INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Corba Version of Concordia Parallel Programming Environment (CPPE)

Shiyi Wu

A Major Report

In

The Department of Computer Science

For the Degree of Master
Concordia University
Montreal, Quebec, Canada

September, 2000

© Shiyi Wu, 2000

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59356-8

Canada

Abstract

Corba Version of Concordia Parallel Programming Environment (CPPE)

Shiyi Wu

In this major report, we present the design and implementation of the Corba Version of the Concordia Parallel Programming Environment (CPPE), a client-server-based simulator for parallel programming environment. The purpose of the client/server version is to provide services and a rich set of graphical user interfaces for remote clients.

The challenge in the design of client/server version is to reuse the existing C/C++ code. To solve the problem, we use Corba's delegation technique to wrap the existing C/C++ application into a standard Corba object, and then we create an IDL interface for it. From the IDL interface specification, we create a Java standalone application wrapped in Corba objects that runs at client site and communicates with the server.

We design and implement client site GUIs for the application in Java/Swing.

This Corba Version of CPPE is very portable since the client site application can be installed and run in any platform. The server supports multiple users.

Acknowledgements

I would like to express my cordial gratitude to Dr. Lixin Tao. By taking Dr. Tao's courses, I learned more computer concepts and skills that were the newest and broadly used in industry . So I hoped that Dr. Tao could supervise me for my major report, and I was a lucky student. During the study of major report, Dr. Tao gave me the most practice support. He always gave a quick answer to my question in detail. They were absolute helpful. When I needed to see him to discuss, he scheduled an appointment as early as possible. His advice, encouragement and kindness gave me a lot of help.

I am grateful to the professors and administrative stuffs in the department of Computer Science, especially Dr. Lixin Tao, who gave the lectures in "Computer system Design" and "Distributing Process With Object Technology", Dr. H.F. Li, who taught a course in "Computer Architecture", Dr. Peter Grogono, who gave the lecture in "Advanced Computer Graphics", and Ms. Halina Monkiewicz, her friendliness and administrative support made my student's life much easier and pleasant.

I also have a very supportive family that backs me up. My wife, Cuilan Li, supported me with her love, sharing and inspiration throughout my study. I am also grateful to my parents for their encouragement.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

1.1 Motivations

Parallel computers can be classified into two categories: shared-memory multi-processor and message-passing multicomputer. With the advantage of hardware technology, high-speed networks and efficient routing techniques have made message-passing multicomputer to be the developing trend for parallel computing, because it is more scalable due to the distributed nature of local memories.

Because of the limitations of the real machines, many studies of parallel algorithms have been conducted on an analytical basis – the analytical modeling. Performance aspects of a parallel program under specific conditions may be estimated using mathematical formulation. However, this approach is suitable only for simple applications and small computer systems. Parallel computer systems and their applications are sufficiently complex to make analytical modeling very difficult and inaccurate.

Because of the difficulty of using real parallel computers and analytical modeling, several research groups have studied the simulation approach. CPPE is one of the projects of that kind. CPPE provides a parallel system simulator called CPSS (Concordia Parallel System

Simulator) which runs sequential software on a uniprocessor to emulate program execution on a real parallel computer. Its objective is to provide a parallel programming environment that allows users to study impacts of system and software factors on program performance, and locate performance bottlenecks in the program.

1.2 Problem

The Concordia CPPE team has developed the project of simulation system of Concordia Parallel Programming Environment (CPPE). It is implemented in language C and language C++ with GUIs of MFC and Motif. It runs as standalone application.

We plan to implement client/server version of CPPE. It can run locally and run through Internet remotely. We use Corba and Java to implement client/server communication and graphic user interface.

1.3 Approach

Java client objects communicate with C/C++ server object through Internet by using Corba technique.

We design and implement Java/Swing GUIs for the application.

We use Corba delegation technique to wrap the existing C/C++ application into a standard Corba object that runs at a server. Java standalone application can be wrapped in Corba objects on client site and communicates with the server.

1.4 Solution Description

Basically, CPPE will be wrapped in a Corba (server) object. An IDL interface will be written for it. This object should be run first on a web server before remote users can use it. A standalone GUI will be implemented in Java. It will communicate with CPPE Corba objects.

From the IDL specification, we use idl2c++ to generate C++ implementation skeleton, and use delegation approach to the methods in this skeleton to call real existing implementation in C/C++. This will wrap up CPPE as standard Corba objects.

Then we compile the same IDL file with idl2java and generate stub Java codes for CPPE. These codes will be used in Java to communicate with the Corba server CPPE.

Chapter 2

2 Survey of Techniques

2.1 Available Techniques

To solve the problem presented in section 1.2, we compared the following available techniques and finally chose CORBA to solve the problem.

2.1.1 Corba

Corba (Common Object Request Broker Architecture) is a set of standard definitions by which objects can interact with each other. The Object Management Group (OMG) -- that includes over 800 companies, created Corba. Corba defines a standard for a layer of middleware called the ORB (Object Request Broker). The way that components interact with other components is specified by IDL (Interface Definition Language). This allows client-server computing where the clients don't need to have any knowledge of the specific operation of the component they are interacting with. For example, the client doesn't need to know the language in which the component was written; it only needs to know the IDL specification of how the component interacts.

2.1.2 Socket

Socket was introduced in 1981. Today, Socket is supported on virtually every operating system. A socket is a peer-to-peer communication endpoint -- it has a name and network address. From a programmer's perspective, a socket hides the details of the network. Typically, socket come in three flavors: stream, datagram, and raw. You specify the type of socket at creation time. In theory, you can extend the socket interface and you can define new types to provide additional services.

2.1.3 HTTP/CGI

HTTP/CGI is currently the predominant model for creating 3-tier client/server solutions over the Internet. The Hypertext Transfer Protocol (HTTP) provides simple RPC-like semantics on top of sockets. You can use it very effectively to access resources that locate in URL space. After HTTP locates a program identified by URL, on the receiving end, the typical Web server only knows how to handle HTML documents. When it receives a request for a program, it invokes the resource named in the URL and tells it to take care of the request. The server passes the method request and its parameters to the back-end program using a protocol called the Common Gateway Interface (CGI). You can write your CGI server programs in any language.

2.1.4 Servlet

A servlet is a Java program that a Web server loads to handle client request. Unlike a CGI application, the servlet code stays resident in memory when the request terminates. In addition, a servlet can connect to a database when it is initialized and then retain its connection across requests. A servlet can also chain a client request to another servlet. This is called servlet chaining. All these features make servlet an excellent workaround for many of CGI's limitations.

2.1.5 RMI

The Remote Method Invocation (RMI) -- now part of JDK -- was designed from ground to seamlessly support remote method invocations on objects across Java virtual machines. RMI directly integrates a distributed object model into the Java language. It makes the Object Request Broker (ORB) almost transparent to the client by adding some new implementation requirements on the server. RMI also extends Java's security net to include the dynamic downloading of stubs. An RMI object is a remote Java object whose methods can be invoked from another Java Virtual Machine, even across a network. You can invoke methods on the remote RMI object like you would on a local Java object.

2.1.6 DCOM

DCOM (Distributed Component Object Model) is a Microsoft-developed standard for allowing software components to interact with each other over a network.

Like Corba, DCOM separates the object interface from its implementation and requires that all interfaces be declares using an Interface Definition Language (IDL). Like Corba, DCOM provides both static and dynamic interfaces for method invocations.

Unlike Corba, DCOM does not support IDL-specified multiple inheritance. However, a DCOM component can support multiple interfaces and achieve reuse by encapsulating the interface of inner components and representing them to a client. So with DCOM, you can achieve object reuse via containment and aggregation instead of inheritance. A DCOM object is not an object in the object-oriented sense.

2.2 Comparing of the Available Techniques

Table 1 shows us the comparing of Corba and their Competitors. Four bold stars is the highest rating and one normal star is the worst rating, it means the function is not there. The table gives us a bird's-eye view of the comparisons. The data in table 1 come from reference [2].

Abstraction level: The higher the level of abstraction, the less work your application must do. For example, Sockets present the lowest level of abstraction; you must create your own conventions for passing the request names and their arguments.

Intergalactic scaling: Sockets via TCP/IP support intergalactic federations of networks. However this support is at a very low level of abstraction. In contrast, CORBA supports loose federations of ORBs.

Open standard: An intergalactic platform must be an open, level-playing field. No single vendor should be allowed to control the platform.

2.3 Why We Choose Corba

Corba extends the reach of applications across networks, languages, component boundaries, and operating systems. Also we can use Corba IDL to wrap existing code. Corba deals with network transparency, while Java deals with implementation transparency. So we chose Corba combined with Java for our application.

Feature	Corba/IIOP	DCOM	RMI/RMP	HTTP/CGI	Servlets	Sockets
Abstraction level	****	****	****	**	**	*
Seamless Java integration	****	****	****	**	**	**
OS platform support	****	**	****	****	****	****
All-Java implementation	****	*	****	**	****	****
Typed parameter support	****	****	****	*	*	*
Ease of configuration	***	***	***	***	***	***
Distributed method invocation	****	***	***	*	*	*
State across invocation	****	***	***	*	**	**
Dynamic discovery and metadata support	****	***	**	*	*	*
Dynamic invocations	****	****	*	*	*	*
Performance (speed)	***	***	***	*	*	****
Wire level security	****	****	***	***	***	***
Wire level transaction	****	***	*	*	*	*
Persistent object references	****	*	*	*	*	*
URL-based naming	****	**	**	****	****	***
Multilingual object invocations	****	****	*	***	*	****
Language neutral wire protocol	****	****	*	****	****	*
Intergalactic scaling	****	**	*	**	**	****
Open standard	****	**	**	****	**	****

Table 1: Comparing Corba and their competitors

Chapter 3

3 Corba Version of CPPE

3.1 Brief Description of CPPE Function

CPPE (Concordia Parallel Programming Environment) consists of two major modules: CPCC and CPSS (Figure 1)

(1) CPCC (Concordia Parallel C Compiler): The CPCC accepts parallel programs written in the CPC (Concordia Parallel C) language and generates virtual machine code (vCode) which will be the input to the CPSS.

(2) CPSS (Concordia Parallel System Simulator): The CPSS reads in the intermediate code produced by the CPCC, simulates execution of the application, yields programs outputs.

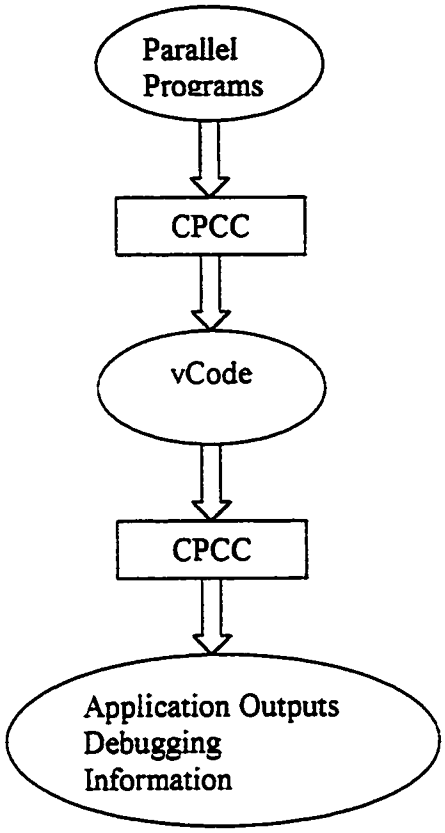


Figure 1: General structure of the CPPE

The core of the CPCC (Concordia Parallel C Compiler) is a compiler. After reading a parallel program written in CPC (Concordia Parallel C) language, the CPCC builds a complete abstract syntax tree to perform syntax and semantics analysis, and produces object code for a generic virtual machine. Such object code is called *vCode* in CPPE. The vCode instruction set is defined based on an analysis of common operations of parallel computer systems. To produce vCode, the compilation process makes use of the virtual architecture and does not call for the physical architecture. The advantage of this design is that the CPC parallel program does not need to be re-compiled every time the underlying target architecture is changed.

The vCode produced by the CPCC will be input to the CPSS. Other inputs to the CPSS are parameters and commands from the user. For example, the user can specify the physical topology on which the program will run and the virtual-to-physical topology mapping. The CPSS then executes the vCode, using the parameters and commands entered by the user. The outputs from the CPSS are the application outputs, performance statistics, and debugging information.

CPSS is a major component of the CPPE and is made of three main components:

(1) Code Execution Module (CEM): The CEM is the processing element of a simulated parallel system. It executes the vCode produced by the CPCC.

(2) Network Module: The roll of the network module is to allocate network resources to messages, route and deliver messages, and detect deadlock in the network.

(3) Performance Debugger: The performance debugger provides a set of software tools to facilitate program testing and debugging.

