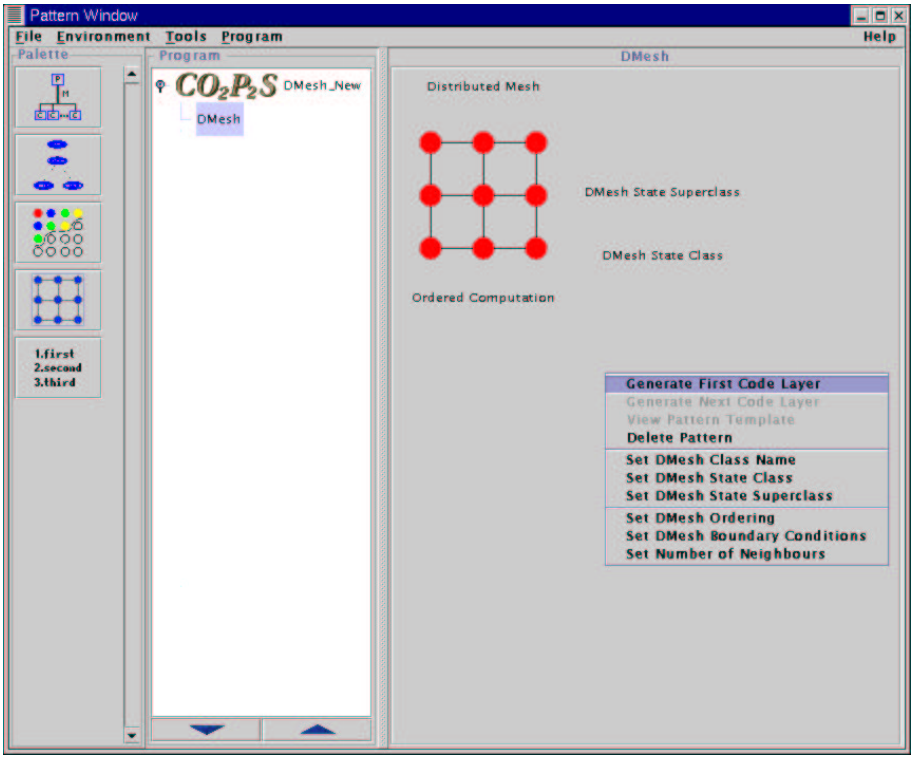


Figure 8.5: The graphical display of the DM_Mesh pattern template

- **Number of Neighbouring Elements:** This parameter defines the neighbours that are required for computing new values of each mesh node. A four-neighbour stencil and an eight-neighbour stencil are supported (Figure 8.2). Each node has four neighbors by default.

- **Boundary conditions**: This parameter defines how to process the boundary mesh nodes. It has four possible values:

  1. **Non-toroidal**: The boundaries do not wrap around. All nodes will be processed using only the connected neighbours.
  2. **Fully-toroidal:** All the boundaries wrap around; all nodes are treated in the same way.
  3. **Horizontal-toroidal:** The horizontal edges wrap around, while the vertical edges do not.
  4. **Vertical-toroidal:**The vertical edges are cyclic and the horizontal edges are not.

  Figure 8.6 illustrates these four boundary conditions based on the four-neighbour stencil. The boundary conditions of the eight-neighbour stencil are similar except that diagonals may also wrap around.

- **Mesh ordering:** The mesh ordering parameter defines whether the computation over the surface is chaotic (Gauss-Saidel) or ordered (Jacobi). If chaotic is selected, a node computation can use the current value or a previous value of

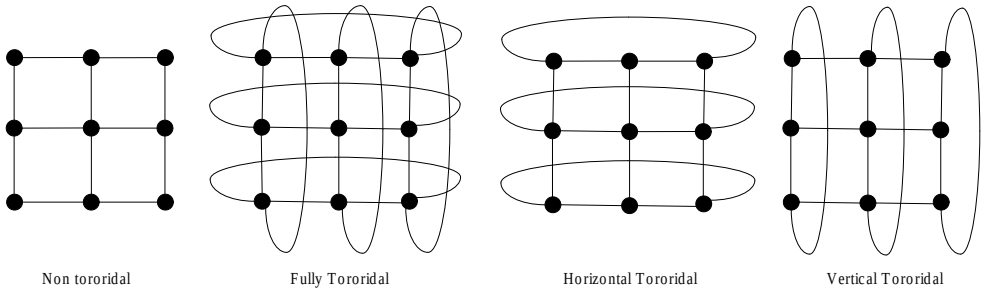


Figure 8.6: The boundary conditions of DM_Mesh

its neighbours. If ordered is selected, each node must use the latest neighbour's value. The default value is ordered.

8.2.6 Use of the template

As described in Chapter 2, the graphical interface provided by the DM_Mesh template (Figure 8.5) helps the programmer adapt the pattern to create a framework. The pop-up menu in the right part of Figure 8.5 allows the user to instantiate the template parameters. After parameter instantiation, the user can generate the framework by selecting the first menu item of Figure 8.5, "Generate first code layer". The menu item named "view pattern template" is enabled after the framework is generated. By clicking it, the programmer can view the framework code, which contains all the hook methods in another pop-up window (refer to Chapter 2 for details). The window helps the programmer input application-specific code to instantiate the framework.

8.2.7 Implementation details

In the shared-memory mesh pattern, there is no real boundary exchange phase since each thread has access to the whole mesh. Moreover, complex thread synchronization mechanisms such as the Barrier can be built easily from Java synchronization primitives. However, the benefits of the Java multi-threading model cannot be used on a distributed-memory architecture. The boundary exchange becomes an explicit phase with messages being exchanged between processes. These extra communications are the limiting factor for achieving high-performance. Java does not support distributed global synchronization.

To solve the boundary-exchange problem, unsynchronized message passing is used. The iteration over a mesh block is divided into two phases: the interior iteration phase and the boundary iteration phase. Each process has two threads that coordinate with each other for the computation of its mesh block. In the first phase, one thread iterates over the block's interior part. All of these mesh nodes can access the required data from their neighbours. At the same time, another thread is assigned the boundary exchange job. This thread sends messages across the network to fetch the boundaries from neighbouring blocks and stores this data. The first thread will be blocked after processing the interior part until the second one finishes the boundary exchange. In the second phase, the first thread alone will process all the boundary nodes. The boundary-exchange scheme is depicted in

Figure 8.3. This approach allows each process to keep the CPU busy while waiting for the data from the network. Therefore, the communication overhead can be overlapped partially by the computation.

The DM_Mesh pattern makes use of the new distributed synchronization mechanism introduced in Chapter 6 to solve the global synchronization problem.

8.2.8 An example DM_Mesh application

Many computer simulation and animation applications usually model the surface of a target object as a two-dimensional mesh. For example, in image processing, the image surface can be simply considered as a two-dimensional mesh, where each mesh node represents one pixel of the image.

Figure 8.7: The zebra strips generated by Reaction-Diffusion

Reaction-diffusion is a chemical process that provides certain stable graphical patterns. In this process, two or more chemicals diffuse over the surface and react with each other to form a stable pattern. In computer graphics, the reaction-diffusion simulation can be used to generate a zebra strip texture (Figure 8.7). In the rest of this section the DM_Mesh template will be applied to a reaction-diffusion application according to the PDP process as described in Chapter 2. In this section, I only focus on the first three steps of the PDP process, leaving performance considerations until the next section.

In the first step of the PDP process, we can easily identify that the DM_Mesh pattern suits the reaction-diffusion process. The algorithm defines a two-dimensional rectangular surface consisting of chemicals, each of which reacts with immediate neighbours. After identifying the DM_Mesh pattern from the reaction-diffusion application, we can choose the DM_Mesh pattern template from the left pane of Figure 8.8.

The next step is to instantiate the following parameters:

- The **DM_Mesh class name** is instantiated as RDMesh.
- The **DM_Mesh state class name** is set to be `MorphogenPair`. An instance of this class represents a cell with a pair of morphogens.
- The `MorphogenPair` class has no specific super class, so the **DM_Mesh state super class name** is set to be `Object`.
- The **topology** of the mesh is fully-toroidal.

- The **number of neighbours** is 4.
- The **mesh ordering** parameter is set to be ordered.

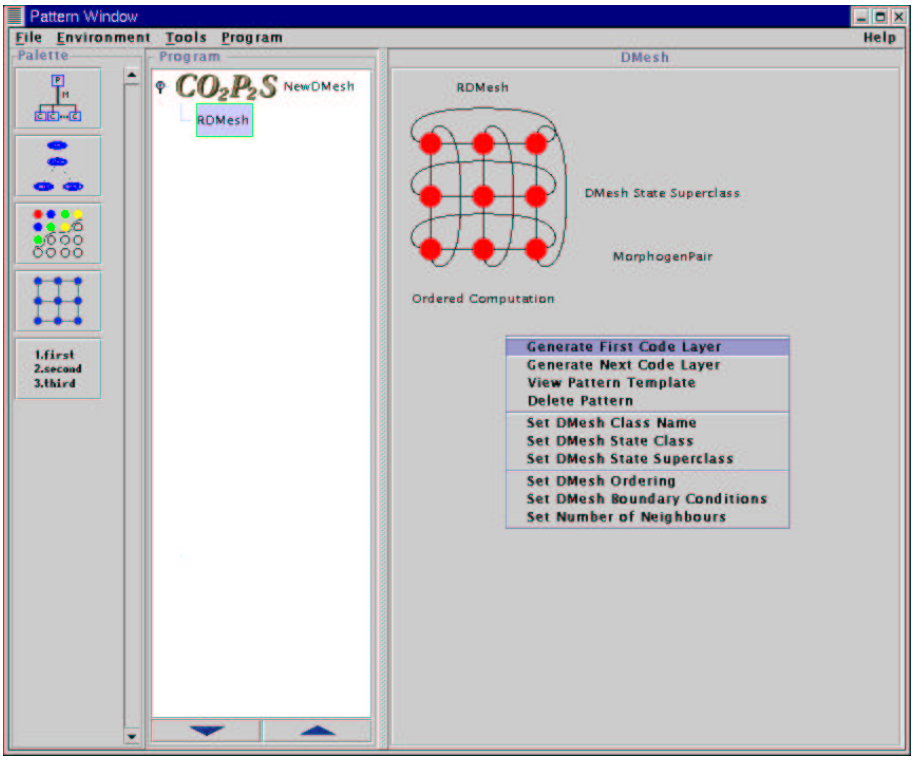


Figure 8.8: The DM_Mesh template with instantiated parameters

The third step is to generate a specialized code framework. The Reaction-Diffusion framework contains a series of hook methods (Table 8.1) that the user must fill in to provide application-specific details. These details include the underlying computation that updates the morphogen concentrations at a node, the threshold for stopping the computations and the reductions to be done on all the nodes once the computation is stopped. Figure 8.9 lists the main execution loop which invokes these hook methods in sequence.

```
public void meshMethod() {
        preProcess();
        while(notDone()) {
                prepare();
                barrier();
                operate();
        }
        postProcess();
}
```

Figure 8.9: Code fragment of meshMethod

Hook method signature	**Implemented functionality**
MorphogenPair(int i, int j, int surfaceWidth, int surfaceHeight, Object initializer)	This method uses the user-specified initializer object to initialize the mesh node at (i,j) of the surface.
void preProcess() void prepare() void postProcess()	These methods allow users to specify customized code fragment at different points of the execution loop.
boolean notDone()	This method evaluates the termination condition for each mesh node until no one returns true.
void reduce(int i, int j, int surfaceWidth, int surfaceHeight, Object reducer)	This method reduces the results of the mesh computation with the user-supplied reducer object.
void operate(MorphogenPair north, MorphogenPair east, MorphogenPair south, MorphogenPair west)	This method is the central part of the mesh computation. It defines how the MorphogenPair reacts with its immediate neighbourhoods. Because the mesh surface in this case is fully-toroidal, all mesh elements are treated the same way. If the topology is non-toroidal, six operate methods—operateLeft, operateRight, operateTop, operateBtm, operateCorner and operateInterior—will be generated to process mesh nodes at different positions on the mesh surface.

