Fast Adaptation of Legacy Code for Server Hosting

Weilan Jiang

A Thesis

In

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March, 2001

0-612-59329-0

ABSTRACT

Fast Adaptation of Legacy Code for Server Hosting

Weilan Jiang

With the advancement of broadband networking technologies, the Internet is starting to function as a comprehensive and global infrastructure for the delivery of data and computing services. The success of such service-based Internet computing depends heavily on whether we can design and implement reliable server hosted applications in large quantity.

In this thesis we study the problem of how to adapt existing legacy code for server hosting so that clients can get remote computing service through web browsers or thin-client applications. While such adaptation cannot take full advantage of the latest computing and networking technologies, it is critical for the industries to make a smooth and fast transition from the traditional in-house computing to Internet-based computing.

We study the motivation of such adaptation, and provide a comprehensive survey of current industry technologies that support such adaptation. We then use Concordia Parallel programming Environment (CPPE), a comprehensive C program, as an example of legacy code, and use CORBA technology to transform it into a server-hosted distributed component globally accessible on the Internet.

ACKNOWLEDGEMENT

I would like to thank Concordia University's Graduate Diploma Program in Computer Science that allowed me to gain computer science expertise on top of my previous electrical engineering background.

I also would like to thank my advisor Prof. Joey Paquet who accepted me as his Master student, and provided effective guidance for me to complete this thesis. I am grateful to Prof. Lixin Tao who introduced me to the area of Internet computing and his CPPE project, and allowed me to use the source code of CPPE. I appreciate the generous offer of Prof. Adam Krzyzak to be one of my examiners in short notice. Thanks also go to Ms. Halina Monkiewicz who made my Master program study at Concordia a pleasant experience.

Finally, but not the least, I would like to thank my family for its complete support and love for my study and research for this thesis.

Table of Contents

List of Figures viii

CHAPTER 1 Introduction 1
1.1 Objectives 2
1.2 Concordia Parallel Programming Environment 2
1.3 Challenges of this Project 4
1.4 Approach 5
1.5 Contributions 6
1.6 Thesis Organization 7
CHAPTER 2 Literature Survey 8
2.1 Techniques for Fast Adaptation of Legacy Code for Server Hosting 8
2.1.1 Microsoft Windows 2000 Terminal Services 8
2.1.2 Citrix's Independent Computing Architecture 9
2.1.3 CORBA 9
2.2 Why CORBA? 10
2.3 CORBA Architecture in a Nutshell 11
2.3.1 Distributed Object Computing 11
2.3.2 CORBA: Object-Oriented Middleware 13
2.3.3 Object Management Architecture (OMA) 17
2.3.4 Sending and Receiving Requests 22
2.3.5 CORBA Component vs. CORBA Object 25
2.4 Summary 26
CHAPTER 3 CPPE Software Architecture 27
3.1 CPPE in a Nutshell 27
3.1.1 Concordia Parallel C Compiler 28
3.1.2 Concordia Parallel Systems Simulator 29
3.1.3 Graphical User Interfaces 38
3.2 Client/Server Architecture of CPPE 39
3.2.1 Open a Source File 41
3.2.2 Compile a Source File 42
3.2.3 Execute a vCode File 42
3.3 CPPE Interface Design and Implementation 43
3.3.1 Data Type Mappings among IDL, C++, and Java 44
3.3.2 CPPE Interface Specification 46
3.4 Wrapping CPPE Inside a CORBA Component 51
3.4.1 Tie vs. Inheritance 51
3.4.2 CPPE Server Component Based on Delegation 52
3.4.3 Generation of CPPE Java Client and C++ CORBA Server 55
3.5 Architecture for Execution both as an Application and as an Applet 57
3.6 General Guideline for Legacy Code Adaptation 58
3.7 Summary 61
CHAPTER 4 CPPE Java GUI Design 62
4.1 Design Objectives 62

4.2 Selected User Interfaces 63
4.2.1 Main Frame 63
4.2.2 Open and Save Windows 65
4.2.3 Source Code Window 66
4.2.4 Step Window 68
4.2.5 Step Size Window 68
4.2.6 Trace Window 69
4.2.7 Show Variable Window 70
4.2.8 Alarm Window 72
4.2.9 Profile Window 72
4.2.10 Status Window 73
4.2.11 Utilization Window 74
4.2.12 Code Window 74
4.2.13 Stack Window 75
4.3 Class Relationship 77
4.4 Summary 78
CHAPTER 5 Accessing CPPE through Web Browser Applets 80
5.1 Applet Sandbox Restriction 80
5.2 Make Trusted CPPE Applet 82
5.3 How to Sign the CPPE Applet 83
5.4 Summary 86
CHAPTER 6 Conclusion 87
Bibliography 88
Appendix A: User Manual 89
A.1 Account Management and Login 89
A.1.1 Account Management 89
A.1.2 Login 89
A.2 Create Application Program Source File 90
A.3 Compile Application Program Source File 90
A.3.1 Open Source Files 90
A.3.2 Compile Source Files 91
A.3.3 Open vCode Files 91
A.4 Execution and Debugging 91
A.4.1 Execution 91
A.4.2 Viewing Source Code and vCode 92
A.4.3 Setting Breakpoints 92
A.4.4 Stepping Through Process 93
A.4.5 Tracing Variables 94
A.4.6 Alarm 95
A.5 Network Architectures and Mapping 95
A.5.1 Specifying Physical Architecture 95
A.5.2 Virtual-to-Physical Architecture Mapping 96
A.5.3 Network Routing Type 96
A.5.4 Program Performance Statistics 96
A.5.5 Execution Time 97
A.5.6 Time 97

A.5.7 Utilization 97
A.5.8 Program Performance Profile 98
Appendix B: Build and Installation Manual 99
B.1 Build and Installation of CPPE Server on Windows NT 99
B.2 Build and Installation of Server on Windows 95/98 100
B.3 Build and Installation of Client 102

List of Figures

Figure 1 OMG's Object Management Architecture 19
Figure 2 CPPE Structure 28
Figure 3 CPSS Structure and Operations 30
Figure 4 Windows CPPE Graphical User Interface 39
Figure 5 Client/Server Architecture of CPPE 40
Figure 6 CPPE Stub and CPPE Skeleton 40
Figure 7 Open a Source File 42
Figure 8 Compile a Source File 43
Figure 9 Execute a vCode File 44
Figure 10 CPPE Delegation Structure 53
Figure 11 CORBA Tie Class 54
Figure 12 CPPE Server Main() 55
Figure 13 Java Client Files and C++ Server Files 59
Figure 14 Client Structure Serving as Both Application and Applet 60
Figure 15 Main Frame 64
Figure 16 Menu in Main Frame 65
Figure 17 Open Window 66
Figure 18 Save Window 67
Figure 19 Source Code Window 68
Figure 20 Step Window 69
Figure 21 Step Size Window 70
Figure 22 Trace Window 71
Figure 23 Show Variable Window 71
Figure 24 Alarm Window 72
Figure 25 Profile Window 73
Figure 26 Status Window 74
Figure 27 Utilization Window 75
Figure 28 Code Window 76
Figure 29 Stack Window 77
Figure 30 Relationship among Client Side Classes 79
Figure 31 Administrator Window 89
Figure 32 Login Window 90

CHAPTER 1 INTRODUCTION

With the advancement of broadband networking technologies, the Internet is starting to function as a comprehensive and global infrastructure for the delivery of data and computing services. The web browsers are becoming universal graphical user interfaces, and applications are typically hosted on application servers and maintained by domain experts. The clients can access the hosted applications through web browsers and the Internet, thus avoiding all overheads due to software installation, upgrade, and maintenance. This Application Service Provider (ASP) model of computing is the beginning of the next wave of Internet computing characterized by specialized and collaborated computing, and selling computing services, instead of selling computer hardware and software licenses.

But the success of the ASP model of computing depends on whether we can produce reliable hosted applications fast. A hosted application is not a traditional client/server application. It has to support extra properties so that client software implemented in any language running on any hardware or software platform can access and interact with the hosted application. To facilitate the integration of the clients and the hosted applications, it will be helpful to let the hosted applications support component technologies and appear on the Internet as distributed components.

The industries have been seeking for fast track adaptation of existing applications to those suitable for server hosting. Even though such adapted applications cannot take full advantage of the latest technologies, there are many good reasons to use them. First, the adaptation saves software development time, and time could be important for the survival of a young company. Second, existing applications have been tested and used for extended period of time, so they are more reliable than applications that have not been thoroughly tested in real operation environments. Third, some existing applications may have special dependencies on particular platforms, or should be implemented in non-object-oriented way to optimize the execution performance.

1.1 Objectives

In this thesis, we study the general problem of how to transform an existing application, or legacy code, to one suitable for server hosting on the Internet. We study the necessary properties that such hosted applications must have, and provide a survey of existing technologies supporting such fast-track transformations. We then use Concordia Parallel Programming Environment (CPPE) as an example of legacy code, and use CORBA technology to transform CPPE from a traditional C program to one supporting server hosting so that remote users can get CPPE service through an applet or a thin-client application over the Internet. The methodology that we demonstrate in this thesis is general and should have wide applications in the industry practice.

1.2 Concordia Parallel Programming Environment

Concordia Parallel Programming Environment (CPPE) is a complete parallel program development environment implemented by Prof. Lixin Tao. It was designed to help find performance bottlenecks of a parallel program on a particular parallel system.

The design of CPPE was based on Prof. Tao's methodologies called "virtual architecture" and "delayed mapping." Since massive parallel computing is mainly useful for computations with regular communication patterns, the underlying algorithms are typically based on high dimensional meshes or tori that have no counterparts in physical parallel system topologies. CPPE allows a domain expert to choose a virtual architecture based on the intrinsic properties of his problem and algorithm, and organize computation based on processes running on the virtual processors. As a result, the source code of a CPPE application is platform-independent, and the virtual architecture encapsulates the basic communication pattern of the parallel program. The mapping of the virtual processors to the physical processors will not happen until program loading, which is an important property for parallel programs to run in dynamically allocated subsystems of a large reconfigurable parallel system.

CPPE uses Concordia Parallel C (CPC) as its programming language. CPC is the C language extended with language constructs to support message passing as well as shared memory parallel programming. The CPC Compiler (CPCC) compiles a parallel program written in CPC into the virtual code of Concordia Parallel Systems Simulator (CPSS). CPSS can simulate thousands of parallel physical processors with various interconnection topologies and various network switching technologies. CPSS reads a virtual code program, simulates its execution on a parallel system of user's choice, and reports execution results as well as various performance feedbacks. CPSS not only supports traditional debugging for syntax or semantic errors, it also helps identify performance bottlenecks of a parallel program. Users can run the same code based on different physical topologies, different mapping algorithms for mapping virtual processors to physical ones, and different network switching technologies without having to recompile the source code.

For efficiency, CPPE was mainly implemented in the C language. It is not based on object-oriented technology. Therefore, CPPE resembles most of the legacy code of the industry that need to be adapted for application hosting.

In this thesis, we use CPPE as an example legacy code to demonstrate how to adapt it for application server hosting with minimal recoding so a remote user can use it through a web browser or a thin-client program as if CPPE was locally available on his computer.

1.3 Challenges of this Project

CPPE is a typical legacy system. To adapt it to a hosted application suitable for remote Internet invocation, we have to address a series of technological challenges of general importance:

CPPE is implemented in C, but web browsers only support Java applets for interactive and sophisticated graphical user interfaces. In general, how can we transform a legacy system so that it can be invoked by graphical user interfaces implemented in any language that may differ from that of the legacy system?

CPPE is not based on object technology. But as an independent server entity deployed on the Internet, it is only natural to present it as a distributed object or component so its public user interface can be effectively separated from its implementation; users only need to get hold of one reference to an instance of CPPE; and CPPE server can upgrade its implementation without changing remote user interfaces.

As a hosted application, CPPE should be accessible from anywhere on the Internet with graphical user interfaces running on any platform. How to support such network interoperability?

How can we achieve the above objectives with minimum implementation work? Ideally we should not touch the components of CPPE core so we would not introduce potential bugs into this system, and avoid multiple versions of CPPE core that may entail significant cost for system reimplementation and maintenance.

How can we achieve the above objectives with minimal execution overhead normally accompanying hosted applications?

1.4 Approach

In this thesis we use CORBA (Common Object Request Broker Architecture) technology to transform CPPE into a hosted application to be deployed on the Internet to provide global service for clients using either a web browser or thin-client programs.

We first abstract the public user interface of CPPE needed for GUI implementation of the CPPE implementation. We use CORBA IDL (Interface Definition Language) to declare this user interface so that both the CPPE server component and the client CPPE stub (image) object have the same user interface.

We then encapsulate the existing C code into a C++ CORBA component that presents itself as a distributed object on the Internet, supports thread pooling, and can communicate with remote user interfaces through the ORB (Object Request Broker) with the industry standard network protocol IIOP (Internet Inter-Object Protocol).

We then implement graphical user interfaces with Java as a web applet and as a standalone application.

When a remote client launches one of the CPPE graphical user interfaces, a local stub (image) object for the remote CPPE server component is instantiated and connected to the CPPE server component transparently. The graphical user interface only interacts with its local CPPE stub, which in turn transparently forwards all method invocations to the remote CPPE server component.

One challenge for using Java applets to provide a CPPE graphical user interface inside a web browser is the sand-box security model of the web browsers, by which an applet normally cannot access users' local files. But accessing users' local files is necessary if we want to support the illusion that the users have a local copy of CPPE running on their machines. We solve this problem by using Java 2 security model to sign CPPE applet as a trusted one, and export a public key certificate for it.

1.5 Contributions

The major contributions of this thesis include

- A concise but comprehensive survey of the state-of-the-art industry technologies for fast adaptation of legacy code to one suitable for application-service-provider (ASP) model of computing.
- Separating GUI user interface of CPPE from its implementation details, and describing it in industry standard IDL language.
- Wrapping up CPPE code in a C++ version of CORBA component so CPPE can present it on the Internet as a distributed component supporting automatic instantiation and thread pooling.
- The design and implementation of a web-based Java applet user interface and its accompanying CPPE stub object for transparent server connection and server method invocation.

- Using applet certificating technology to break the web browser sand-box security model so CPPE applet running inside a remote user's web browser can access the user's local file system to support the illusion that the user has CPPE running on his own machine.

1.6 Thesis Organization

We start with a comprehensive but concise literature survey for technologies that support fast transformation of legacy code for Internet computing. We explain why CORBA is the right technology for adapting CPPE for server hosting. We also provide a concise survey of the key CORBA concepts and software framework.

Chapter 3 will focus on the software architecture of my distributed CPPE. I provide a concise survey of the major components of CPPE, and my IDL declaration used to abstract the interface of CPPE for client/server implementation. Then we describe how we wrap up CPPE legacy code in a CORBA component, what architectural design that we have adopted so our single version of Java client implementation can run both as a applet inside a web browser and as a standalone application.

Chapter 4 describes the design and implementation of the client's side user interface for CPPE. Chapter 5 explains how we use certification technology to break the sand-box security model of web browsers. Chapter 6 concludes the thesis with some observations and future work.

Finally, Appendix A provides a user manual for our distributed CPPE, and Appendix B provides an installation manual detailing how to build and install the distributed CPPE system.

CHAPTER 2 LITERATURE SURVEY

In this chapter, we first survey the current key technologies that support fast adaptation of legacy code for server hosting. Then we explain why we choose CORBA for our CPPE project. We will also provide a concise survey of the CORBA software architecture to prepare readers unfamiliar with CORBA to understand the remainder of my thesis.

2.1 Techniques for Fast Adaptation of Legacy Code for Server Hosting

There are two basic approaches for such adaptation. The first approach runs the legacy code on servers, and logically extends the cord between servers and client PCs' I/O devices. Microsoft Windows 2000 Terminal Services and Citrix's Independent Computing Architecture are examples of this approach. The second approach encapsulates the legacy code inside a distributed component, and use delegation to support the illusion that the legacy code is designed for server hosting. CORBA technology represents this approach.

2.1.1 Microsoft Windows 2000 Terminal Services

Microsoft Windows 2000 Terminal Services (WTS) [10] is an example technology for fast adaptation of legacy code for server hosting. WTS allows standard Windows-based applications to run on the server instead of on a client PC. Clients running Windows terminal software can then access the sessions through Microsoft Remote Display Protocol (RDP).

2.1.2 Citrix's Independent Computing Architecture

As another example in this category, Citrix [15], the original developer of the core technology underlying WTS, has its own Independent Computing Architecture (ICA) for delivering sessions to clients, which supports non-Windows clients on platforms such as Java and Unix as well as the Windows clients supported by Microsoft RDP. It also offers a technology called Application Launching and Embedding (ALE), which allows Windows applications on the server to be accessed from any browser without the need to install special Citrix or Microsoft client software.

2.1.3 CORBA

Common Object Request Broker Architecture (CORBA) is the dominant distributed component model for ASP applications for which the components need to be deployed across various types of networks and on various platforms. The Object Management Group (OMG), an industry consortium consisting of over 800 IT companies, with the noticeable exception of Microsoft, specified CORBA.

CORBA uses Object Request Broker (ORB) to provide network connectivity for its components. It uses a neutral Interface Definition Language (IDL) to separate interface specification from the implementation of a component. Currently all Netscape web browsers have a built-in ORB to support CORBA-based applications embedded in web contents.

A special feature of CORBA is that it can easily wrap up legacy code in CORBA wrapper components, thus providing a fast-track approach to adapt legacy code for server hosting. The ultimate goal of CORBA is system integration. OMG uses IDL to standardize the specification of vertical and horizontal common facilities, which are discussed in Subsection 2.3.3.2, for system integration.

