A 2-D PLOTTING UTILITY WITH CORBA

WEI PAN

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

JUNE 2002

0-612-68475-X

ABSTRACT

A 2-D Plotting Utility with CORBA

Wei Pan

The Commom Object Request Broker Architecture (CORBA) is an important and open standard for distributed objects. CORBA uses objects as a unifying metaphor for brings existing applications to the bus. At the same time, it provides a solid foundation for a component-based future. The 2-D plotting utility directly visualizes mathematical functions for remote users over Internet. It provides a complete and friendly graphical user interface for remote client sites, and a reliable, portable and extensible parser/evaluator on a server site. This 2-D plotting utility is designed basing on client/server model. It is implemented and deployed using CORBA technology. This report demonstrates that CORBA's language, location, and platform independence provides a strong base towards its use in wrapping the legacy applications.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Peter Grogono, for his guidance as well as patience during the entire course of preparing this report.

I would also want to thanks my wife Yi Shen for her untiring love, encouragement, patience and wise words. I could not have made it without her.

Contents

List of Figures

Chapter 1

1 Introduction

The Common Object Request Broker Architecture (CORBA) is an emerging open distributed object computing infrastructure. CORBA uses an object-oriented approach for creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well-defined interface, which reduces application complexity. CORBA specifies a framework for the transparent communication between application objects. The client application can get services provided by distributed server objects without worry about where these server objects are located, what platform they are running on, and which language is used to implement them.

In this project, I have designed and implemented a distributed 2-D plotting utility based on client/server model and using CORBA technology. This report is organized as follows. In Chapter 2, the background of the project will be addressed. In Chapter 3, I will give brief introduction to client/server architecture and CORBA. In Chapter 4, the project system design and user interface design is discussed. Chapter 5 summarizes the implementation and deployment of the application. Selected pieces of codes are presented in the Appendix.

Chapter 2

2 Background

2.1 Problem Definition

A 2-D plotting utility is a helpful tool for people who want to visualize mathematical function(s) on a computer screen. Such a simple 2-D graph plotter has been developed by Ahn Phong Tran [1]. Tran's plotter is a standalone application requires users to install and run the whole application on a single machine. There are many disadvantages for standalone application, including the impact on this 2-D plotter when updating the software. A new version has to be distributed to each user and the user has to re-install it.

The explosive growth of the Web, the increasing popularity of PCs and the advances in high-speed network access has brought distributed computing into the mainstream. The client/server distributed computing model is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, time-sharing computing. The client/server paradigm provides a perfect solution for the problem described above. Basically the presentation logic and data processing logic in this 2-D plotter application can be divided into two separated components which could be installed and run at any two computer machines connected by a network. This solution has many advantages. First of all, it saves computing resources. The user needs only the client portion (user interface component) of the application while the server portion (data processing component), residing at host machine, is shared by many users. Second, it gives flexibility to implement the client component with freedom to chose the programming languages and platform on which it runs. Third, perhaps more important for this simple 2-D plotter application, it allows a manager to update the server portion component without has any effect on the user as long as the outward interface of the server component is not changed.

Recently, Meng Cai [2] completed a significant step in redesigning and implementing this 2-D plotter graph utility using client/server distributed computing model. The new 2-D plotter is a distributed object system, which is simple, reliable and cost-effective. However, since it uses Java Remote Method Invocation (RMI) as the middleware, it has limitations of its own. RMI is a pure Java solution and is specifically designed to operate in the Java environment. With RMI, all client and server objects must be written in Java. This makes it more difficult to reuse a legacy system written in other programming languages. In our case, the parser/evaluator component was written in C++. To reuse this component under RMI extra programming effort is needed. Java is good for client implementation because it provides strong language support for GUI and works well in a Web environment. However its lower performance compared to C/C++ language limits its application for server implementation especially when the application scales up.

CORBA, as an industry standard middleware for object-oriented distributed system, is another obvious choice for our 2-D plotter system. Wrapping legacy applications and hiding language and platform heterogeneity are two of the main application areas for CORBA, which particularly fits our needs. With CORBA the most appropriate programming language can be chosen for each object, based on the need for legacy integration, prior experience of a development team, or the suitability of the language for implementing the object's semantics.

2.2 A Summary of the Previous Work

2.2.1 Tran's standalone implementation using C++ language

Tran's standalone 2-D graph plotter [1] can be used to plot simple algebraic expressions. The software consists of two major components, namely, a *parser/evaluator* component and a *user interface* (including *plotting*) component.

The *user interface* component provides an *Input* dialog and a *View* on which curves can be drawn. This component was implemented with *Microsoft Foundation Class* (MFC). The *Input* dialog inherits MFC's *CDialog* class, and the View derives from MFC's *CView* class.

The *parser/evaluator* component is used to parse and validate the input expression, evaluate the expression according to the input data and return the value of the expression. The user-input expression has been checked for its syntactic correctness by conformance to the following grammar.

Expr -> ['-'] Term {('+' | '-') Term}.

Term -> Factor {('' | '/') Factor}.*

Factor -> Primary ['^' Primary].

Primary -> NUM | VAR ['('Expr') '] | '('Expr')'.

In this grammar, *NUM* represents a number, *VAR* represents either a simple variable or a built-in function (for example, *sin*, *cos*, *tan*, *atan*, *exp*, *log*, *fabs*, or *sqrt*). Things of the form [X] means that X may appear or not. Things of the form (X1|X2) indicate that exactly one of the Xi must appear. If the expression is not syntactically correct, an exception is thrown. Otherwise, a parse tree is built. The evaluator then assigns input values to the nodes in the parse tree, and returns the result of evaluating the parse tree. The parser was based on the *recursive descent method* [3] and was written in the C++ programming language.

2.2.2 Cai's implementation based on RMI technology

Recently, Meng Cai [2] redesigned the 2-D graph plotter based on client/server software architecture and implemented it using the Java programming language. The system has been divided into two major components: client and server. The client component is responsible for graph user interface and the server component is responsible for parsing and evaluating mathematical expression. The client and server can be distributed to any two computers with network connected between them. The distributed object middleware RMI [4] has been used to control the communication between client and server.

The client component is implemented as a Java Applet. An end user uses a Web browser to load a Java Applet which implements the graphical user interface of the application. The choice of using Java applet has its own advantages. It provides a complete GUI with minimum resource requirements from the end user. It can also overcome a major problem caused by software updates and their distribution. A Web browser automatically does the shipping and installation process. A variety of graphic class libraries provided by *Java Foundation Classes* (JFC, also known as *Swing*) [5] has been used for detail GUI implementation of this client component.

The server component is used to accept the client's requests, uses its application logic to process the requests, and returns the results to the client. All the services of the server are defined in its RMI interface, which is registered into the registry service (running on the server machine) provided by RMI. Using JRMP (Java Remote Method Protocol) protocol, the client sends requests over network to the RMI registry service which in turns invokes the methods in the Java implementation of the server object. To be able to reuse the existing C++ parser module, the Java Native Interface (JNI) facility [6] is used. JNI allows a native Java interface be build upon non-Java legacy system and through this interface the Java code running inside a Java virtual machine can interoperate with applications and libraries written in other programming languages, such as C, C++ and assembly.

2.3 Project Goals

Based on previous works [1][2] of the 2-D graph plotter application, the goal of this project is to develop a distributed 2-D graph plotter using industry-standard object-oriented middleware CORBA technology. More specifically the VisiBroker, Inprise's implementation of the CORBA specification, is used in this project.

The new 2-D plotter should be designed based on distributed client/server architecture. The system should be divided into two major components; a client that interacts with user and a server that parses and evaluates expressions. The two components need to be well separated and able to run at two different computers in the network with possible different platforms (Windows, Unix, Linux, etc.). The communication between the two components is handled by the middleware and must be reliable, efficient, and scalable. The coupling between the two components should be kept as low as possible. The only thing that the client needs to know about the server is the server's public interface, details of the server's implementation should be hidden from the client. Great care should be taken on the design of the public interface. It should be complete and concise, providing all but not more than required for the client's need to do its job. The interface should also be able to remain unchanged during the evolution of the software system. In this way any future updates on the server side do not require recompilation of the client code.