Table 8.1: Hook methods of the DM_Mesh Pattern

By filling in the hook methods with application-specific details, the DM_Mesh pattern has been fully instantiated into a specific mesh computation. The last thing left to do is to write a driver program which creates an instance of the RDMesh and launches the computation (Figure 8.10).

```
mesh = new RDMesh(surfaceWidth, surfaceHeight,
meshWidth, meshHeight, initializer, reducer) ;
mesh.launch() ;
```

Figure 8.10: Code fragment of mainMethod

8.2.9 Application performance

The DM_Mesh pattern can be used to generate parallel applications that achieve a speedup in a distributed environment. The performance data of the reaction-diffusion application are provided in Table 8.2.

Mesh Size	1 process	4 processes	Speedup
400x400	31	103.2	0.3
800x800	120	110.2	1.1
1200x1200	267	131.5	2.0

Table 8.2: The Reaction-Diffusion application performance (in seconds)

The program was run using meshes with three different sizes. The central JVM for the main program was started with a 512MB heap space. The distributed JVMs were started with a 256MB heap space. The speedups are based on the average wall clock time for ten executions compared to the sequential execution time using a HotSpot virtual machine. Note that the timings consider only the computation time and boundary-exchange time; the initialization and result gathering times are not included. The results given in the table are not very satisfying. Using the real-time monitoring function provided in DCO_2P_3S, each participating processor was found to be less than 40 percent utilized when the mesh size was less than 1200x1200. Most of the time was wasted on synchronization and boundary exchange. As the mesh size rose to 1200x1200, each processor received more work to do that compensated for the high communication overhead.

Based on the layered design approach supported in the PDP process, the user still has a chance to tune the RDMesh application if current performance is not sufficient. By following the last two steps in the PDP process, the user can modify the parallel-structure implementations in the Intermediate Code Layer and even modify some primitives in the Native Code Layer. In this case, the user needs to modify the code in both the Intermediate Code Layer and the Native Code Layer.

Consider the main execution loop in the framework code of Figure 8.9. All processes need to synchronize by calling the barrier method twice for each iteration. One call is between the `prepare()` method and the `operate()` method inside the execution loop. There is a hidden barrier inside the `notDone()` method that can be found using the Native Code Layer. At the hidden barrier, all processes must

submit the current status of their own computation. Based on this information, the collector decides whether the done condition is satisfied or not. Before such a decision is made, all processes have to synchronize at the hidden barrier. The second barrier is to ensure that no process can continue the computation until the last one finishes updating its block. The preparation work is some pre-processing that has to be done before each iteration.

The distributed barrier is very expensive in that it involves passing multiple messages among all the participants. If the number of barrier synchronizations can be reduced by one for each iteration, it will be a huge win in performance since it will reduce the total synchronization overhead by 50%. By inspecting the framework code and the code at the native layer, the user can choose to incorporate the `prepare()` method into the `notDone()` method before the synchronization point of the `notDone()` method. The new framework code is shown in Figure 8.11. Of course this modification comes with a price: now each block has to prepare even if it is not needed, so some cpu cycles maybe wasted.

```
public void meshMethod() {
        preProcess();
        while(notDone()) {
                operate();
        }
        postProcess();
}
```

Figure 8.11: Code fragment of the modified meshMethod

Mesh Size	1 process	4 processes	Speedup
400x400	30	63.9	0.5
800x800	120	71	1.7
1200x1200	267	90.8	2.94

Table 8.3: The Reaction-Diffusion application performance (in seconds)

Table 8.3 shows the performance results after modifying the code. As shown in Figure 8.11, these results are better than Table 8.2, since the synchronization points are reduced by half. Thus by using the PDP process, the programmer can further incorporate application-specific information into the generated program if such information cannot be captured by pattern parameters and hook methods.

Now that the code in Figure 8.11 can run faster than the code in Figure 8.9, we can modify the code generated by the framework to use this faster version. However, if the `prepare()` method is computation-intensive, always incorporating it into the `notDone()` method will waste CPU cycles in the last iteration, where the `prepare()` method is not needed. So this choice of code should become a performance parameter in the pattern. The user can choose which code should be generated.

8.3 The Phases pattern

8.3.1 Intent

Help decompose a program into distinct phases, which will be executed sequentially.

8.3.2 Motivation

The existing CO_2P_3S system supports the Phases design pattern template. The Phases pattern is not a real parallel design pattern; rather, it is a tool to facilitate program design. The Phases pattern helps the programmer to divide the overall control flow into consecutive phases, which are implemented as separate methods that are called in a sequential order. Each phase undertakes a distinct task. As a result, the Phases pattern facilitates the construction of an application with different kinds of parallelism in different stages throughout its life-cycle. The Phases pattern simplifies the design of complex parallel programs.

The Phases pattern can be supported in DCO_2P_3S without modification since it contains no information about parallelism in itself.

Structure

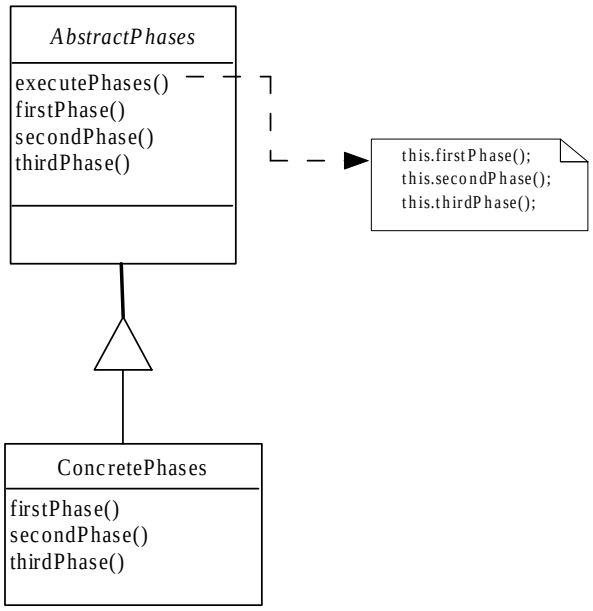


Figure 8.12: The class diagram of the Phases pattern [23]

Figure 8.12 illustrates the class diagram [23] of the Phases pattern. The structure of the Phases pattern is simple, and includes only two main classes: the abstract phases class and the concrete phases class. The user can modify the latter to add different methods; each represents a distinct phase of an application. The `executePhases()` method defined in the abstract class will invoke all these methods in the same order as they are added.

Parameters

- The **name** of the Phases class.
- A **sequence of methods**

8.3.3 Using the template

Figure 8.13 shows the Phases template window involving an example application named `SamplePhases`, which is the first pattern parameter. The method list in the upper right part of the window includes all current phases that the programmer wants to add to the application. By clicking the bottom entry of the pop-up menu in Figure 8.13, the programmer can enlarge or reduce the method list.

Figure 8.14 displays all the hook methods that the programmer needs to implement after the framework has been generated. The implementation of these hook methods can use different patterns. Therefore the generated Phases framework acts as a bridge that connects different parallel patterns together. Section 8.4.6 has an example application that uses the Phases pattern along with another pattern called DM_Distributor.