2.2 Why CORBA?

Windows 2000 Terminal Services and Citrix Independent Computing Architecture basically run the legacy code in its native windows and processes, and maintain image windows of the native ones on remote client machines window system. The following drawbacks of this approach are inherent in its simple software architecture:

1. Significant overhead for each update of the window contents. The window is usually divided into a coarse grid, and each update of some pixels in a grid will lead to the transfer of all pixels of the grid between the native and client windows to synchronize them. While it will work fine on high-speed local area networks, such high volume of data exchange will introduce significant delays in the update of window contents based on today's Internet bandwidth.

2. Limited number of concurrent client sessions. Operating systems have a limit on how many processes or windows it can support at the same time, and this limit is usually small, no more than a few hundreds. But we are looking for technologies that can support thousands or millions of concurrent client sessions.

3. No native support for thread pooling, state management, or persistency service.

Legacy code transformed to distributed components based on CORBA technology enjoys the following major advantages:

- It is globally accessible with a universal reference.

- Client applications can run on any hardware or software platform.
- Client applications can be implemented in most existing programming languages.
- It presents itself as a local object to a client application, thus greatly simplifying the client side programming.
- Automatic thread pooling and management on the server.
- The CORBA environment provides automatic persistency service: the clients get the illusion that the server process never dies.

Other alternative component technologies include Microsoft's COM+ and Sun's Enterprise JavaBeans. The former mainly work on windows platform only, and the latter mainly works on the Java platform only. Neither of them support fast adaptation of legacy code for server hosting. CORBA is therefore the best choice we have.

2.3 CORBA Architecture in a Nutshell

In this subsection, we provide a concise survey of the CORBA architecture so readers unfamiliar with CORBA can understand the remainder of this thesis. This subsection also serves to describe the general framework in which my CPPE software architecture fits and works.

2.3.1 Distributed Object Computing

In many ways CORBA is the next evolution of the client/server paradigm. Client/server topologies came about in the 1980s in response to the need to share centralized data with a large number of end users, all employing the processing power of increasingly powerful desktop computers. Typical two-tier client/server software architecture partitions functionality in such a manner that the client application performs both business processing and user interface operations. The server is used as a data manager: file server or data repository. The architectural limitations and development pitfalls of this approach have become apparent. Software maintenance in a two-tier architecture is tedious at best. Due to the fact that the user interface makes direct calls to the database, changes to the database have a widespread impact. Domain-level reuse is nearly impossible because the rules of the business are so tightly coupled with presentation and data storage logic.

The application of sound software engineering principles has resulted in the separation of responsibilities into three areas, or tiers: presentation, business logic, and data storage and retrieval. The logical separation and loose coupling of tiers isolates them from changes in the other tiers. The second tier, encapsulating the business and application logic, has become known as the *middle tier*. Services to help support the implementation of the middle tier have become known as *middleware.*

Middleware has very different meanings to many people. Paramount in any middleware is that it increases developer productivity and reduces (or abstracts) system complexity. Good middleware will also share several important features: It will allow for clear separation of business logic; it will aid in supporting reuse; it will be standards-based; it must exhibit high reliability and availability; and, to be part of an enterprise's strategic architecture, it must be highly scalable.

Good middleware must support interoperable objects, implemented in various languages, living on different types of platforms, located on a network. A client and a server implemented using object-oriented languages that cannot directly invoke operations transparently, regardless of where those objects actually live, is not a distributed object system.

2.3.2 CORBA: Object-Oriented Middleware

The marriage of the object-oriented paradigm with a client/server topology, with the intention of facilitating the interaction of objects in a client/server relationship, has given rise to CORBA, the Common Object Request Broker Architecture. An industry consortium, the Object Management Group (OMG), created the CORBA standard as an answer to distributed object interoperability. CORBA is the heart of the OMG's architectural framework, the Object Management Architecture (OMA), which will be discussed shortly.

CORBA is superior to other middleware products for many reasons, not the least of which is that it is a nonproprietary, industry-supported standard. Other benefits of CORBA include the following:

- Separation of an object's interface and its implementation.
- Scalability.
- Inherent reusability.
- Language and platform transparency.
- Vendor independence through interoperability.
- A la carte functionality.
- Abstraction of Network communication from the developer.

CORBA clients and servers are developed against a common interface specification, written in the OMG's Interface Definition Language (IDL), which is essentially a contract between a server and potential clients. The IDL for a server specifies the interfaces, or objects; attributes, and operations available for that interface. The IDL file is compiled and supporting files are created that map the IDL specification to a target implementation language. For instance, the VisiBroker for Java IDL compiler generates several *.java* source files for each IDL interface. Some of these source files are used by the object implementation, the server, and some by the client. The client and server need not be developed in the same language. One IDL file may be compiled into different implementation languages – a server written in C++ doesn't know, or care, that a client was written in Java. The client and server communicate via an Object Request Broker (ORB), which is the core of CORBA middleware. The client does not have to be aware of the object location, the network protocol used, the language used to implement the object, or the operating system hosting the server. The only aspects the client has knowledge of are those specified in the IDL interface.

So what is happening underneath? The IDL compiler, ORB, and object adapters conspire to abstract the complexities of distributed object communication. The key layers in this framework follow.

2.3.2.1 Object Request Broker (ORB)

The ORB is the heart of any CORBA implementation. It is responsible for enabling objects to transparently make requests and receive responses in a distributed environment, whether that environment is a heterogeneous or a homogeneous system of computers and networks. The ORB assumes responsibility for so much object management, and the mechanisms of routing invocations to their target objects have been abstracted away so completely, that client applications are relatively simple. To the client, it appears as though every object is always active even though that's not the case. CORBA does not even provide a separate command for a client to start up an object implementation; the client just sends a request and the ORB does everything else. With the exception of some ORB initialization requirements, a typical client views the distributed system solely through the IDL interface specifications and the client is completely separated from the implementation details.

2.3.2.2 Object Adapter (OA)

An Object Adapter (OA), in the CORBA sense, is a logical set of server-side facilities that serves to both extend the functionality of the ORB and to provide a mechanism for the ORB and the object implementation to communicate with each other. Rather than bundling this functionality into the ORB Core, adapters can be used to offer specialized services that have been optimized for a particular environment, platform, or object implementation. The OA is layered on top of the ORB Core to provide an interface between the ORB and the object implementation. A typical OA provides services such as the following:

- Registration of servers (implementations)
- Activation and deactivation of object implementations
- Instantiation of objects at run time and the generation and management of object references
- Mapping of object references to their implementations
- Dispatching of client requests to server objects via a skeleton or DSI (Dynamic Skeleton Interface)

While many types of object adapters are possible for unique situations, the CORBA specification only requires implementations to provide a Basic Object Adapter (BOA). The recent adoption of the Portability Specification calls for the deprecation of the BOA in favor of a Portable Object Adapter (POA).

2.3.2.3 Interface Definition Language (IDL)

The Interface Definition Language (IDL) defines the types of server objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. IDL is the means by which a particular object implementation informs its potential clients of the operations available and how to invoke them. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

An object implementation provides the semantics of the object, usually by defining data to represent the state of the object instance and code (methods) to implement the object's behavior. Each IDL interface is ultimately implemented in code, and that code is collectively called an object implementation. In Java, this typically corresponds to one (though it need not be limited to one) public Java class. To fully implement the object, that class will often use other objects and classes and define variables and methods not part of the IDL interface.

2.3.2.4 Clients and Servers

Quite simply, a client is any entity that issues requests for services upon an object. An entity plays the role of a client relative to a particular object. An object that assumes the role of client on one invocation may in turn respond to requests for services from other objects. A client of an object must have an invocable reference to that object, a so-called object reference.

Typically, the term *server* is used to describe an executable program or a specific process executing that program. The server contains one or more object implementations, that is, a Java class that implements the operations corresponding to an IDL interface.

2.3.2.5 Internet Inter-ORB Protocol (IIOP)

The goal of ORB interoperability is to allow communication between independent implementations of the CORBA standard. ORB interoperability allows a client of one vendor's ORB to invoke operations on an object in a different ORB. Invocations between client and server objects are independent of whether they are on the same or different ORBs. To make this happen, all ORBs must communicate via a standard protocol. The OMG protocol for ORB interoperability, the General Inter-ORB Protocol (GIOP), defines the on-the-wire data representation and message formats for all inter-ORB communication. The OMG also defined a specialization of GIOP, called the Internet Inter-ORB Protocol (IIOP), which uses TCP/IP as the transport layer. Specialized protocols for other transports are expected to be defined in time. All compliant ORBs are required to at least provide support for IIOP.

2.3.3 Object Management Architecture (OMA)

The Object Management Architecture is a larger framework within which all OMG-adopted technology resides. It provides two basic models on which CORBA and other standard interfaces are based: the Core Object Model and the Reference Model. The Core Object Model defines the concepts that allow distributed application development to be facilitated by an Object Request Broker (ORB). It describes the theoretical basis of CORBA. The Core Object Model is an abstract definition that does not attempt to detail the syntax of object interfaces or any other part of an ORB. It also defines a framework for refining the model to a more concrete form. The model provides the basis for CORBA, but is more relevant to ORB designers and implementers than to distributed object application developers. It is thoroughly described in the OMG's *Object Management Architecture Guide* and will not be dealt with at any level in this thesis.

The Reference Model (Figure 1) places the ORB at the center of groupings of objects with standardized interfaces that provide support for application object developers. The following groups are identified: *CORBAservices*, which provide infrastructure; *vertical CORBAfacilities*, which provide special support to applications from various industry domains; *horizontal CORBAfacilities*, which provide application-level services across domains; and *application objects*, which is the set of objects developed for a specific application.

The Reference Model is important to CORBA developers, as it presents a development model through which developers can create and use frameworks, components, and objects.

2.3.3.1 CORBAservices

The *CORBAservices* offer fundamental services for use by the developers of implementation objects. Among other things, the object-level functionality specified by these interfaces provides services to store, manage, and locate objects, to enforce relationships between objects and groups of objects, and to provide frameworks for licensing and security. The published services include:

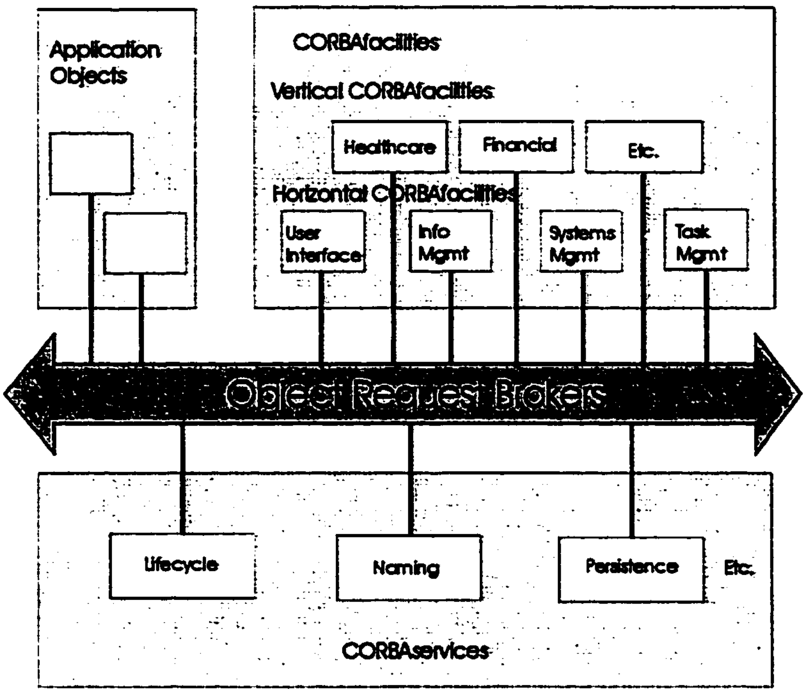


Figure 1 OMG's Object Management Architecture

Naming The Naming Service provides the ability to bind a name to an object relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context. Through the use of a very general model and dealing with names in their structural form, Naming Service implementations can be application specific or based on a variety of naming systems currently available on system platforms.

Events The Event Service provides basic capabilities that can be configured together flexibly and powerfully. The service supports asynchronous events (decoupled event suppliers and consumers), event fan-in, notification fan-out, and, through appropriate event channel implementations, reliable event delivery.

The Event Service design is scalable and is suitable for distributed environments. There is no requirement for a centralized server or dependency on any global service. Both push and pull event delivery models are supported; that is, consumers can either request events or be notified of events.

Life Cycle The Life Cycle Service defines operations to copy, move, and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects. Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references: Role objects in a relationship (see explanation for relationships below) can be located with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.

Persistent Object Service The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The object ultimately has the responsibility of managing its state, but it can use or delegate to the Persistent Object Service for the actual work. A major feature of the Persistent Object Service (and the OMG architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where mechanisms useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers do not apply to mainframes.

Relationships The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: relationships and roles. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the Role interface can be extended to add role-specific attributes and operations. Type and cardinality constraints can be expressed and checked. Exceptions are raised when the constraints are violated.

Externalization The Externalization Service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth), and then it can be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service Specification. The Externalization Service is related to the Relationship Service and parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services, and file services.

2.3.3.2 CORBAfacilities

Whereas the CORBAservices provide functionality for use by objects, the CORBAfacilities provide standards for services aimed at applications. This is an in-progress effort that intends to provide a set of interfaces to generic functionality needed by many applications. Facilities such as printing, document management, and e-mail have been proposed. The horizontal CORBAfacilities are those providing generic services to multiple applications. The vertical CORBAfacilities are those providing specialized services for applications in a specific application domain.

2.3.4 Sending and Receiving Requests

A *request* refers to a client invoking an operation upon an object residing in a server. In order to do this, the client's request is handled locally by a *stub* (also called a *proxy*). The server-side counterpart of a stub is a *skeleton*, which resides between the ORB and CORBA server objects. Client stubs and server skeletons are generated by the IDL-to-language precompiler. It appears to the client that the stub is the actual target object, when in reality the stub is used as a placeholder for the remote object. The stub and the ORB cooperate to marshal any parameters and transmit the request to the remote object. An instance of the skeleton is waiting on the remote system for the client's request. The skeleton (and the ORB) unmarshals the arguments, executes the requested operation, and creates a reply if necessary.

2.3.4.1 Stubs and Skeletons

For the mapping of a non-object-oriented language, there will be a programming interface to the stubs for each interface type. Generally, the stubs will present access to the OMG IDL-defined operations on an object in a way that is easy for programmers to predict once they are familiar with OMG IDL and the language mapping for the particular programming language. The stubs make calls on the rest of the ORB using interfaces that are private to, and presumably optimized for, the particular ORB Core. If more than one ORB is available, there may be different stubs corresponding to the different ORBs. In this case, it is necessary for the ORB and language mapping to cooperate to associate the correct stubs with the particular object reference.

Object-oriented programming languages, such as C++ and Java, do not require stub interfaces, but may still utilize them to provide additional functionality.

For a particular language mapping, and possibly depending on the object adapter, there will be an interface to the methods that implement each type of object. The interface will generally be an up-call interface in that the object implementation writes routines that conform to the interface and the ORB calls them through the skeleton.

The existence of a skeleton does not imply the existence of a corresponding client stub (clients can also make requests via the dynamic invocation interface).

2.3.4.2 Dynamic CORBA

An interface is available that allows the dynamic construction of object invocations. This functionality is called the Dynamic Invocation Interface (DII). Rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (perhaps obtaining it from an Interface Repository or other run-time source). The nature of the Dynamic Invocation Interface may vary substantially from one programming language mapping to another.

On the server side, an interface exists that allows dynamic handling of object invocations. This is called the Dynamic Skeleton Interface (DSI). The DSI allows an object's implementation to be reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface, rather than relying on a specific skeleton to access an operation. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may be also used, to determine the parameters.

The implementation code must provide descriptions of all the operation parameters to the ORB, and the ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the values of any output parameters, or an exception, to the ORB after performing the operation. The nature of the dynamic skeleton interface may vary substantially from one programming language mapping or object adapter to another, but will typically be an up-call interface.

Dynamic skeletons may be invoked through both client stubs and the dynamic invocation interface; either style of client-request construction interface provides identical results.

2.3.4.3 Interface and Implementation Repositories

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at run time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet be able to determine what operations are valid on the object and make an invocation on it.

In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects.

For example, debugging information, libraries of stubs or skeletons, or routines that can format or browse particular kinds of objects might be associated with the Interface Repository.

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, and security might be associated with the Implementation Repository.

2.3.5 CORBA Component vs. CORBA Object

A CORBA component is a module of executable code that is designed to support run-time system integration. With suitable IDE (Integration Development Environment) tools, a system integrator can integrate remote and local components visually into a new application or component without the source code of any of the components.

CORBA objects are, by definition, components because of the way they are packaged. In distributed object systems, the basic unit of work and distribution is a component. The CORBA object infrastructure makes it easier for components to be more autonomous, self-managing, and collaborative. This undertaking is much more ambitious than anything attempted by competing forms of middleware. CORBA's distributed object technology allows us to put together complex client/server information systems by simply assembling and extending components. You can modify objects without affecting the rest of the components in the system or how they interact. A client/server application becomes a collection of collaborating components.

In this thesis we will use "CORBA components" and "CORBA objects" interchangeably.

2.4 Summary

In this chapter we have surveyed three technologies supporting fast-track adaptation of legacy code for server hosting, and concluded that only CORBA can support scalable server performance.

We have provided a concise survey of CORBA software architecture. We explained how the CORBA framework and services could abstract networking details from the programmers, thus greatly simplify the design and implementation of Internet-based client/server applications. The key CORBA features that support our CPPE project include CORBA Interface Definition Language, language mapping between IDL and C++, language mapping between IDL and Java, Object Request Broker, Object Adapters, and server stubs and skeletons.

CHAPTER 3 CPPE SOFTWARE ARCHITECTURE

In this chapter we first summarize the objective, structure, and function of CPPE. We then outline the software architecture for our distributed version of CPPE. We present our platform-independent IDL interface design for CPPE, and highlight its important operations. We also explain how we use CORBA delegation approach to wrap the legacy CPPE code inside a CORBA distributed component, and how to design the software architecture so that the same client code can run both as a standalone application and as an applet inside a web browser.