The client component is responsible for the application's representation logic. It runs on the user's machine and provides a complete graphical user interface (GUI) to interact with the user. It should also provide an interactive interface to allow the user to enter a mathematical formula and its parameters, select a number of options, and view the resulting curves directly. It should also be able to validate the user's input, catch exceptions, and inform the user whenever an error occurs. The GUI design should let users carry out their task effectively, efficiently, enjoyably, and safely.

The server component is responsible for the application's business logic. It runs on a host machine. The component must accept the client's requests and responses by sending the results back after parsing and evaluating expressions. The server should be able to manage the run-time errors in an orderly fashion. Considering that there are possible many requests from different clients in the network, the design and implementation of the server component should be efficient and reliable.

It is important for this project to be able to reuse exist software code and modular. To efficiently reuse Cai's Java GUI implementation [2] in client component and Tran's C++ parser/evaluator modular [1] in server component, programming language independence in this project's design and implementation should be emphasized. As discussed in section 2.1, it is suitable for this project to use Java language on the client side and C++ language on the server side.

2.4 Project Requirements

In this section, I provide a description of the project requirement. The section gives details on what the plotter can do for the user.

The plotter must be able to allow the user to input an expression, and then parse and validate the expression to conform to the grammar (see section 2.2.1). The plotter accepts expression with the following constituents:

- Numbers in C++ format. All of these are acceptable numbers: 1, .25, 99., 2.71828, 1E-10, and 6.3e24.

- Variables consisting of one or more letters. Variable names are case sensitive.
- The operators '+', '-', '*', and '/' with the usual precedence. Also '^' can be used for exponents and has higher precedence than the other operators.
- Absolute value expressions of the form '| ... |'. The expression '|x|' and 'fabs(x)' are equivalent.
- Parentheses '(' and ')' used for grouping.
- Function invocations in the form f(e), in which f is the name of a function and e is an expression. Frequently used mathematical functions must be available, such as sin, cos, exp, etc. New functions must be easily added into the system if needed.
- Blank and tab characters, they are allowed in an expression.

Here are some typical expressions:

X^2+1	^ is the exponent operator
(x-1)*(x+1)	parentheses allowed; no implicit multiplication
sin(x)+cos(x)	function calls
exp(x)^2	$= (e^x)^2$
x - b	blanks are allowed

The plotter must allow user to define the following roles of the letters in an expression.

- Variable.

  There must be exactly one independent variable, which will often be x or t, but the program allows user to specify any letter. The user must specify the range of the independent variable by entering a minimum and a maximum value. The default range is from 0 to 1.

- Parameter.

  One letter may be specified as a parameter. If there is a parameter, the user must specify the first value of the parameter, the final value, and the interval. The default values are 0, 5 and 1 respectively.

- Constant.

  Any letters other than that defined as variable and/or parameter in expression are assumed to be constants. The default value of a constant is 0.

The plotter can be in either one of two states: the *initial* state and the *plotted* state. The application starts with *initial* state. In this state the user sees an empty graph. There are two axes without labels or scales. After the user has entered an expression and its settings have been accepted by the plotter, the plotter computes the values of the expression, finds the minimum and maximum values in the desired range and choose suitable scales, writes labels and scales and plots the curve. When the first graph is plotted, the plotter is in the *plotted* state. In this state, the label, the scale and the range of the axes is fixed. Settings for the plotted graph cannot be changed. However, the user can introduce new expressions that will be plotted in the same display area as the first graph. The only constraint for the new expression is that the variable name should not changed and the range of the variable should be within that for the first graph. The application should also provide a control to clear the current graph and return the system to the *initial* state.

The plotter should be able to handle the obvious singularities in the expression when plotting the curve. Here are some examples of singularities:

1/x	when x = 0
sqrt(x)	when x < 0
log(x)	when x<=0

In this situation, the plotter should still plot the expression but with the singular point or points ignored and inform the user with appropriate warning information.

The new plotters should provide additional functionality including:

- The user should be able to select the resolution of the graph, that is, the user should have a choice of "quick and rough" plotting or "slow and accurate" plotting.
- The user should be able to choose colors for axes and the graph background.
- The user should be able to get the coordinates of a point by clicking in the graph display area.
- The user should be able to highlight a single curve among many curves and has its expression shown by clicking right on the curve.

Chapter 3

3 Client/Server Application

3.1 Client/Server Architectures

The term client/server was first used in the 1980s [7] in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. Client/Server Architecture is a class of software architectures in which processing is distributed among one or more information requesters (clients) and one or more information providers (servers), as well as in the interfaces (network, protocols, and middleware) between them. Client/Server architectures include two-tiered and three-tiered (or even n-tiered) architectures. Client/Server architectures are in contrast to mainframe architectures in which the processing is self-contained.

3.1.1 Clients and Servers

Client/server computing systems are comprised of two logical parts [8]: a server that provides services and a client that requests services from the server. Together, the two form a complete computing system with a distinct division of responsibility. Many clients have a modern graphical user interface (GUI) that presents each resource accessible by the user as an independent object. Normal models of client/server computing place no limit on the number of servers a client can access simultaneously. Servers typically fill one specific need and encapsulate the provided services so that the state of the server is protected and the means by which the service is provided are hidden from the client. It is often the case that many clients request services of a server independently but uses the same interface. Sometimes it is possible that a server can execute a task by dividing it into subtasks and then have other servers completed the subtasks: in this situation the server act as a client to other servers.

3.1.2 Middleware

The distributed software required to facilitate client/server interaction is referred to as "middleware" [8]. Transparent access to non-local services and resources distributed across a network is usually provided through middleware, which serves as a framework for communication between client and server portions of a system. Middleware can be thought of as the networking between the components of a client/server system; it is what allows the various components to communicate in a structured manner. Middleware is defined to include the APIs used by clients to request a service from a server, the physical transmission of the request to the network (or the communication of the service request to a local server), and the resulting transmission of data for the client back to the network. Middleware is run on both the client and server ends of a transaction.

3.1.3 Two-Tiered Client/Server Architectures

An application program can be divided into the following three layers [9]:

- Presentation logic: The part of the application program responsible for interfacing to the user interface.

- Business logic: The actual program rules of the application responsible for controlling program execution and enforcing business rules.
- Database logic: The part of the application program responsible for interfacing to the database management system.

Two-tiered client/server architectures deliver the presentation logic on the client and the database logic on the server. The business logic may be distributed as follows:

- On the client in a two-tiered architecture known as "fat client".
- On the server in a two-tiered architecture known as "fat server".

Most client/server systems are flexible with regard to the distribution of authority, responsibility, and intelligence. However, shifting intelligence from the client to the server or vice versa shifts the capabilities and strengths of the system. For example, if a fat server is being used, it usually is easy to update business logic without affecting the distributed clients. However, if fat clients are being used, the server need not be touched and system stability is not jeopardized.

Fat clients let users create applications and modify complex front-ends to systems easily, but this comes at the price of reduced data encapsulation; as more responsibility is placed on a client, the client requires a more intimate knowledge regarding the organization of data on the serving end. Recently, the use of fat servers has been increased because of:

- The industry trend towards greater object orientation, which favors a high degree of data encapsulation. With encapsulation, the server can provide more abstract services and more meaningful responses to service requests can be send back to the clients.

- Fat servers are easier to manage and deploy since the data and code exists in a centralized location.
- Fat servers reduce the problem of limited bandwidth by carrying out more of the work where the data resides, reducing the need for costly data transfers over the network.
- Fat servers ensure greater compatibility between clients and servers: the more work the server does, the less dependent it is on the client.

3.1.4 Three-Tiered Client/Server Architectures

The three-tiered (or sometimes referred to as n-tiered) client/server architectures deliver the presentation logic on the client, the business logic on one or more dedicated servers, and the database logic on one or more superservers or mainframes.

The three-tiered model is more advanced and flexible than the traditional two-tiered model because the separation of the business logic from the client and server gives business logic processes a new level of autonomy. The processes become more robust by providing more insulation and separation between layers. Also because the executable components are more fine-grained, three tiers give more flexibility in the deployment of an application.