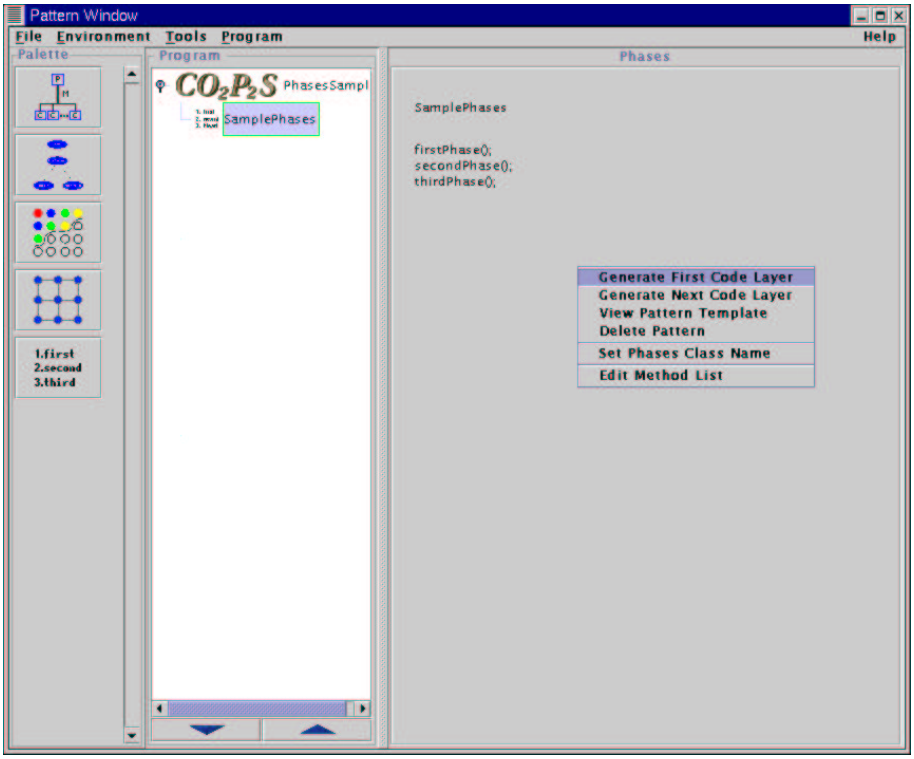


Figure 8.13: The template GUI of the Phases pattern

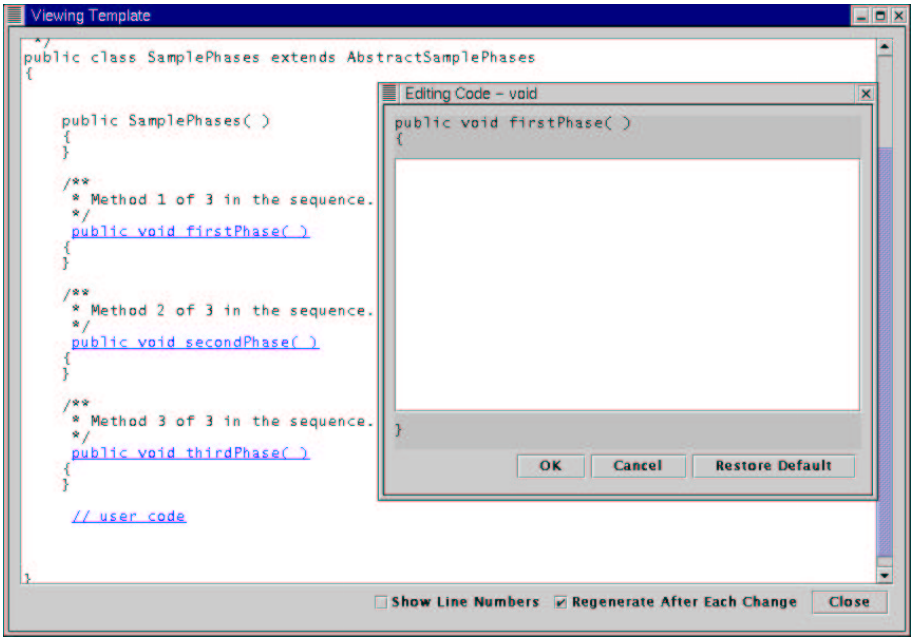


Figure 8.14: The code template of the Phases pattern

8.4 The DM_Distributor pattern

8.4.1 Intent

The DM_Distributor pattern template defines a data-parallel style of computation. It applies the master/slave paradigm: a master has a fixed set of slaves and works as a supervisor of them. The master divides up the work to be done and distributes it to the slaves. The master also gathers results from the slaves and returns the final result to the user.

8.4.2 Motivation

The master/slave paradigm is a fundamental and commonly-used model in parallel and distributed computing. As illustrated in Figure 8.15, a group of slaves work independently under the supervision of a master, who accepts tasks and divides them into sub-tasks, each of which is assigned to a slave. No direct access to the slaves from outside is allowed except through the master. The slaves act passively based on the master's commands. A slave does not have direct access to the master, nor does it have direct accesses to other slaves. In contrast, the master can gather necessary information from its slaves and supply them with new information. This approach improves parallelism by avoiding direct references between slaves. However, in a distributed-memory environment, requiring the master to be involved in all communications will inevitably increase the communication overhead. Enabling direct references between slaves may incur less communication overhead so they can spend more time on computational tasks.

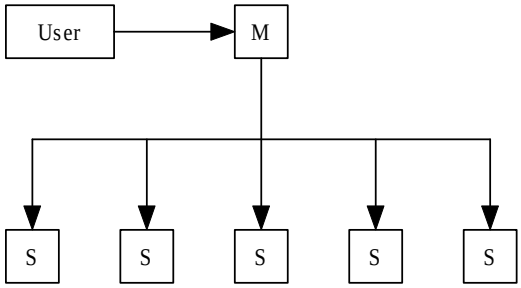


Figure 8.15: The object diagram of DM_Distributor

Master/slave applications have a common structure, which includes the task dividing, slave control and sub-tasks dispatching, and can be abstracted using a parallel design pattern template. The DM_Distributor parallel design pattern template supports general master/slave computations in a distributed-memory environment. This template allows the programmer to provide a list of methods, each representing a task that should be conducted in parallel by all slaves. The methods are defined as parallel methods that are implemented in both the master class and the slave class. The master's implementation is generated automatically, while the slave's implementation is provided by the programmer to deal with each individual subtask. In addition to these methods, the master can also have a set of sequential methods to manipulate the states of its slaves.

8.4.3 Structure

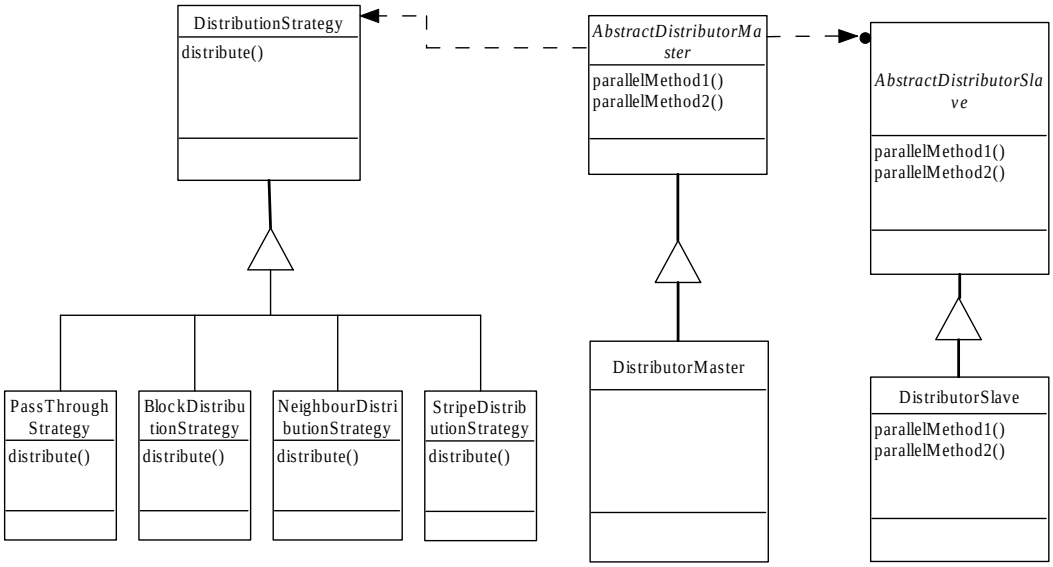


Figure 8.16: The class diagram of DM_Distributor

Figure 8.16 shows the class structure [23] of the distributor pattern, including two major components—the `DistributorMaster` and the `DistributorSlave`. An instance of the `DistributorMaster` refers to one or more `DistributorSlave` instances. The *strategy* pattern [15] is also used to help the master divide each task and dispatch resulting subtasks to slaves.

8.4.4 Parameters

The parameters [23] for the DM_Distributor design pattern template are the following:

- The **name** of the master class. The slave class name will simply have "Slave" appended after the master class name.
- **A list of parallel methods**. Each is implemented in both the master class and the slave class. The master's implementation is generated by the template. It encapsulates the details of job dividing, subtasks dispatching and invoking the methods on the slaves. The slave implementation contains user-supplied sequential code which defines the actions to be undertaken on each individual subtask.

  Each method element contains three sub-parameters:

  - The **return type**. Each slave will return an instance of this type once the method is done. The results are gathered by the master into an array to be returned to the invoker.
  - The **method name**.

– The **arguments** of the method. Each argument contains the type, name and distribution strategy. Once a client invokes the parallel method on the master with the required arguments, the master will invoke the same method on all the slaves. The arguments will also be distributed to the slaves to be used in the slave's implementation of the parallel method. The distribution strategy associated with each argument is used to decide the scheme for dividing and dispatching it. Four different distribution strategies [23] are supported:

  1. **Pass through**. Each slave will have a duplicate copy of the argument that the master has.
  2. **Block distribution**. The data (should be a one-dimensional array in this case) will be divided into several consecutive equal-sized non-overlapping data blocks, the number of which equals the number of slaves. Each data block will be assigned to one slave.
  3. **Neighbour distribution**. The data will be divided into several consecutive data blocks each of which overlaps with its neighbors somehow.
  4. **Stripe distribution**. Data elements evenly-spaced in the array will be chosen to form a data block to be assigned to each slave.

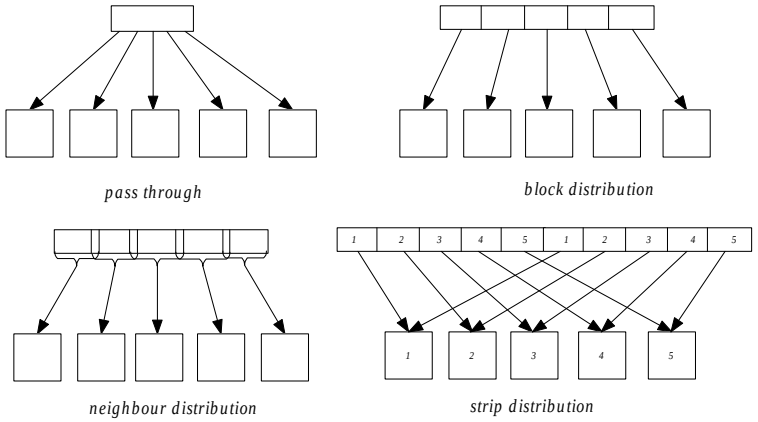


Figure 8.17: Distribution strategies

8.4.5 Use of the template

The DM_Distributor template GUI is shown in Figure 8.18. The pattern pane titled with "Distributor" depicts the object structure of the DM_Distributor pattern and all of its instantiated parameters for a sample application. The master class name in this case is DM_DistSample and the slave class name is DM_DistSampleSlave. The method-list parameter on the right part of the pane shows the user-specified methods, including the complete method signatures as well as the distribution strategy for every argument.

Methods can be added, deleted or edited by the use of the method input dialog shown in Figures 8.19 and 8.20. In the window shown in Figure 8.20, all the sub-parameters relating to one new method can be specified. For each method, the user

has to specify its return type, method name and arguments if necessary. The user has to specify the distribution scheme only for arguments that are one-dimensional arrays. Non-array arguments will be simply passed to the slaves by the master using the pass-through scheme.