3.1 CPPE in a Nutshell

The Concordia Parallel Programming Environment (CPPE) [14] was designed to study the interplay of algorithm, hardware and software platforms, interconnection network, mapping of processes to processors, and network switching technologies on a parallel program running on a variety of multi-computers and multiprocessors. It can also support a complete programming environment for developing system-independent, portable parallel programs on any platform supporting a C compiler, and provide powerful debugging features not only for traditional syntax and semantics bugs but also for performance bottlenecks. Figure 2 shows the major components of CPPE.

The CPPE is based on the *virtual architecture* and *delayed mapping* approaches to parallel computation. Given a problem, the domain expert will declare a virtual architecture for his parallel program based on the intrinsic properties of the problem and the algorithm of his choice, and specify computation and communication based on this

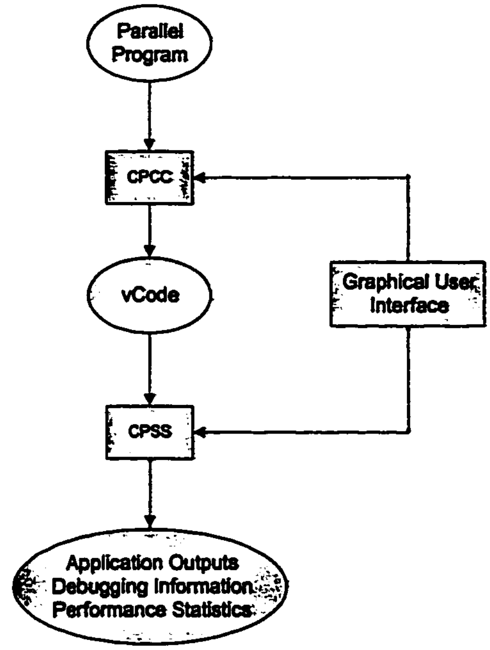


Figure 2 CPPE Structure

virtual architecture. The language supported by CPPE is the Concordia Parallel C (CPC), which is basically the C language extended with language features for parallel computing. The Concordia Parallel C Compiler (CPCC) reads the CPC source code, generates complete abstract syntax tree for it, perform extensive semantics analysis, and output the virtual code (vCode) for the program. The vCode is still system-independent. The Concordia Parallel Systems Simulator reads in the vCode, simulates various parallel architectures and network switching technologies, and performs the mapping of virtual processors to the simulated physical processors at program loading time. The user can change the simulated physical architecture, change the network switching technology, or change the process mapping without recompiling the source code.

3.1.1 Concordia Parallel C Compiler

The core of CPCC (Concordia Parallel C Compiler) is a compiler. After reading a parallel program written in CPC (Concordia Parallel C) language, the CPCC builds an 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 a 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.

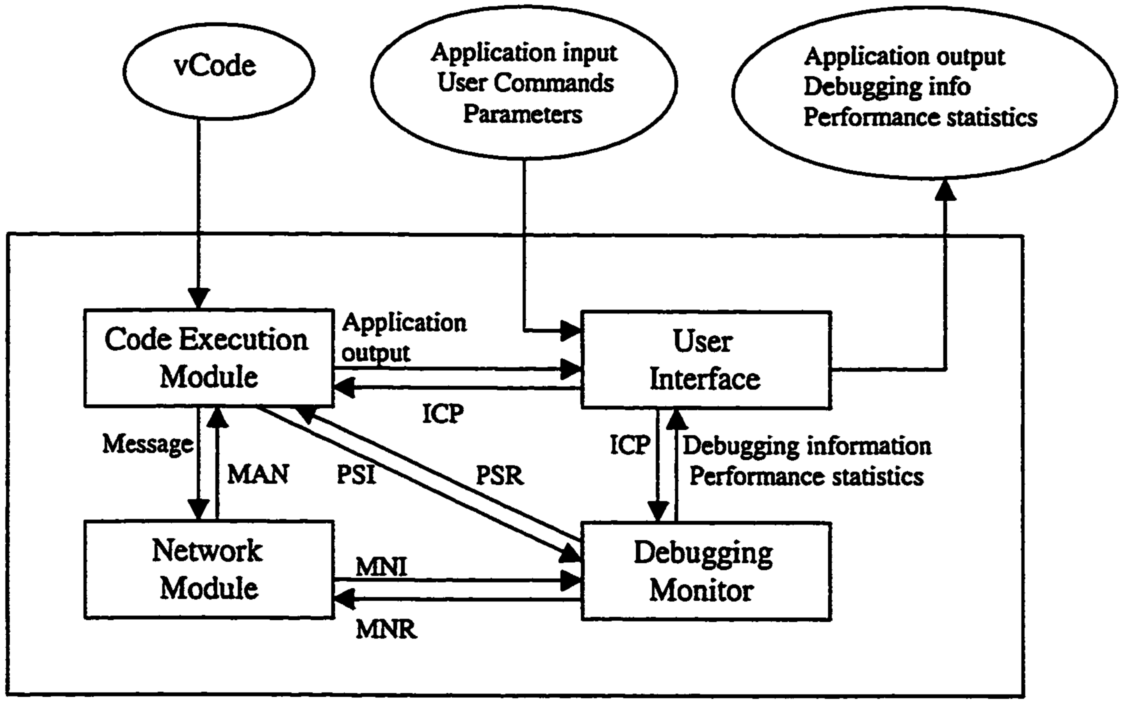
3.1.2 Concordia Parallel Systems Simulator

The vCode produced by the CPCC is the input to the CPSS. Other inputs to 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 of the CPSS are the application outputs, performance statistics, and debugging information.

The CPSS consists of two major components: the code execution module and the network module. The code execution module models the processing elements of a parallel computer system: it executes the parallel code specified by the parallel program. The network module is to manage the inter-processor communication via message passing. There are two other utility modules interacting with the code execution module and network module in CPSS: they are the debugging monitor and the user interface.

The interactions among the components in CPSS are illustrated in Figure 3.

The debugging monitor and the interface not only closely interact with the code



MAN: Message Arrival Notification
MNR: Message/Network Request
MNI: Message/Network Information
ICP: Input/Commands/Parameters
PSR: Process/Processor Status Request
PSI: Process/Processor Status Information

Figure 3 CPSS Structure and Operations

execution module and the network module of CPSS during execution, their implementation is also dependent on the high-level design of these two larger components.

3.1.2.1 The Code Execution Module

The Code Execution Module (CEM) plays the role of a processing element of a parallel computer system: it executes the parallel code specified by the parallel program. There are three key issues that influence the design of the debugging monitor: simulation at the function level, sequential execution model and the way timing system is implemented. CPSS uses a functional simulation technique, which uses a sequential execution model to emulate the parallel execution and interprets the parallel object code instructions at the function level. This technique offers the most accurate results among the existing simulation techniques [14]. In addition, this technique provides a good basis for performance debugging. The following are the key features of CPSS:

- Functional simulation: CEM interprets the virtual parallel code at the function level as if they were executed on the target machine. Each instruction of the target machine is usually expressed as a host macro or procedure whose size depends on the complexity of the instruction and the desired level of simulation accuracy. This technique permits the simulator to have complete control over program execution. It allows users to establish the connection between their program statements and object instructions. Thus the user can set breakpoints, examine trace variables, or step-through the program fragment of a particular process. Monitor code can be added to the simulating code without affecting the execution outcomes, because the CEM is able to distinguish between application code and monitor code and the execution time for monitoring code is not accumulated.
- By using a functional simulation technique, we can parameterize system measurements (e.g., system clock cycle, execution time of object code instructions, network packet size, link buffer size, network delay, message and packet startup overheads). Performance statistics are based on these parameterized measurements.
- The sequential simulation is deterministic in nature. Therefore repetition of execution of a parallel program will always produce the same results and performance under the same system parameters. This provides a stable environment to study the performance of parallel programs at different levels of detail and from different perspectives.

- Timing system: CPSS does not use the machine clock for performance timing. There is a global clock for the simulated parallel computer system that is updated periodically by the CEM. Each process has a local clock that keeps track of the present time of this process. In the CPSS, parallelism is simulated by time slicing: each application process is given a quantum to run and processes are scheduled in a round-robin fashion. During each quantum, the job scheduler traverses the list of physical processors and schedules one process on a processor at a time for execution. The local clock of the scheduled process is updated after each instruction is executed. The user can define the cost to execute an instruction based on the complexity of the instruction. A process runs until its time quantum expires or it is put to sleep by some event. The job scheduler then schedules a process on the next processor for execution. When every processor has finished its quantum, the global clock is advanced to the next quantum. By using this timing system, CPSS can provide accurate and repeatable performance statistics for performance debugging.

3.1.2.2 The Network Module

The Network Module is responsible for inter-process communication via message passing. It is under control of the network manager. The network manager allocates network resources to messages to be sent, routes messages and delivers them to destination processors, and detects and resolves deadlock, if any.

The following design issues of the network module is crucial for accurately simulating the communication behavior, yet providing feasibility for performance evaluation for parallel applications:

By using the functional simulation technique and the same global clock mentioned in the CEM design, it can effectively simulate the network behavior and communication cost such as message startup overhead, routing overhead and congestion delay. New messages that are being initialized for routing are queued at a new message list. The waiting time at this list simulates message startup overheads. When the startup overhead time of a new message expires, the message will be removed from the list and appended to a list of active messages. In each quantum, all active packets that are not blocked are advanced by one link, and it simulates the movement of packets by advancing their ID numbers.

- Most of the network and communication parameters are well defined with appropriate data structures. User can configure most of the network parameters such as packet size, flit size, routing scheme, link bandwidth, communication delay, network topologies and virtual-to-physical mapping without having to recompile the simulator software or application programs. This provides a flexible performance debugging environment.

3.1.2.3 CPPE Debugging Tools

The system architecture of CPSS provides a good debugging environment that is inherited from the functional simulation technique and the timing system employed in the design of the two major components (Code Execution Module and Network Module) in CPSS. We now identify the major debugging tools that have been implemented in CPSS and describe the high-level design of these debugging tools.

➢ **Correctness Debugging**

CPSS supports the following features for correctness debugging, with the design concepts borrowed from sequential programming environments:

- Set and clear instruction breakpoints. Users can set breakpoints in the source program to automatically interrupt the program execution. Users can also clear breakpoints.
- Set and clear trace variables. Users can set a trace flag on a variable. Whenever the variable is referenced during execution, the program execution is suspended so that users can inspect the execution status. Users can also clear the trace flags.
- View the value of a variable. Users can view the value of a variable in an active process when execution is suspended.
- Step through a process. Users can suspend a program and then let the execution continue line by line or by a specified number of lines.
- Set a particular process to be the current process for debugging. The user may then use the above tools to debug the current process.
- View the program source code (written in the CPC language). Users can specify the range of the source code to be displayed.
- View the vCode corresponding to specified range of the source code.
- View the status of the processes. Information about each process includes
  - the processor on which this process is run
  - the process status (e.g., ready, running, blocked, etc.)
  - the stack of the process
  - the function that is currently executed by this process
  - the line in the source code that is currently executed by this process.

The main data structures related to correctness debugging are:

- Source code breakpoint table: CPSS maintains a global break table that stores all breakpoints set by the user at run time. When executing in debugging mode, the code execution engine checks the line number of an instruction in the global break table before executing it.
- Trace variable table: CPSS maintains a global trace table that stores all traced variables set by the user at run time. Since CEM references variables by their addresses in the memory pool, traced variables are stored in the trace table with their memory addresses. When a vCode instruction references a variable, CEM will check whether the referenced variable is in the trace table.
- Source code and vCode table: Source code table stores the application source code, and vCode table stores the compiled virtual machine code. CEM executes the vCode in the vCode table, with the index of the vCode table serving as the program counter (PC).
- Source-to-vCode table: each source code is usually compiled into several vCode instructions for execution. The source-to-vCode table src2codTable will associate the source line number with the vCode line number so that we can trace the program execution. The first vCode line number of each corresponding source code line will be stored in this table.
- Memory pool: Memory blocks will be allocated from the memory pool and distributed to running processes for program execution. Memory pool is implemented as a fixed-size array, with the index of the array serves as the memory address. Variables are accessed by their addresses in the memory pool.

➢ **Performance Debugging**

CPSS not only provides the overall performance profile of a parallel program execution, but also provides the following functionalities for studying the performance of any portion of the parallel program.

- Set and clear time breakpoints. Users can set an alarm to automatically suspend the program execution when a certain time is reached. When program execution is suspended, users can query various performance data and statistics. Users can also clear the alarm.
- Parallel execution time. Reporting the estimated execution time of the program run on a target multicomputer or multiprocessor.
- Sequential execution time. Reporting the estimated execution time of the program run on a uniprocessor computer.
- Reporting execution time of any portion of the program, either sequential or in parallel.
- Reporting the computation time of the program: time the program spent on computation task.
- Reporting the communication time of the program: overhead involved in inter-processor communication such as message sending/receiving and congestion delay.
- Reporting processor utilization.
- Reporting profile of processor utilization as a function of time.
- Reporting process creation/termination information, such as time, processor number, and parent process ID.

- Reporting message send/receive information, such as time, source processor, destination processor, and message length.
- Reporting message routing information, such as path and time traces.

Users can reconfigure the network by specifying different topologies, redefine the virtual-to-physical mapping, and redefine most of the communication parameters in order to study the performance at different levels and from different angles. CPSS also provides the utility to log these performance statistics into files that can be retrieved later for further analysis as the user wishes.

The main data structures related to performance debugging are:

- The global clock: CPSS uses relative timing with a user defined clock cycle. The global clock is advanced to the next quantum only when every application process has finished its quantum. The Global clock simulates the actual elapsed execution time on a real parallel machine. Both parallel and sequential execution times are accumulated based on this global clock.
- Process local clock: each parallel process maintains a local clock using the same clock cycle as the global clock. During parallel execution, each parallel process is given a quantum to run until its quantum expires. The process's local clock is used to keep track of the time spent during the given quantum.
- Network and communication parameters: network and communication parameters are all configurable variables defined in CPSS. The network architecture is defined by network type, network dimensionality, and size of each dimension. Communication parameters are defined in a parameter structure. Different network architectures can be simulated by simply redefining the corresponding variables or parameters.

- Virtual-to-physical mapping table: this table is used to store the data that reflect a specific pattern of virtual-to-physical-architecture mapping. Using this table at run-time, a virtual processor will be mapped to a physical processor on which a parallel process will be running.

3.1.3 Graphical User Interfaces

Due to the complexity and multiple dimensionalities of parallel performance data, CPSS provides graphical user interfaces (GUI) to help programmers in their development and debugging process. The user interface enables the user to interactively communicate with the simulator. The user interface receives parameters and commands from the user, validates the received information, and passes valid parameters and commands to the appropriate module (the CEM, the network module or the debugging monitor). During execution of a parallel program, the user interface is used to interact with the debugging monitor and display performance statistics and debugging information. Program outputs are also transferred from the CEM to the user interface for displaying.

Currently, CPPE runs on UNIX workstations and MS-Windows PCs. The GUI for UNIX workstations is developed with MOTIF toolkit, and the GUI for MS-Windows PCs is developed with MFC (Microsoft Foundation Classes). Figure 4 shows the Windows version of CPPE graphical user interface.

To modularize the development, the user interface code is isolated from the other components in CPPE as much as possible. Since CPPE is continuously under development, modularization helps to de-couple and therefore reduce the dependence among CPPE modules. To support portability, compilation condition flags are used to adapt the simulator to different development environment.

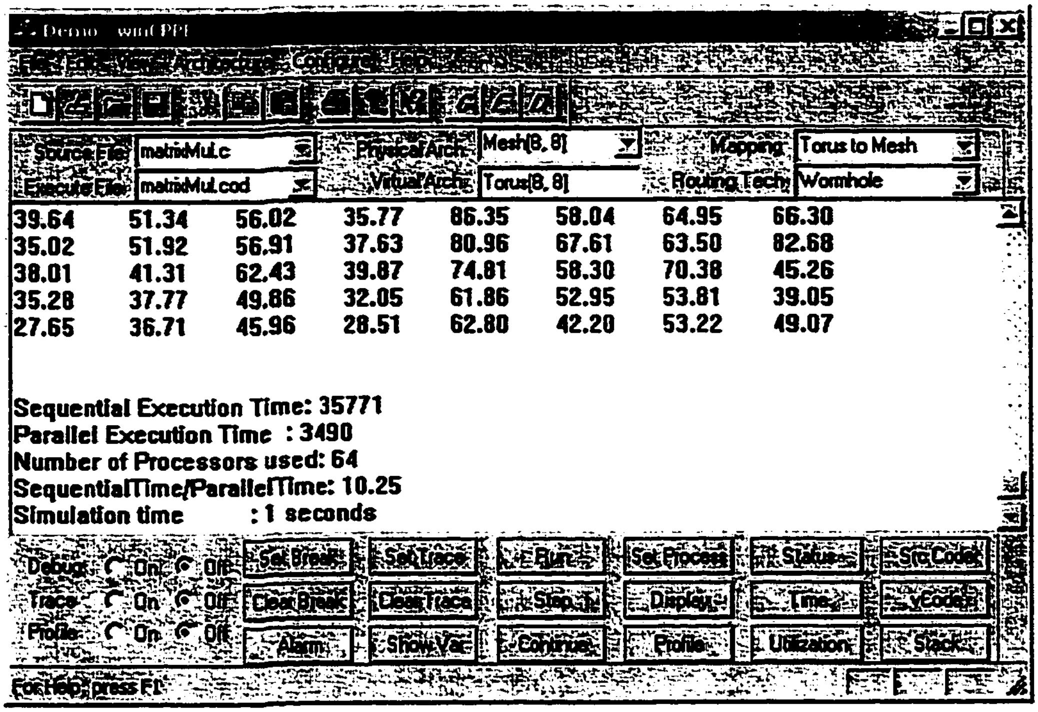


Figure 4 Windows CPPE Graphical User Interface

3.2 Client/Server Architecture of CPPE

The existing CPPE is basically a C/C++ program compiled into a single executable file, which includes the CPCC compiler, the CPSS parallel systems simulator, and a graphical user interface.