3.2 Client/Server Design

3.2.1 Corba Technology

Corba (Common Object Request Broker Architecture) is a middleware project from Object Management Group (OMG). Corba is designed to allow intelligent components to discover each other and interoperate on an object bus. Corba objects can live anywhere on a network. They are packaged as binary components that remote clients can access via method invocations. Clients do not need to know where the distributed object resides or what operating system it executes on or how the server object is implemented. What the client needs to know is the interface that its server object publishes. This interface serves as a binding contract between clients and servers. Corba IDL (Interface Definition Language) is purely declarative and it provides operating system and programming language independent interfaces to all the services and components that reside on a Corba bus. It allows client and server objects written in deferent languages to inter-operate. (see Figure 2)

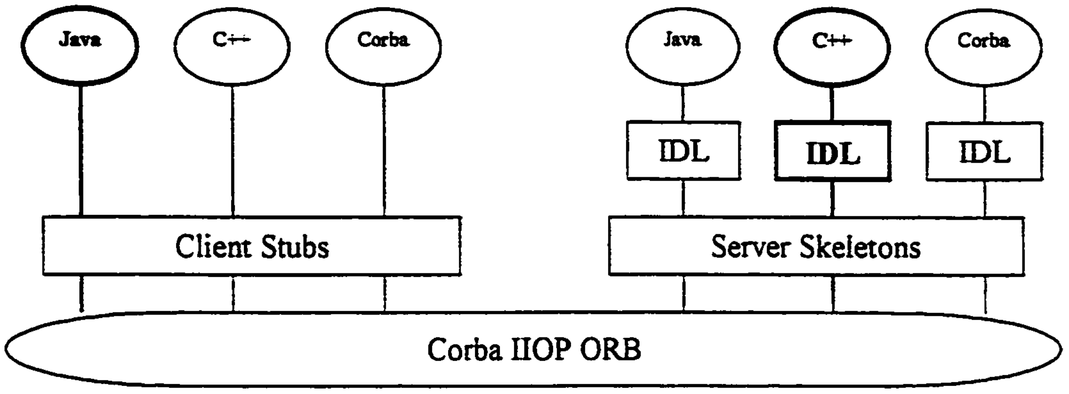


Figure 2: Corba IDL language bindings provide client/server interoperability

3.2.2 Corba Method Invocation

Figure 3 shows the steps to create server classes and provide interface stubs for them. The arrows represent the step order.

1. To Define server interface using the Interface Definition Language (IDL). The IDL is the tool by which objects tell clients what operations are available and how to innovate them. The IDL defines the types of objects, attributes, the methods and the method parameters.
2. To Run the IDL file through a language precompiler. VisiBroker and other vendors can provide the precompiler. The compiler generates three types of output files:
    - Client stubs for the IDL-defined methods. These stubs are invoked by a client program;
    - Server skeletons that call the methods on the server;
    - An example class for implementation.
3. To Add the servant implementation code. User must only provide the code that implements the server interface
4. To Compile the server code.
5. To Implement the client code.
6. To Compile the client code

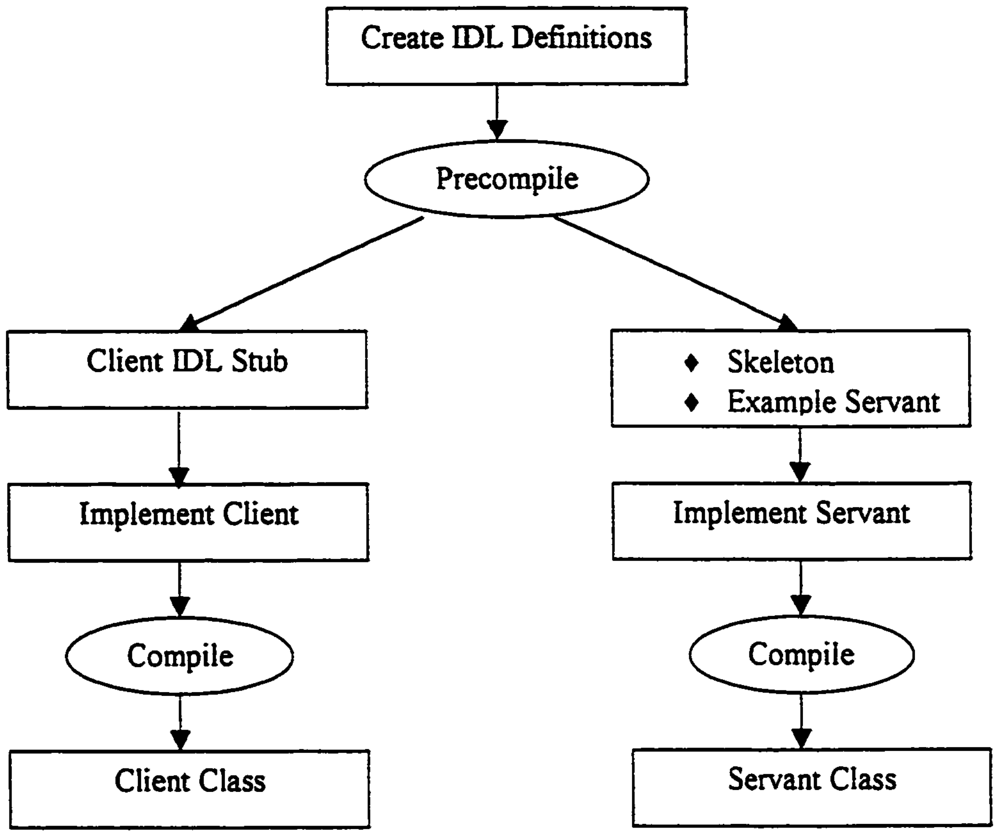


Figure 3 : Defining services: from IDL to interface stubs

3.2.3 Mapping Corba IDL to C++

When we have an IDL file, we can map it into something that C++ clients and severs can understand. So we will need an IDL-to-C++ compiler. We chose the VisiBroker idl2cpp compiler (or precompiler). It generates four C++ files from the IDL inputs (to see Figure 4). Corba supports two styles of programming: inheritance-based and delegation-based. We use delegation to integrate existing code because we want to reuse the existing CPPE codes and can not inherient from the IDL.

1) cppe_s.cpp is the server skeleton for the methods of CPPE class.
2) cppe_s.hh is a server header file that includes the class definitions for the server skeletons.

3) cppe_c.cpp contains a class called cppe that serves as a proxy on the client for the Cppe object.
4) cppe_c.hh is a client header file that includes declarations and class definitions for the stub implementation in cppe_c.cpp.

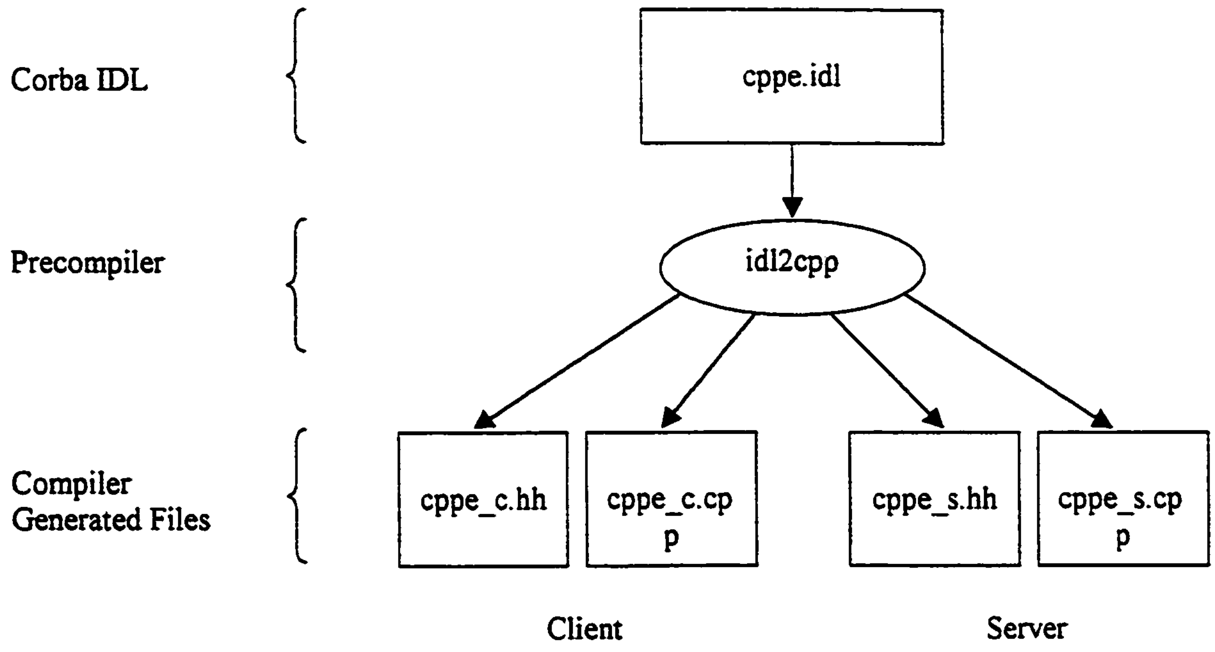


Figure 4 : The C++ files generated by the idl2cpp compiler

3.2.4 Mapping Corba IDL to Java

When we have an IDL file, we can map it into something that Java Clients and severs can understand. So we will need an IDL-to-Java compiler. We chose the VisiBroker idl2java compiler (or precompiler). It generates a Java package that contains five Java classes and one Java interface (see Figure 5)

1) _cppeImplBase is a Java class that implements the Corba server-side skeleton for CPPE.
2) _st_cppe is a Java class that implements the Corba client-side stub for CPPE.
3) cppeHelper is a Java class that provides useful helper function for CPPE clients.
4) cppeHolder is a Java class that holds a public instance member of type CPPE.
5) Cppe is a Java interface. It maps the CPPE interface to the corresponding Java interface.
6) _example_cppe is an example class for the CPPE object implementation.

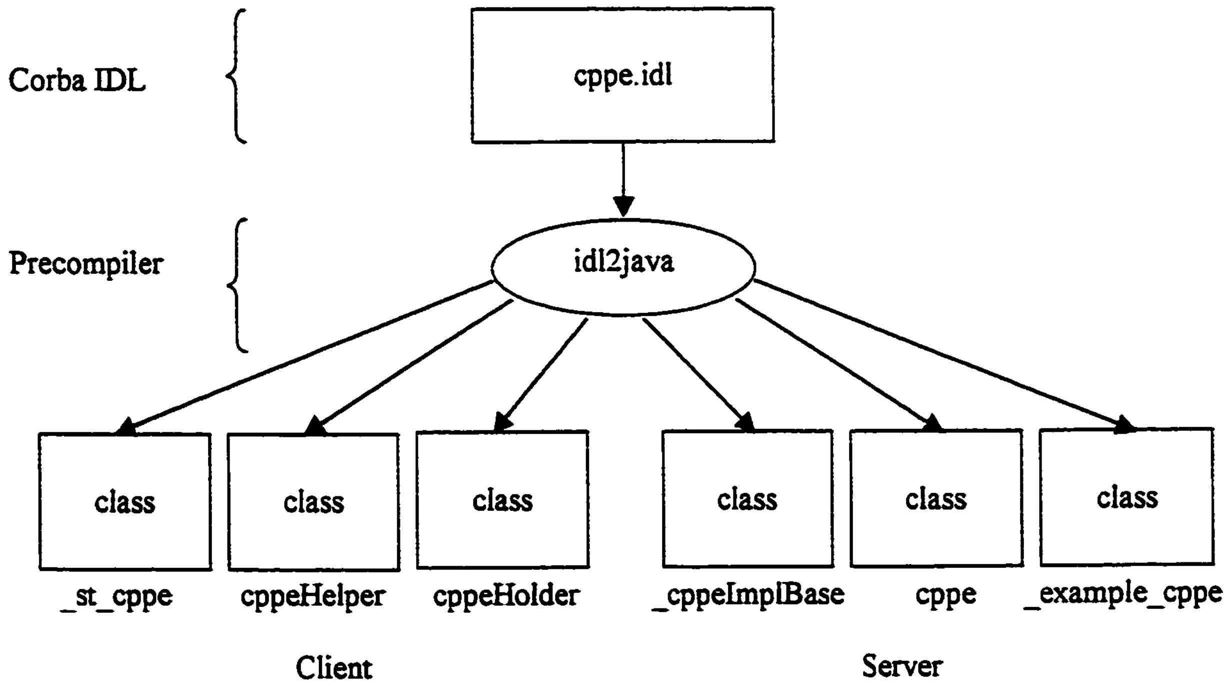


Figure 5 : The Java classes generated by the idl2java compiler

3.2.5 Mapping Corba IDL to C++ and Java

For our special application, we choose the combination of the mapping from Corba IDL to C++ for server and the mapping from Corba IDL to Java for client. The reasons are that we want to reuse the existing CPPE codes implemented in C/C++ and we want to provide the Java codes for client site. We choose the VisiBroker idl2java precompiler for the mapping from Corba IDL to Java and the VisiBroker idl2cpp precompiler for the mapping from Corba IDL to C++. They generate Java and C++ packages (see Figure 6)

For client side, we just keep client side files as the followings:

- _st_cppe
- CppeHelper
- CppeHolder.

and drop all server side files (_cppeImplBase, Cppe, _example_cppe).
Java files (cppeFrame.java and others) are the main client side codes that provide user interface for the application.