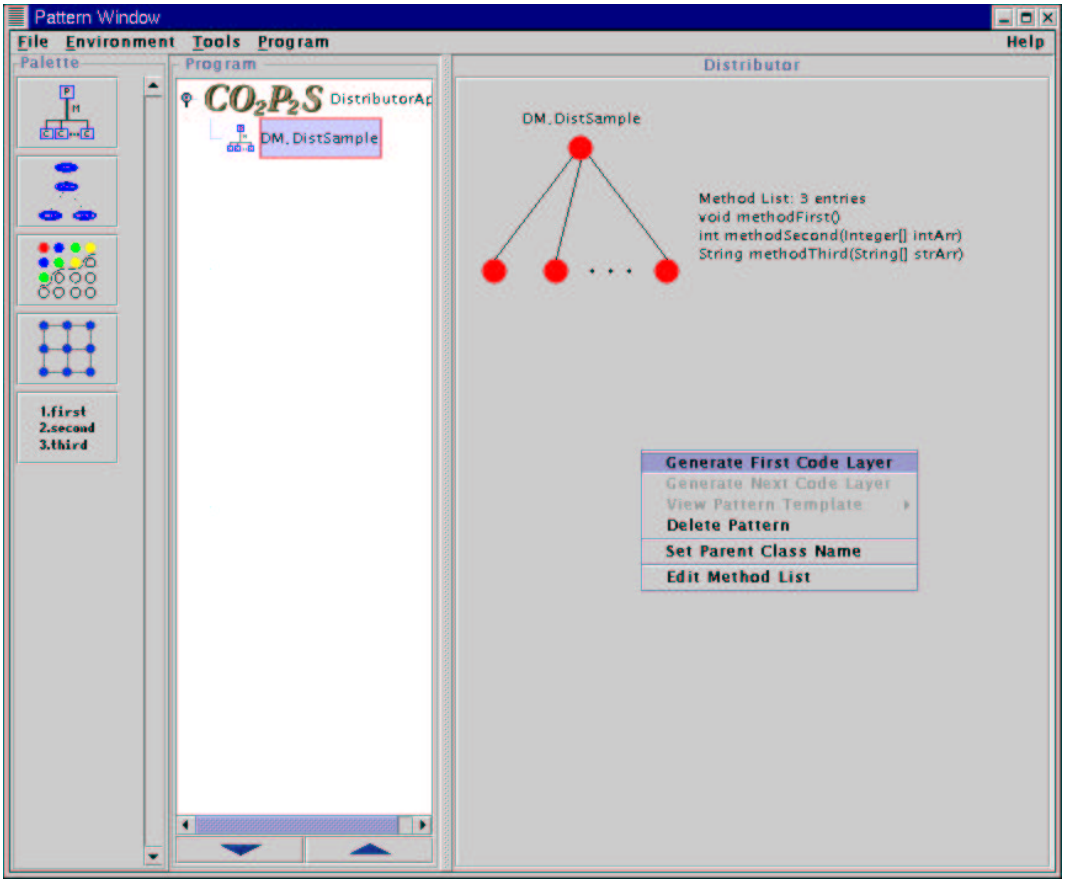


Figure 8.18: The template gui of DM_Distributor

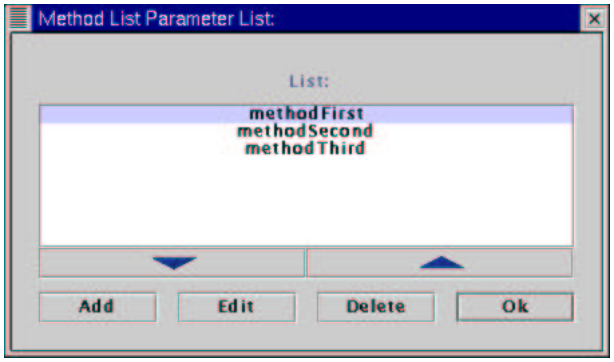


Figure 8.19: The method dialog of DM_Distributor

After specifying all the required parameters, the user can execute the code generation command by clicking the pop-up menu option shown in Figure 7.2.4. A new framework will be generated according to the specified class name parameter

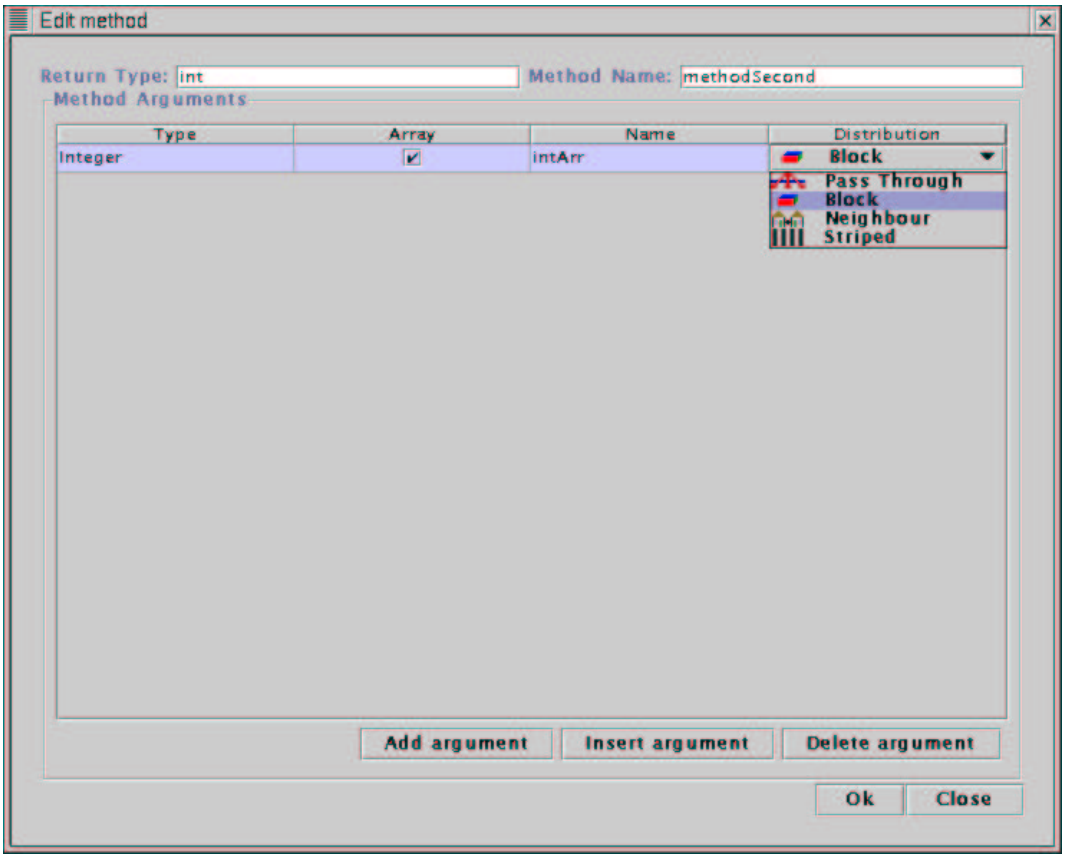


Figure 8.20: Editing one parallel method

and the method list parameter. The hook methods provided in the framework are simply the methods in the method list parameter. They provides a means for the user to specify application-specific sequential operations that the slaves should do under the master's control.

8.4.6 Example application

PSRS (Parallel Sorting by Regular Sampling) [36] is a parallel sorting algorithm suitable for a diverse range of parallel architectures. Its most notable characteristic is its regular-sampling load-balancing heuristic, which yields good analytical and empirical performance. During the sorting process, this strategy helps sample the data to find out pivots that are used for an even redistribution of the data across processors.

PSRS consists of four stages, as illustrated in Figure 8.21 [23]. The sequential execution order of these stages has to be guaranteed for correct results.

Suppose p processors are used in the PSRS algorithm, the four phases are:

- Divide the whole data into p equal-sized consecutive segments and each processor is assigned one. In parallel, processors sort their own parts and select evenly-spaced elements as samples to be used in the next phase.

- One processor is assigned to gather all the samples and sort them. Another set of samples (called pivots) are picked out from these sorted samples afterwards.
- Each of the other *p-1* processors receives a copy of the pivots and divides its sorted data according to them. All p processors exchange the resulting subparts according to different pairs of pivots. For instance, processor 1 fetches all the 1st subparts from all the other processors, processor 2 will have all the 2nd subparts, and so on. After the exchange is finished, subparts owned by one processor will be in the same range.
- In parallel, p processors merge their subparts into contiguous parts and one processor will collect all the sorted parts and concatenate them into a whole array.

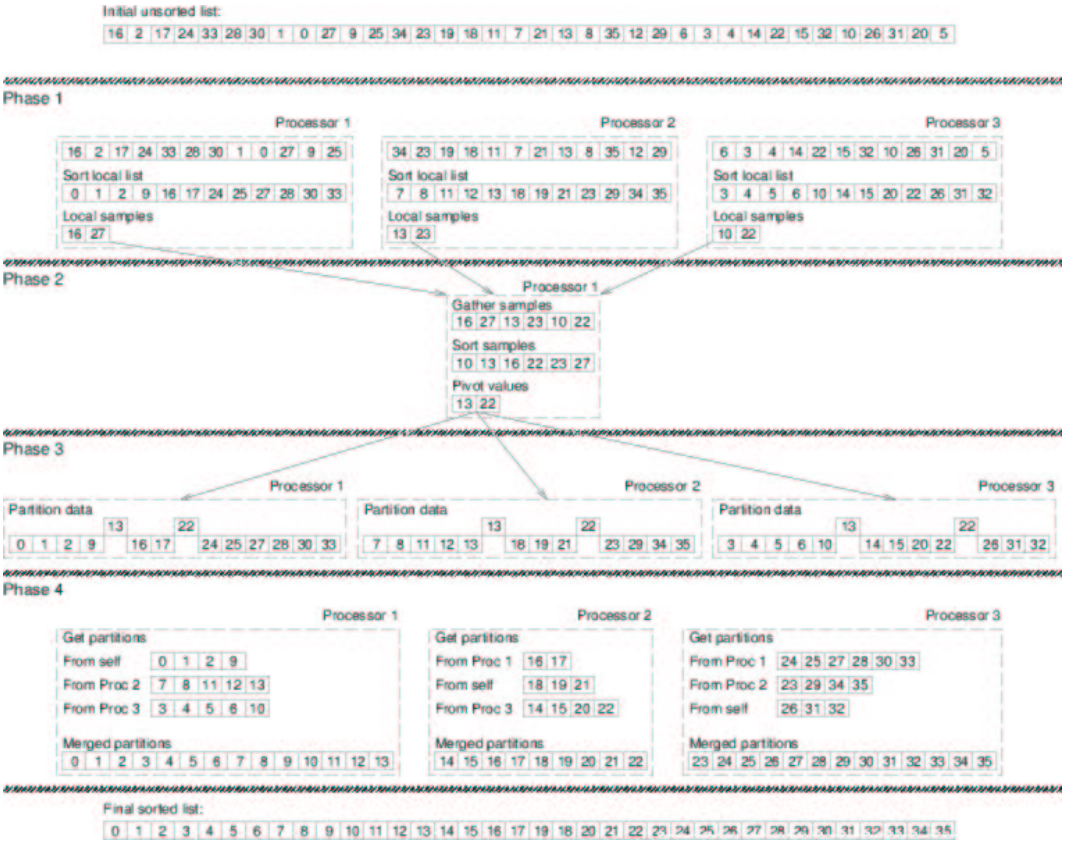


Figure 8.21: The four phases of the PSRS algorithm [25]

In a shared-memory environment, all processors share the same memory space. Data dispatch and exchange can be simply realized by passing pointers and offsets between threads. However, things are completely different in a distributed-memory environment, where message passing must be used. It can be anticipated that phase 3 will be a bottleneck as *(p-1)/p* (theoretically) of the data will be sent across the network. In our implementation, a non-blocking message passing scheme is adopted to reduce the communication overhead involved. A main thread spawns a set of children threads in which a number of RMI calls take place. By this means, we