The next logical step in the evolution of application architectures is the distributed system model. This architecture takes the concept of multitier client/server to its natural conclusion. Rather than differentiate between business logic and data access, the distributed system model simply exposes all functionality of the application as objects, each of which can use any of the services provided by other objects in the system, or even objects in other systems. The architecture can also blur the distinction between "client" and "server", because the client components can also create objects that behave in server-like roles. The distributed system architecture provides the ultimate flexibility.

3.1.5 The Benefits of Client/Server System

As client/server systems have grown more robust, the computing community has acknowledged their many distinct advantages [8]. Perhaps the most important advantage is the natural mapping of applications into a client/server framework. As a result of the availability of compatible middleware for multiple platforms and recent advances in binary interoperability, client/server systems can usually put clients on one platform and the server on another. This allows users to customize their environments for maximum efficiency and system administrators to upgrade transparently to more powerful, capable, or less expensive servers without notifying the users (clients). Application development is also simplified since clients and servers each fill a specific need, and each properly designed server supports an interface directly related to the realization of one common goal. Client/server models leverage the advantages of commodity-like hardware prices since resource-intensive applications can be designed to run on multiple low-cost systems. Systems can scale both horizontally and vertically, meaning that clients can be added with little performance penalty and that extra performance can be extracted from a client/server system by adding faster server hardware.

3.2 CORBA: Object-Oriented Middleware

3.2.1 Overview

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 [10,11]. An industry consortium, the Object Management Group (OMG), which now numbers over 800 members, defined the CORBA standard as a solution to distributed object interoperability.

CORBA was designed basically to allow objects to discover each other and invoke methods on remote objects and interoperate on the object bus. It also specifies an extensive set of bus-related services for creating and deleting objects, accessing them by name, storing them in persistent stores, externalizing their states, and defining ad hoc relationships between them. CORBA lets you create an ordinary object and then make it transactional, secure, lockable, and persistent by making the object multiply-inherit from the appropriate service. This means that you can design an ordinary component to provide its regular function, and then insert the right middleware mix when you build it or create it at run time.

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 [12]:

- It forces the separation of an object's interface and its implementation.
- It is scalable.
- Support for reuse is inherent.
- There is programming language transparency. In addition to C++ and Java, CORBA support is available for C, Smalltalk, Ada and LISP.
- There is location transparency. Application components which use CORBA can communicate over shared memory, a backplane, a local area network or the Internet. Object location is completely transparent to application code.
- There is platform transparency. CORBA support is available for over 50 diferent operating systems, including VxWorks, pSOS, Windows 98, Windows NT/2000, all major Unix variants, Linux and mainframe operating systems.
- It provides vendor independence through interoperability.
- CORBA Services provide à la carte functionality.
- Network communication is abstracted from the developer.

3.2.2 Object Request Broker

An object request broker (ORB) is the central component of CORBA [8,10,11,13]. The ORB defines the object model and provides bi-directional location-transparent object access. The ORB is what shields clients from the necessity of dealing with the complexities of remote object communication; the ORB handles all of the difficulties in coordinating the task. The CORBA 2.0 specification mandates inter-vendor ORB compatibility, which is accomplished via the required Internet Inter-ORB Protocol (IIOP). IIOP provides a common communication backbone between different ORBs by adding several CORBA-specific messages to the TCP/IP schema already widely used today. The ORB provides most of the middleware-like services that a robust distributed object system should provide.

It is the ORB that establishes client-server relationships between objects. The ORB intercepts method invocations from client objects and routes them to an appropriate server. The serving component can be a specific object or a general server that delivers the services required to meet the demands of a generic client request. By using an ORB with such capabilities, CORBA shields the programmer from (e.g. the language used to write the cooperating component) as well as run-time variables (e.g. the details of which machine hosts a given component). The ORB does not bind a given component to a client or a server role: the same component acts as a client to other objects yet still delivers requested services, making it also a server.

Figure 1 shows the CORBA ORB structure. Let's first go over the components on client side.

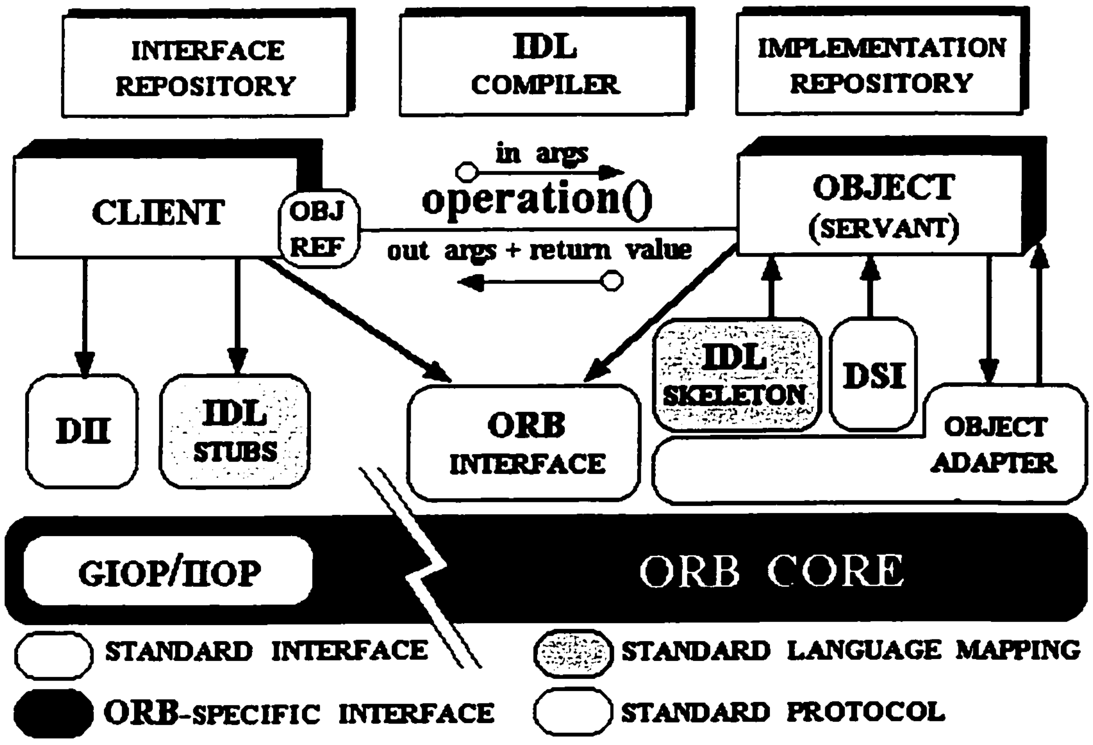


Figure 1. The CORBA ORB structure [11]

- **Client IDL Stubs** provide static interfaces to object services. These precompiled stubs define how clients invoke corresponding services on the servers. From a client's perspective, the stub acts like a local call – it is a local proxy for a remote server object. A client must have an IDL stub for each interface it uses on the server. The stub includes code to perform marshalling. This means that it encodes and decodes the operation and its parameters into flattened message formats that it can transfer over the network to the server. It includes header files that enable you to invoke the method on the server from a higher level programming language like C, C++, Java or Smalltalk without worrying about the underlying protocols or issues such as data marshalling. You simply invoke a language method within your program to obtain a remote service.

- **Dynamic Invocation Interface (DII)** lets you discover methods to be invoked at run time. CORBA defines standard APIs for looking up the metadata that defines the server interface, generating the parameters, issuing the remote call, and getting back the results.
- **Interface Repository** is a dynamic metadata repository of the ORBs which contains machine readable versions of the IDL-defined interfaces. API's allow to obtain, store and modify the descriptions of all the server components interfaces the methods they support and the parameters they require. Thus, the interface repository allows every component that exists on ORB to have self-described interfaces.
- **ORB Interface** consists of few API's for local services in client. For example, CORBA provides API's to convert an object reference to a string and vice versa. These calls are useful to store as well as communicate object references across the network.

The support for both static and dynamic client/server invocations – as well as the Interface Repository – gives CORBA a leg-up over competing middleware. Static invocations are easy to program, faster, and self-documenting. Dynamic invocations provide maximum flexibility, but they are difficult to program; they are very useful for tools that discover services at run time.