For server side, we just keep server side files as the followings:

- cppe_s.hh
- cppe_s.cpp

and drop all client side files (cppe_c.hh, cppe_c.cpp).
C++ files (server.cpp, cppeimp.h and cppeimp.cpp) are the main server side codes that provide function for client to call.

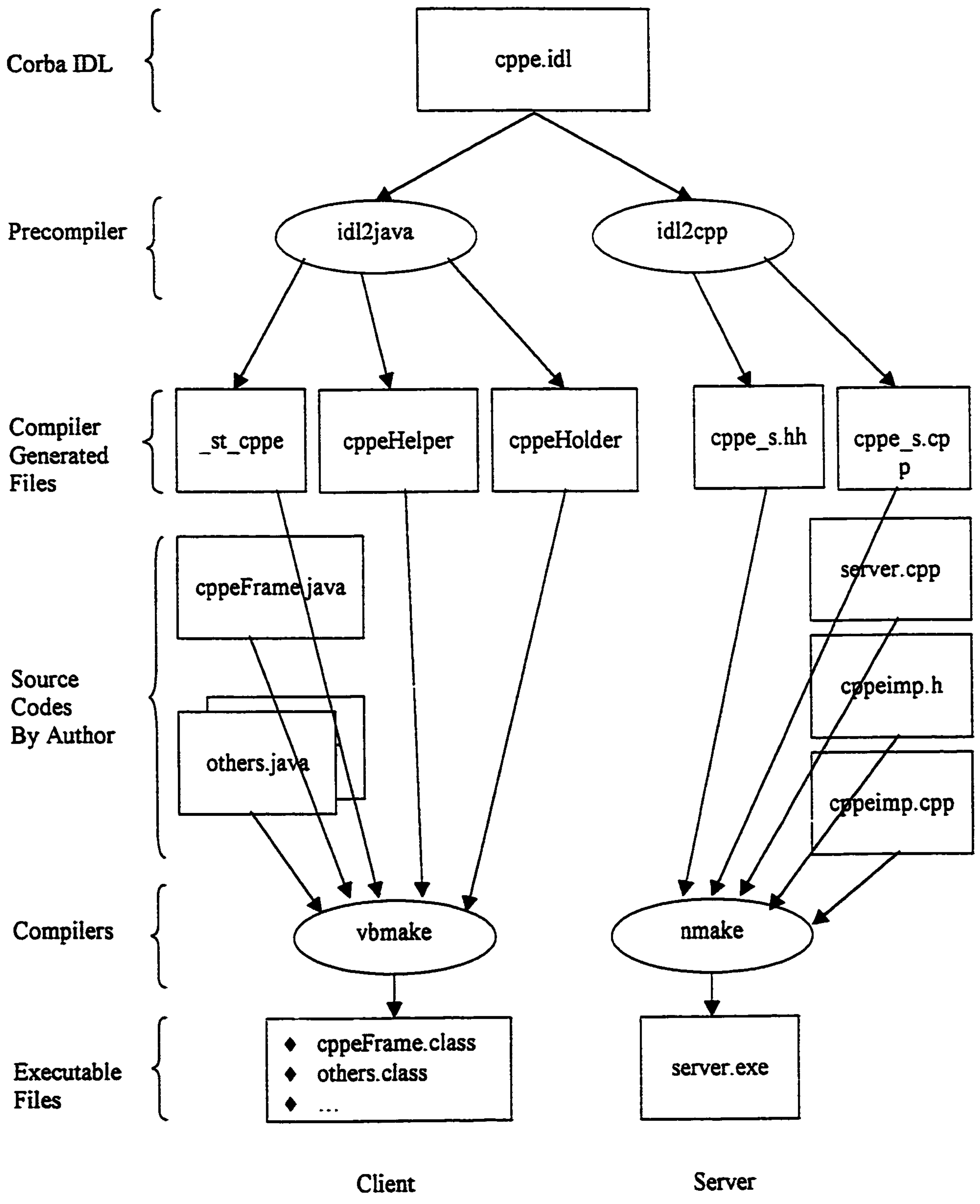


Figure 6 : The Java codes in client side and C++ codes in server side

3.3 Client/Server Implementation

3.3.1 IDL

The CPPE has the following data type needed to map from C++ to IDL, and then map from IDL to Java:

boolean

int

char

char*

array[] of int

array[][] of char

enum

struct{int, int}

struct{char array[], int}

array[] of struct

We can use the following table to map the above data types.

C++ to IDL Mapping		IDL to Java Mapping	
C++	IDL	IDL	Java
boolean	boolean	boolean	boolean
int	short	short	short
char	char	char	char
char*	string	string	java.lang.string
int array[]	short array[]	short array[]	short array[]
char array[][]	char array[][]	char array[][]	char array[][]
enum	enum	enum	Java class with same name as enum type
struct	struct	struct	Java class with same name as struct type

Table 2: The Corba C++ to IDL mapping and Corba IDL to Java mapping

3.3.2 File Structure

In the server side, there are 4 major subdirectories in Cppe\, they are

cppe\cpcc

cppe\cpss

cppe\include

cppe\CorbaCPPE

In the CorbaCPPE subdirectory, there are the following files:

server.cpp

CppeImp.h

CppeImp.cpp

cpssExportCorba.h

Config.h

Config.cpp

CorbaPrint.h

CorbaPrint.cpp

Cppe.idl

Makefile

3.3.3 Operations

Open: when the button "Open" is pressed, then

- ♦ A file choosing window pops up, a local source file (*.c) at the client side can be chosen.
- ♦ The chosen source file is transferred to server side and saved in subdirectory: ~\user.

Compile: when the button "C" is pressed, then

- ♦ The source file located in server side is compiled and its executable file (*.cod) is generated and saved in server side.
- ♦ Compilation displaying will be saved in server side in a temp output file and the contents of the file will be sent back to client to display in the first text area (up).

Execute: when the button "E" is pressed, then

- ♦ The executable file located in server side is executed.
- ♦ Execution displaying will be saved in server side in a temporal output file and the contents of the file will be send back to client to display in the second text area (down).

Save: when the button "Save" is pressed, then

- ♦ The execution results displayed in second text area will be saved into text file in client side. Users can give it any name.