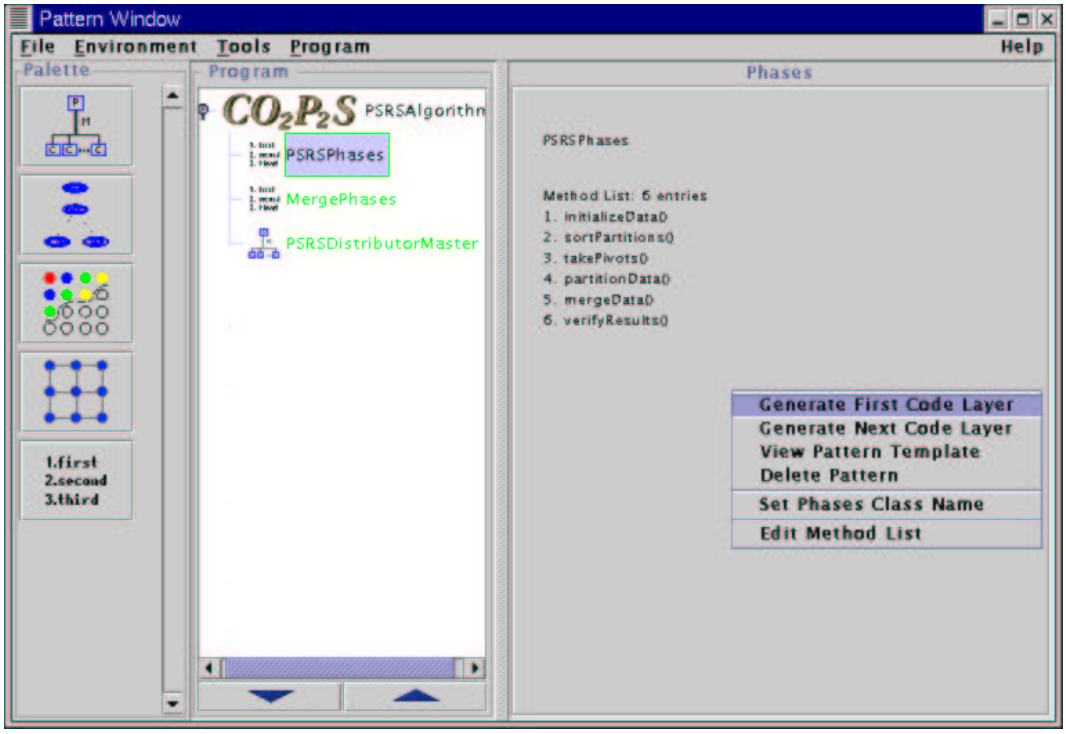


Figure 8.22: The PSRSPhases pattern template

can issue many messages once and check the results of them together later, just like what non-blocking MPI provides. All this is done in the framework code.

As the PSRS algorithm consists of four distinct stages, and the second and fourth stages can use data-parallelism. a combination of the Phases pattern and the DM_Distributor pattern was chosen to implement the PSRS algorithm. The implementation uses two instances of the Phases pattern template and one instance of the DM_Distributor pattern template.

The first Phases template instance is called `PSRSPhases` [23], which contains the four PSRS phases listed above plus two additional phases (Figure 8.22). One of these additional phases initializes the data and the other verifies the final result.

The fourth PSRS phase contains two sub-phases, which are modeled by the second Phases pattern— `MergePhases` [23]. These two sub-phases are the following:

- Processors merge their sub-parts in parallel;
- A final merge which concatenates all the results of last phase into a single array.

A single instance of the DM_Distributor template is used in the application. Two instances of the Phases pattern share one `PSRSDistributor` framework which works both in the sort phase and the merge phase. This approach saves the effort of redistributing the data through the slaves which would occur if two instances of the `PSRSDistributor` pattern were used.

Figures 8.22, 8.23 and 8.24 illustrate these three patterns with instantiated parameters. Table 8.4 gives some performance results (in milliseconds). The program

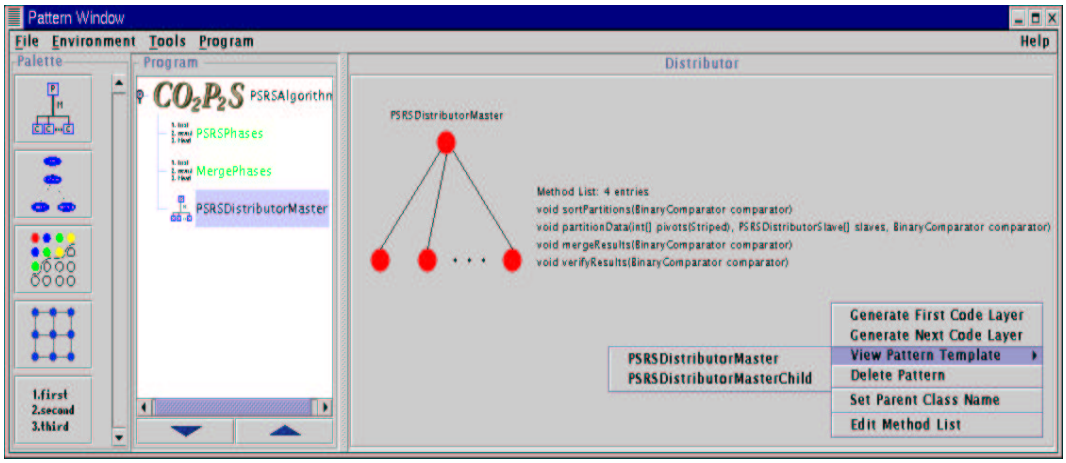


Figure 8.23: The PSRSDistributor pattern template

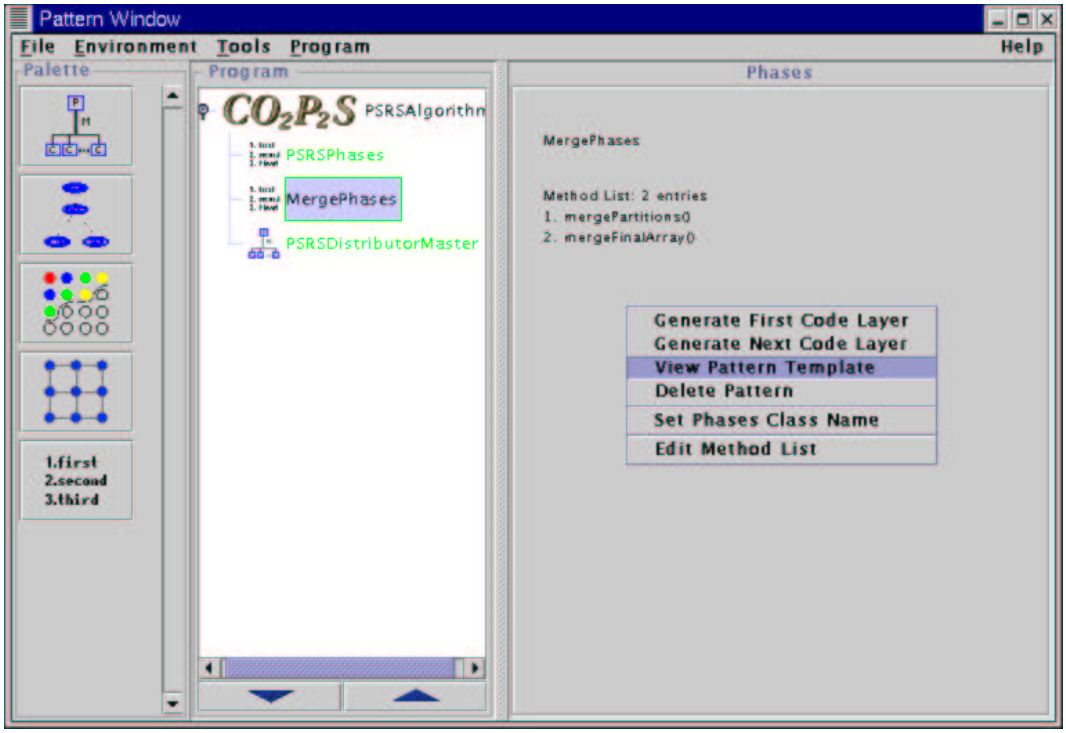


Figure 8.24: The MergePhases pattern template

was run using an int array with 4 different sizes and with randomly generated values. The runtime and hardware configuration is the same as in Section 8.2.9.

8.5 The DM_Wavefront pattern

8.5.1 Intent

Similar to the DM_Mesh pattern, the Wavefront pattern supports computations over a two-dimensional rectangular data structure [3]. However, the Wavefront pattern

Size	Sequential	2 processes		4 processes		6 processes		8 processes	
		time	speed up	time	speed up	time	speed up	time	speed up
2M	730	743	0.98	649	1.12	690	1.06	881	0.83
4M	1580	1216	1.13	973	1.63	919	1.72	867	1.8
8M	3417	2214	1.55	1515	2.26	959	3.60	809	4.22
12M	5340	3633	1.47	1871	2.90	1494	3.60	1170	4.56

Table 8.4: The PSRS application performance (in seconds)

provides more complex computation semantics. The computing of each element depends on the results of the computations of a series of prerequisite elements. This dependency relationship among elements results in data flowing through an acyclic graph.

8.5.2 Motivation

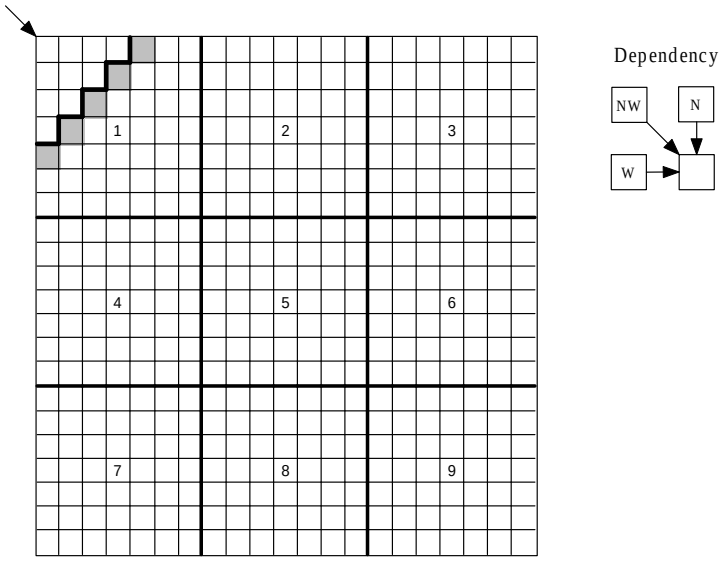


Figure 8.25: The Wavefront pattern [3]

The Wavefront pattern models the computation over regular two-dimensional data structures with complex dependency strategies. Each element can depend on one or more of its immediate or non-adjacent neighbours. Rules are enforced to avoid cyclic dependencies among elements so that all elements can be processed in a specific order resembling a flow moving from one region of the surface to another.