The server side cannot tell the difference between a static or dynamic invocation; they both have the same message semantics. In both case, the ORB locates a server object adapter, transmits the parameters, and transfers control to the object implementation through server IDL skeleton. Here's what CORBA elements do on the server side of Figure 1.

- **Server IDL Stubs** (OMG calls them "skeletons") provide static interfaces to each object exported by the server. The stubs are created using an IDL compiler.
- **Dynamic Skeleton Interface (DSI)** provides a run-time binding mechanism for servers that need to handle incoming method calls for components that do not have IDL-based compiled skeletons. The Dynamic Skeleton looks at parameter values in an incoming message to figure out whom it's for – that is, the target object and method. In contrast, normal compiled skeletons are defined for a particular object class and expect a method implementation for each IDL-defined method. Dynamic skeletons are very useful for implementing generic bridges between ORBs. They can also be used by interpreters and scripting languages to dynamically generate object implementations. The DSI is the server equivalent of DII. It can receive both static and dynamic method invocations.
- **Object Adapter** sits on top of the ORB core communication services and accepts requests on behalf of the server objects. It provides the run-time environment for instantiating server object, passing requests to them, and assigning them object IDs – CORBA calls the Ids object references. The Object Adapter also registers the classes it supports and their run-time instances (i.e. objects) with Implementation Repository. CORBA 2.0 specifies that each ORB must support a standard adapter called the Basic Object Adapter (BOA). Servers may support more than one object adapter. CORBA 3.0 introduces a portable version of BOA called the Portable Object Adapter (POA).

- **Implementation Repository** provides a run-time repository of information about the classes a server support, the objects that are instantiated, and their IDs. It also serves as a common place to store additional information associated with implementation of ORBs. Examples include trace information, audit trails, security, and other administrative data.
- **ORB Interface** is similar to the client side which has certain API's to support local services.

3.2.3 Interface Definition Language (IDL)

IDL is the key for interoperability in CORBA [8,12,13]. IDL is the neutral intermediate language that specifies a component's boundaries, interfaces with potential clients or any description of any resource or service that the server component wants to expose to its client. The CORBA IDL is declarative, that is, it separates interfaces from implementation details. IDL specified methods can be written in and invoked from any language that provides CORBA bindings. It acts basically as an intermediate neutral interface that allows client and server objects written in different languages to interoperate across networks and operating systems.

Clients necessarily use IDL for two purposes:

- Clients that invoke existing services.
- Developers who use IDL to extend an existing component's functions by subclassing.

IDL can be used to specify component's attributes, the parent classes it inherits from, the exceptions it raises, typed events, interfaces and the methods an interface supports - including the input and output parameters and their data types.

IDL grammar is a subset of C++ with additional keywords to support distributed concepts. It supports C++ like syntax for constants, type and operation declarations. IDL provides a direct path between its interfaces and the compiled code that implements it. IDL precompiler can directly generate client header files and server implementation skeletons.

3.2.4 CORBA Object Services

CORBA also provides CORBAservices, which define system-level object frameworks that extend the CORBA model. In this report, instead of cover all services that CORBA provide, I will only simply describe two services that can be used to locate objects in a system.

The first is the Naming Service. This service is analogous to the white pages in a phone book: an object looks up another object by the name under which the object registered itself with the ORB on initialization. This method of finding an object relies on unique signatures: a client must know the exact name a server gave when it registered for use. The second service is the Trader Service, which is like the yellow pages: objects can ask the Trader Service what objects with certain service characteristics have registered. The trading repository then returns references to salient objects and gives the client information regarding the properties of the services. The client then chooses a server to contact for the needed services.

3.3 VisiBroker: an implementation of CORBA specification

VisiBroker [14, 15] is the Borland's solution to CORBA; it is fully compliant with the latest CORBA specification. VisiBroker makes it easy for you to develop distributed, object-based clients and servers. In addition to providing the features defined in the CORBA specification, VisiBroker offers enhancements that increase application performance and reliability. VisiBroker provides high availability and load balancing through its agent-based architecture, which is implemented as two software components: SmartAgents (osagent) and Object Activation Daemons (OAD).

VisiBroker's osagent is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. When a client application invokes the bind method on an object, the osagent locates the specified implementation and object so that a connection can be established between the client and the implementation. Object implementations register their objects with the osagent so that client applications can locate and use those objects. When an object or implementation is destroyed, the osagent removes them from its list of available objects.

An osagent may be started on any host. To locate an osagent, client applications and object implementations send a broadcast message, and the first osagent to respond will be used. Once an osagent has been located, a point-to-point UDP communication is established for registration and look-up requests. The UDP protocol is used because it consumes fewer network resources than a TCP connection. All registration and locate requests are dynamic, there are no required configuration files or mappings to maintain.

When multiple instances of the osagent are started on different hosts, each osagent will recognize a subset of the objects available and communicate with other osagents to locate objects it cannot find. If one of the osagent processes should terminate unexpectedly, all implementations registered with that agent will be notified and they will automatically re-register with another available osagent.

The Object Activation Daemon (OAD) is VisiBroker's implementation of the Implementation Repository. The Implementation Repository provides a runtime repository of information about classes a server supports, the objects that are instantiated, and their IDs. In addition to the services provided by a typical Implementation Repository, the OAD is used to automatically activate an object implementation when a client references the object. It cooperates with the osagent to ensure that server objects are activated on demand and are shut down when they are no longer needed.

It is sufficient to run a single SmartAgent on a given local area network. However, running multiple SmartAgents provides higher availability, with no additional effort on the part of the application developer.

- Multiple SmartAgents cooperate with Object Activation Daemons to ensure uninterrupted service even if the SmartAgent or a server object becomes unavailable during processing. If the host running a SmartAgent becomes unavailable, objects registered with that SmartAgent are re-registered with another available SmartAgent. This is transparent to the applications.

- If a server object becomes unavailable, its clients cooperate with the SmartAgent to re-establish a connection, whether this means locating another instance of the server object or using the Object Activation Daemon to start a new instance. In either case, service to clients proceeds uninterrupted.

Load balancing is accomplished using a simple round-robin load-balancing algorithm. SmartAgents allocate client location requests to multiple object instances, ensuring that no single object instance becomes overloaded and providing a consistent level of service to client objects. With the 3.0 release of VisiBroker, more sophisticated load balancing algorithms can be added by the application developer.

Chapter 4

4 Project Design

4.1 System Architecture

The 2-D plotter is an application that lets anyone using the Internet to visualize his/her mathematical expressions on their own computer screen. In design phase, this software system is divided into two subsystems. One subsystem is responsible for interacting with the user. It accepts user-input expression and its settings, and then sends request to another subsystem for parsing and evaluating the expression. After it gets the results back from that subsystem, it plots the graph on the screen. The other subsystem is responsible for providing services that parse and evaluate an expression. It parses the expression to check its correctness and evaluates the expression using the data provided in the request. It is easy to see how the 2D-plotter system is naturally mapped into the client/server framework. It makes sense to construct the subsystem that is responsible for parsing and evaluating expressions as a server. Since the parsing and evaluation processes are uniform to all users of the system, and the optimal parse algorithm can be changed and the build-in functions can be added at any time. It is also logical to treat the other system as a client that has a GUI and needs to request services to accomplish its task.

For rapid and effective development of distributed applications, we need a middleware that can provide a framework to handle heterogeneous data representation, hardware, and software environments across networks. In this project, I chose CORBA/VisiBroker as the middleware. The reason to choose VisiBroker, besides the general benefits of using

CORBA (described in section 3.2.1), is also based on the following consideration particularly for this project.

- The previous work [2] has provided Java RMI solution for this application; another option is using CORBA technique.
- One of the project requirements is that this system can be applied to heterogeneous platforms across networks, which makes the choice of CORBA preferable to DCOM, which runs only on Windows operating system.
- To efficiently reuse existing code and modules developed previously for this application is an important goal of this project. Since the nice GUI component in Cai's work [2] was developed using Java language and the original parser module implemented with C++ language, it requires the chosen middleware to be language independent. Wrapping legacy applications and hiding language heterogeneity are just two of the main application areas for CORBA, which fit our needs perfectly.
- The reason to choose VisiBroker instead of other CORBA implementations such as Orbix is that the software VisiBroker is available in the department's lab and is free for students to use.