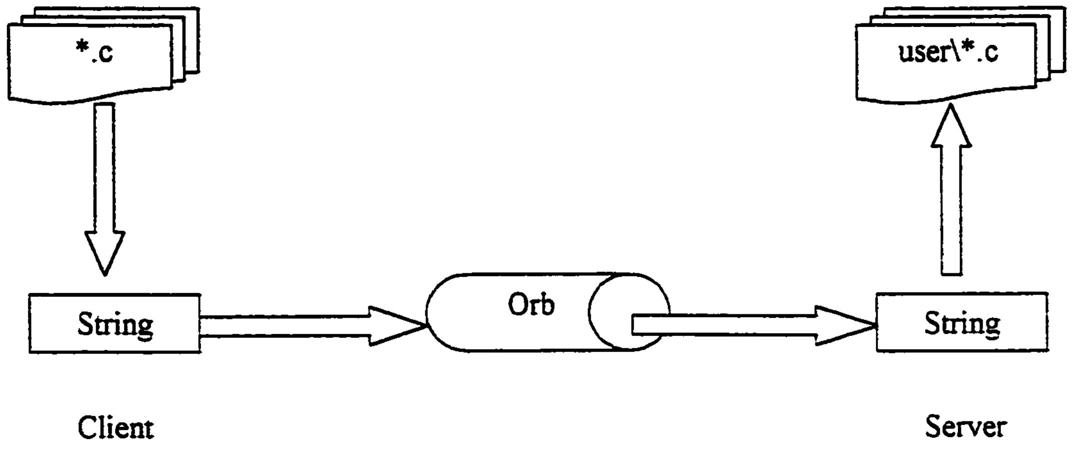


Figure 7 : Open

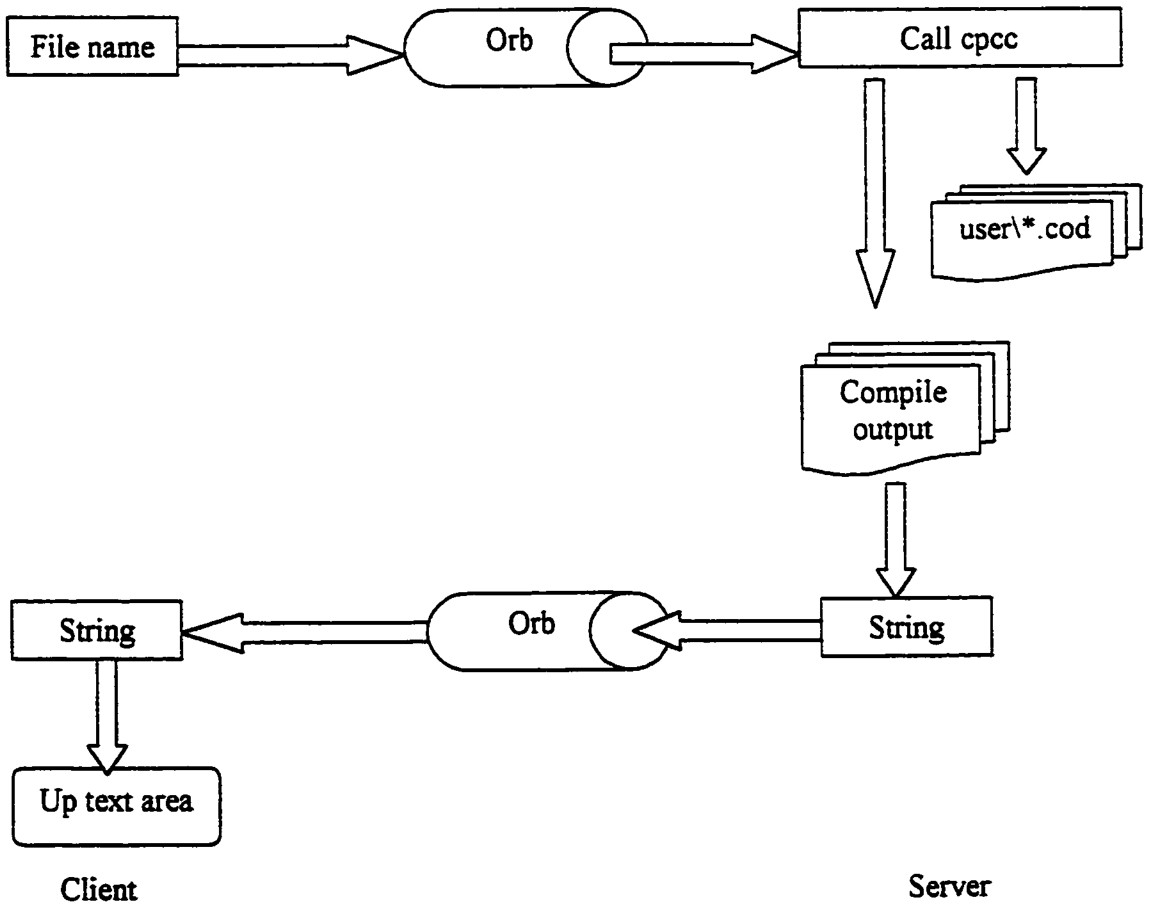


Figure 8 : Compile

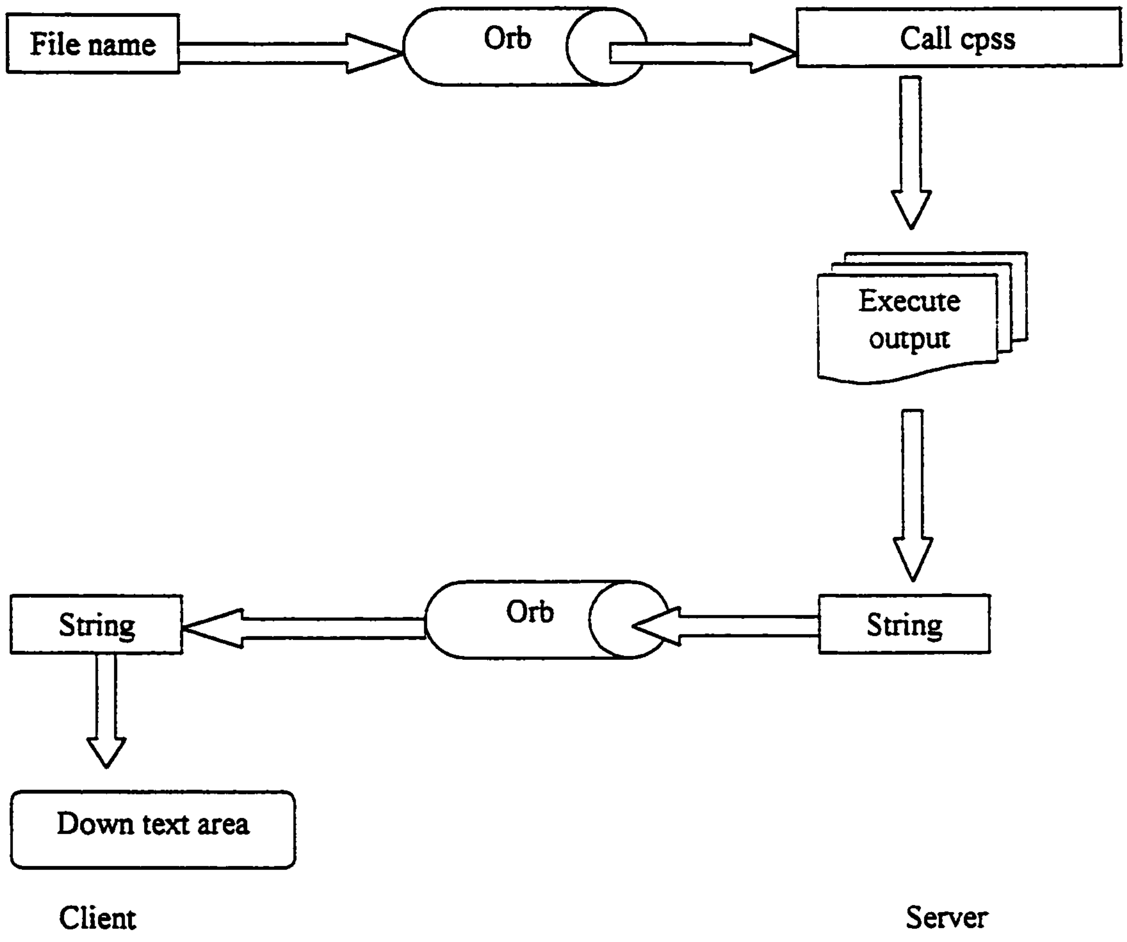


Figure 9 : Execute

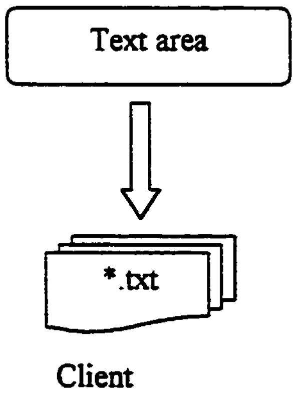


Figure 10 : Save

Chapter 4

4 Java GUI Design

4.1 Design Objectives

A graphic user interface (GUI) is used to communicate with the major components in CPPE (CPCC and CPSS). Through the interaction with the GUI, user can accomplish the whole development process from source code listing, compilation, execution, debugging, and performance profiling. The designing goal of the GUI is to make the developing and debugging process easy for the parallel application programmers.

CPPE needs two major functions to be a complete development environment: compile of the application programs written in CPC, simulation of parallel system and program execution.

The compile function in CPPE is provided by CPCC. If used from command line, CPCC is an independent executable with its own command line input and console output. We need to modify its main function into a CPPE global function, which is called from an interface function. A generic output function is needed to replace the original output function to display the program output into a designated text field in the CPPE GUI.

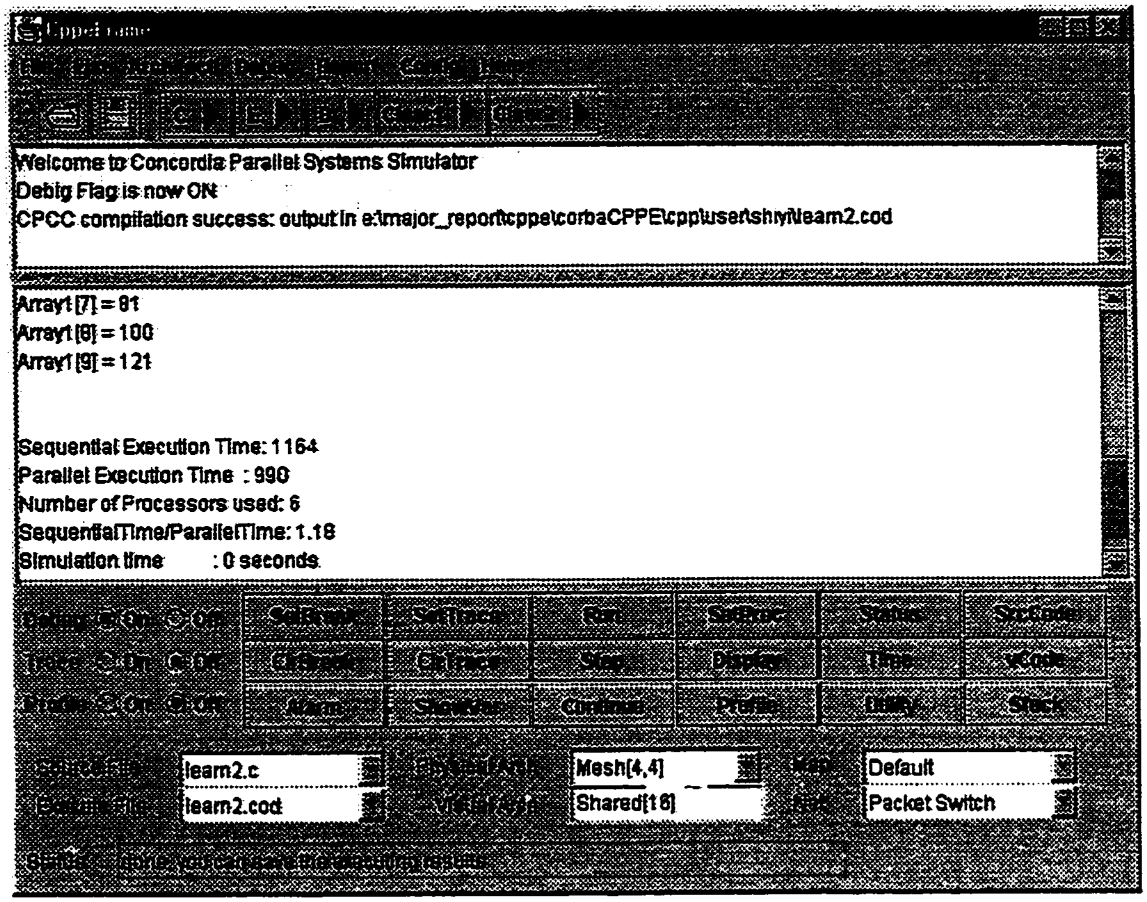
CPSS is the parallel system simulator that includes the functionality of network module, code execution module, and debugging monitor. When used from command line, CPSS has an interpret function that accepts the user command and invokes the corresponding functions from the network module, code execution module or debugging monitor. The function of interpret of CPSS will not be used when incorporated into the Corba CPPE. Instead, CPSS functions can be invoked from the function calls of GUI components. The same generic output function can be used to display the program output into designated text field in the CPPE GUI.

4.2 Main Frame

Figure 11 shows a typical layout of the Java GUI. The window with the title *CppeFrame* is the main frame of the CPPE. To make a friend GUI, we designed many types of GUI widgets. From top to bottom, there are many widgets as the following:

- Menu (including sub menu) with shortcut keys. The menu provides the comprehensive item for CPPE functionality. To refer to Figure 12.
- Floating toolbar with icons and shortcut keys. The buttons in the toolbar can be used to invoke those most frequently used functions in CPPE, such as, opening and compiling a source file, or executing a vCode.
- Splitting text areas that display the application execution result, debugging information and performance statistics.
- Pop up debug pane. Clicking on button D will show or hide the debug pane. Figure 12 shows that the debug pane is hidden.
- Combo box pane.
- Status text field.

The above GUI widgets are implemented in Java/Swing.



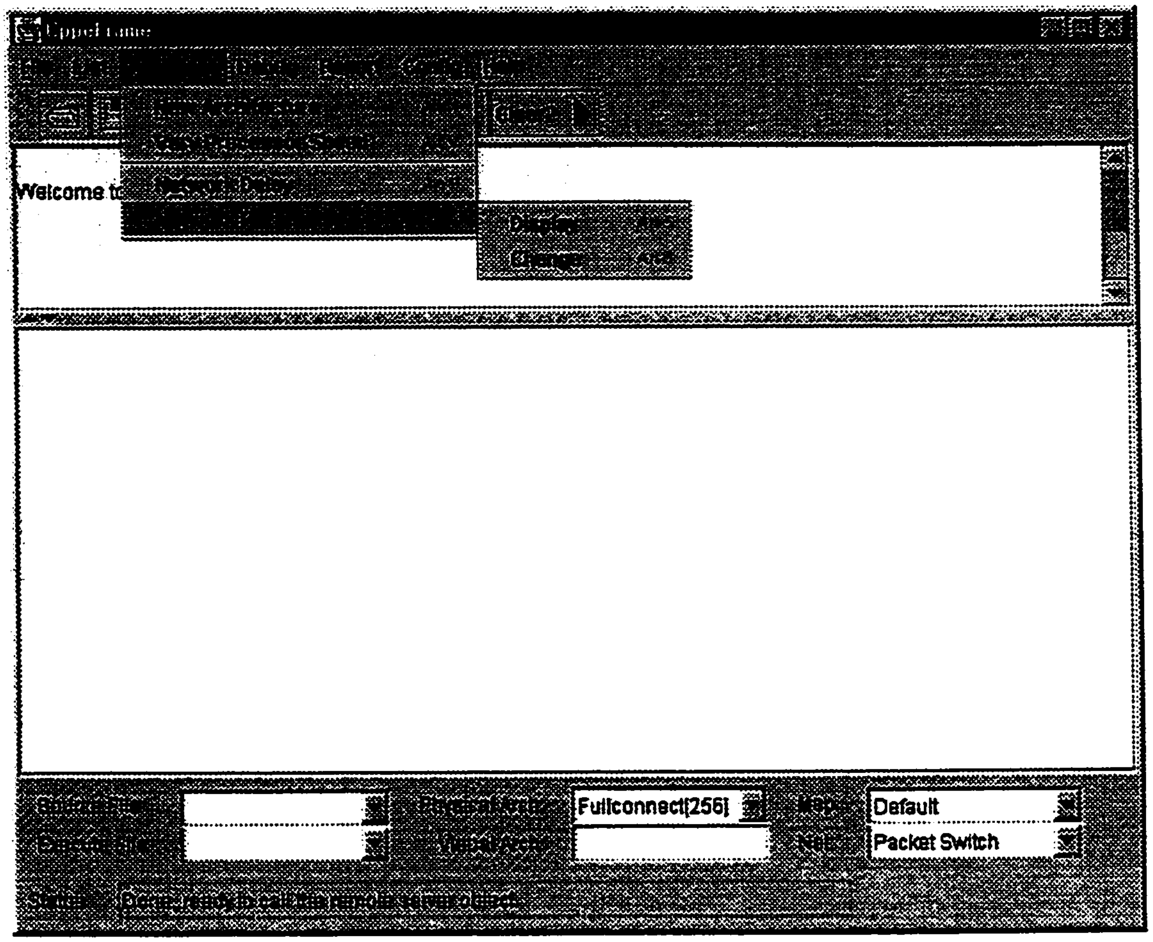
```
Welcome to Concordia Parallel Systems Simulator
Debig Flag is now ON
CPCC compilation success: output in e:\major_report\cppe\corbaCPPE\cpp\user\shiyi\learn2.cod
```

```
Array1[7] = 81
Array1[8] = 100
Array1[9] = 121


Sequential Execution Time: 1164
Parallel Execution Time  : 990
Number of Processors used: 6
SequentialTime/ParallelTime: 1.18
Simulation time       : 0 seconds
```

learn2.c
learn2.cod
Mesh[4,4]
Shared[16]
Default
Packet Switch

Figure 11 : Main frame



Status: Done, ready to call the remote server object.

Figure 12 : Menu in main frame

4.3 Open and Save Shells

The window with the title *Open* is used to open a file from local machine, to refer to Figure 13. It is Java Swing utility. Users can browse the local file system by clicking combo box. Users can select a fold or a file by double clicking on the desired item, or clicking on the desired item and press *Open* button. If users select a file, then this file name will be added in source file combo box and become current one in main frame, also to refer to Figure 11.