Our new distributed version of CPPE is based on the client/server software architecture, as shown in Figure 5. Its user interface will be implemented as a Java client that can run either as a standalone application or as an applet inside a web browser. CPCC and CPSS will be wrapped inside a CPPE CORBA server component, and run on an application server, which may be on the same machine as a web server. When a user launches a CPPE graphical user interface, the thin client will first connect to the CPPE server to validate the identity and password of the user, and then create a small local CPPE stub object representing the CPPE server component. All invocations to CPCC and CPSS will go transparently through the local CPPE stub, the Internet, and the server skeleton code to reach the CPPE server, as shown in Figure 6.

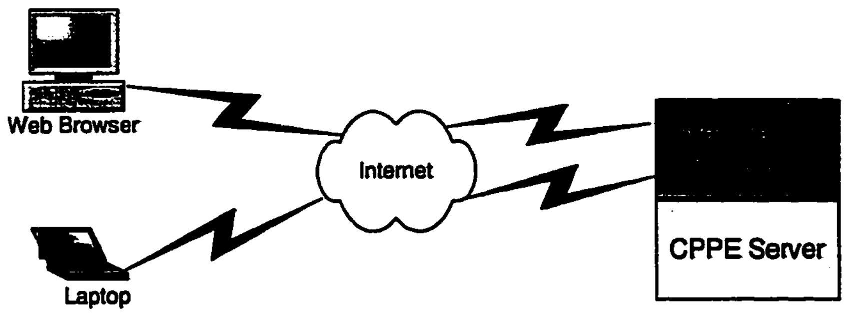


Figure 5 Client/Server Architecture of CPPE

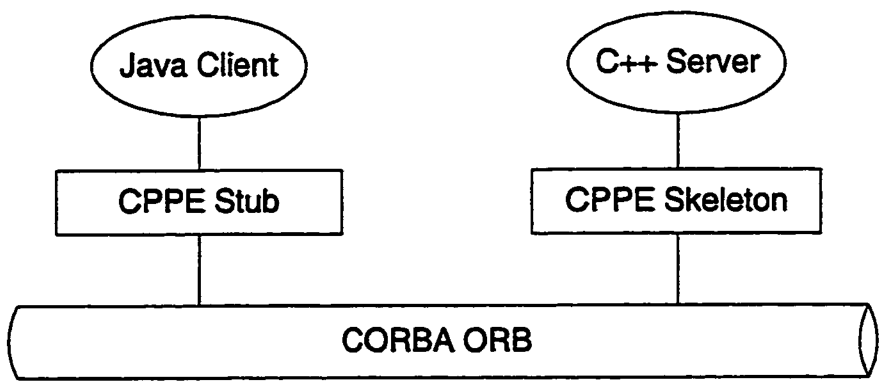


Figure 6 CPPE Stub and CPPE Skeleton

All users need the help of a super-user to open an account on the CPPE server machine, and choose a login name and a password. When a user logs on successfully, a disk directory is created on the server machine named by the user's login name. All the intermediate files needed for the CPPE session will be maintained in that directory. When the user finishes his CPPE session, all of his files as well as his disk directory on the server will be deleted. The user always gets the illusion that all the files of his CPPE session reside on his local machine.

In the following we outline the execution of three major operations: open a source file, compile a source file, and execute a vCode file. The designs for other operations are similar.

3.2.1 Open a Source File

When the user invokes the Open method to open a new source file, the following events will follow:

- A file-choosing window pops up, a local source file (*.c) on the client side can be chosen.
- The chosen source file is transparently transferred over the ORB as a string of characters to the server side and saved in a file of the same name under subdirectory: "~\user," where "user" represents the user's login name.
- As long as the user and CPPE server do not modify the file, CPPE will use its cached server copy of the file for efficiency.
- If CPPE server modifies the file, a copy of the file will be sent back to the client's side to update the client's version.
- If the client modifies the file on his side, he needs to reopen the file to update the server version of the file.

Figure 7 shows the flow chart of the open source file operation.

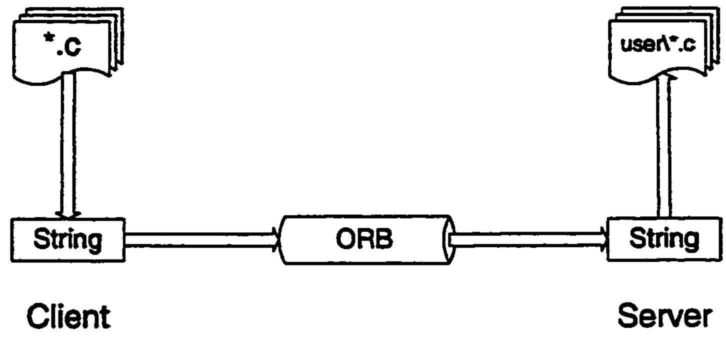


Figure 7 Open a Source File

3.2.2 Compile a Source File

When the user invokes the `Compile` method to let CPCC compile his source file, the following sequence of events follows:

- The server side copy of the source file is compiled and its executable file (*.cod) is generated and saved on the server side under "~\user."
- Compilation messages will be saved on the server side in a temporary output file and the contents of the file will be sent back as a string to the client side to be displayed in the Monitor panel of the CPPE graphical user interface.

Figure 8 shows the flow chart of compiling a source file by the CPPE server.

3.2.3 Execute a vCode File

When the user invokes the `execute` method to let CPSS simulate the execution of a virtual code file generated by CPCC, the following sequence of events follows:

- The vCode file generated by a previous call to CPCC on the server side is executed.

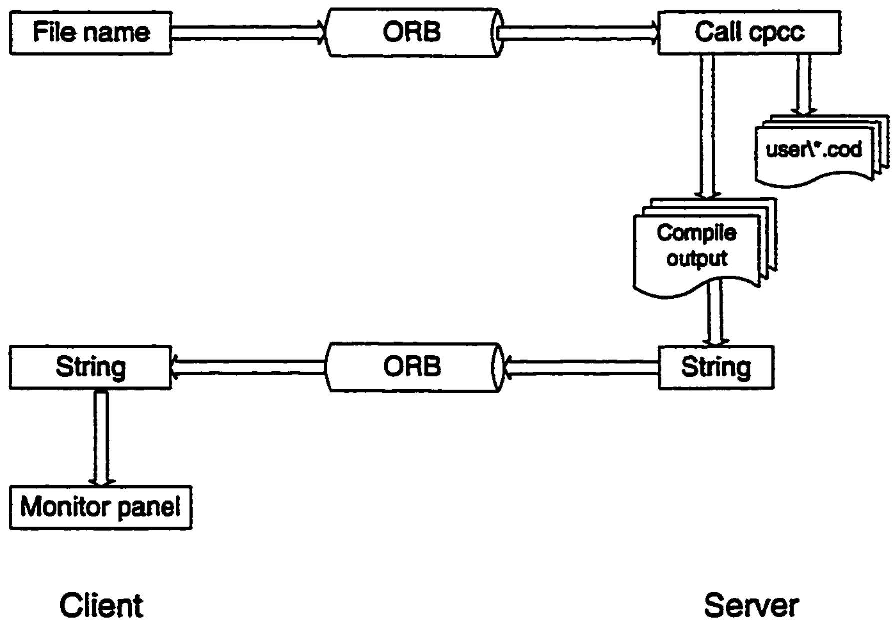


Figure 8 Compile a Source File

- Execution results will be saved on the server side in a temporal output file, and the contents of the file will be sent back as a string to the client side to be displayed in the Output panel of the CPPE graphical user interface.
- The user can save the execution results displayed in the Output panel in a text file on the client side with a name of his choice.

Figure 9 shows the flow chart of executing a vCode file on the CPPE server.

3.3 CPPE Interface Design and Implementation

The key component for the design of a CORBA-based client/server application is the abstraction of the public interface from an application, and the accurate specification of it in CORBA's Interface Definition Language (IDL). This IDL specification is the contract between the clients and the server for method invocations, and the foundation to support network interoperability.

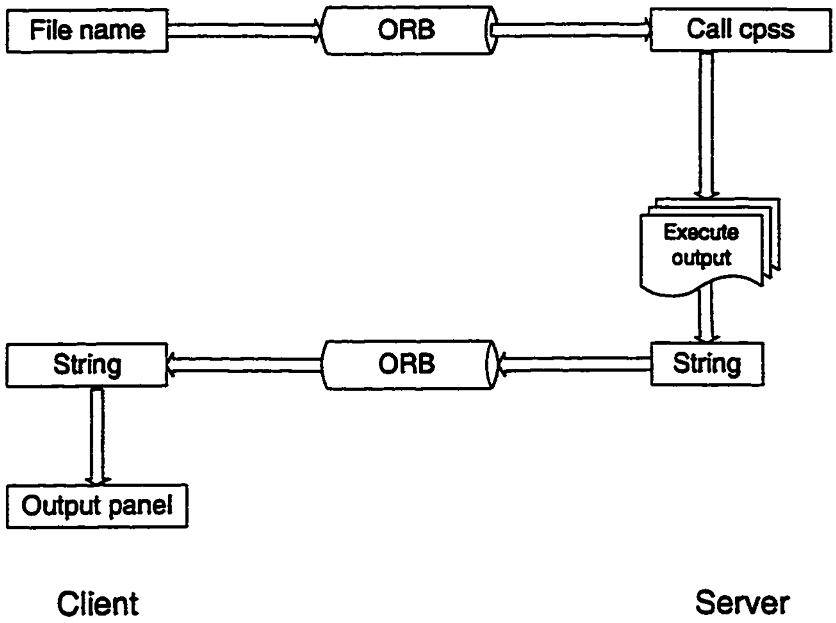


Figure 9 Execute a vCode File

An IDL specification is made up of a module containing a list of type definitions, attributes, and interfaces. Each interface contains a list of type definitions, attributes, and operations. Each operation may have a return type, and a sequence of parameters each with its own data type. Each parameter also has a direction: *in* for sending data from a client to the server, *out* for sending data from the server to a client, and *inout* for both.

In the following subsections, we first specify the mapping of data types among IDL, C++, and Java. We then abstract and specify the CPPE interface in IDL, and briefly explain the key operations of the CPPE interface.

3.3.1 Data Type Mappings among IDL, C++, and Java

The data types defined in the IDL are platform-independent. On the other hand, the same type name may have different meanings in different languages. To support network interoperability, all data passing through the ORB must first be transformed from the client's programming language data type into IDL's equivalent data type on the sender's side, and then be transformed from IDL's platform-independent data type to the equivalent receiver's programming language data type. Sometimes data types can only be approximated since no equivalent data types can be found in two languages.

In a typical CORBA application development process, we should first use object-oriented analysis to identify the interfaces, specify them in IDL, and then map them to interfaces and classes in the implementation programming languages with CORBA IDL compilers. But this process does not apply to this project, since we need to take full advantage of the existing legacy code of CPPE, which was implemented in C++. Therefore we have to first find out all C++ data types used by CPPE interface methods/functions, and then find out the IDL data types that will map to these C++ data types. After this process, we can use IDL compilers to map all C++ data types in CPPE to data types in any programming language supported by CORBA.

3.3.1.1 C++ Data Types in CPPE

The CPPE has the following data type that need to be mapped from C++ to IDL, and then mapped 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

3.3.1.2 Data Type Mappings among IDL, C++, and Java

We can use the following table to map the above data types. These mappings are based on our study of the language specifications of CORBA IDL, C++, and Java.

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 the same name as enum type
`struct`	`struct`	`struct`	Java class with the same name as struct type

Table 1 C++ to IDL Mapping and IDL to Java Mapping

3.3.2 CPPE Interface Specification

We thoroughly studied the CPPE design and implementation, and identified all public data members and methods invoked by CPPE graphical user interfaces. We then specify an IDL module mCppe that includes a list of type definitions and one CORBA interface Cppe.

In general, we first transform the legacy code so that the user interfaces only interact with the business logic through methods (no direct modification of internal variables, for example). Then all these methods invoked by user interfaces become the operations in the CORBA IDL interface specification for this distributed object.

3.3.2.1 IDL Specification of CPPE Server

The following is a partial listing of our IDL specification for the CPPE server.

```
module mCppe {
  typedef char SrcTable[2000][128];
  typedef short PhyDimSizes[4];
  enum NetworkTypesEnum {WrongNetType, PckSwitch, Wormhole, SimuPack,
                         CalcDist};
  enum MappingEnum {DefaultMap, IdentityMap, RandomMap,
                    Ring2lineMap, Torus2meshMap, UserdefinedMap };
  enum ArchTypeEnum {LINE, RING, MESH, TORUS, HYPERCUBE,
                     FULLCONNECT, SHARED};
  ......
  interface Cppe {
    void compile(in string file_name, out string print_out);
    void execute(in string file_name, out string print_out);
    void send2server(in string userDir, in string file_name,
                     in string file_in);
    void send2client(out string print_out);
    void sendVcode2client(in long offset, in string file_name,
                          out string file_out);
    void alarmProc(in string flagState, in float time);
    void archProc(in short topo, in short dim, in IntArray dimSizes);
    void clearBreakProc(in short number);
    void clearTraceProc(in string argStr);
    void codeProc(in short from, in short to);
    void continueProc();
    void curArchName(in boolean isPhysical, out string archName);
    void debugProc(in string flagState);
    void displayProc();
    void helpProc();
    void initialSetup();
    void listProc(in short fromLine, in short toLine);
    void loadProc(in string file_name);
    void mapProc(in short selection, in string mapString);
    void netTypeProc2(in string flagState);
    void netSetLinkDelay(in short delay);
    void profileProc(in string flagState, in short from, in short to,
                     in short step);
    void quitProc();
    void runProgProc();
    void setBreakProc(in short lineNbr, out boolean flag);
```

```
    void setStepProc(in short stepProcNbr);
    void showVarProc(in string varName, in short processNum,
                     in short from, in short to);
    void stackProc(in short from, in short to);
    void statusProc(in short fromProcess, in short toProcess);
    void stepProc(in short stepNum);
    void timeProc();
    void traceProc(in string varName, in short processNum,
                   out short flag);
    void utilizationProc(in short fromProcessor, in short toProcessor);
    void varySpeedProc(in string flagState, in short seed);
    ......

    attribute char alarmOn;
    attribute char codeLoaded;
    attribute char debugOn;
    attribute char initialized;
    attribute char logOn;
    attribute string mapFuncStr;
    attribute char profileOn;
    attribute char varySpeedOn;
    ......

  };
};
```

3.3.2.2 Key Operations

In this subsection we explain the semantics of the key operations of the CPPE interface.

- *void compile(in string file_name, out string print_out)*

  CPPE client GUI passes the file name of a source file, which has already been opened and copied to the CPPE server, to the CPPE server through parameter *file_name*; the CPPE server compiles the file with CPCC, and passes the CPCC compilation messages back to CPPE client GUI through parameter *print_out*.

- *void execute(in string file_name, out string print_out)*

CPPE client GUI passes the file name of a vCode file, which has already been generated on the CPPE server, to the CPPE server through parameter *file_name*; the CPPE server executes the file with CPSS, and passes the CPSS execution results back to CPPE client GUI through parameter *print_out*.

- *void send2server(in string user_dir, in string file_name, in string file_in)*

  CPPE client GUI passes a file from the client's local file system to the CPPE server: the relative path of the file through parameter *user_dir*; the file name through parameter *file_name*; and the contents through parameter *file_in*. The CPPE server will save the file under "~\user" with the specified relative path, file name, and file contents.

- *void send2client(out string print_out)*

  The CPPE server passes a message through parameter print_out to the Monitor panel or the Output panel of the CPPE client GUI.

- *void sendVcode2client(in long offset, in string file_name, out string file_out)*

  The CPPE client asks for a vCode file from the CPPE server. The CPPE client specifies the starting line number of the vCode file through parameter *offset*, and the file name through parameter *file_name*. The CPPE server in turn sends back the contents of the requested vCode file through parameter *file_out*.

- *void alarmProc(in string flagState, in float time)*

  The CPPE client notifies the CPPE server whether the alarm should be on or off through parameter *flagState*; and if the alarm should be on, then when the alarm should ring and break the execution of the parallel program for debugging through parameter *time*.

- *void archProc(in short topo, in short dim, in IntArray dimSizes)*

  The CPPE client notifies the CPPE server that the simulated physical architecture should be of topology specified by parameter *topo*, of dimensionality specified by parameter *dim*, and with lengths of the dimensions specified by parameter *dimSizes*.

- *void setBreakProc(in short lineNbr, out boolean flag)*

  The CPPE client notifies the CPPE server that a breakpoint should be set on source code line *lineNbr*; the CPPE server will return a Boolean value through parameter *flag* indicating whether the specified source line can be set as a breakpoint (whether it is executable).

- *void traceProc(in string varName, in short processNum, out short flag)*

  The CPPE client notifies the CPPE server that variable varName of process processNum should be traced: the execution should be stopped for debugging if the variable is accessed by the specified process. The CPPE server will return a status through parameter flag indicating whether the operation is successful (whether the variable is active in the specified process).

- *void showVarProc(in string varName, in short processNum, in short from, in short to)*

  The CPPE client requests the value of variable *varName* of process *processNum*; if the variable is an array, then parameters *from* and *to* are used to specify an optional index range.

- *void utilizationProc(in short fromProcessor, in short toProcessor)*

The CPPE client requests the processor utilization statistics for physical processors of ID numbers in the range from *fromProcessor* to *toProcessor*.