In Figure 8.25 [3], a 21x21 matrix represents the data be processed in the whole computation. The computation is based on the dependency graph depicted in Figure 8.25, in which each interior element is dependent on its north, northwest and west neighbours. Boundary nodes and corners are treated slightly differently as the topology is not toroidal in any direction. For instance, the nodes on the top edge only depend on their west neighbours; the nodes at the left edge depend only on

their north neighbours. Unlike the mesh computation, each element is only computed once.

In Figure 8.25, the upper-left corner is the only one that does not depend on any other node. Thus it is the start point of the computation. The black stair-like thick line denotes the current wavefront in the computation. Nodes above the line have been processed; nodes directly underneath (in shaded color) are ready to be processed as all of their prerequisites have been satisfied. All the other elements are still waiting because the nodes they depend on have not been processed yet.

The nodes that are ready will be generated as the wavefront moves forward; each ready node can be computed based on itself and all of its prerequisites. By assigning these nodes to different processors, the wavefront computation can be parallelized. To increase the granularity of the parallel processing, the matrix can be divided into blocks, each of which is assigned to a processor. Figure 8.25 shows 9 blocks, each containing 7x7 elements. The dependency graph described in the figure still applies to the blocks, but at a larger-scale. For example, block 5 depends on blocks 1, 2 and 4. At the beginning, only one processor will be assigned a task, block 1, for processing. As the computation continues, the parallelism grows as more and more blocks are added to the work list.

However, the evaluation of the boundary nodes in each block needs information from the neighbouring blocks. The boundary exchange is treated in the same way as the DM_Mesh pattern. Each block will distribute its boundaries to its dependents once it is finished.

The distributed-memory Wavefront design pattern template encapsulates parallel design complexities such as the matrix surface division and distribution, the communication scheme for boundary exchange, the synchronization support for mutually exclusive access to the work list, etc. Different dependency strategies are also recorded in the form of a template parameter. Provided with a combination of user-specified parameters, the DM_Wavefront template will generate a framework with all the code necessary to define the overall structure and the parallelism for the application.

8.5.3 Structure

Figure 8.26 shows the class structure of the Wavefront pattern template. Two main components are the Wavefront controller class and the Wavefront process class. An instance of the controller class divides a wavefront surface to initialize a central work list. A series of Wavefront processes will mutually exclusively access the work list to obtain mutually exclusive blocks to compute. Two instances of the Strategy pattern are used in DM_Wavefront. One is called `OperateStrategy`, which is used by each process to compute its acquired block based on different dependencies. The other is called `BoundaryStrategy`, which helps the Wavefront controller determine how to send the boundaries of a finished block to its dependents. `BoundaryStrategy` is used if the computation of each node only needs access to its immediate neighbors. If the computation also needs non-adjacent nodes, the controller will provide another series of methods to fetch the required nodes. This condition only exists in the distributed-memory environment and can be captured by a pattern template parameter described later.

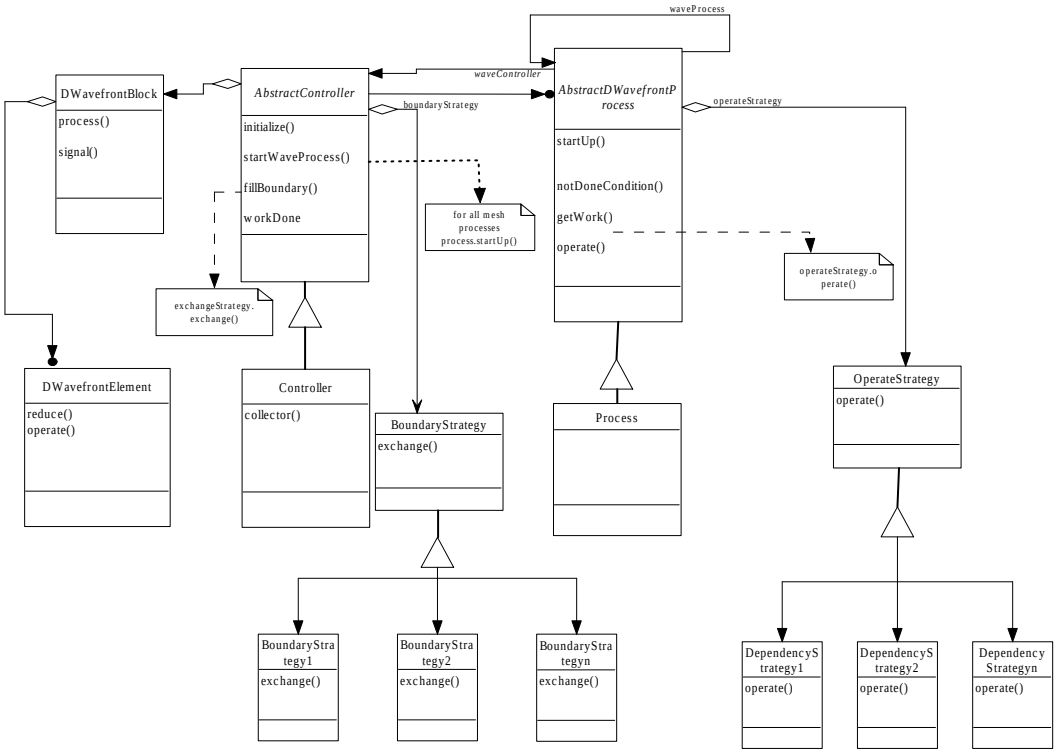


Figure 8.26: The DM_Wavefront pattern structure

8.5.4 Pseudo code

The pseudo code for a DM_Wavefront application is as follows:

1. The Wavefront controller will do the following:

   (a) create an instance of the DM_Wavefront class, divide the wavefront surface into a number of blocks (according to the number of available processors),

   (b) initialize the blocks and create dependencies between them,

   (c) and put the block which is the starting point of the wavefront computation into a worklist.

2. After the initialization work has been done, the controller will start the execution of all the remote slave processes. For each slave, the following steps are done in a loop:

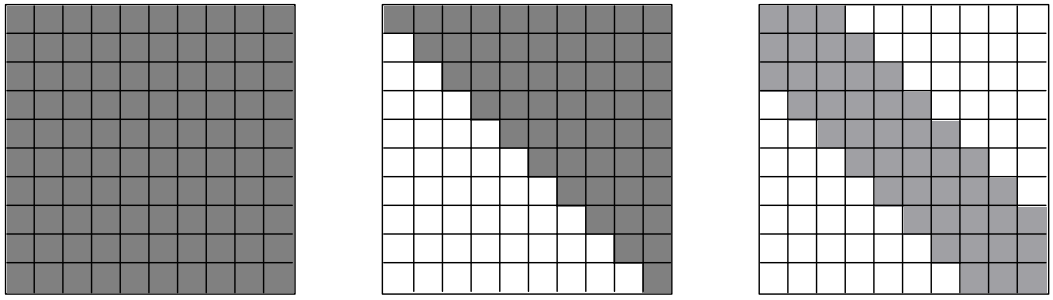
```
While (computation not done)
          Get one block from the controller's worklist,
          Process this block.
          Once the block is finished, notify its dependents
             and send back its boundaries to the controller
             to fill the ghost boundaries of the dependents.
Endwhile
```

8.5.5 Parameters

The DM_Wavefront template parameters can be classified into two categories: design parameters and performance parameters. The design parameters have an impact on the overall parallel structure of the resulting framework for the target problem. The performance parameters are only concerned with the efficiency of the generated framework code, not the structural designs.

The DM_Wavefront pattern template has three design parameters:

- The **name** for the Wavefront element class instances which represent nodes in the Wavefront surface.
- The **shape** of the element matrix (see Figure 8.27). The value of it is defined as follows:
  - Full Matrix in which all elements of a rectangular matrix are populated with data to be computed. It is the default value.
  - Triangular matrix in which only half of the matrix is processed.
  - Banded matrix in which only several element bands parallel to the diagonal are computed.



(a) Full Matrix (b) Triangular Matrix (c) Banded Matrix

Figure 8.27: Three matrix shapes supported in DM_Wavefront

- The **dependency set** for all nodes on the surface. A graphical user interface (Figure 8.29) is provided to the user to specify the dependency set. Not all combinations of the eight directions are legal dependency sets. Combinations that include opposite directions are illegal since they may create cyclic dependency graphs, resulting in deadlock during the computation.

The three performance parameters are the following:

- The **notification** method used to inform wavefront nodes whose dependency constraint has been satisfied. This parameter has two values:
  - the **Push** notification scheme which lets an element notify its dependents once it is done;
  - the **Pull** notification scheme which has all the dependents actively test their prerequisites for completion.

- The **type** of Wavefront elements. The user can choose to specify the type of the element. If the element type is primitive, a specific primitive type such as int or char can be chosen to generate static methods for computing each element. If the element type is the object type, instance methods are generated to process instances of the element class. Executing static methods with primitive data can be more efficient than invoking instance methods on objects.

- The **Neighbours only** conditions defines whether each element only accesses its immediate neighbourhood or not. In a shared-memory environment, this parameter will not make much difference since the whole surface is accessible to every thread; in a distributed-memory environment, accessing non-immediate neighbourhoods needs more explicit messages to be passed between machines and may incur more overhead than accessing immediate neighbours