Figure 2 shows the overall architecture of the plotter application. In this diagram:

- *PlotterClient* is the client-side component that represents the application's presentation logic. In this component, many classes that consist of GUI elements are largely reused from Cai's previous Java implementation [2].

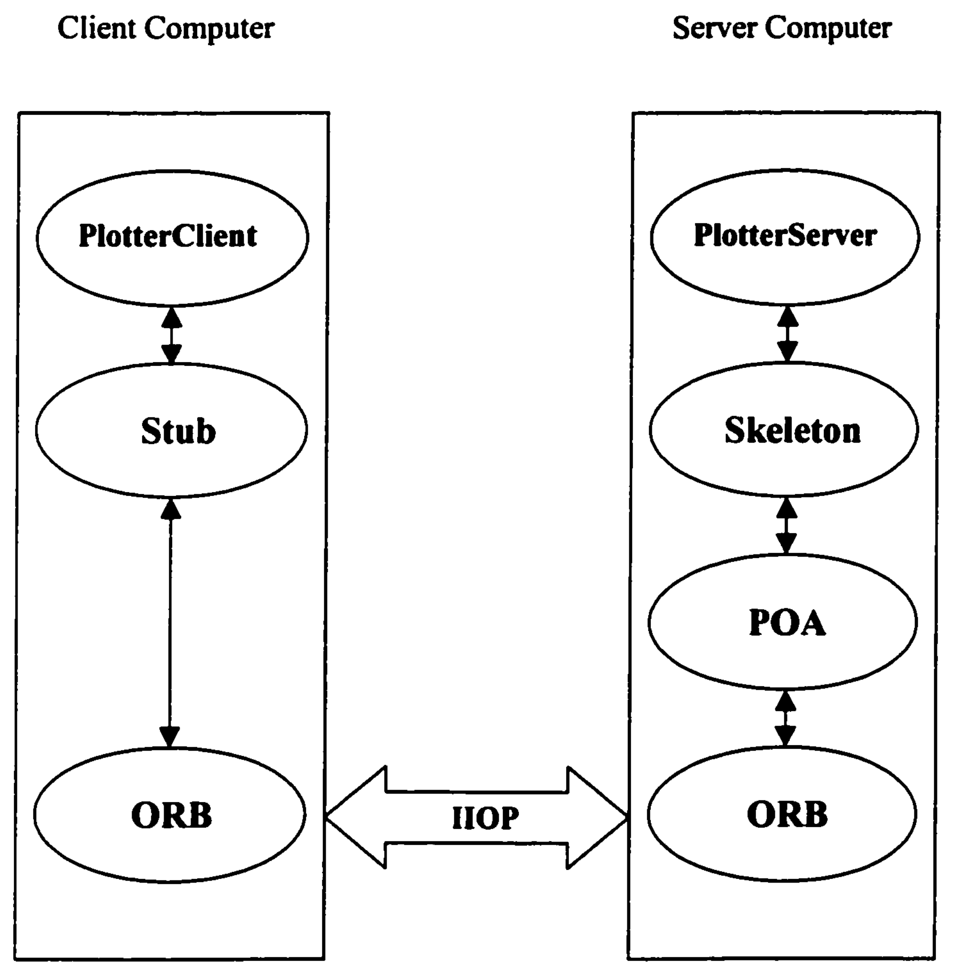


Figure 2. The architecture of the plotter application.

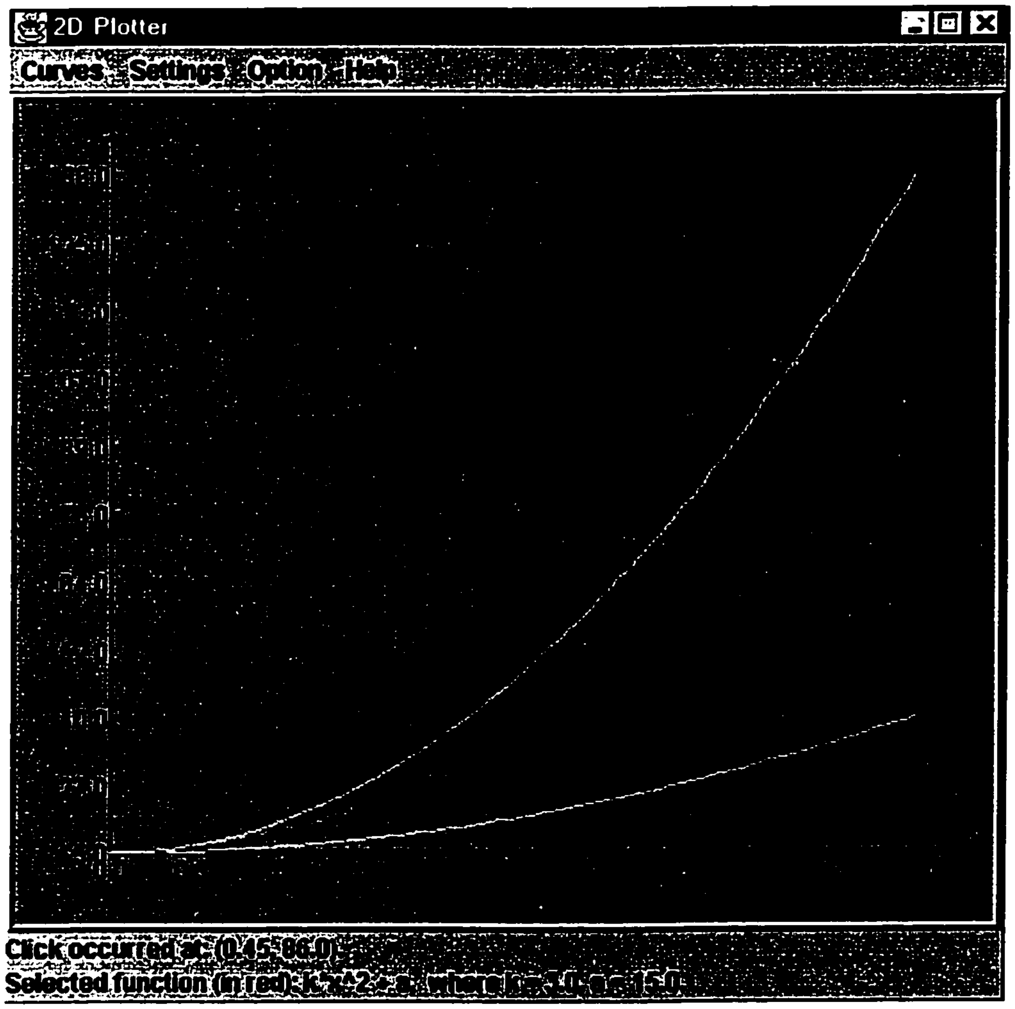
- *PlotterServer* is the server-side component that represents the application's business logic. The component is basically a wrapper for the existing parser C++ module provided with the server's public interface. This public interface is a crucial part of CORBA application because it tells what services the server can provide and how the client can request them, that is, it is a contract between client and server. The interface can be declared using CORBA IDL language and will be discussed in detail in the next chapter.

- *Skeleton/Stub* provides static interfaces for objects exported by the server in server/client side. The skeleton/stub is a local proxy for the remote client/server. They are automatically generated from the server's public interface by VisiBroker's IDL-to-language compiler.
- *ORB(Object Request Broker)/POA(Portable Object Adapter)* are, for our purposes, objects that are instantiated in application code. They come from the Visibroker libraries that ship with the Visibroker installation and are essentially "black boxes" that you can use without totally understanding their underlying complexity. The server objects interface with the POA (via generated skeleton code) and the POA interfaces with the ORB so that the ORB may communicate to the client. The ORB's purpose in life is to provide a communication mechanism between applications [13].