- *void mapProc(in short selection, in string mapString)*

  The CPPE client specifies a library process mapping through parameter *selection* or specifies a user-specified mapping through parameter *mapString*. This mapping determines how the virtual processors are mapped to the simulated physical processors, or how the processes are mapped to the simulated physical processors.

- *void varySpeedProc(in string flagState, in short seed)*

  The CPPE client requests that the CPPE server use randomized speeds for the simulated physical processors to help find timing-sensitive bugs. Parameter *flagState* specifies whether this feature should be turned on or off. Parameter *seed* specifies a seed for the random number generator.

3.4 Wrapping CPPE Inside a CORBA Component

3.4.1 Tie vs. Inheritance

There are two basic approaches for CORBA component implementation: inheritance and delegation.

With either approach, an IDL interface will be used to generate a C++ abstract class or a Java interface with the same name and containing all the method signatures for the operations in the IDL interface.

With the inheritance approach, a CORBA implementation class will inherit *CORBA::Object*, which provides basic CORBA functionalities, and inherit the C++

abstract class or implement the Java interface. The programmer is responsible to fill in business logic implementation for each of the abstract methods in this class.

With the delegation approach, the CORBA precompiler will generate a template *tie* class that inherits *CORBA::Object*, inherits the C++ abstract class or implements the Java interface, and accept an implementation object as a template argument. The class for the implementation object is a non-CORBA class, implementing the same interface. Inside the tie class, each method corresponding to an interface operation is implemented by invoking the same method against the embedded implementation object, passing all arguments unmodified.

3.4.2 CPPE Server Component Based on Delegation

CPPE legacy code is a non-CORBA C++ program. To minimize the modification of the legacy code, we use CORBA tie (delegation) approach to wrap the non-CORBA CPPE object inside a CORBA distributed component. Figure 10 shows the flow chart of the process and the relationships among the related classes.

- Visibroker CORBA precompiler *vbcpp* reads interface specification *cppe.idl*, and generates an abstract class *mCppe::Cppe*.

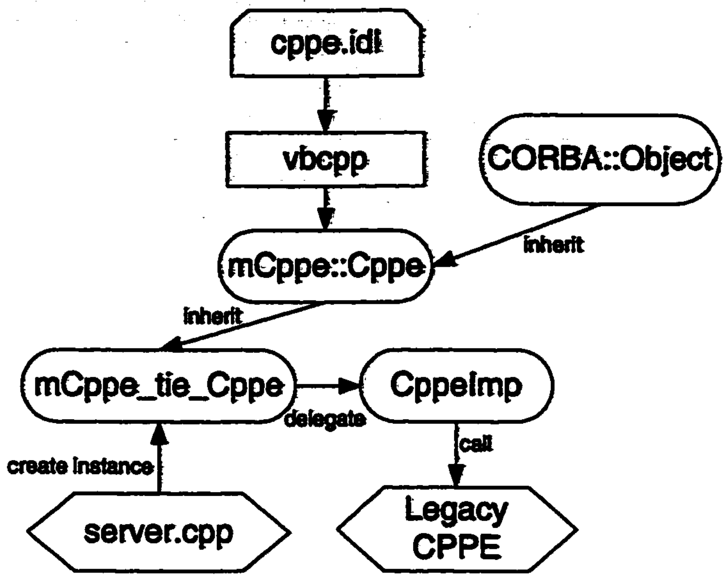


Figure 10 CPPE Delegation Structure

- *mCppe::Cppe* inherits class *CORBA::Object*, which implements basic CORBA functionalities.

- Class *CppeImp* is created to implement each of the methods in interface *mCppe:Cppe*. For each of such methods, the implementation is based on invoking existing legacy CPPE C++ code. *CppeImp* itself is a non-CORBA normal C++ class.

- Precompiler *vbcpp* also generates a C++ template class *mCppe_tie_Cppe*, as outlined in Figure 11. Class *mCppe_tie_Cppe* inherits class *mCppe::Cppe*, therefore it is a CORBA object implementing interface *Cppe*. *MCppe_tie_Cppe* accepts an implementation object (of type *CppeImp* in our case) as its template argument, which is not a CORBA object and must implement interface *Cppe*. The constructor of *mCppe_tie_Cppe* saves the reference of the implementation object in one of its private data members. For each of the methods of *mCppe_tie_Cppe* corresponding to a *Cppe* operation, its implementation just invoke the same method against the *CppeImp* implementation object, passing all arguments unmodified.

```
template <class T>
class mCppe_tie_Cppe : public mCppe::Cppe {
private:
  T& _ref;

public:
  mCppe_tie_Cppe(T& _t) :_ref(_t) { }
  void cpss_main() { _ref.cpss_main(); }
  void send2server(const char* userDir, const char* file_name,
                const char* file_in) {
    _ref.send2server(userDir, file_name, file_in);
  }
  void send2client(char*& print_out) {
    _ref.send2client(print_out);
  }
  ......
}
```

Figure 11 CORBA Tie Class

- The function *main()* of file *server.cpp* initializes Object Request Broker (ORB) and Basic Object Adapter (BOA), creates an instance of implementation class *CppeImp*, passes the new object as an argument to the instantiation of a new object of type *mCppe_tie_Cppe*, which is the CORBA object that we post on the Internet to serve the remote CPPE clients. Figure 12 shows the listing of *server.cpp*.

```
// server.cpp
#include "Cppeimp.h"

int main(int argc, char* const* argv) {
  try {
    // Initialize the ORB.
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    // Initialize the BOA.
    CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
    // Create the Cppe object.
    CppeImpl Cppe;
    // Create the delegator
    mCppe_tie_Cppe<CppeImpl> tieServer(Cppe, "CPPE");
    // Export the newly created object.
    boa->obj_is_ready(&tieServer);
    // Ready to service requests
    boa->impl_is_ready();
  }
  catch(const CORBA::Exception& e) {
    cerr << e << endl;
  }
  return (0);
}
```

Figure 12 CPPE Server Main()

3.4.3 Generation of CPPE Java Client and C++ CORBA Server

Figure 13 shows the major files for the CPPE Java client and the CPPE C++ server, and the process in which they are generated.

3.4.3.1 CPPE Java Client Generation

- Apply Visibroker for Java precompiler *idl2java* on CPPE idl specification file *cppe.idl* to generate Java source files including

- *_st_cppe.java*: stub class representing the remote CPPE server, responsible for network connectivity and parameter marshaling.
- *cppeHelper.java*: helper class for CPPE component providing utilities to bind to CPPE server and casting of generic references to those for CPPE component.
- *cppeHolder.java*: a utility class to support *out* or *inout* parameters of CPPE type. Since CORBA server cannot modify the value of an *out* or *inout* parameters of type *Cppe* in IDL specification, such parameters will be modified to be of type *cppeHolder* class after the language mapping. The client will pass an object of such Holder class as argument, and the server can modify the *value* field of this argument object to pass value back to the client.

- Write *cppeFrame.java* to implement Java GUI as a standalone thin-client or as an applet. A CPPE stub object is created in cppeFrame.java to represent the remote CPPE server object.
- Use Visibroker batch file *vbmake* to compile all Java files mentioned above as well as some other utility files. Byte code file *cppeFrame.class* is the file for launching CPPE GUI.

3.4.3.2 CPPE C++ Server Generation

- Apply Visibroker for C++ precompiler *idl2cpp* on CPPE idl specification file *cppe.idl* to generate C++ source files including
    - *cppe_c.cpp*: C++ implementation file for class *mCppe::Cppe*, which among other things implements the skeleton object for the CPPE server.

This skeleton object sits between ORB and the CPPE server object, and perform the necessary data marshaling and unmashaling.

  - *cppe_s.cpp*: C++ implementation file for class *mCppe_tie_Cppe*

- Write *cppeimp.h* and *cppeimp.cpp*, the header file implementation file for the CPPE implementation class. This class implements all public methods defined by the CPPE interface, and invoke the legacy CPPE code for its method implementation.
- Write *server.cpp*, which has the *main()* for the CPPE server.
- Use Microsoft Visual C++ command *nmake* to execute a makefile to compile the above C++ source files into executable *server.exe*.

3.5 Architecture for Execution both as an Application and as an Applet

Our objective is to support CPPE client both as a standalone application and as an applet to run inside a web browser.

An applet has its special structure different from applications. An applet usually does not have a *main()* method. It must inherit class *javax.swing.Japplet*. An applet usually implements the following methods:

- *init()*: It is used to initialize the applet. It is called immediately after the applet is downloaded from a web server.
- *start()*: It is called after the applet is initialized, and also every time the applet becomes visible again inside a web browser after it was hidden for a while. It is mainly used to restart the execution of some threads.

- *stop():* It is called when the applet becomes invisible inside a web browser, maybe due to visiting a different HTML page. It is mainly used to suspend the execution of some threads to conserve computing resources.
- *destroy():* It is called when the web browser is shut down to release all resources used by the applet.

For easy maintenance, it is better to have a single version of CPPE Java source code. Therefore we designed the following framework for the *CppeFrame* class, as shown in Figure 14, so it can be run both as an application and as an applet.

3.6 General Guideline for Legacy Code Adaptation

The adaptation of CPPE represents a general approach to adapt legacy code for server hosting. In this subsection we outline this general approach and its scope.

Our approach can be used to adapt any legacy code that has been implemented in language C, C++, Ada, COBOL, Smalltalk, or Java.

The following are the major steps of the adaptation:

1. Identify the methods or free functions of the legacy code that are called by the user interfaces.
2. Identify all the data types used by these methods or free functions.
3. Find CORBA IDL type definitions that will map to the above types in the implementation language of the legacy code.
4. Use the CORBA IDL language to specify a CORBA interface including all the above methods or free functions.

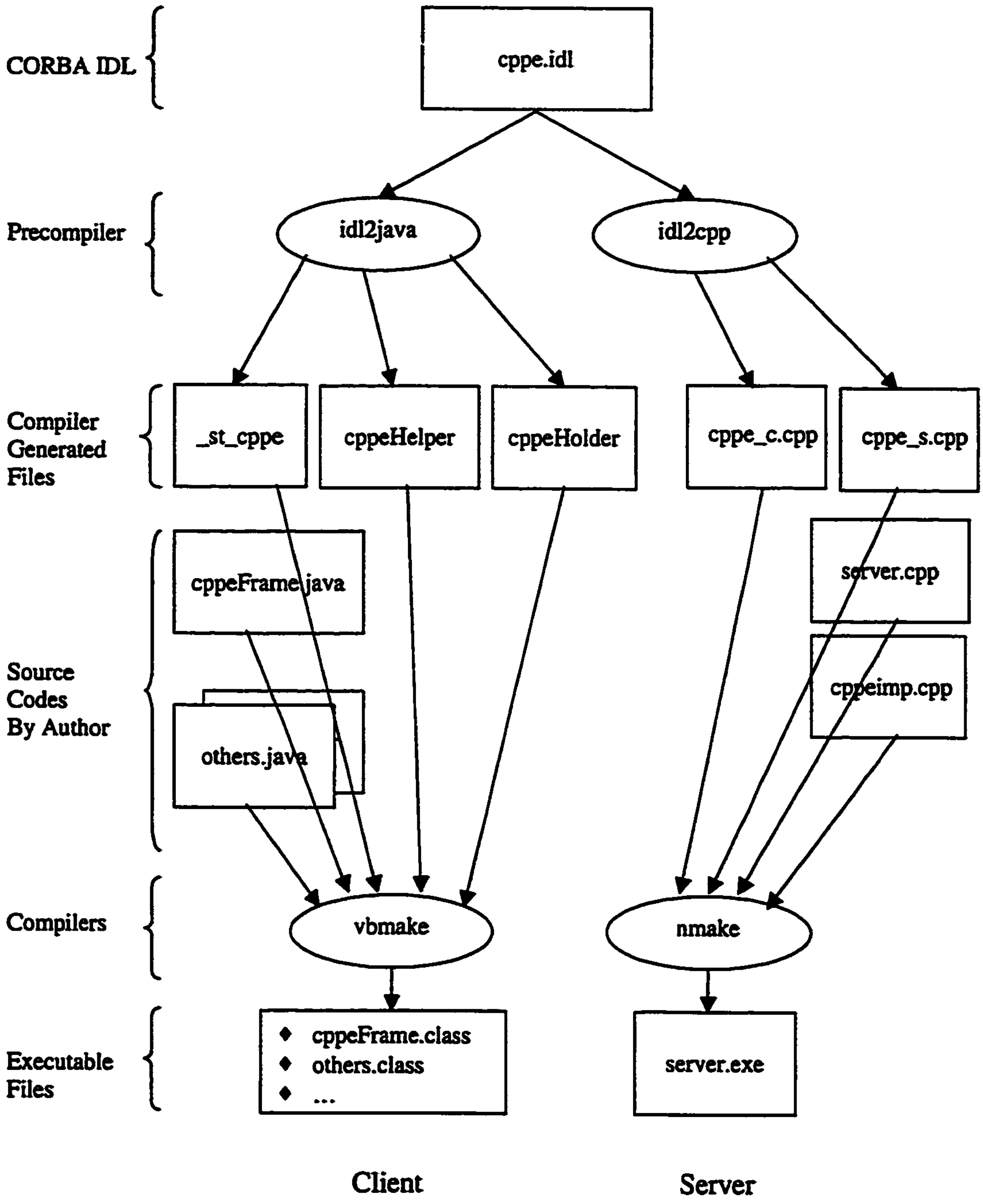


Figure 13 Java Client Files and C++ Server Files

```
public class CppeFrame extends JApplet {
  public CppeFrame() { ...... }
  public void init() {
    // initialize the GUI
  }
  public void start() { ...... }
  public void stop() { ...... }
  public void destroy() { ...... }

  public static void main(String[] args) {
    // Create a frame
    Frame frame = new Frame("CPPE Client");

    // Create an instance of CppeFrame
    CppeFrame applet = new CppeFrame();

    // Add the applet instance to the frame
    frame.getContentPane().add(applet,
          BorderLayout.CENTER);

    // Invoke init() and start()
    applet.init();
    applet.start();

    // Display the frame
    frame.setSize(680, 550);
    frame.setVisible(true);
  }
}
```

Figure 14 Client Structure Serving as Both Application and Applet

5. Use the CORBA precompiler to compile the above IDL interface specification into the implementation language of the legacy code. Throw away the client-side code automatically generated by the precompiler. Use the delegation approach to use the existing legacy code to support the implementation of the CORBA server object.

6. Write a server program to create instances of the CORBA server object, register them with object adapters, and wait for client calls.

7. Use the CORBA precompiler to compile the above IDL interface specification into the implementation language of the client application. Throw away the server-side code automatically generated by the precompiler. In the client code, create a local instance of the CORBA stub class generated by the precompiler, and treat it as if it were the remote CORBA server object.

3.7 Summary

In this chapter we have studied the architecture of the legacy CPPE, and provided the design and implementation highlights of our distributed CPPE project. Since CPPE is a typical legacy code, and the CORBA delegation approach has basically no special requirements for legacy code, our approach discussed in this chapter can be used to adapt any legacy code for server hosting as long as there is a language mapping between CORBA IDL and the implementation language of the legacy code.

CHAPTER 4 CPPE JAVA GUI DESIGN

So far we have finished the design and implementation of the CPPE server object. But our users will use Java applet or thin-client applications to support graphical user interfaces to get services from the CPPE server object transparently. Since CPPE is a comprehensive software development environment, we need to use Java to reimplement all graphical user interfaces for CPPE.

In this chapter we provide highlights of sample graphical user interfaces that we have reimplemented in Java. While they do not contribute original designs, they represent a big proportion of design and implementation work for this project. This is typical of any legacy code adaptation project for server hosting.

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, a user can accomplish the whole development process from source code editing, compilation, execution, debugging, and performance profiling. The design goal of the GUI is to make the developing and debugging process easy and intuitive for the parallel application programmers.

CPPE needs two major functions to be a complete development environment: the compilation of the application programs written in CPC; and execution of parallel programs on simulated parallel systems.

The compilation function in CPPE is provided by CPCC. If used from the 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 to be called from the CPPE client user interface. A generic output function is needed to replace the original output function to display the program output into a designated text field in the client CPPE GUI.

CPSS is the parallel systems simulator that includes the functionalities of network module, code execution module, and debugging monitor. When used from the command line, CPSS has an *interpret()* function that accepts the user commands and invokes the corresponding functions from the network module, the code execution module, or the debugging monitor. The function *interpret()* will not be used in the distributed version of CPPE. Instead, CPSS functions can be invoked directly from the GUI components. The same generic output function can be used to display the program output into designated text field in the client CPPE GUI.

4.2 Selected User Interfaces

4.2.1 Main Frame

Figure 15 shows a typical layout of the Java GUI. The window with the title *CppeFrame* is the main frame of the CPPE. From top to bottom, there are the following widgets:

- Menu (including sub-menu) with shortcut keys. The menu provides the comprehensive selection for CPPE functionalities. Refer to Figure 16.

- Floating toolbar with icons and shortcut keys. The buttons in the toolbar can be used to invoke the most frequently used functions in CPPE, such as opening and compiling a source file, or executing a vCode file.
- 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 16 shows the main frame when the debug pane is hidden.
- Combo box pane.
- Status text field.

The above GUI widgets are implemented in Java/Swing.