8.5.6 Use of the pattern template

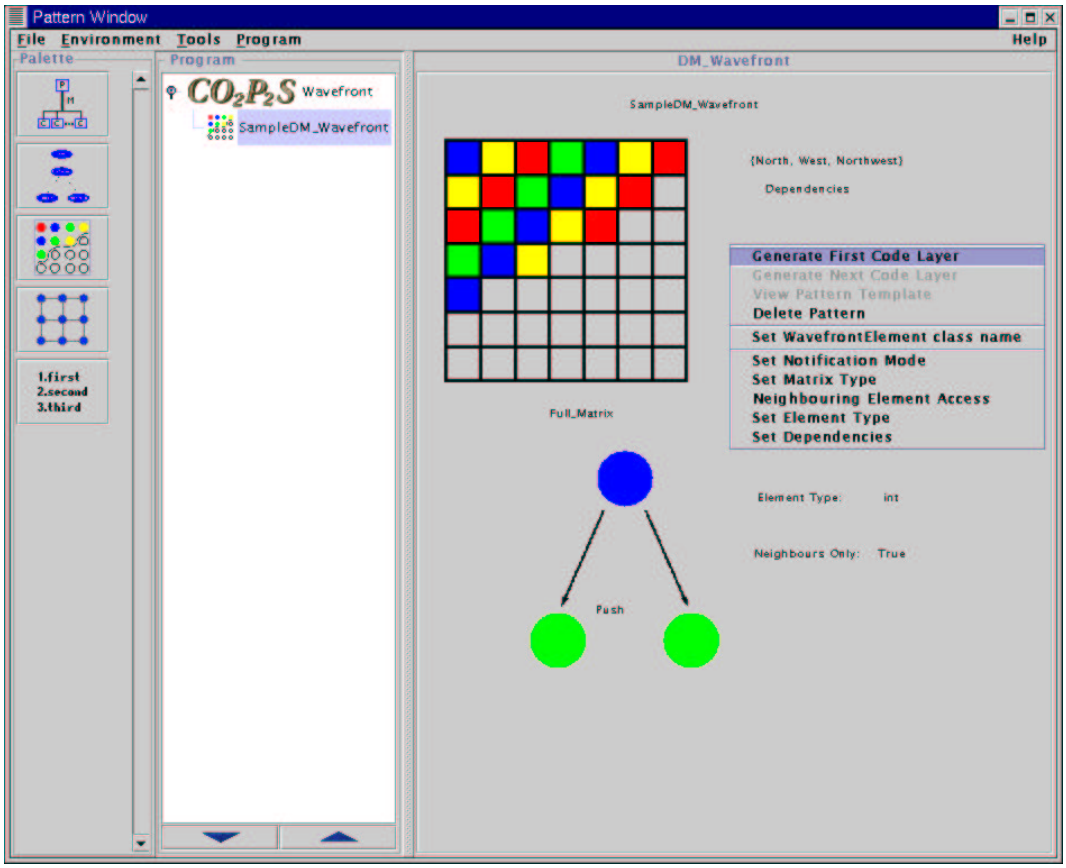


Figure 8.28: The DM_Wavefront template GUI

The DM_Wavefront template is shown in Figure 8.28. The pattern pane titled with "DM_Wavefront" depicts the diagram of the DM_Wavefront pattern and all of its instantiated parameters for a sample application. In this case, the Wavefront element class name is SampleDM_Wavefront. The matrix shape is full matrix. The

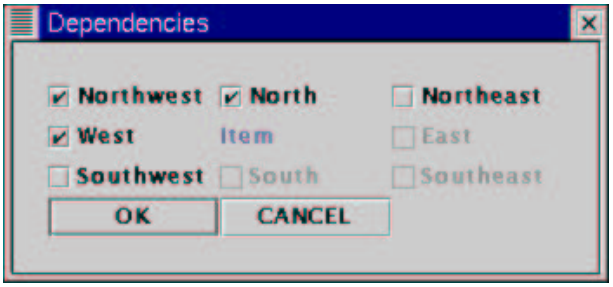


Figure 8.29: The dependency dialog

dependency set is North, Northwest and West, as depicted in the right corner of the window. The notification method is Push and the neighbours-only parameter is true. Pop-up windows in the figure provide options to set different parameters. For example, Figure 8.29 shows a dialog used to set the Dependency parameter.

After the instantiation of all template parameters, the user can choose the code generation option to generate a new framework. This framework provides a series of hook methods (Figure 8.30), which define the computation for each element based on its prerequisites. The user must also provide sequential code to create a concrete application. As depicted in Figure 8.30, a set of hyperlinks are provided to direct the programmer to different dialogs to input the sequential code for different hook methods (Figure 8.31).

8.5.7 Applications

The Wavefront pattern has applications in many different domains. For example, in bioinformatics it can be used for genetic sequence alignment. In the scientific computing area, it can help solve the LU-decomposition of matrices and the matrix product chain problem.

8.5.8 Example application

Sequence alignment [10] is one of the fundamental techniques used in computational biology research. Different genetic sequences can be aligned using this algorithm to gather evidence of similar functions or common biological origins. DNA or protein sequences are converted to arrays of letters and aligned. The alignment process will compare these sequences with cost functions. The purpose of such functions is to find an optimal comparison score so as to maximize the similarities between the sequences.

The sequence alignment problem can be solved using a dynamic programming matrix, which can be modeled by the Wavefront pattern. In essence, aligning two sequences with lengths of m and n reduces to finding a maximum cost path through a matrix of size (m+1)x(n+1). An extra row and column are added to represent initial scores of the matrix to start the computation. The computation starts from the top left corner and proceeds to the bottom right corner. The resulting paths across the matrix represent different combinations of the possible operations: letters are matched, mismatched and matched with gaps inserted into the original sequences.

The instantiated design template parameters for this application are:

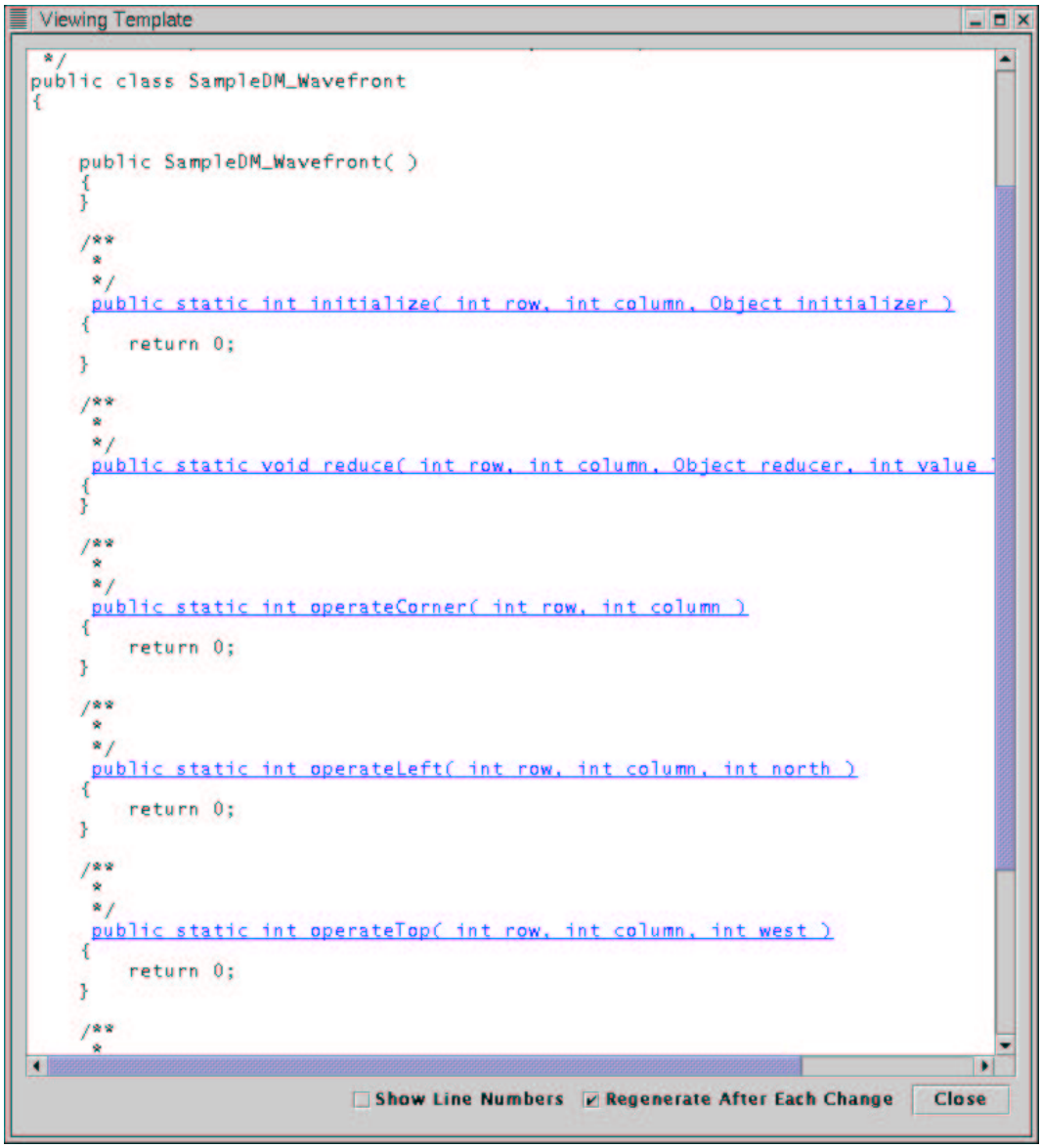


Figure 8.30: The code template with all hook methods

- The Wavefront element class name is SAElment
- the matrix shape is full matrix, and
- the dependency set is North, Northwest and West.

The performance parameters are the following:

- the notification method is Push,
- the type of wavefront elements is int, and
- instances of SAElement only needs to access its immediate neighbours.