4.2 User Interface

The term "User Interface" refers to the methods and devices that are used to make the interaction between machines and the humans who use them. The user interface is very important for software systems. Users can use software only through its user interface. Good user interface design should let users carry out their task effectively, efficiently, enjoyably and safely. For this application, the goal of the graph user interface (GUI) design is easy to learn and intuitive to use. To achieve this goal, the GUI is designed to use as many standard window GUI elements as possible. Users who have any experience with window applications can use this application easily.

Figure 3 shows the 2D plotter's main user interface. The plotter's user interface is consist of three parts. There is a menu bar on the top, a graph display area on the middle and an information bar on the bottom.



Click occurred at: (0.45, 86.0)
Selected function (in red): k*x^2 + a, where k = 1.0, a = 15.0

Figure 3. The main user interface of the plotter

4.2.1 The Graph Display Area

The graph display area is the major part of the application's GUI. It is the place that the user's expressions are visualized. In the application's 'initial' state, the user can see an empty graph shows only two bare axes without labels or scales. When the application enters its 'plot' state, that is, the plotter is starting to plot first curve, labels and scales are written on the axes based on the data ranges of the curve and the name of the independent variable. The axes and their labels and scales will keep unchanged as long as the plotter stays on its plot state. However, user can plot multiple curves on display area. Each set of curves (curves that belong to same expression but with different parameter values are treat as one set of curves) is shown with a different color. Red is dedicated for highlighting the curve that is selected by the user by clicking on it.

4.2.2 The Information Bar

The information bar is designed for two purposes. First, it is used to display the coordinates of the point if the user clicks in the graph display area. This enables the user to get interesting data from the curve(s) by simply clicking the mouse. Another purpose is to provide a direct connection between the plotted curve and its expression. When the user clicks on a curve on the graph display area, the color of the curve will turn to red indicating its selection and at same time its expression is shown on the information bar.

4.2.3 The Menus

Menus are widely used in window applications; they help the user to make selection easier. In this application, the menu bar contains four menus: *Curves, Settings, Options,* and *Help.*

4.2.3.1 The *Curves* Menu

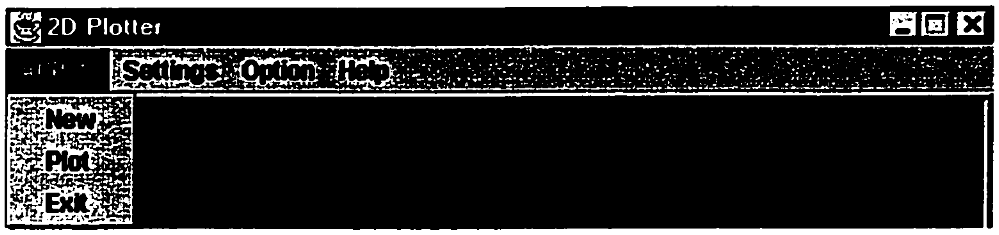


Figure 4. The *Curves* Menu

As can be seen from Figure 4, The *Curves* menu consists of three items: *New*, *Plot* and *Exit*. The *New* menu item is used when the plotter is in its 'plot' state, the selection will change state to 'initial', which means it will clean all graphs in the display area and leaves only two bare axes. The selection of *Plot* menu item may happen in both 'initial' and 'plot' states. In the former case, the plotter just starts plotting the first curve or set of curves and brings the plotter to the 'plot' state, while in the second case, the plotter continually stay in its 'plot' state and adds more curve(s) on the same display area. In both cases, the plotter uses the current settings to plot the curve(s). If in current settings, the expression is not valid or the independent variable is not set, an error message will be displayed in a separate window. In the drawing process, if any singularities are detected, a warning message window will be popped up while the program continues to draw the rest of the curve. The selection of the *Exit* menu item at any time will quit the application.

4.2.3.2 The *Settings* Menu

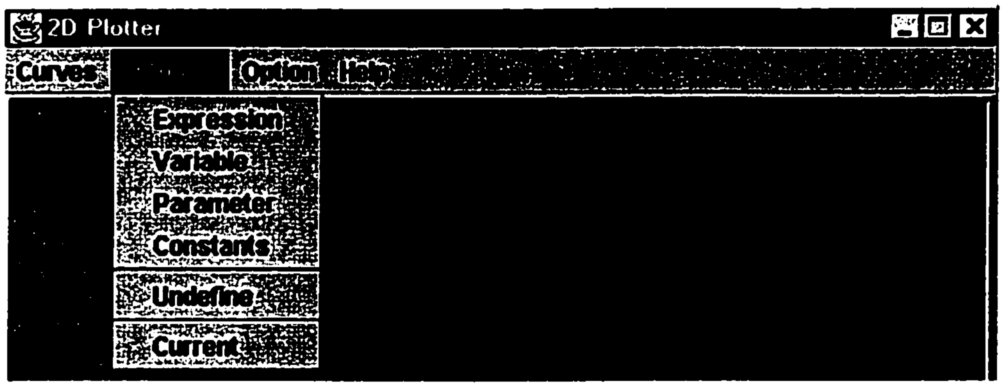
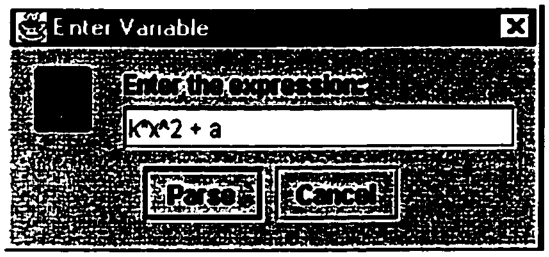


Figure 5. The Settings Menu

Figure 5 shows the *Settings* menu, it has six menu items with the following functionalities.

- *Expression* menu item

  This menu item is used to allow the user to enter an expression. Selecting this item will pop up a dialog box as shown in figure 6. The user types in the expression and clicks the *Parse* button; this will send a request to server for parsing the expression. After the request has been answered, another message window (see figure 7) will pop up to inform the user of the parsing result (success or fail).



k*x^2 + a

Figure 6. The expression dialog box.

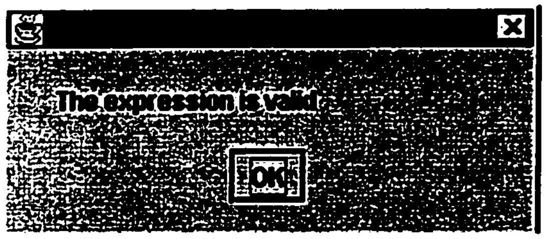

The expression is valid

Figure 7. The message window for parsing result.

- *Variable* menu item

  Select this item will bring up a 'Enter Variable' dialog box (see figure 8) to allow user to specify the expression's variable name and its data range.

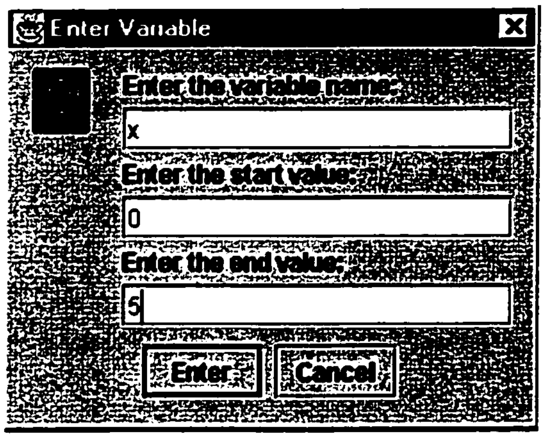


Figure 8. The dialog box for setting the expression variable.

- *Parameter* menu item

  This menu item allows user to enter the expression's parameter name, its start value, end value and the step value from its 'Enter Parameter' dialog box (see figure 9).

- *Constant* menu item

  This menu item allows user to enter the expression's constant names and their values from the 'Add Constant' dialog box (see figure 10).