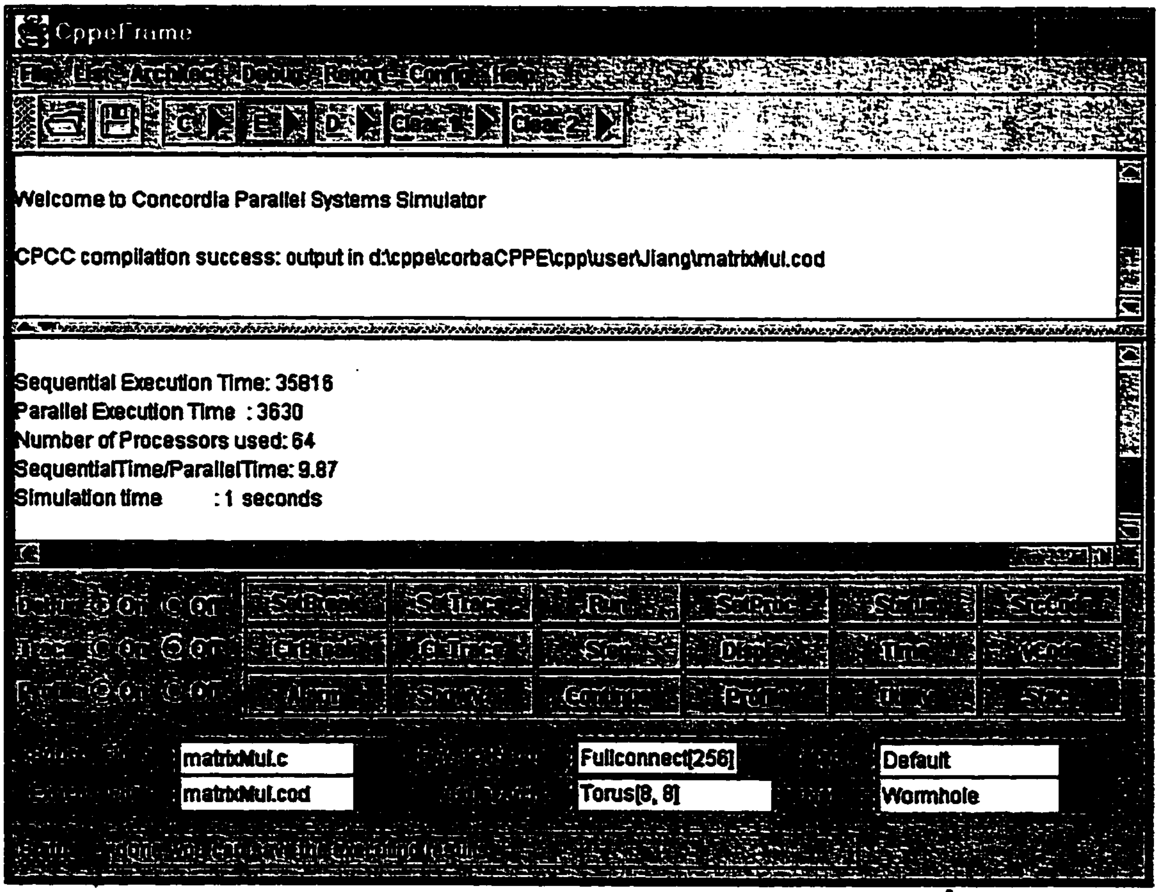


Figure 15 Main Frame

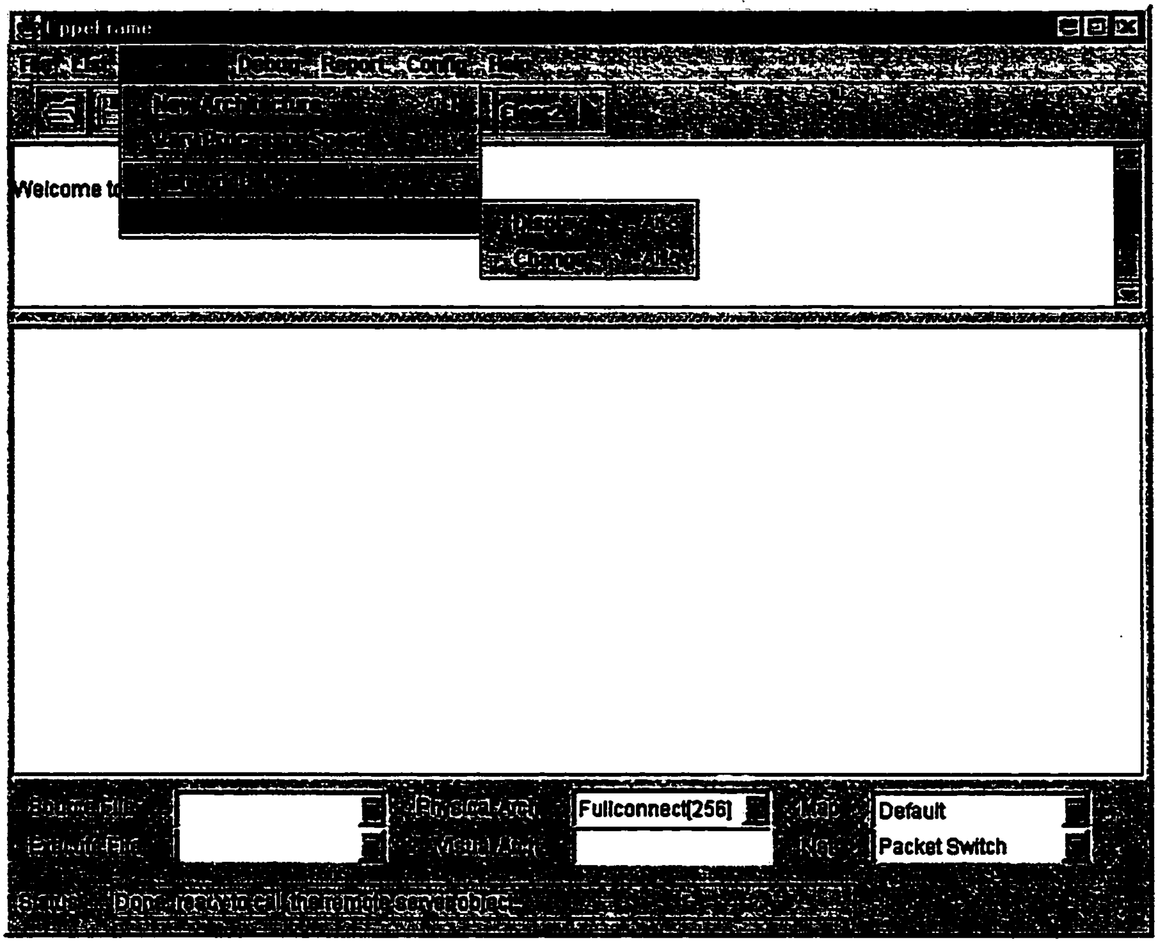


Figure 16 Menu in Main Frame

4.2.2 Open and Save Windows

The window with the title *Open* is used to open a file from local machine, as shown in Figure 17. It is a Java Swing utility. Users can browse the local file system by clicking the combo box. Users can select a folder or a file by double-clicking on the desired item, or clicking on the desired item and pressing the *Open* button. If users select a file, then this file's name will be added in the source file combo box and become current one in the main frame, as shown in Figure 15.

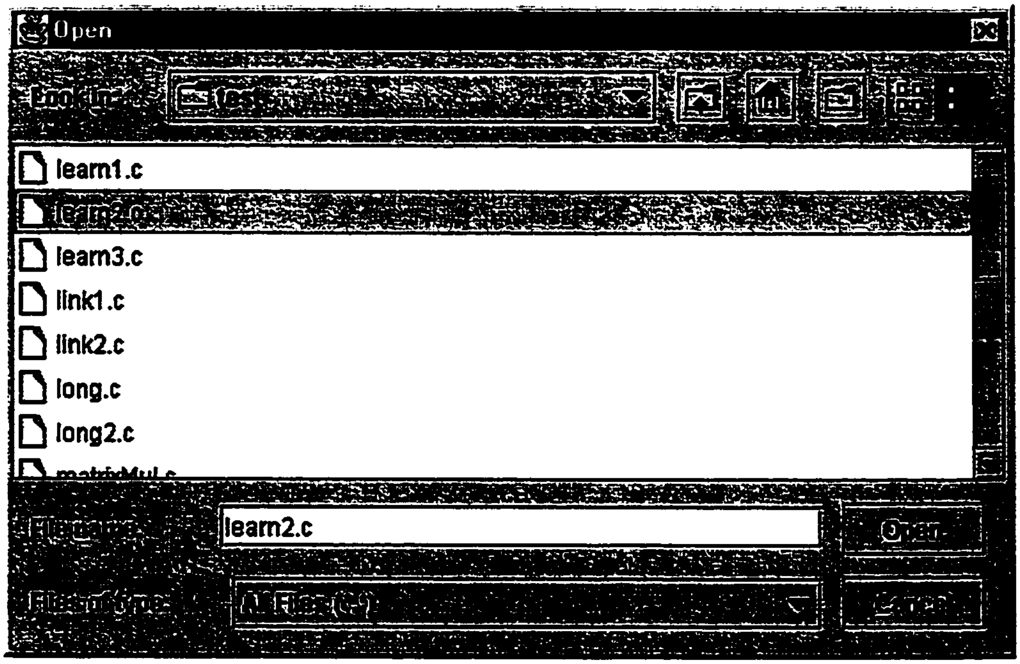


Figure 17 Open Window

The window with the title *Save* is used to save a file to the local machine, as shown in Figure 18. Users can browse the local file system by clicking the combo box. The user can select a folder by double-clicking on the desired item. After giving a file name, the user can press the *Save* button to save the contents in the text area of the main frame, as shown in Figure 15.

4.2.3 Source Code Window

The window with the title *SourceShell* is the source code listing window, as shown in Figure 19. 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, called *source list*, is used to display the source code with line numbers; and the list in the lower part of

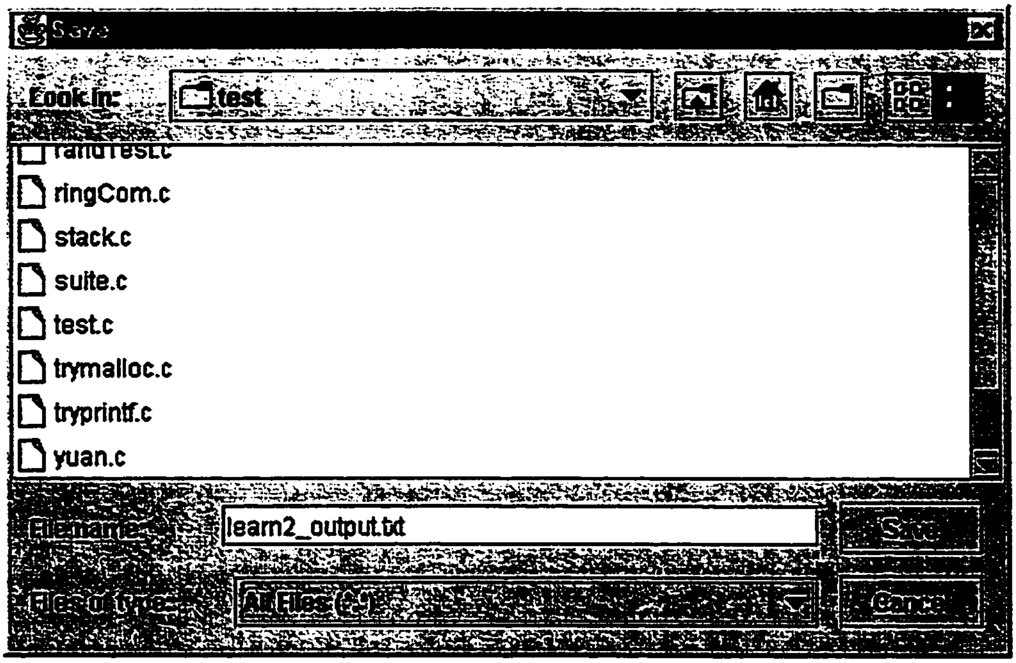


Figure 18 Save Window

the window, called *breakpoint list*, is used to display the source lines of the breakpoints. 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 the *Break* button. As a result, 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 click on the desired source line and press the *UnBreak* button. As a result this source line will be deleted from the *breakpoint list*.

By default, lines 1 to 20 of the source file will be displayed. If users enter invalid data, such as "-1" or "abc," in the fields, then the invalid data will be replaced by the default values automatically. If the user enters a number bigger than the maximum line number of the vCode in the second text field, then the invalid number will be replaced by the maximum line number automatically.

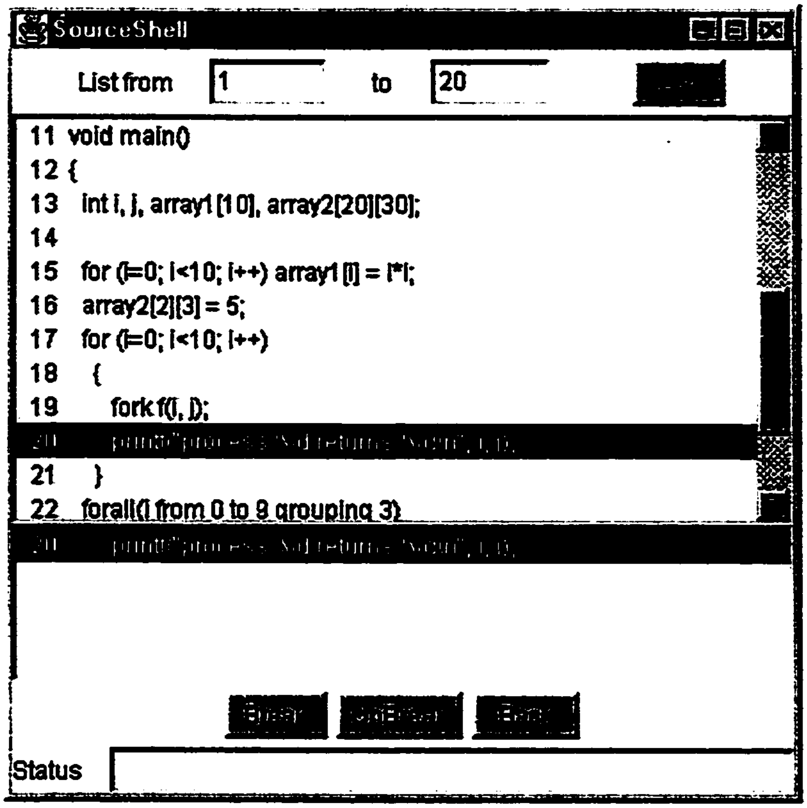


Figure 19 Source Code Window

4.2.4 Step Window

The window with the title *StepShell* is the window for source code stepping. After setting a break or a few breaks, users can debug the code by pressing the *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 users press the *Step* button or the *Continue* button.

4.2.5 Step Size Window

The window with the title *Step Size Window* is used to set the step size: how many consecutive source lines will be executed each time the user clicks on the *step* button.

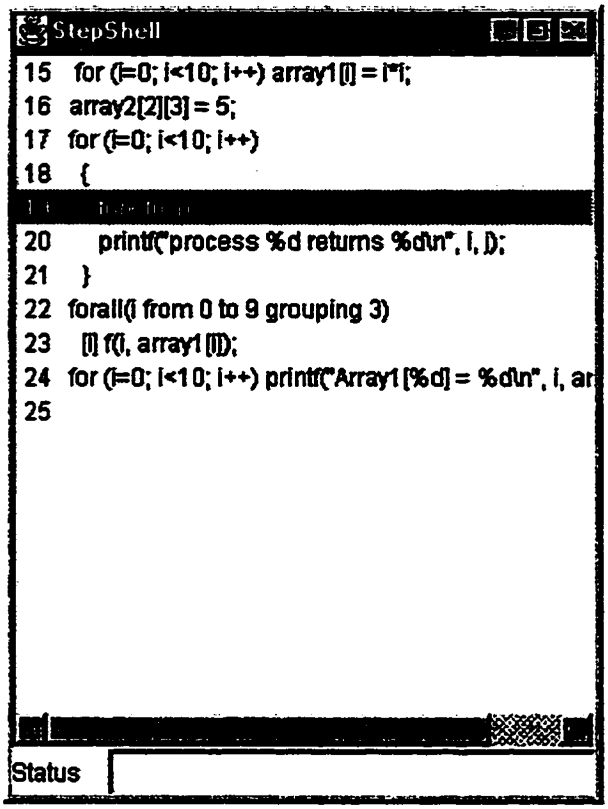


Figure 20 Step Window

The window is shown in Figure 21. Users can specify a step size in the text field. To update the step size, users can press the *ok* button. To keep previous value, press the *Cancel* button, and this window will be closed.

By default, the step size displayed is 1. If users enter invalid data, such as "-1" or "abc," in the field, then the invalid data will be replaced by the default value automatically. At the same time, the status field will show users some warning message.

4.2.6 Trace Window

The window with the title *TraceDialog* is used to set trace variables in a certain process, as shown in Figure 22. The *Trace Variable List* in the window is used to display the

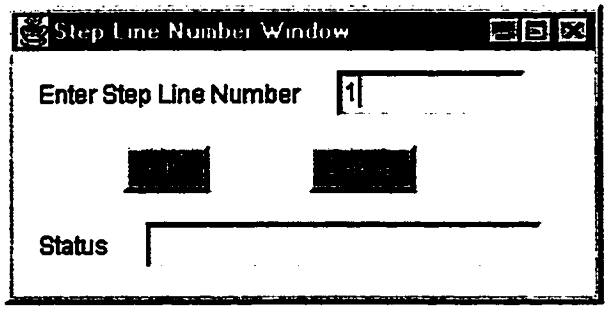


Figure 21 Step Size Window

variable names with their associated process numbers. Users can set a variable to trace by pressing the *Trace* button. To clear a variable, users can double-click the desired line in the *Trace Variable List*, or click on the desired line and press the *UnTrace* button, and this line will be deleted from the *Trace Variable List*. By pressing the *back* button, this window will be closed.

By default, the value of the Trace Process field is 0. If users enter invalid data, such as "-1" or "abc," in the field, the invalid data will be replaced by thedefault value automatically. At the same time, the status field will show users some warning message.