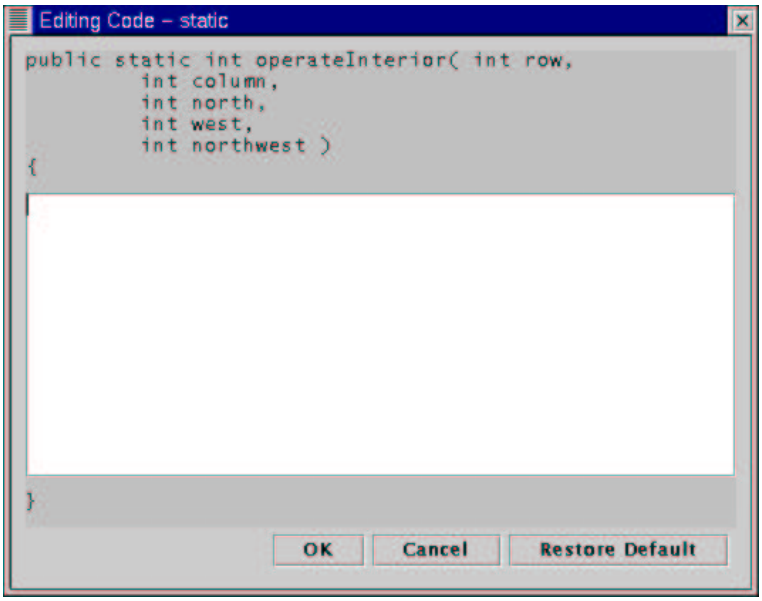


Figure 8.31: The hook method instantiation dialog

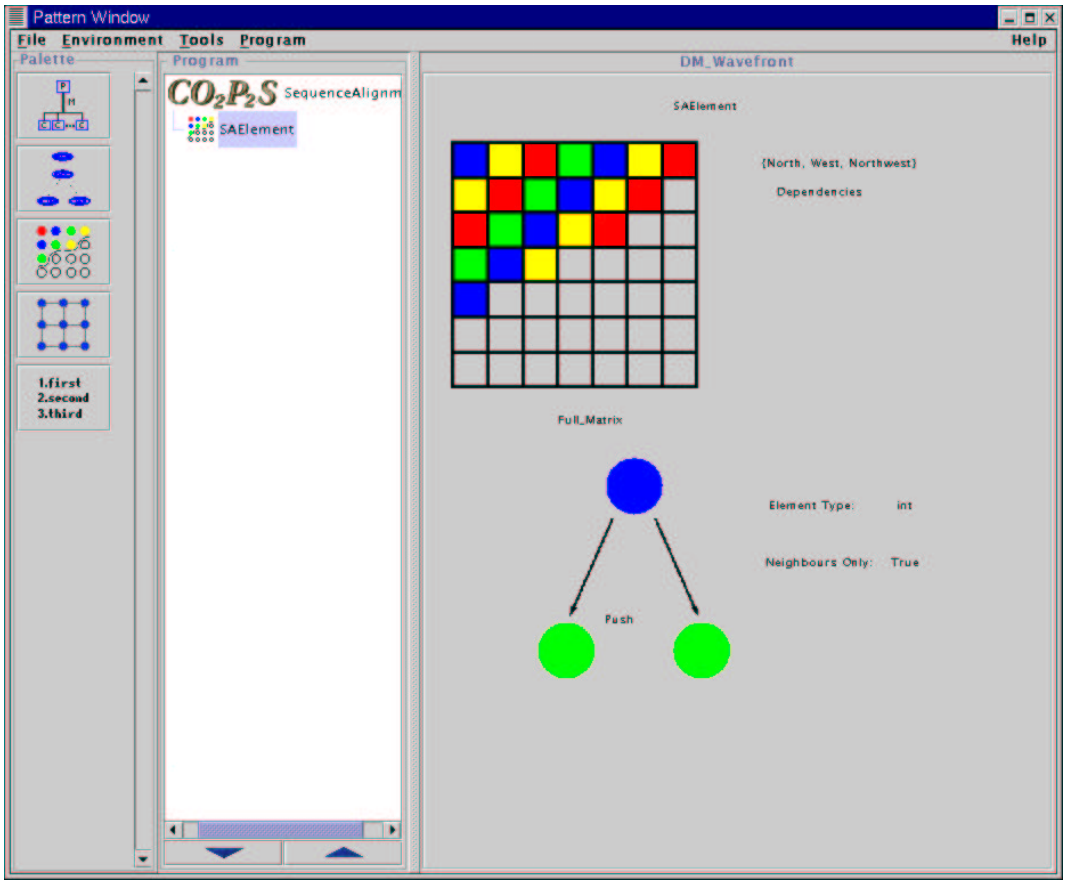


Figure 8.32: The sequence alignment application

Figure 8.32 is a screen shot of the CO_2P_3S GUI showing the Wavefront template instantiated with these six parameter values.

Six hook methods are provided by the Wavefront framework after pattern template parameters are instantiated. In the sequence alignment application, they are implemented as follows.

- `operateCorner()`: it returns 0 to set the top left corner.
- `operateLeft()`: it calculates the elements along the left edge based on their north neighbours.
- `operateTop()`: it computes elements along the top edge based on their west neighbours.
- `operateInterior()`: this method is the core of the sequence alignment. It computes the element value based on its adjacent neighbour elements (north, west and northwest) along with a cost function.
- `initialize()`: in this example application, this method does nothing since the matrix elements need no initialization.
- `reduce()`: this method is used to save the results of the computation. In this application, the method will return the value at the bottom right corner, which is the optimal comparison score accumulated during the computation.

Table 8.5 lists the performance (in milliseconds) of the sequence alignment application using the Wavefront pattern. The speedup rises slowly as more processors are added. Although these speed-up numbers are not great, they still indicate some speed-up in exchange for quick parallelization.

Size	Sequential	2 processes		4 processes		6 processes		8 processes	
		time	speed up	time	speed up	time	speed up	time	speed up
10000x 10000	6386	7321	0.87	3733	1.71	2633	2.43	2115	3.02

Table 8.5: The sequence alignment application performance (in milliseconds)

8.6 Conclusions

This chapter introduced four distributed design pattern templates—DM_Mesh, Phases, DM_Distributor and DM_Wavefront. Programmers can use these pattern templates to design distributed applications in the DCO_2P_3S environment. From the descriptions of these pattern templates and some real applications, we can see that these patterns are easy to use and that they generate parallel designs with reasonable performance.

Chapter 9

Summary and Conclusions

This dissertation describes a research project that supports parallel programming in a distributed-memory environment. It is an important extension of the existing CO_2P_3S environment which is restricted to shared-memory environment. My extended CO_2P_3S, called Distributed CO_2P_3S (DCO_2P_3S), minimizes the user involvement in the intricacy of distributed, parallel programming, while providing reasonable performance gains.

9.1 Contributions of this research

This research made a number of contributions to the CO_2P_3S project. First of all, a full-fledged distributed-memory runtime environment (DCO_2P_3S) has been built. Jini, RMI and JDK-serialization are three main technologies used in the implementation for the infrastructure and communication support. Other important components of DCO_2P_3S include the process manager, synchronization controller and performance monitor, all of which are implemented using techniques such as Jini, JNI and JavaSpaces. Secondly, I developed a customized version of JDK-serialization— CO_2P_3S-serialization. The existing RMI is slightly modified to make use of the new component. The new routine applies a more efficient wire format and more aggressive data compression schemes. Using CO_2P_3S-serialization, the RMI system has less overhead in the argument marshalling phase and the transportation process. The pattern designer can enjoy the performance improvement for free and use RMI the same as before.

9.2 Future work

One on-going research project is to devise a meta-level pattern language to facilitate the design of new design pattern templates. To support the design of distributed pattern templates, I should encapsulate the details related to Jini (such as the service discovery and service lease renewal) into the pattern language using macros or explicit language constructs.

Another possible avenue of the DCO_2P_3S research is to extend the distributed environment to an even larger scale. Grid computing addresses the ever-increasing need for computing power to solve scientific applications. The term "the Grid"

refers to an emerging infrastructure technology that provides security control, resource accessing, information sharing, process coordination and other services to build computing systems with resources that are geographically distributed [40]. While it is similar to Jini in some sense, the Grid technology focuses more on high performance scientific computing. The Globus Toolkit (www.globus.org) is the base API to build computational grids. It would be wonderful if the programming details of using the Globus API could be captured in DCO_2P_3S design pattern templates. Thus the computational science community could benefit from our DCO_2P_3S system.

Finally, existing pattern templates are dedicated to either distributed-memory or shared-memory, but not both. In future research, we can add an additional environment parameter in each parallel design pattern template. This parameter can direct the template to generate shared-memory or distributed-memory framework code. A potential advantage of this approach is that different kinds of parallelism can be more easily specified.

Bibliography

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[3] J. Anvik. Asserting the Utility of CO2P3S using the Cowichan Problems. Master's thesis, Department of Computing Science, University of Alberta, September 2002.

[4] B. Appleton. Patterns and Software: Essential Concepts and Terminology, 1997. http://www.enteract.com/ bradapp/docs/patterns-intro.html.

[5] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[6] Kent Beck and Ralph Johnson. Patterns Generate Architectures. *In Proceedings of the 8th European Conference on Object-Oriented Programming*, 821:139–149, 1994.

[7] S. Bromling. Meta-programming with Parallel Design Patterns. Master's thesis, Department of Computing Science, University of Alberta, 2001.

[8] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, October 1989.

[9] J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. Network Multicomputing Using Recoverable Distributed Shared Memory. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 519–527, 1993.

[10] K. Charter, J. Schaeffer, and D. Szafron. Sequence Alignment Using FastLSA. *International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 239–245, 2000.

[11] Amnon H. Eden, Yoram Hirshfeld, and Amiram Yehudai. Towards a Mathematical Foundation for Design Patterns. Technical Report Technical report 1999-004, 1999. http://www.math.tau.ac.il/eden/bibliography.

[12] Gabriel Antoniu et. al. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[13] Raphael A. Finkel. *ADVANCED PROGRAMMING LANGUAGE DESIGN*. Addison-Wesley, 1st edition, 1995.

[14] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides and Patsy S. Yu. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[17] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool Support for Object-Oriented Patterns. *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 472–495, 1997. volume 1241 of Lecture Notes in Computer Science.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Mass., 1994.

[19] Object Management Group. CORBA (Common Object Request Broker Architecture). http://www.corba.org/.

[20] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *Programming Languages and Systems*, 23(6):747–775, 2001.

[21] libGTop. Library that Provides System Information, 2001. http://www.gnu.org/directory/Software_Libraries/LibGTop.html.

[22] F. Lau M. Ma, C. Wang and Z. Xu. JESSICA: Java-Enabled Single System Image Computing Architecture. In *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 2781–2787, June 1999.

[23] S. MacDonald. From Patterns to Frameworks to Parallel Programs. Ph.D. thesis, Department of Computing Science, University of Alberta, 2002.

[24] S. MacDonald. Two-Dimensional Mesh Design Pattern Template for CO_2P_3S. Technical report. http://www.cs.ualberta.ca/ stevem/papers/COPS_meshDocs.ps.gz.

[25] S. MacDonald, D. Szafron, and J. Schaeffer. Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks. *Proceedings of the 5th USENIX Conference on Object-Oriented Tools and Systems (COOTS'99), San Diego, CA*, May 1999.

[26] Microsoft. DCOM (Distributed Component Object Model). http://www.microsoft.com/com/wpaper/default.asp#DCOMpapers.

[27] Sun Microsystems. Java Class Loading Mechanism, 1997. http://java.sun.com/docs/books/tutorial/ext/basics/load.html.

[28] Sun Microsystems. Java Object Serialization Specification, 1997. http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html.

[29] Sun Microsystems. Java Remote Method Invocation Specification, JDK 1.1 FCS, 1997. http://java.sun.com/products/jdk/rmi_ed.

[30] Sun Microsystems. JNI Specification, 2000. http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html.

[31] Sun Microsystems. Jini Architectural Overview. 2001. http://wwws.sun.com/software/jini/whitepapers/architecture.pdf.

[32] Sun Microsystems. Jini Network Technology An Executive Overview. 2001. http://wwws.sun.com/software/jini/whitepapers/jini-execoverview.pdf.

[33] Jan Newmarch. *A Programmer's Guide to Jini Technology.* Apress, November 2000.

[34] M. Philippsen, B. Haumacher, and C. Nester. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[35] R. Sansom, H. Kung, and S. Schlick. Network-based Multicomputers: an Emerging Parallel Architecture. In *Supercomputing'91*, pages 664–673, 1991.

[36] Hanmao Shi and Jonathan Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[37] GigaSpaces Technologies. GigaSpaces Cluster White Paper, 2002. http://www.gigaspaces.com/download/GSClusterWhitePaper.pdf.

[38] ModelMaker Tools. Design Patterns in ModelMaker. http://www.modelmakertools.com/mm_design_patterns.htm.

[39] Bill Venners. *Inside the Java 2 Virtual Machine.* McGraw Hill, 2nd edition, 1999.

[40] Michael Philippsen Vladimir Getov, Gregor von Laszewski and Ian T.Foster. Multiparadigm communications in Java for grid computing. *Communications of the ACM*, 44(10):118–125, 2001.

[41] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 49(7), July 1999.

[42] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.

[43] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 1st edition, 1999.

[44] Weimin Yu and Alan L. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.