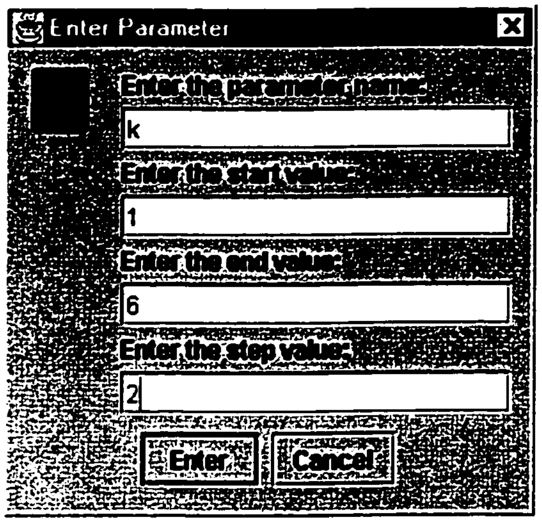


Figure 9. The dialog box for setting the expression parameter.

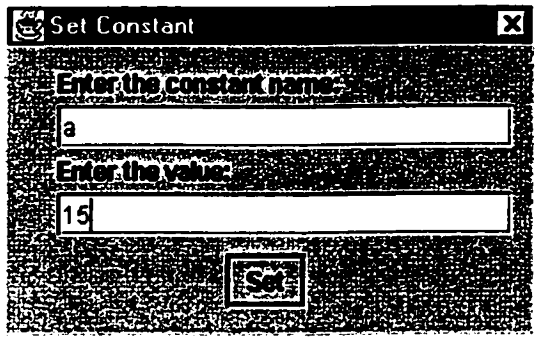


Figure 10. The dialog box for setting expression constant.

- *Undefine* menu item

  This menu item allow user to remove the definition of specify letter in the expression in order to have a new definition. Figure 11 shows the dialog box for this purpose.

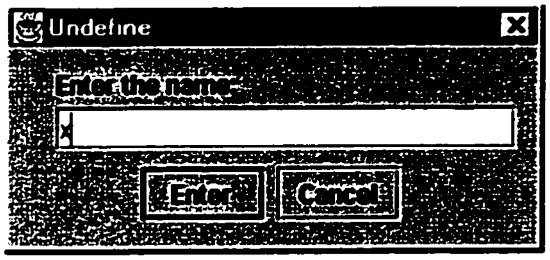
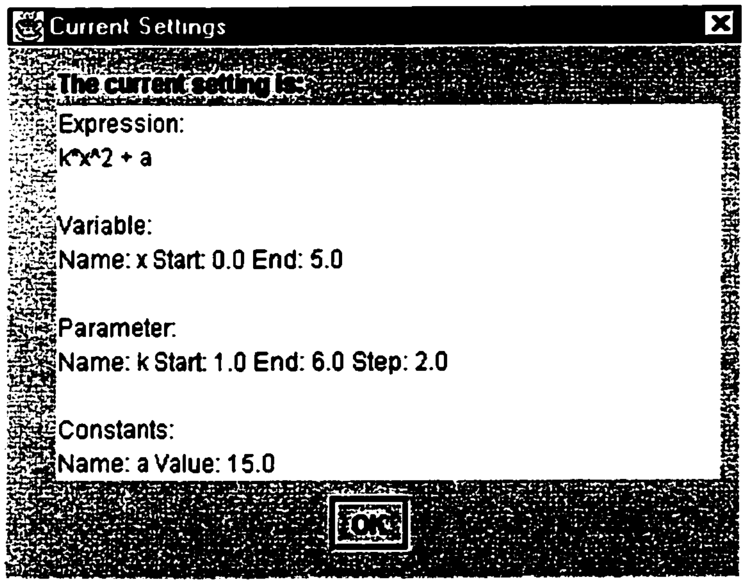


Figure 11. The Undefine dialog box.

- *Current* menu item.

The selection of this menu will bring up a window (see figure 12) to inform user all current settings for the expression, including the expression itself, the variable name and its data range, the parameter and its start, end and step values, and the constant names and values.



```
The current setting is:
Expression:
K*x^2 + a

Variable:
Name: x Start: 0.0 End: 5.0

Parameter:
Name: k Start: 1.0 End: 6.0 Step: 2.0

Constants:
Name: a Value: 15.0
```

Figure 12. The window showing the expression's current settings.

4.2.3.3 The *Option* Menu

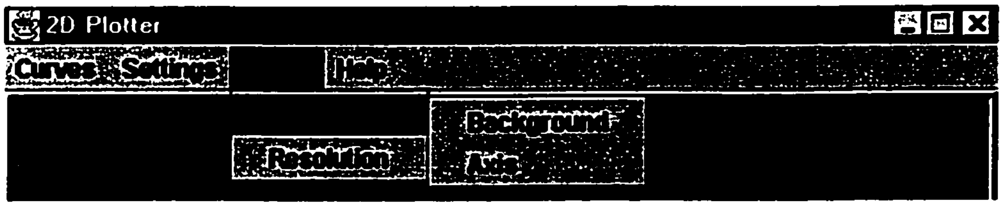


Figure 13. The *Option* menu.

As figure 13 shows, this menu has two menu items: *Color* and *Resolution*. These menu items provide options for user. If they not selected, default values will be used.

- *Color* menu item.

  The *Color* menu item has two submenus, it allows user to choose their favorite colors for background of the plotter's display area and the axes from a Java color choosier window (see figure 14).

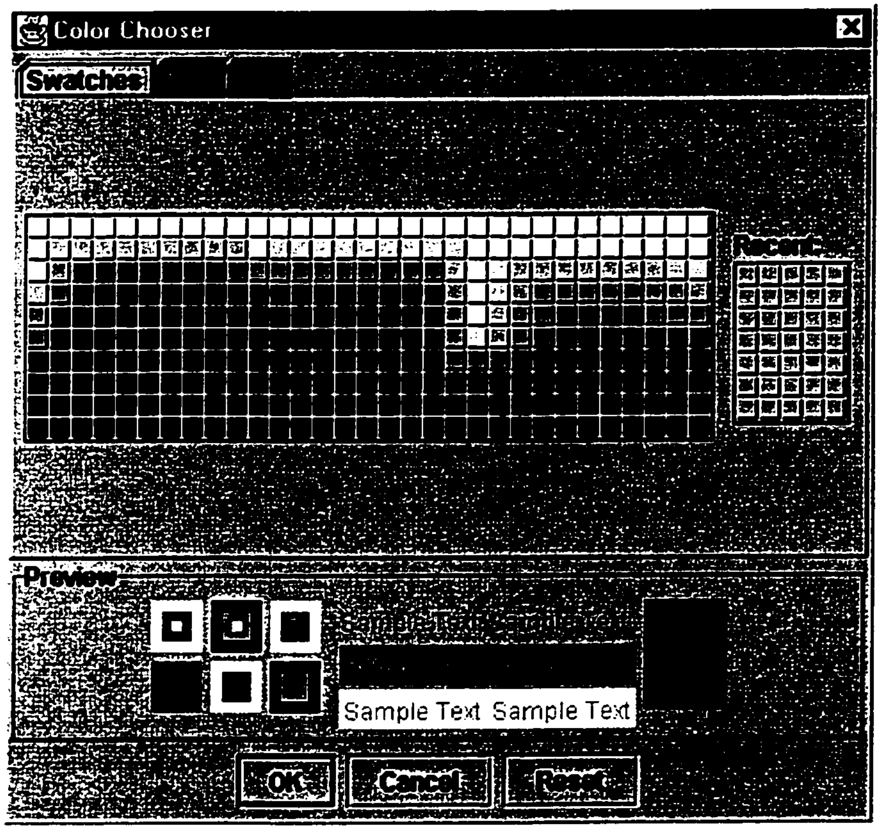


Figure 14. The color chooser.

- *Resolution* menu item.

  This menu item allow user to choice different plotting resolution levels. It provides three levels: low, medium and high level. Higher resolution give more smooth curves but is slower, while lower resolution is rougher but faster. Figure 15 shows the dialog box for this purpose.

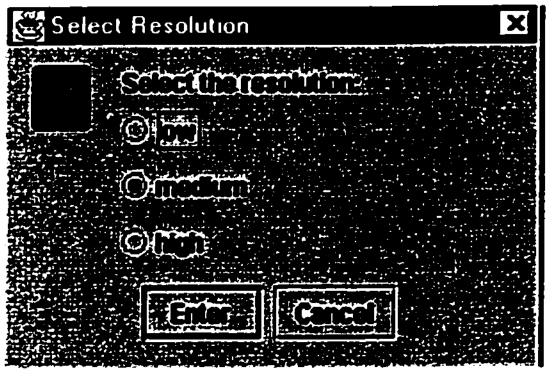


Figure 15. The dialog box for selecting plotting resolution.