4.2.7 Show Variable Window

The window with the title *ShowVarDialog* is used to display the value of a variable in a certain process, as shown in Figure 23. For an array variable, users can set the index range. To update the new settings, users can press the *ok* button. To keep previous settings, press the *Cancel* button, and this window will be closed. After the execution is suspended, users can use this Show Variable Dialog Window to select a variable to show its current value. This value will be displayed in the upper text area in the main frame, as shown in Figure 15.

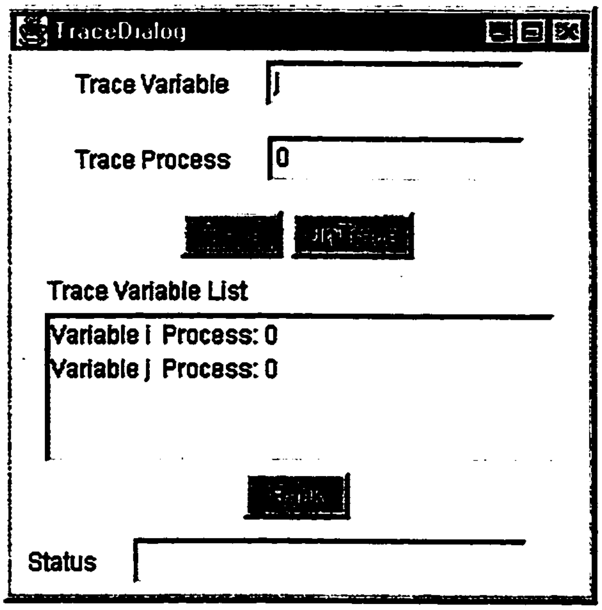


Figure 22 Trace Window

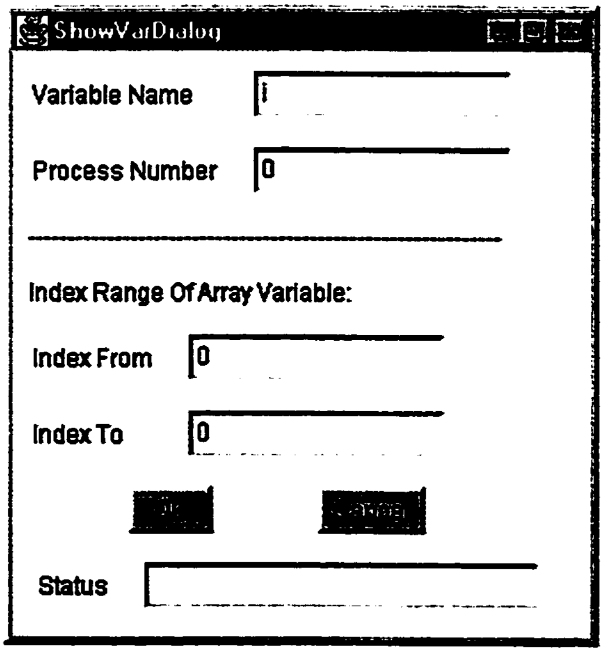


Figure 23 Show Variable Window

4.2.8 Alarm Window

The window with the title *AlarmDialog* is used to set alarm ON or OFF by clicking on the corresponding ratio button, as shown in Figure 24. An alarm time in basic simulation time units can be entered to set the alarm time. To confirm the new settings, users can press the *ok* button. To keep the previous settings, press the *Cancel* button, and this window will be closed.

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

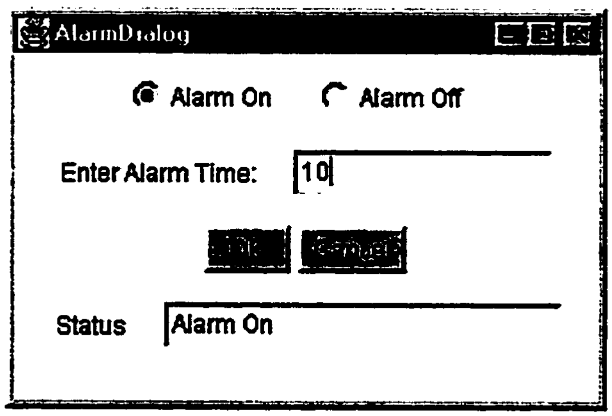


Figure 24 Alarm Window

4.2.9 Profile Window

The window with the title *ProfileDialog* is used to profile the utilization of a sequence of processors with the specified time interval, as shown in Figure 25. To confirm the new settings, users can press the *ok* button. To keep the previous settings, press the *Cancel* button, and this window will be closed.

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

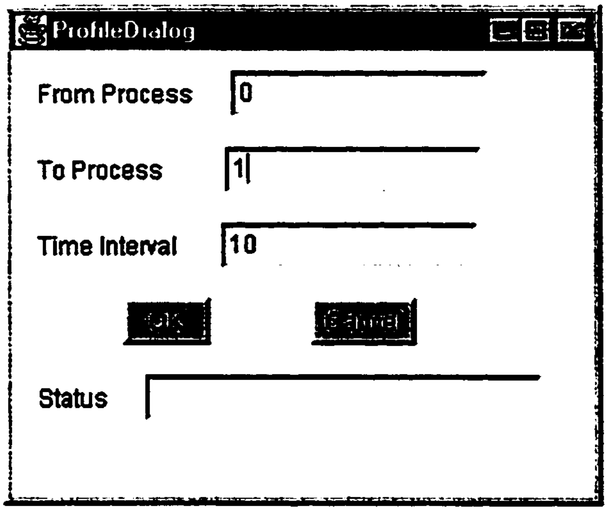


Figure 25 Profile Window

4.2.10 Status Window

The window with the title *StatusDialog* is used to show the status of some processes, as shown in Figure 26. To confirm the new settings, users can press the *ok* button. To keep the previous settings, press the *Cancel* button, and this window will be closed.

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

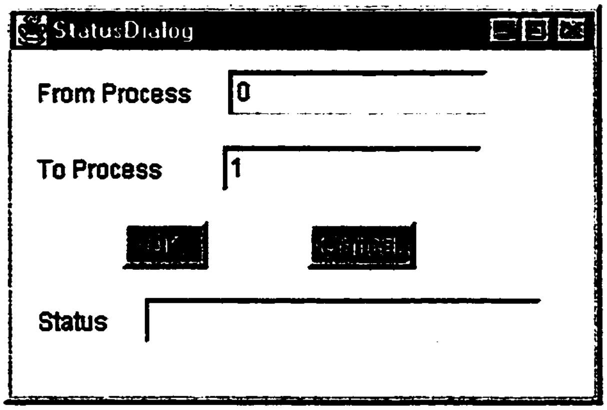


Figure 26 Status Window

4.2.11 Utilization Window

The window with the title *Processor Utilization* is used to check the utilization of some processors, as shown in Figure 27. To confirm the new settings, users can press the *ok* button. To keep the previous settings, press the *Cancel* button, and this window will be closed.

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

4.2.12 Code Window

The window with the title *vCode window* can be used to list a range of vCode lines, as shown in Figure 28. Users can enter the starting and ending line numbers of the vCode to

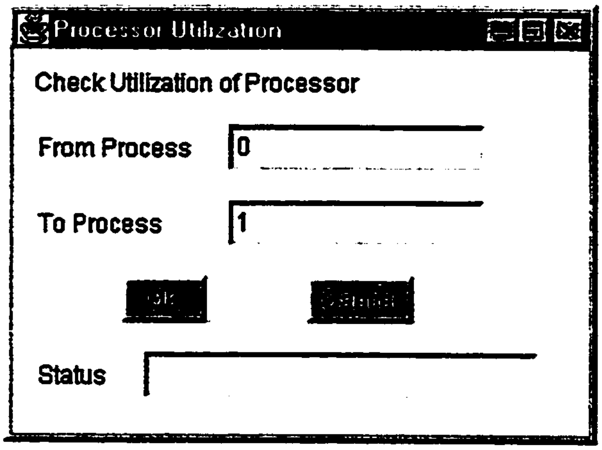


Figure 27 Utilization Window

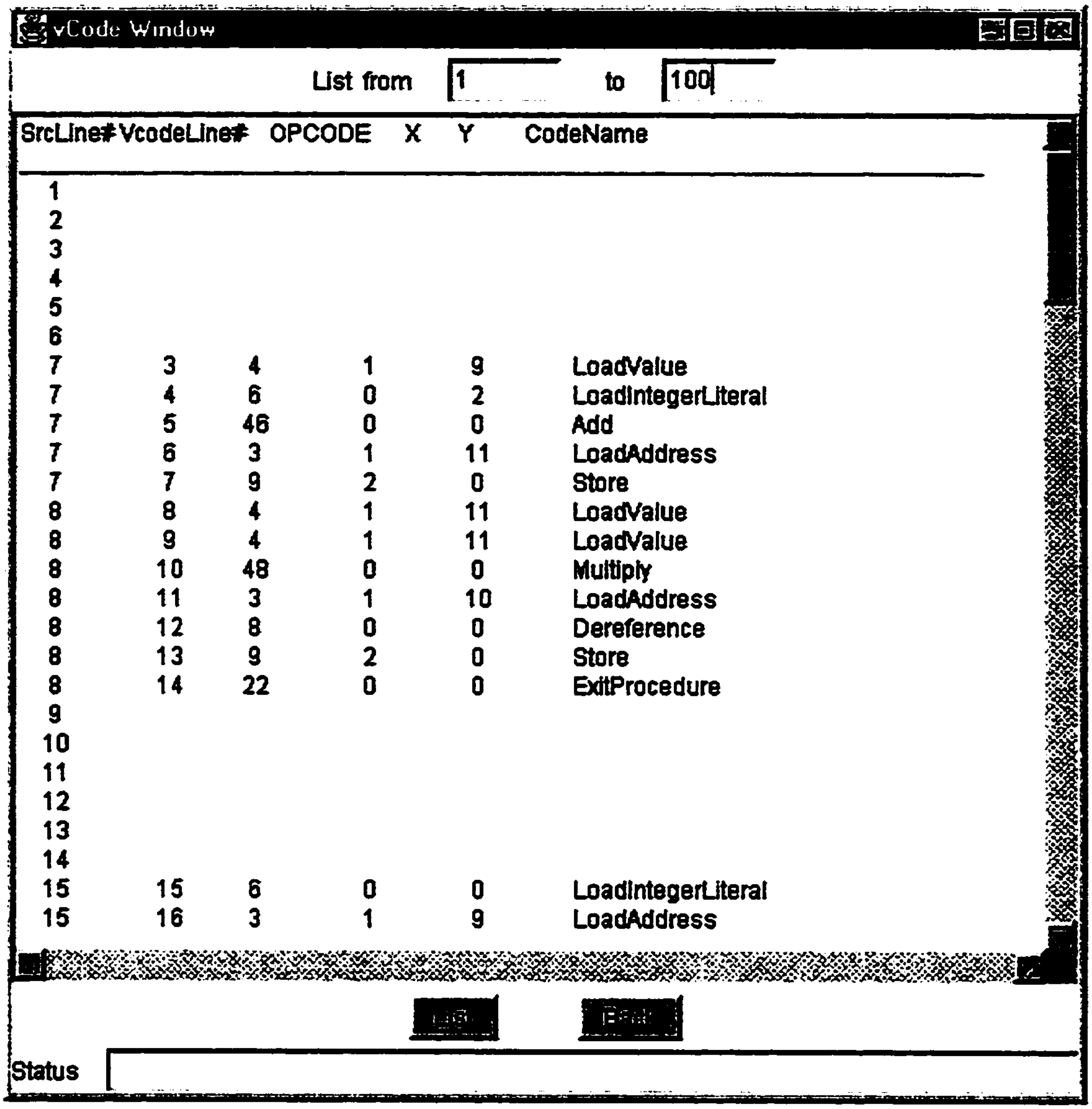
be displayed, and press the *List* button. To close the window, the user can press the *Back* button.

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

4.2.13 Stack Window

The window with the title *StackDialog* is used to display the contents of execution stack, as shown in Figure 29. To confirm the new settings, users can press the *ok* button. To keep the previous settings, press the *Cancel* button, and this window will be closed.

By default, the values of fields From Line and To Line are 0 and 10. If users enter invalid data, such as "-1" or "abc," in the fields, the invalid data will be replaced by the



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

Figure 28 Code Window

default values automatically. At the same time, the status field will show users some warning message.

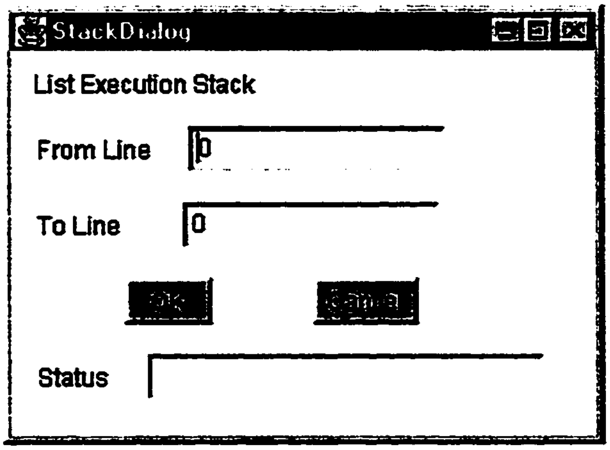


Figure 29 Stack Window

4.3 Class Relationship

The CPPE Java client has the following files

- *CppeFrame.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*

Figure 30 shows the relationship among the classes contained in these files. The first and third columns are names of GUI classes. The second column is the names of buttons or menu items on the CppeFrame user interface used to activate the objects described in the third column.

4.4 Summary

This chapter provides sample designs and implementations of CPPE graphical user interface in Java. Since the Java client code represents the CPPE server object on the client machine and is the only user interface between the remote server object and the client, the quality of the Java user interface is very important. Any project for legacy code adaptation must allocate enough manpower and time for this stage of the project.

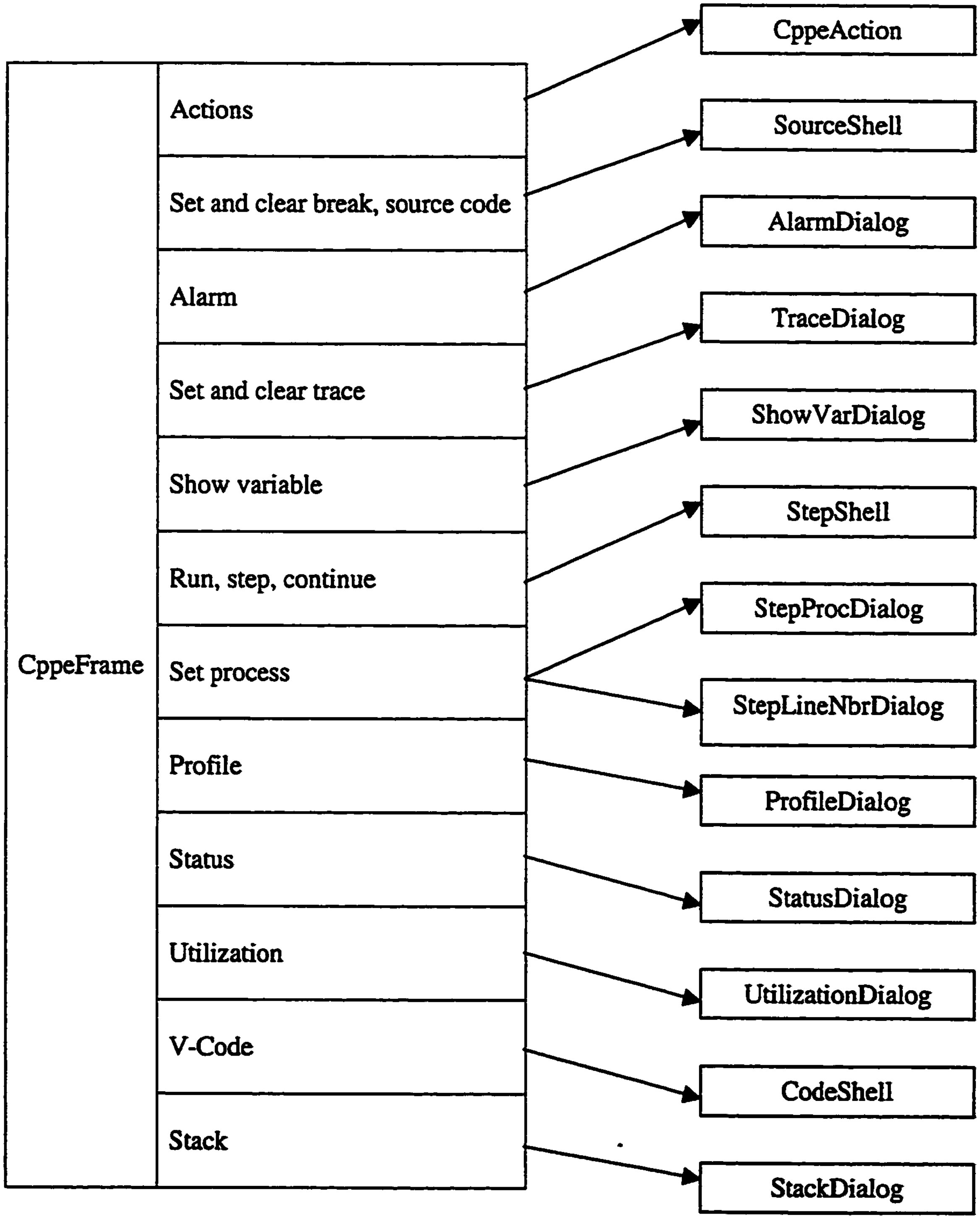


Figure 30 Relationship among Client Side Classes

CHAPTER 5 ACCESSING CPPE THROUGH WEB BROWSER APPLETS

At this point we have finished the design and implementation of CPPE server object, the linkage between CPPE server object and CPPE client, and the global CPPE software architecture in Chapter 3, and the design and implementation of CPPE client in Chapter 4.