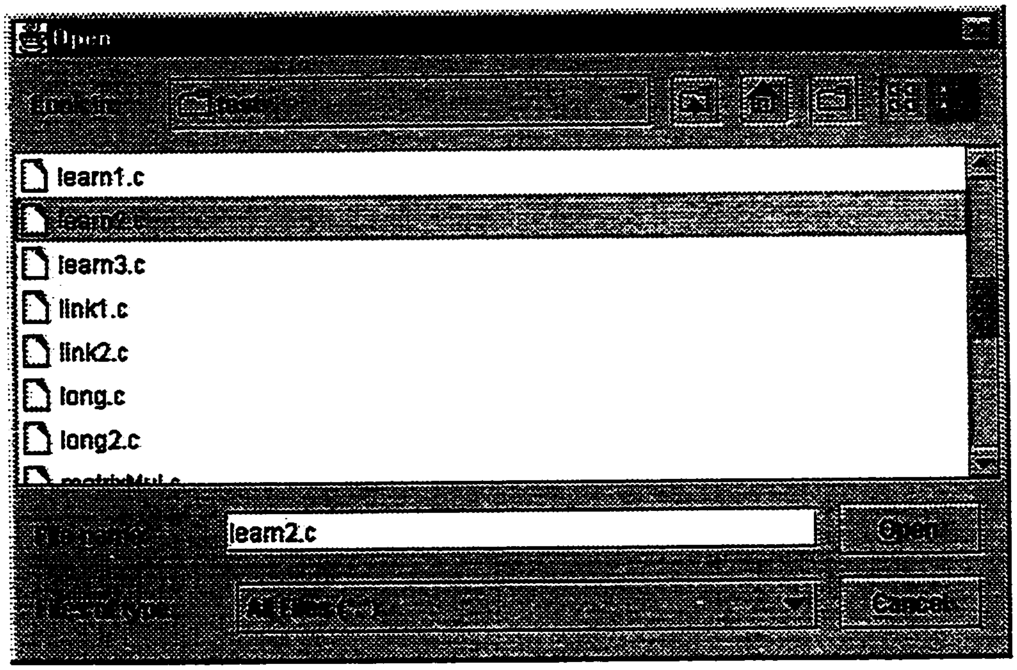


Figure 13 : Open window

The window with the title *Save* is used to save a file to local machine, to refer to Figure 14. Users can browse the local file system by clicking combo box. Users can select a fold by double clicking on the desired item, or clicking on the desired item. After giving a file name, user can press *Save* button to save the contents in text area of main frame, also to refer to Figure 11.

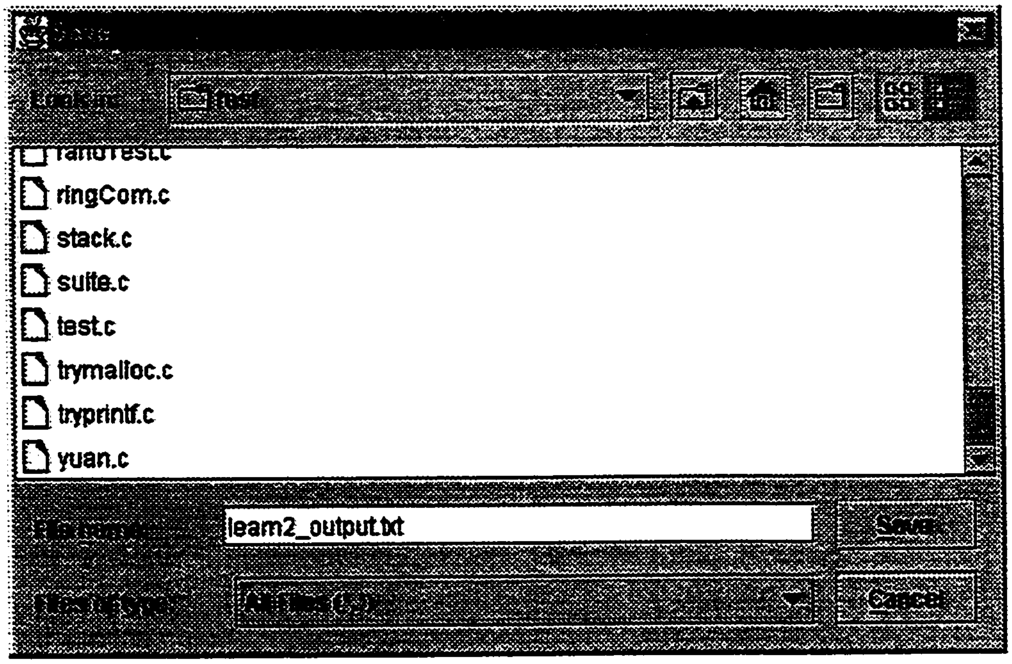
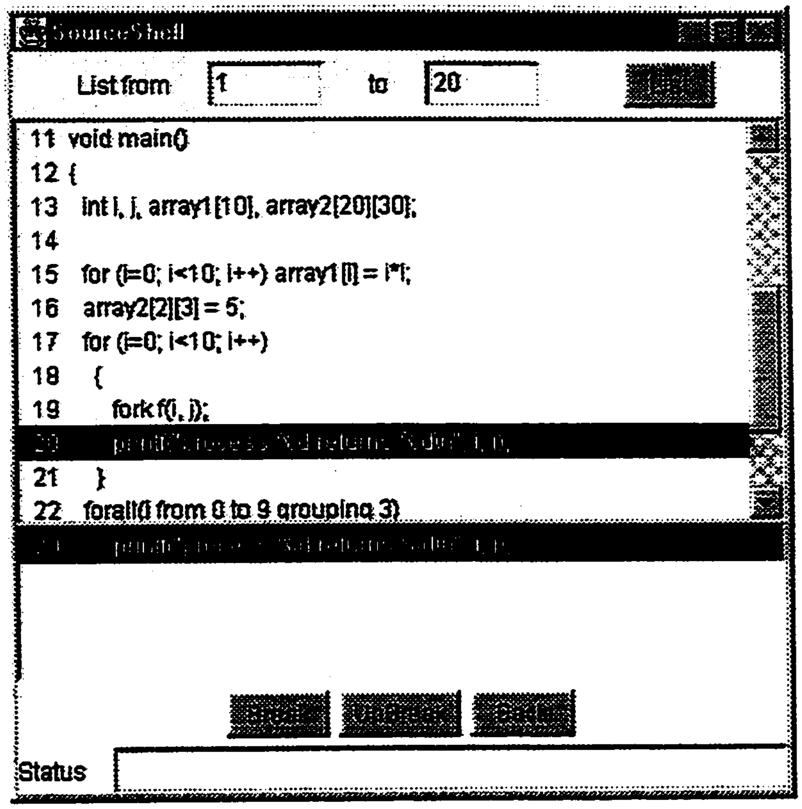


Figure 14 : Save window

4.4 Source Code Shell

The window with the title *SourceShell* is the source code-listing window, to refer to Figure 15. Users can specify the starting and ending line numbers of the source code to be displayed. This window can also be used to set source code breakpoints. There are two source code lists in this window. The list in the upper part of the window is used to display the source code with line number, called *source list*, and the list in the lower part of the window is used to display the source lines of the breakpoints, called *breakpoint list*. Users can set a breakpoint by double clicking on the desired source line in the *source list*, or clicking on the desired source line and press *Break* button, then this source line will be listed in the *breakpoint list*. To clear a breakpoint, users can double click the source line in the *breakpoint list*, or clicking on the desired source line and press *UnBreak* button, then this source line will be deleted from the *breakpoint list*.

By default, the line numbers to be display are from 1 to 20. If users enter invalidate data, such as, "-1" or "abc" in the fields, then the invalidate data will be found and to be replaced by default values automatically. If usera enter a bigger number than the maximum line number of the vCode in the second text field, then the invalidate number will be replaced by the maximum line number automatically.



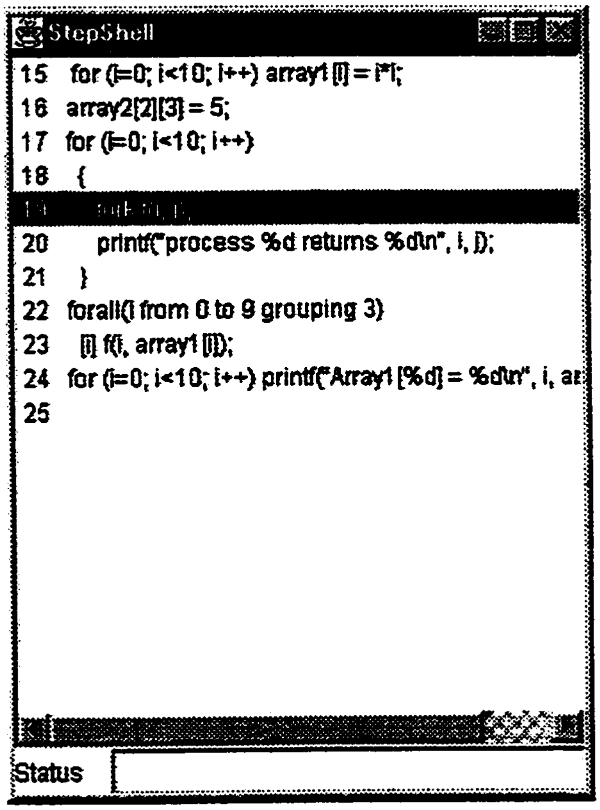
```
void main()
{
  int i, j, array1[10], array2[20][30];

  for (i=0; i<10; i++) array1[i] = i*i;
  array2[2][3] = 5;
  for (i=0; i<10; i++)
   {
     fork f(i, j);
     printf("process %d returns %d\n", i, i);
   }
  forall(i from 0 to 9 grouping 3)
     printf("process %d returns %d\n", i, i);
```

Figure 15 : Source code shell

4.5 Step Shell

The window with the title *StepShell* is the source code step shell window. After setting a break or breaks, users can debug the code by pressing *Run* button in debug pane. Users can find that the currently executed line in the *StepShell* window will be highlighted. The highlighted line will move when user press *Step* button or *Continue* button.



```
 for (i=0; i<10; i++) array1 [i] = i*i;
array2[2][3] = 5;
for (i=0; i<10; i++)
 {
    [illegible]
    printf("process %d returns %d\n", i, j);
 }
forall(i from 0 to 9 grouping 3)
 [i] f(i, array1 [i]);
for (i=0; i<10; i++) printf("Array1 [%d] = %d\n", i, ar
```

Figure 16 : Step shell

4.6 Step Process Dialog

The window with the title *Step Process Dialog* is used to set step line number, to refer to Figure 17. Users can specify a line number of the source code to step. To update the step line number, users can press *ok* button. To keep previous value, press *Cancel* button, then this window will be closed.

By default, the line number to be display is 1. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.

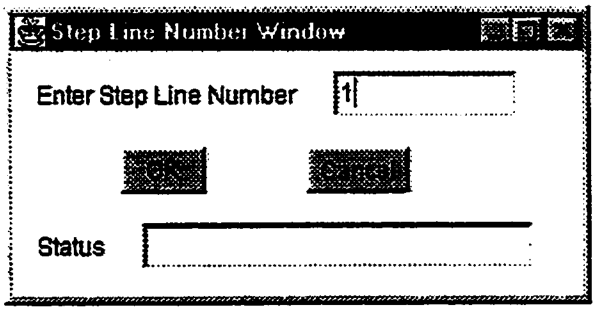
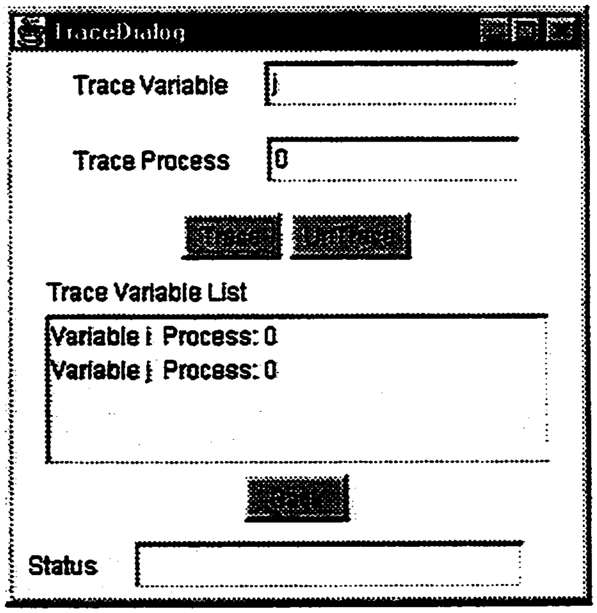


Figure 17 : Step process dialog window

4.7 Trace Dialog Window

The window with the title *TraceDialog* is used to set trace variables in certain process, to refer to Figure 18. The *Trace Variable List* in the window is used to display the variable names with their process numbers. Users can set a variable to trace by pressing *Trace* button. To clear a variable, users can double click the desired line in the *Trace Variable List*, or clicking on the desired line and press *UnTrace* button, then this line will be deleted from the *Trace Variable List*. By pressing *back* button, this window will be closed.

By default, the value of Trace Process field is 0. If users enters invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.



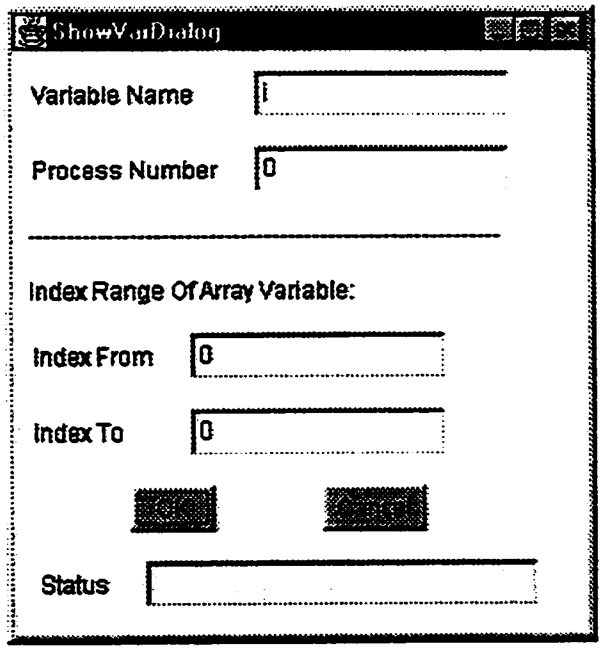
Variable i Process: 0
Variable j Process: 0

Figure 18 : Trace dialog window

4.8 Show Variable Dialog Window

The window with the title *ShowVarDialog* is used to set variable name in certain process, to refer to Figure 19. For an array variable, users can set the index range. To update the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

After setting the trace variables in Trace Dialog Window (to refer to Figure 18), users can use this Show Variable Dialog Window to select a variable to show its value. This variable value will be display in upper text area in main frame (to refer to Figure 11).



Variable Name i

Process Number 0

Index Range Of Array Variable:

Index From 0

Index To 0

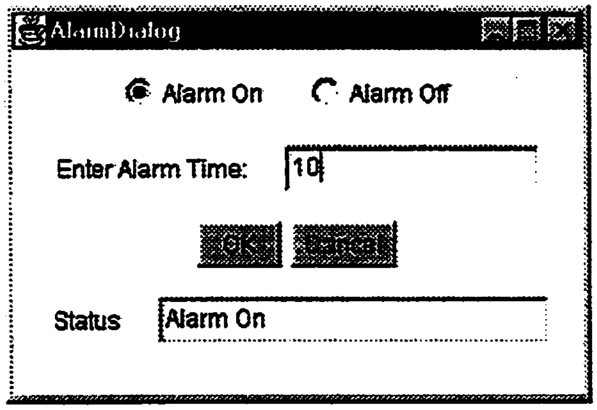
Status

Figure 19 : Show variable dialog window

4.9 Alarm Dialog Window

The window with the title *AlarmDialog* is used to set alarm ON or OFF by clicking on corresponding ratio button, to refer to Figure 20. An alarm time in millisecond can be entered to set alarm time. To confirm the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

By default, the value of Alarm Time field is 10. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.



Alarm On [✓] Alarm Off

Enter Alarm Time: 10

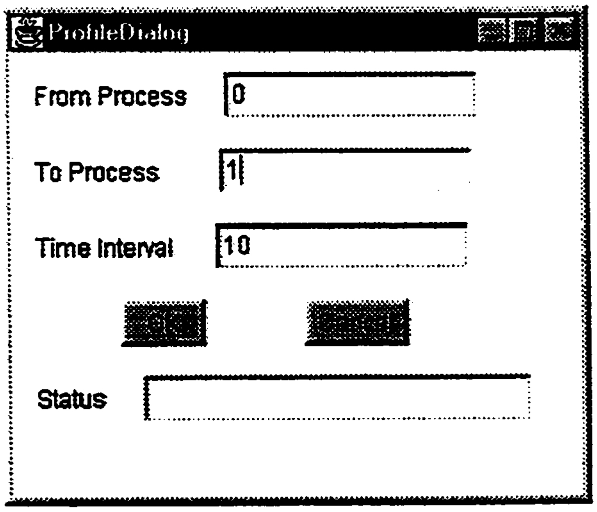
Status: Alarm On

Figure 20 : Alarm dialog window

4.10 Profile Dialog Window

The window with the title *ProfileDialog* is used to set profile for some processes and time interval, to refer to Figure 21. To confirm the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

By default, the field values of From Process, To Process and Time Interval are 0, 1 and 10. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.



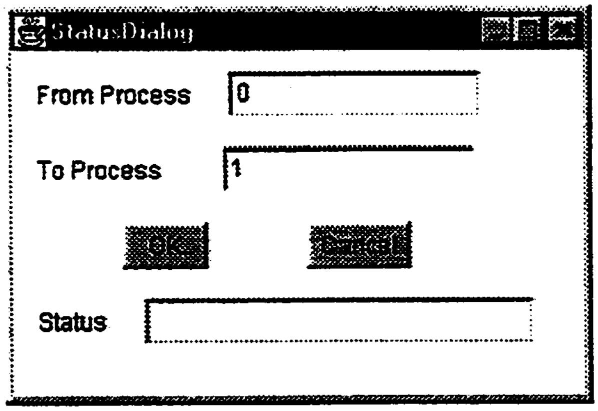
From Process 0
To Process 1
Time Interval 10
Status

Figure 21 : Profile dialog window

4.11 Status Dialog Window

The window with the title *StatusDialog* is used to set to show the status for some processes, to refer to Figure 22. To confirm the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

By default, the field values of From Process and To Process are 0 and 1. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.



From Process 0

To Process 1

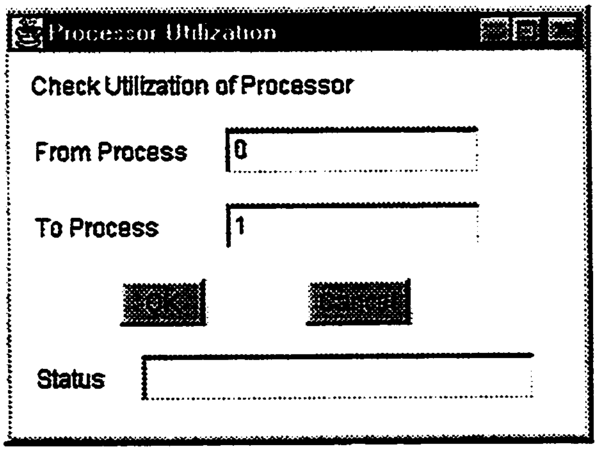
Status

Figure 22 : Status dialog window

4.12 Utilization Dialog Window

The window with the title *Processor Utilization* is used to check the utilization for some processes, to refer to Figure 23. To confirm the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

By default, the field values of From Process and To Process are 0 and 1. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.


Processor Utilization

Check Utilization of Processor

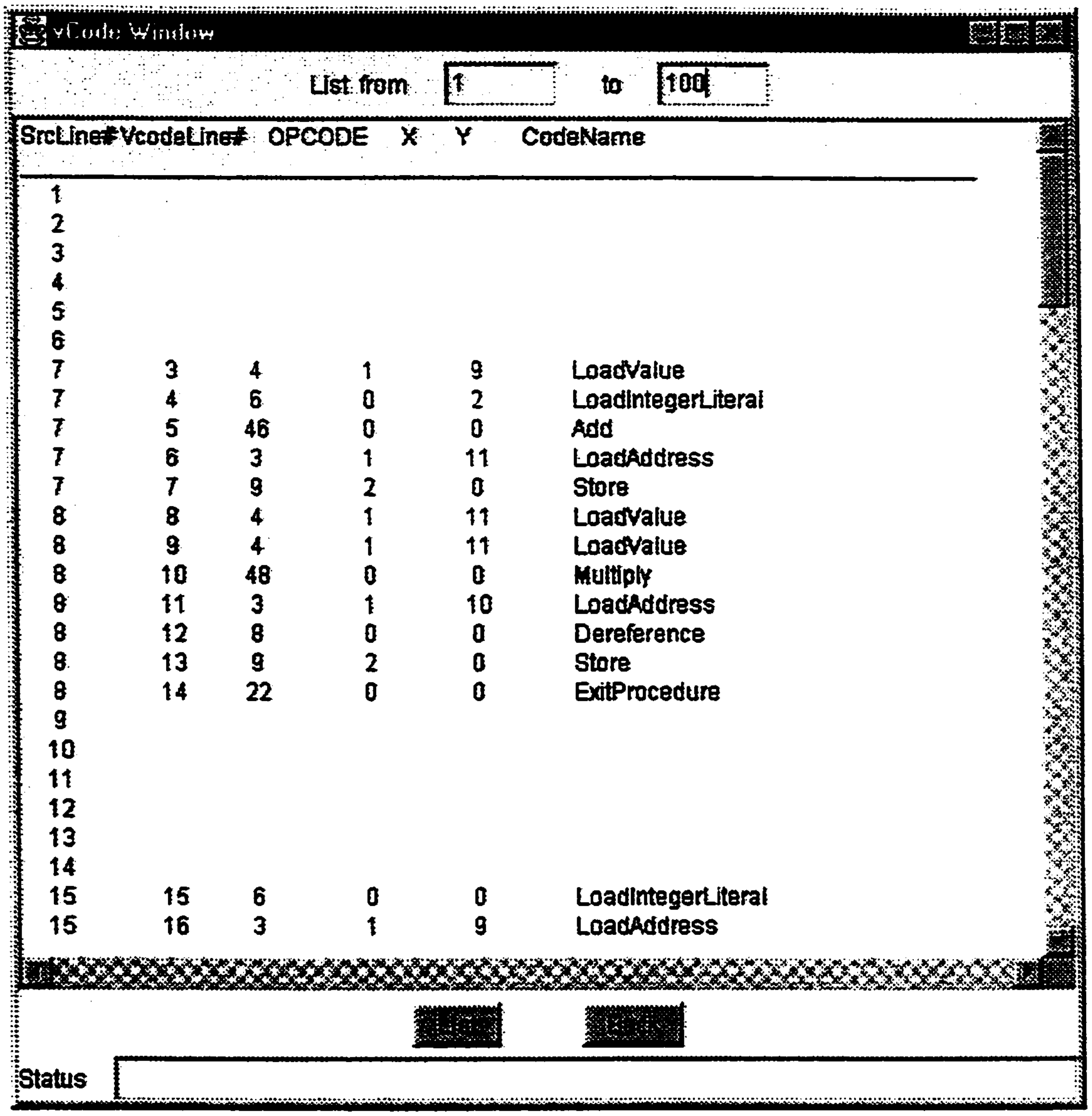
From Process	0
To Process	1
Status	

Figure 23 : Utilization dialog window

4.13 Code Shell

The window with the title *vCode window* is the executable code-listing window, to refer to Figure 24. Users can enter the starting and ending line numbers of the executable code to be displayed, and press *List* button. To close the window, press *Back* button.

By default, the line numbers to be display are from 1 to 20. If users enter invalidate data, such as, "-1" or "abc" in the fields, then the invalidate data will be found and to be replaced by default values automatically. If users enter a bigger number than the maximum line number of the vCode in the second text field, then the user's number will be replaced by the maximum line number automatically. At the same time, the status field will show user some warning message.


vCode Window

List from 1 to 100

SrcLine#	VcodeLine#	OPCODE	X	Y	CodeName
1					
2					
3					
4					
5					
6					
7	3	4	1	9	LoadValue
7	4	6	0	2	LoadIntegerLiteral
7	5	46	0	0	Add
7	6	3	1	11	LoadAddress
7	7	9	2	0	Store
8	8	4	1	11	LoadValue
8	9	4	1	11	LoadValue
8	10	48	0	0	Multiply
8	11	3	1	10	LoadAddress
8	12	8	0	0	Dereference
8	13	9	2	0	Store
8	14	22	0	0	ExitProcedure
9					
10					
11					
12					
13					
14					
15	15	6	0	0	LoadIntegerLiteral
15	16	3	1	9	LoadAddress

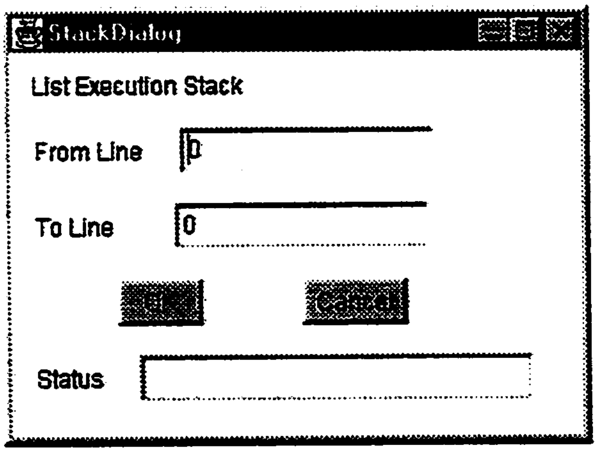
Status

Figure 24 : Code Shell

4.14 Stack Dialog Window

The window with the title *StackDialog* is used to set some lines to be shown in execution stack, to refer to Figure 25. To confirm the new settings, users can press *ok* button. To keep previous settings, press *Cancel* button, then this window will be closed.