While our project is complete if we run the client code as a standalone application, we have to overcome one extra difficulty to let our client code run inside a web browser and access the user's local file system. This is critical to providing the illusion that the user has a local CPPE installation.

In this chapter we discuss how we can use Java 2 applet certificating technology to break the applet sandbox restrictions so that the CPPE client running inside a web browser can have full access to the user's local file system.

5.1 Applet Sandbox Restriction

An applet, being a Java program that is run from inside a Web browser, is subjected to some security restrictions. The original Java applet security model implements a sandbox that imposes strict controls on what certain kinds of Java programs can and cannot do. In general, applets loaded over the Internet are prevented from reading and writing files on the client file system; they can only access resources from or make network connections to the applet-originating host [7]. By default all downloaded applets are considered untrusted, which means that these applets are only permitted to run in client-side applet sandbox and can't access local resources.

All Netscape Navigator versions subsequent to 3.0 are Java enabled. All Microsoft Internet Explorer versions subsequent to 3.0 also support Java. The two browsers' security policies are, at the present time, very similar. Both are somewhat strict. The following rules apply to all untrusted applets running under Netscape Navigator and Internet Explorer [7]:

- Applets cannot read or write files on client's local disks.
- Applets cannot open a client-side network connection to any machine other than the applet's origin host.
- Applets after JDK 1.1 can open a server socket as long as the port number is greater than the privileged port number on the machine (usually 1024).
- Applets can read only nine system properties. Applets are not permitted to read any other system properties.
- If an applet is loaded using the file URL, and it does not reside in a directory on the CLASSPATH, it will be loaded by an Applet Class Loader.

Both browsers offer complex security models based on digital signatures and partial trust. Although the models are quite similar to the model defined by Java 2, there are many subtle differences that annoy developers and users. To solve this problem we have to make a trusted applet to break the applet sandbox restriction. There are two ways for an applet to be considered trusted: the applet is installed on the local hard disk, or the applet is signed by an identity marked as trusted in your identity database. CPPE applet is expected to read and write files on client's machines, so we need to make it a trusted applet.

5.2 Make Trusted CPPE Applet

CPPE applet uses Java 2 Swing. We need to use the Java 2 plug-in to deploy the system and we can naturally use the Java 2 security model to leverage the full ability of our CPPE applet. The Java 2 security model is policy-based and it will force the Java application abide the policy file definition. Since CPPE applet only reads CPC source files specified by the users, generates safe vCode files on the local file system, and never modifies other files or system attributes on the client machine, we give CPPE applet all the needed permission to perform its tasks.

Permission represents access rights to system resources. In order for a resource to be accessed by an applet (or an application running with a security manager), the corresponding permission must be explicitly granted to the code attempting the access. A policy file represents the policy for Java application resource accessibility. The default policy implementation obtains its information from static ASCII policy configuration files. A policy file can be composed via a simple text editor, or via the graphical Policy Tool utility ***policytool*** in JDK. The user policy file "java.policy" is by default located at:

C:\Winnt\Profiles\login_name on multi-user Windows NT systems

C:\Windows\Profiles\login_name on multi-user Windows 95 systems

C:\Windows on single-user on Windows 95 systems

user.home/.java.policy (Solaris)

When the policy is initialized, the system policy is loaded in first, and then the user policy is added to it. If neither policy is present, a built-in policy is used. This built-in policy is the same as the original sandbox policy. Policy file locations are specified in the security properties file, which is located at

java.home/lib/security/java.security (Solaris)

java.home\lib\security\java.security (Windows)

Using the Policy Tool saves typing and eliminates the need for you to know the required syntax of policy files, thus reducing errors. There is a tool named ***keytool*** from JDK that can be used to create public/private key pairs and self-signed X.509 v1 certificates [7], and to manage keystores. A keystore is a protected database that holds keys and certificates for an enterprise. Access to a keystore is guarded by a password (defined at the time the keystore is created, by the person who creates the keystore, and changeable only when providing the current password). In addition, each private key in a keystore can be guarded by its own password. Another tool ***jarsigner*** is used to digitally sign Java applications or applets using produced keys and certificates from keystores. Using keytool, it is possible to display, import, and export X.509 v1, v2, and v3 certificates stored as files, and to generate new self-signed v1 certificates. By using these tools JDK 1.2 provides the basic technology for loading and authenticating signed classes. This enables browsers to run trusted applets in a trusted environment. CPPE applet uses these tools for finer-grained policy control to get flexible security policies. We will show the detailed steps to demonstrate how to apply this.

5.3 How to Sign the CPPE Applet

In order to deploy CPPE applet and let it run as a trusted applet inside the client browsers, we need to sign the class archive file and export the signed certification file to the client side. The Client then imports this certification and adds a new Java security policy entry to the client policy file. The steps are as follows:

Server side

- Step 1: Produce owner's public key

  JDK_HOME\bin\keytool -genkey -keystore C:\WINNT\Profiles\jiang\.keystore -alias jiang

  Enter keystore password: ***cppe***

  What is your first and last name?

  [Unknown]: ***Weilan Jiang***

  What is the name of your organizational unit?

  [Unknown]: ***CS Dept.***

  What is the name of your organization?

  [Unknown]: ***Concordia***

  What is the name of your City or Locality?

  [Unknown]: ***Montreal***

  What is the name of your State or Province?

  [Unknown]: ***Quebec***

  What is the two-letter country code for this unit?

  [Unknown]: ***ca***

  Is <CN=Weilan Jiang, OU=MRI, O=MRI, L=Montreal, ST=Quebec, C=ca> correct?

  [no]: ***y***

  Enter key password for <jiang> (RETURN if same as keystore password):[return]

- Step2: Use the above public key to sign a jar file

```
JDK_HOME\bin\jarsigner -keystore
  C:\WINNT\Profiles\jiang\.keystore -signedjar cppe.jar
  cppe.jar jiang
```

Enter Passphrase for keystore: ***cppe***

- Step 3: Export the Public Key Certificate file and signed jar file

```
JDK_HOME\bin\keytool -export -keystore
  C:\WINNT\Profiles\jiang\.keystore -alias jiang -file
  cppe.cer
```

Enter keystore password: ***cppe***

Certificate stored in file < cppe.cer>

<u>**Client side**</u>

- Step 4: Import signer's certification file to produce client's keystore file

  JDK_HOME/bin/keytool -import -alias jiang -file cppe.cer -keystore c:\userStore

- Step5: Modify java policy file

  Use java policy tool ***policytool*** to create or modify existing file "*java.policy*" in the user's profile root folder (i.e. *C:\WINNT\Profiles\jiang*). Eventually the following two lines should be in file "*java.policy*".

  keystore "C:/ userStore"

  grant signedBy "jiang" {permission java.security.AllPermission;};

Now the signed CPPE applet can break the applet sandbox restriction and access local machine resources as a standalone application.

5.4 Summary

In this chapter we have shown how to use Java2 applet certificating technology to allow the CPPE client to access users' local file systems. This technology can be also used to allow an applet to access users' system attributes. One advantage of this technology is that we have fine control as to what system resources should be accessible by an applet.

CHAPTER 6 CONCLUSION

In this thesis we have provided a concise survey of technologies supporting fast adaptation of legacy code for server hosting so that the functionalities of the legacy code can be accessed remotely through web browsers or standalone thin-client applications. We use Concordia parallel Programming Environment (CPPE) as an example legacy code to demonstrate how we can use CORBA technology to achieve this adaptation effectively and with minimal modification to the existing source code.

The approaches and technologies demonstrated in this thesis are of general importance for the information technology industry to support the new Application Service Model of computing without immediate significant investment in terms of manpower training, software development, and performance testing. While such fast adaptations may not be able to take full advantage of the latest Internet technologies, they can support smooth transition for industries from in-house computing model to the new open-to-the-public model.

As future work, we can further investigate the performance of such adaptations, especially the scalability of such adaptations. This is another hot issue that has drawn a lot of attention from the IT industries in the recent months.

BIBLIOGRAPHY

[1] Inprise Corporation. "Programmer's Guide: VisiBroker for C++," 2000

[2] Inprise Corporation. "Programmer's Guide: VisiBroker for Java," 2000

[3] James Gosling et al.. "Java Programming Language," Addison Wesley, 1998

[4] Orfali Harkey. "Client/Server Programming with Java and CORBA," Wiley, 1998

[5] F. Thomas Mowbray and Ron Zahavi. "The Essential CORBA: Systems Integration Using Distributed Objects," Wiley, 1995

[6] Cay Horstmann and Gray Cornell. "Core Java 1.1," Prentice Hall, 1997

[7] Gary McGraw and Edward W. Felten. "Securing Java," 2nd Edition, Wiley, ISBN: 047131952X

[8] Thomas Mowbray and Raphael Malveau. "CORBA Design Pattern," Wiley, 1997

[9] Robert Orfali, et al. "Instant CORBA," Wiley, 1997

[10] David S. Platt. "Understanding COM+," Microsoft Press, 2000

[11] Jon Siegel, et al.. "CORBA Fundamentals and Programming," Wiley, 1996

[12] Lixin Tao. "Delayed Program Mapping in Concordia Parallel Programming Environment," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 28-July 1, 1999. pp. 1709-1715

[13] Andreas Vogel and Keith Duddy. "Java Programming with CORBA," Wiley, 1997

[14] Jing Zhang. "A Visual Performance Debugger for Concordia Parallel Programming Environment," Master Thesis, Concordia University, 2000

[15] www.citrix.com

APPENDIX A: USER MANUAL

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

Parallel application program compilation;

Parallel application program execution;

Correctness and performance debugging.

A.1 Account Management and Login

A.1.1 Account Management

To support multiple users, CPPE provides an account for each user. An administrator has the permission to open an account with the GUI shown in Figure 31. This window will pop up when you run "accManage" on the server site. An administrator can add an account, delete an account, and list the names and passwords for all current accounts.

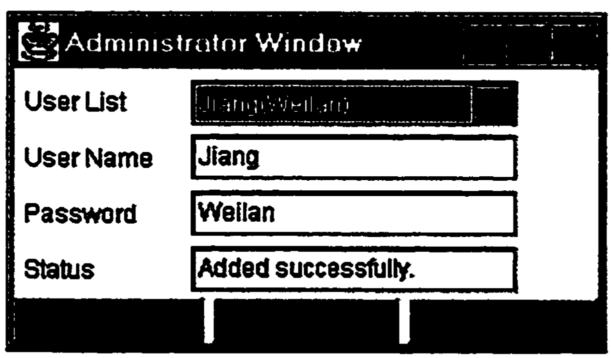


Figure 31 Administrator Window

A.1.2 Login

To login to an account, a user must have an account with user name and password in advance. The login window, as shown in Figure 32, will pop up when you run "cppe" or click on the CPPE hyperlink inside a web browser on client site. After entering right user name and password, the main frame, as shown in Figure 15, will pop up.

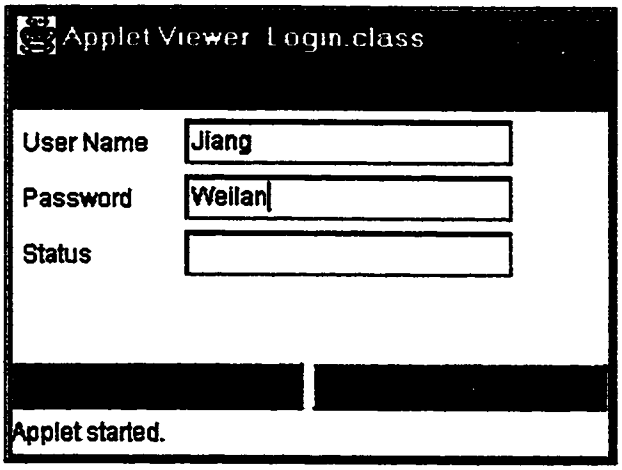


Figure 32 Login Window

A.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 environment.

A.3 Compile Application Program Source File

A.3.1 Open Source Files

To open a source file, users can click the button *Open* in the main frame GUI. A file selection dialog box will pop 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, as shown in Figure 17.

A.3.2 Compile Source Files

The 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 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.

A.3.3 Open vCode Files

Button *Open* can also be used to open a vCode file directly. The name of the opened vCode file 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.

A.4 Execution and Debugging

A.4.1 Execution

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

A.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 button *D* to pop up the Debug pane and then click button *SrcCode*. A new *List* window will pop 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, as shown in Figure 19.

To display the vCode, from the main menu in the main frame, users can click button *D* to pop up the Debug pane, and then click button *vCode*. A new *vCode* window will pop 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, as shown in Figure 28.

A.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 locate bugs in the program. The breakpoints are set in the Source Code Shell window, as shown in Figure 19.

Breakpoints are set in the CPPE by referring to source program line numbers. To set a breakpoint, users can click button *SetBreak* in the debug pane. A new window titled *SourceShell* will pop up. From the *SourceShell* 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 *SourceShell* 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 the *Unbreak* button in the *SourceShell* 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, CPPE is set to Debug mode, and a user executes the 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.

A.4.4 Stepping Through Process

When any running process tries to execute a line in a program set as 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 the continue function is used, the execution will continue until any process encounters the next breakpoint. If the step function is used, the execution will continue line by line from the breakpoint in the suspended process. Users can also specify the number of source lines to be executed in each step, and set the "Step Process" to a different running process, as shown in Figure 20 and Figure 21.

To use the Continue function, users can click button *Continue* in the main frame. To use the Step function, they can click button *Step* in the main frame. In either case, if the program execution stops at a new breakpoint, the new breakpoint of source code will be highlighted in the *Step Shell* window.

A.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 button *Show* in the debug pane, and specify a variable name and the ID of an active process. If the variable is an array, users can optionally specify the index range that the user wants to display, then click button *OK*. The output will be displayed in the up output window in the main frame. The Show function is performed with the *ShowVarDialog* window, as shown in Figure 23.

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 pop up, as shown in Figure 22. 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.

A.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 set an alarm, users can click button *Alarm* in the debug pane. An *Alarm Dialog* window 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 basic simulation time units in the text field Enter Alarm Time.

A.5 Network Architectures and Mapping

A.5.1 Specifying Physical Architecture

When CPPE starts, the default architecture is a 4 x 4 2D-mesh multi-computer. Users may override this default and specify a wide range of other architectures based on common interconnection topologies. This allows the performance of the parallel program to be evaluated on a wide range of parallel computer architectures according to the choice of the users.

CPPE has predefined some most common architectures that can be used directly by selecting from the option menu *PhyArch* in the main frame.

A.5.2 Virtual-to-Physical Architecture Mapping

We encourage programmers to write message-passing parallel programs using virtual topologies, the topologies most natural to express the program communication structures. However, the virtual topology may be different from the topology of the physical system on which the program will run. CPPE supports flexible mappings of the virtual processors to the physical processors. The objectives of virtual-to-physical architecture mapping are to minimize communication cost by minimizing the distance between communicating processes, and balancing the workload among the physical processors.

CPPE currently supports six types of mapping: Default, Identity, Random, Ring-to-Line, Torus-to-Mesh and User-Defined Mapping. With User-Defined Mapping, users can specify the mappings with C language expressions at program loading time.

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

A.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. The network type can be selected from the option menu *Network* in the main frame.

A.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.

A.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, uses can estimate the performance improvement by parallel computing.

A.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 button *Time* on the *debug* pane. The output will be displayed in the main output window in the main frame.

A.5.7 Utilization

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

To use the Utilization function, users can click button *Utilization* on the *debug* pane. A *utilizationDialog* dialog box will pop up, as shown in Figure 27. Users can specify the range of processors that the user wishes to see the utilization value with the dialog box.

A.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 turn on or off the profile option. The default range of processors in the profile is all the 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 button *Profile* in the debug pane, a *profileDialog* dialog box will pop up, as shown in Figure 25. Users can specify the range of processors and the time interval with this dialog box.

APPENDIX B: 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.

B.1 Build and Installation of CPPE Server on Windows NT

Install *JDK1.2.2.*

Install *Visibroker for C++ 3.3.*

Install *Microsoft Virtual C++ 6.0.*

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

Change attributes of all files under CPPE from read only to writable.

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

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

Flex.exe

Bison.exe

Bison.hai

Bison.sim

Lex.exe

Lex.par

Add *c:\tools* to PATH.

Define environment variable *BISON_HAIRY* to be *c:\tools\Bison.hai.*

Define environment variable *BISON_SIMPLE* to be *c:\tools\Bison.sim.*

Define environment variable *CPPE* to be *workdir\cppe\test.*

Compile the *cpcc* and *cpss* in DOS prompt:

workdir\cpp\ cpcc\>nmake

workdir\cppe\cpss\>nmake

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

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 directory *c:\pkg\inprise*.

Compile the server in DOS prompt:

workdir\cppe\CorbaCPPE\cpp\>nmake

B.2 Build and Installation of Server on Windows 95/98

Install JDK1.2.2.

Install *Visibroker for cpp 3.3.*

Install *Microsoft Visual C++ 6.0.*

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

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

Set PATH in *autoexec.bat* as the following:

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

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

Flex.exe

Bison.exe

Bison.hai

Bison.sim

Lex.exe

Lex.par

Modify *autoexec.bat* by adding:

set PATH=c:\tools

set BISON_HAIRY=c:\tools\Bison.hai

set BISON_SIMPLE=c:\tools\Bison.sim

Modify *autoexec.bat* by adding:

set CPPE= workdir\cppe\test

Compile *cpcc* and *cpss* in DOS prompt:

workdir\cppe cpcc\>nmake

workdir\cppe cpss\>nmake

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

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.*

Compile the server in DOS prompt.

workdir\cppe\CorbaCPPE\cpp\>nmake

B.3 Build and Installation of Client

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

Install *JDK1.2.2*

Install *Visibroker for Java 3.4*

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

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

Compile the client in DOS prompt.

workdir\cppe\CorbaCPPE\java\>vbmake