By default, the field values of From Line and To Line are 0 and 10. If users enter invalidate data, such as, "-1" or "abc" in the field, then the invalidate data will be found and to be replaced by default values automatically. At the same time, the status field will show user some warning message.


StackDialog

List Execution Stack

From Line 0

To Line 0

Status

Figure 25 : Stack dialog window

4.15 Class Relationship

Java client side has the following files

- CppeFrame.java
- MyFrameWithExitHandling.java
- CppeAction.java
- SourceShell.java
- AlarmDialog.java
- TraceDialog.java
- ShowVarDialog.java
- StepShell.java
- StepProcDialog.java
- StepLineNbrDialog.java
- ProfileDialog.java
- StatusDialog.java
- UtilizationDialog.java
- CodeShell.java
- StackDialog.java
- Cppe.idl
- Vbmake.bat

The Figure 26 shows the relationship of client side classes.

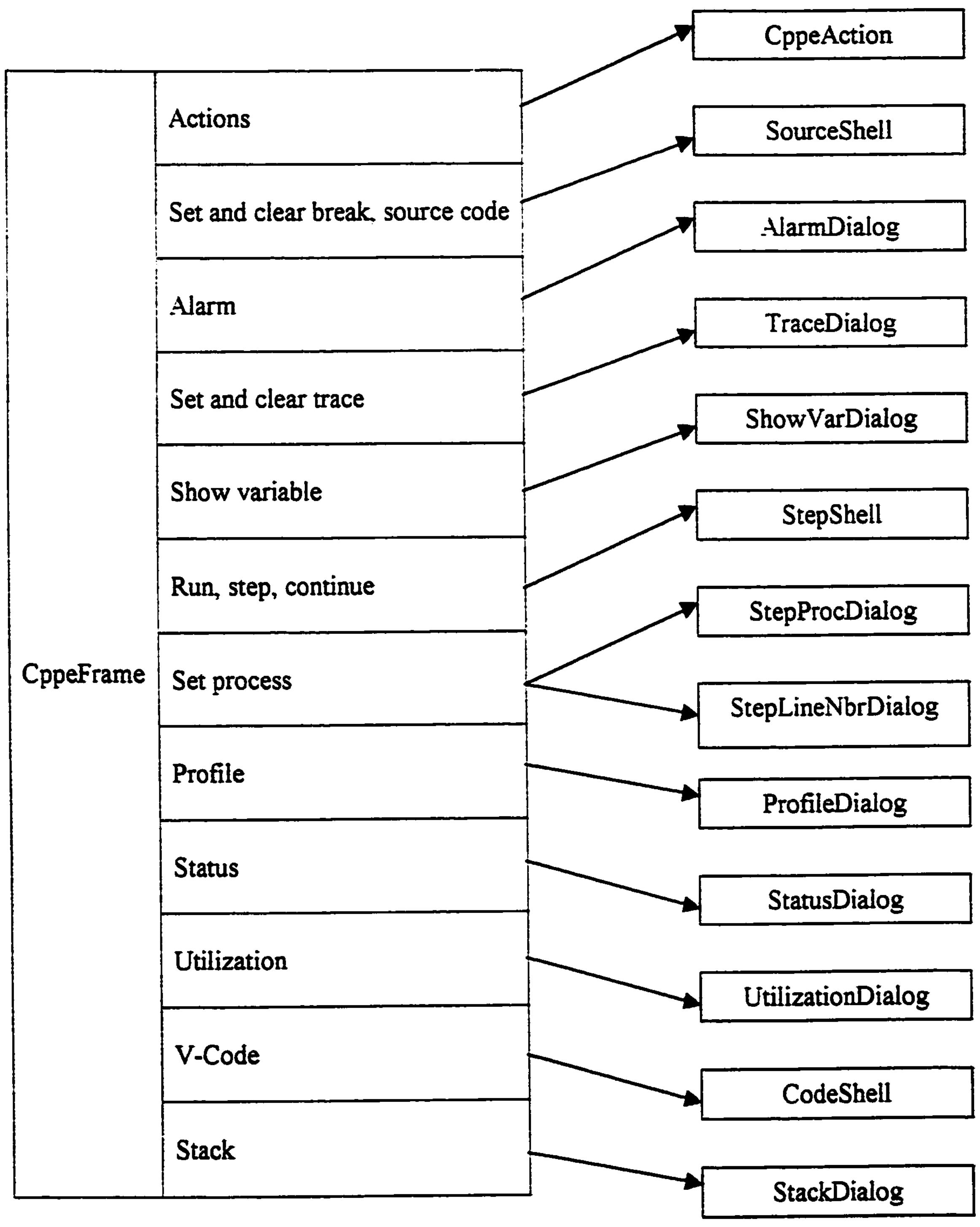


Figure 26 : Relationship of client side classes

Chapter 5

5 User Manual

The Corba version CPPE supports multiple users and contains three major functions:

Parallel application program compiling;

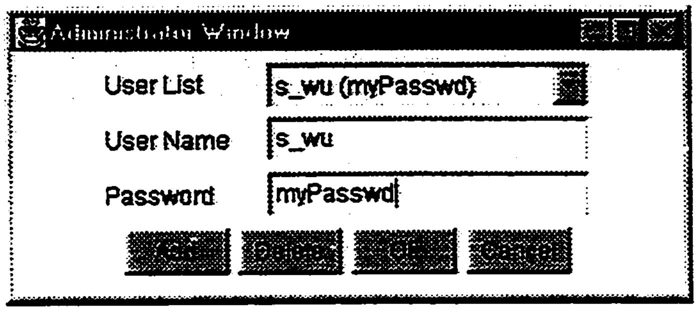
Parallel application program execution;

Correctness and performance debugging.

5.1 Account Management and Login

5.1.1 Account Management

To support multiple users, the Corba version CPPE provides an account for each user. An administrator has the permission to open an account by the GUI shown in Figure 27. This window will pop up when you run "accManage " in server site. An administrator can add, delete an account and list the names, passwords for all current accounts.



User List s_wu (myPasswd)

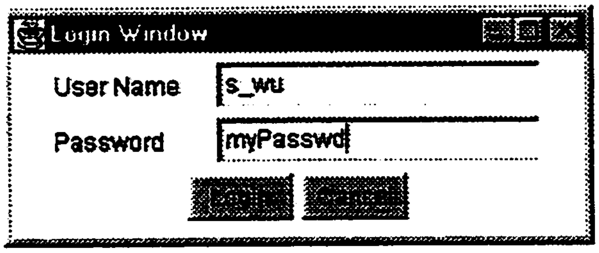
User Name s_wu

Password myPasswd

Figure 27 : Administrator window

5.1.2 Login

To login to an account, a user must have an account with user name and password in advance. The window as shown in Figure 28 will pop up when you run "cppe" in client site. After entering right user name and password, the main frame as shown in Figure11 will pop up.



User Name s_wu

Password myPasswd

Figure 28 : Login window

5.2 Create Application Program Source File

A parallel application program source file should be created first as a text file, with file name extension ".c". Users can use any text editor to create the source file outside the CPPE program.

5.3 Compile Application Program Source File

5.3.1 Open Source Files

To open a source file, users can click the function button *Open* in the GUI main frame. A file selection dialog box is popped up. To select a directory, users can double click the requested directory in the *Directories* list. Then the files in that directory will be displayed in the *Files* list besides the *Directories* list. To select a file, users can double click the requested file or single click the requested file and then click *OK*. The file name will be displayed in the *Source* option menu, to refer to Figure 13.

5.3.2 Compile Source Files

Compile process is used to compile an application program source file into a virtual machine code (vCode) file, with file name extension ".cod", for execution in CPPE.

A source file should be opened before it can be compiled. To compile a source file, users can click the function button *Compile*. If the compilation is successful, the corresponding code file name is displayed in the *Cod* option menu. At the same time, the virtual architecture of the program will be displayed in the *Virtual Arch* text field and the default mapping will be displayed in the *Mapping* option menu.

5.3.3 Open vCode Files

Function button *Open* can also be used to open a vCode file directly. The opened vCode file name is displayed in the *Cod* option menu. At the same time, the virtual architecture of the program will be displayed in the *Virtual Arch* text field and the default mapping will be displayed in the *Mapping* option menu. The opened code file is immediately ready for execution.

5.4 Execution and Debugging

5.4.1 Execution

After a source file is compiled successfully or a vCode file is opened, click the function button *Run* in the main frame. The execution result and any debugging message will be displayed in the main window of the main frame.

5.4.2 Viewing Source Code and vCode

After a source file is compiled successfully or a vCode file is opened, or after program execution is suspended, users can specify a range of source code or vCode to be displayed by referring to the line numbers.

To display the source code, from the main menu in the main frame, users can click the button *D* to pop up Debug pane and click the button *SrcCode*. A new *List* window will be popped up. In the *List* window, users can specify the line numbers in the *List From* and *To* fields and then click the *List* button. If no line numbers are specified in the *List From* and *To* fields, the whole source file will be displayed. To close the *List* window, users can click the *Back* button in the *List* window, to refer to Figure 15.

To display the vCode, from the main menu in the main frame, users can click the button *D* to pop up Debug pane and click the button *vCode*. A new *vCode* window will be popped up. In the *vCode* window, users can specify in the *List From* and *To* fields the starting and ending line numbers for the vCode to be displayed, and then click the *List* button. If no line numbers are specified in the *List From* and *To* fields, the whole vCode file will be displayed, to refer to Figure 24.

5.4.3 Setting Breakpoints

Users can set breakpoints on executable instructions in the source program to automatically interrupt the program execution. This function is useful for helping users to locate bugs in the program, to refer to Figure 15.

Breakpoints are set in the CPPE by referring to program line numbers. To set a breakpoint, users can click the function button *SetBreak* in the debug pane. A new window titled *List* will be popped up. From the *List* window, users can get a list of program source code in a source code list. To set a breakpoint, users can single click on the source line in the source list where the break point will be. Then users can click the *Break* button in the *List* window. The selected break point line will be displayed in the breakpoint list below the source code list. To clear a breakpoint, users can single click on the breakpoint line in the break point list, then click *Unbreak* button in the *List* window. Both setting a breakpoint and clearing a breakpoint can also be done by double clicking the source line in the source list or in the breakpoint list.

If a breakpoint is set and CPPE is set to Debug mode and an user executes an application program, the execution will stop at the breakpoint and a new window titled *Step Execution From Breakpoint* will pop up showing the program source code with the breakpoint highlighted.

5.4.4 Stepping Through Process

When any running process tries to execute a line in a program with a breakpoint, the whole program execution will be suspended. At this point, the execution of the program may be continued with two functions: continue or step. If continue function is used, the execution will be continued until any process encounters the next breakpoint. If step function is used, the execution will be continued line by line from the breakpoint in the suspended process. Users can also specify the number of lines in each step and set the "Step-Process" to a different running process, to refer to Figure 16.

To use Continue function, users can click the function button *Continue* in the main frame. To use the Step function, if users want to step through the currently suspended process line by line, they can click the function button *Step* in the main frame. In either case, if the program execution stops at a new breakpoint of execution, the new breakpoint of source code will be highlighted in the *Step Execution From Breakpoint* window.

5.4.5 Tracing Variables

Whenever the execution of the parallel program is suspended, Users may want to examine the current value of variables in the current environment of each process. CPPE provides two functions for this purpose: show and trace.

Function show is used to display the value of a variable when program execution is in suspension state. To use function show, the variable should be in a currently active process, users can click the function button *Show* in the debug pane, and specify a variable name and an active process id. If the variable is an array, users should specify the index range that user wants to display, then click the button *OK*. The output will be displayed in the up output window in the main frame, to refer to Figure 18.

Function trace is used to trace a particular variable during the execution process. To use function Trace, from the debug pane, users can click button *Trace*. A trace dialog box will be popped up. Users can specify the variable name and process id in the trace dialog box and then click button *Trace*. The traced variables are displayed in the *Trace Variable List* in the dialog box. Users can clear a trace variable later by selecting the variable from the *Traced Variable List* and clicking the button *UnTrace*.

Users have the option to turn on or off the trace function before or during program execution. There is a group of *Trace On/Off* radio buttons in the main frame window for users to turn Trace function on or off.

5.4.6 Alarm

Alarm is used to suspend the program execution when a certain amount of time is reached. The functionality of Alarm is similar to setting a breakpoint so that users can examine execution status in the process of program execution, to refer to Figure 20.

To set an alarm, users can click the function button *Alarm* in the debug pane. An *alarm_popup* dialog box will pop up. Users can turn the alarm function on or off from this dialog box. When the alarm is turned on, users can specify the alarm time in the text field Enter Alarm Time.

5.5 Network Architectures and Mapping

5.5.1 Specifying Architecture

When CPPE starts, the default architecture is a 2D-mesh parallel computer with size of each dimension being 4 (mesh 4x4). Users may override this default and specify a wide range of other architectures, including many of the common parallel topologies. This allows the performance of the parallel program to be simulated and evaluated on a wide range of parallel computer architectures according to the choice of the users.

CPPE has predefined some most common architecture in the system, which can be used directly by selecting from the option menu *PhyArch* in the main frame.

5.5.2 Virtual-to-Physical Architecture Mapping

We are encouraged to write message-passing parallel programs using virtual topologies, the topologies most natural to express the program communication structure. However, the virtual topology may be the same as or different from the topology of the physical system on which the program is running. CPPE supports virtual-to-physical architecture mapping. The objectives of virtual-to-physical architecture mapping are to minimize communication cost by minimizing the distance between communicating processes, and to balance the workload among physical processors.

CPPE currently supports six types of mapping: Default, Identity, Random, Ring-to-Line, Torus-to-Mesh and User-defined Mapping.

To specify a virtual-to-physical mapping, users can select a mapping type from the option menu *Mapping* in the main frame.

5.5.3 Network Routing Type

CPPE can simulate different network types. Currently it supports packet switching network, simulated packet switching network, shortest path network and wormhole-routed network.

Users can select a network type before program execution starts or change network type during program execution. The network type can be selected from the option menu *Network* in the main frame.

5.5.4 Program Performance Statistics

When CPPE executes a program, it keeps track of the relative timing of all processes and generates a range of performance statistics at the end of execution to help the user understand the behavior and evaluate the performance of the program.

5.5.5 Execution Time

At the end of execution, CPPE will display the total Sequential Execution Time and the total Parallel Execution Time. Sequential Execution Time is the estimated execution time on a uniprocessor computer. Parallel Execution Time is the estimated execution time on an actual target multicomputer or multiprocessor. From the ratio of sequential/parallel execution time, user can estimate the performance improvement by parallel computing.

5.5.6 Time

Time function can be used whenever program execution is suspended to give the total elapsed time since the beginning of the program execution.

To use the time function, users can click the function button *Time* in the debug pane. The output will be displayed in the main output window in the main frame.

5.5.7 Utilization

Utilization function is used to show the usage of physical processors in a particular parallel architecture.

To use the Utilization function, users can click the function button *Utilization* in the debug pane. A *utilizationDialog* dialog box will pop up, to refer to Figure 23. Users can specify the range of processors that the user wishes to see the utilization value from the dialog box.

5.5.8 Program Performance Profile

To create a performance profile, the user needs to turn on the profile option. In the CPPE main frame there is a group of *Profile* radio buttons that let users to turn on or off the profile option. The default range of processors in the profile is all processors used in the program. The default time interval in the profile is 10 time units. Users can also specify a different range of processors and time interval, and click the function button *Profile* in the debug pane, a *profileDialog* dialog box will pop up. Users can specify the range of processors and the time interval from this dialog box, to refer to Figure 21.

Chapter 6

6 Build and Installation Manual

To install the application, the following web sites are very useful:

http://www.sun.com.

http://www.javasoft.com.

http://www.visigenic.com.

http://www.microsoft.com.

6.1 Build and Installation Server on Windows NT

(1) Install JDK1.2.2 (20kb).

(2) Install Visibroker for cpp 3.3 (8kb).

(3) Install Microsoft Virtual C++ 6.0.

(4) Unzip the CPPE project file *CorbaCPPE.zip* into your directory *workdir*.

(5) Changed attributes of all files under CPPE from read only to be writable.

(6) Add *"c:\Program Files\Microsoft Visual Studio\VC98\bin\Vcvars32"*into PATH variable.

(7) Copy the following tools (6 files) into directory: *c:\tools*

Flex.exe

Bison.exe

Bison.hai

Bison.sim

Lex.exe

Lex.par

(8) Add *c:\tools* into PATH variable.

Add *c:\tools\Bison.hai* into *BISON_HAIRY* variable.

Add *c:\tools\Bison.sim* into *BISON_SIMPLE* variable.

(9) Add *workdir\cppe\test* into *CPPE* variable.

(10) Compile the cpcc and cpss in DOS prompt:

workdir\cppe cpcc\>nmake

workdir\cppe cpss\>nmake

to create executable files in directory *workdir\cppe\bin*

(11) Change the directory of VBROKERDIR in file *workdir\cppe\CorbaCPPE\cpp\stdmk* to the directory where Visibroker is installed, for example, *VBROKERDIR = c:\pkg\inprise\vbroker* if Visibroker is installed in the directory *c:\pkg\inprise*.

(12) Compile the server in DOS prompt.

workdir\cppe\CorbaCPPE\cpp\>nmake

6.2 Build and Installation Server on Windows 95/98

(1) Install JDK1.2.2 (20kb)

(2) Install Visibroker for cpp 3.3 (8kb)

(3) Install Microsoft Virtual C++ 6.0

(4) Unzip the CPPE project file *CorbaCPPE.zip* into your directory *workdir*.

(5) Changed attributes of all files under CPPE from read only to be writable. (need to do it in several directories. Go there, select all files, then start properties window).

(6) Set PATH in autoexec.bat as the following:

set PATH="c:\Program Files\Microsoft Visual Studio\VC98\bin\Vcvars32"

(7) Copy compilation the following tools (6 files) into directory: *c:\tools\as*:

Flex.exe

Bison.exe

Bison.hai

Bison.sim

Lex.exe

Lex.par

(8) Modify autoexec.bat by adding:

set PATH=c:\tools

set BISON_HAIRY=c:\tools\Bison.hai

set BISON_SIMPLE=c:\tools\Bison.sim

(9) Modify autoexec.bat by adding:

set CPPE= workdir\cppe\test

(10) Compile the cpcc and cpss in DOS prompt:

workdir\cppe cpcc\>nmake

workdir\cppe cpss\>nmake

to create executable files in directory *workdir\cppe\bin*

(11) Change the directory of VBROKERDIR in file *workdir\cppe\CorbaCPPE\cpp\stdmk* to the directory where Visibroker is installed, for example, *VBROKERDIR = c:\pkg\inprise\vbroker* if Visibroker is installed in the directory *c:\pkg\inprise.*

(12) Compile the server in DOS prompt.

workdir\cppe\CorbaCPPE\cpp\>nmake

6.3 Build and Installation Client

The client programs are implemented in pure Java, so they can be installed in any platform. The installation of client programs is very simple, just 4 steps as the following:

(1) Install JDK1.2.2 (20kb)

(2) Install Visibroker for Java 3.4 (12kb)

(3) Unzip the CPPE project file *CorbaCPPE.zip* into your directory *workdir*.

Changed attributes of all files under CPPE from read only to be writable. (select all files, then start properties window).

(4) Compile the client in DOS prompt.

```
workdir\cppe\CorbaCPPE\java\>vbmake
```

Chapter 7

7 Conclusion and Future Work

7.1 Advantages

Our Corba version CPPE has the following advantageous features:

- Distributed application: This version is client/server distributed application. It is Internet accessible.
- Multiple users: The server can provide services for multiple users who have an accounts in advance.
- Reusing of the existing C/C++ code: By using Corba delegation mechanism, the existing C/C++ codes can be reused.
- High-speed server: The server runs fast because it is implemented in C/C++.
- Platform independent client: The client programs can be installed in any platform because they are implemented in pure Java.
- Friendly GUIs: There are rich graphical user interfaces for client users and account manager users. The GUIs make the application easy to use.

7.2 Future Work

In the future, a Java applet can be used to implement the GUI that will be embedded on a web page. A clicking on it will download the codes to the user's web browser, and this applet will communicate with the CPPE Corba server object through gatekeeper. Most of parts of Java applet could be the same as this standalone application.

Bibliography

[1] Jing Zhang, *A Visual Performance Debugger for Concordia Parallel Programming Environment*, Master Degree Thesis of Concordia University, 2000

[2] Orfali Harkey, *Client/Server Programming with Java and Corba*, Wiley, 1998

[3] Thomas Mowbray and Ron Zahavi, *The Essential Corba: Systems Integration Using Distributed Objects*, Wiley, 1995

[4] Thomas Mowbray and Raphael Malveau, *Corba Design Pattern*, Wiley, 1997

[5] Jon Siegel, et al., *Corba Fundamentals and Programming*, Wiley, 1996

[6] Andreas Vogel and Keith Duddy, *Java Programming with Corba*, Wiley, 1997

[7] Robert Orfali, et al., Instant *Corba*, Wiley, 1997

[8] Inprise Corporation, Programmer's Guide: VisiBroker for C++, 2000

[9] Inprise Corporation, Programmer's Guide: VisiBroker for Java, 2000

[10] Y. Daniel Liang, *Introduction to Java Programming*, An Imprint of Macmillan Computer Publishing, 1999

[11] Cay Horstmann and Gray Cornell, *Core Java 1.1*, Prentice Hall, 1997

[12] James Gosling et al., *Java Programming Language*, Addison Wesley, 1998

[13] B. P. Lester, *The art of parallel programming*, Prentice Hall, 1993

[14] E. A. Brewer, et al, "Proteus: A high performance parallel architecture simulator", Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Laboratory of Computer Science, September 1991

[15] E. Reiher, H. H. J. Hum, and A. Singh, "Simulating networks of superscalar processors", *Proceedings of the Supercomputing Symposium*, 1993, pp.125-133

[16] E. Olk, "PARSE: Simulation of message passing communication networks", *Proceedings of the 27th Annual Simulation Symposium*, 1994, pp.115-124

[17] B. A. Delagi, et al, "An instrumented architectural simulation system", Technical Report KSL 86-36, Knowledge System Laboratory, Stanford University, January, 1987

[18] B. A. Delagi, et al, "Instrumented architectural simulation", Technical Report KSL

87-65, Knowledge System Laboratory, Stanford University, November 1987

[19] G. Gao. et al, "Towards a portable parallel programming environment", *Proceedings of the Supercomputing Symposium*, June 1992, pp.219-228

[20] B. P. Lester and G. R. Gutherie, "A system for investigating parallel algorithm architecture interaction", *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp.667-670

[21] A. Saha, "A simulator for real-time parallel processing architectures", *Proceedings of the 28th Annual Simulation Symposium*, 1995, pp.74-83

[22] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993

[23] L. M. Ni and P. K. Mckinley, "A survey of wormhole routing techniques in direct networks", *Computer*, 1993, pp.62-76

[24] Virginia M. Lo, Kurt Windisch, and Rajen Datta. METRICS: A Tool for the Display and Analysis of Mappings in Message-passing Multicomputers. Proceedings of the Sixth Distributed Memory Computing Conference, April 1991

[25] Kurt Windisch, Jayne V. Miller, Virginia Lo. ProcSimity: An Experimental Tool for Processor Allocation and Scheduling in Highly Parallel Systems. Department of Computer and Information Science, University of Oregon, Eugene, OR

[26] Michael T. Heath, Jennifer A. Etheridge. Visualizing the Performance of Parallel Programs. IEEE Software, 8(5), September 1991, pp.29-39

[27] James Gosling, Frank Yellin, and The Java Team. The Java Application Programming Interface Volume 1: Core Package; Volume 2: Window Toolkit and Applets. Sun Microsystems, June 1996

[28] F. Thomson Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, 1991

[29] Michael J. Quinn, Designing Efficient Algorithms for Parallel Computers. McGraw-Hill, 1987

[30] William A. Shay, Understanding Data Communications and Networks, PWS Publishing Company, 1995

[31] S. Chittor and R. Enbody, "Predicting the effect of mapping on the communication performance of large multicomputers", Proceedings of the International Conference

on Parallel Processing, 1991, vol.2, pp.1-4

[32] S. Chittor and R. Enbody, "Performance degradation in large wormhole-routed interprocessor communication networks", Proceedings of the International Conference on Parallel processing, 1990, vol.1, pp.424-428