4.2.3.4 The *Help* Menu

This menu is intended to teach user how to use the plotter quickly. In its simplest implementation, it may bring up a window and give a description of how to use the menus described above.

Chapter 5

5 Implementation and Deployment

In this chapter, we describe the implementation and deployment of the 2D-plotter, starting with the development process.

5.1 The Development Process

As shown in Figure 16, the following steps of development have been used to implement this 2D-plotter CORBA server and client.

1. Identify the objects that will be used in a distributed object system: Object Analysis and Design.
2. Write the IDL specification for the objects identified.
3. Compile the IDL for the desired language mappings, using the specific IDL compiler for whatever language and platform that are being used. In this project, the VisiBroker's IDL-to-Java compiler idl2java is used for client to generate client stub code, and the VisiBroker's IDL-to-C++ compiler idl2cpp is used for server to generate server skeleton code.
4. Code the server. Create object server by completing the object implementation in C++.
5. Code the client. Complete client code by integrating the client stub code with user interface. The client is implemented in Java.

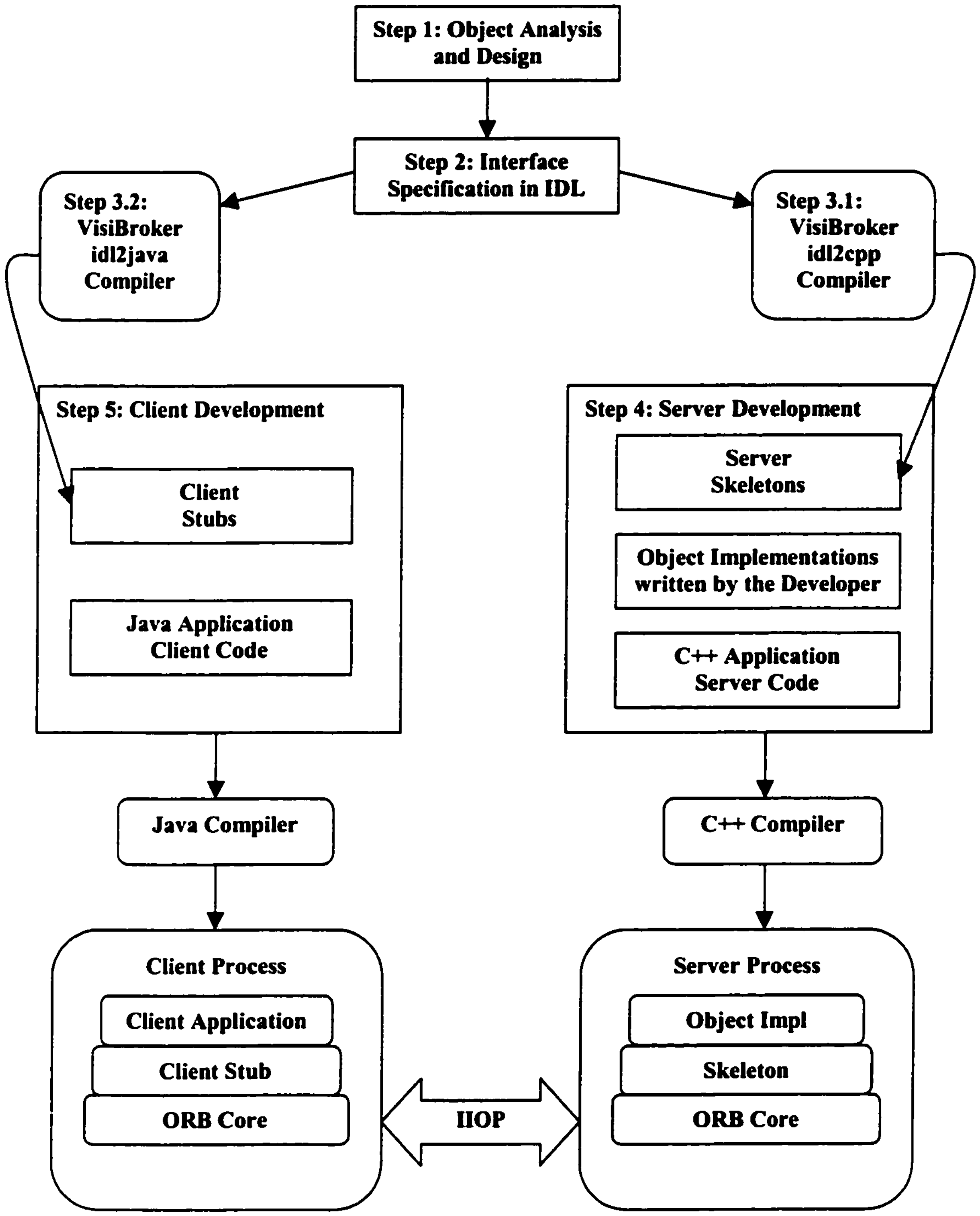

Step 1: Object Analysis and Design
Step 2: Interface Specification in IDL
Step 3.2: VisiBroker idl2java Compiler
Step 3.1: VisiBroker idl2cpp Compiler
Step 5: Client Development
Client Stubs
Java Application Client Code
Step 4: Server Development
Server Skeletons
Object Implementations written by the Developer
C++ Application Server Code
Java Compiler
C++ Compiler
Client Process
Client Application
Client Stub
ORB Core
IIOP
Server Process
Object Impl
Skeleton
ORB Core

Figure 16. The steps of development

5.2 The IDL Specification

The Interface Definition Language (IDL) describes only the public attributes and operations that any client would need to successfully interoperate with the distributed system. By inspecting the IDL of the system, we should be able to ascertain all our needs to know about how to interface with the system. Here is the complete IDL for the 2D-plotter system:

```
#ifndef _plotter_idl_
#define _plotter_idl_

module Plotter
{
      struct ResStruct
      {
            boolean valid;
            double value;
      };

      typedef sequence<ResStruct> ResStructSeq;
      typedef sequence<ResStructSeq> Results;

      exception PlotterException
      {
            string message;
            string exceptionName;
      };

      interface PlotterServer
      {
            // parse the expression
            void checkExpr(in string expr)
                  raises(PlotterException);

            // set variable in expression
            void setVariable(in string vname, in double vmin, in double vmax)
                  raises(PlotterException);

            // set parameter in expression
            void setParameter(in string pname, in double pfirst,
                              in double plast, in double pstep)
                  raises(PlotterException);

            // set constant in expression
            void setConstant(in string cname, in double cval)
                  raises(PlotterException);

            // evaluate the expression
            void evaluate(in double step, out Results res)
                  raises(PlotterException);

            // clear
            void clear();
      };
};
#endif
```

Let's inspect the IDL in more detail:

A module is used as naming scope to avoid name clashes when using several IDL declarations together; that is to say that an interface or type name used within one module will not conflict with the same name used in another module. In this project, I named the module *plotter*.

Struct is a data type in IDL that allows related items to be grouped together in a useful fashion. *ResStruct* is declared as a *Struct* to represent the evaluated result of an expression at given point. *ResStruct* contains two fields: *valid* and *value*. The field *valid* is a boolean type, it will get value `false` if the expression has a singularity at this point. The other field *value* is used to store the expression's value at this point.

The IDL *typedef* statement is generally used to create an alias for a defined type. In this case, we want to define a data structure to represent evaluated results for an expression for all points in the given independent variable data range. The *Results* is declared as an IDL sequence type (A sequence is similar to a one-dimensional array, but has its length determined at run time). Since one expression may correspond to one set of curves if the expression has a parameter and the parameter has multiple values (see section 2.4), so in general case each element in *Results* sequence contains data for one curve that is represented as another sequence *ResStructSeq*. The *ResStructSeq* is a sequence of *ResStruct* that we just discussed.

The standard way of processing errors in CORBA is through exceptions. An IDL operation may raise an exception indicating that an error has occurred. Exceptions provide a clean way to allow an operation to report an error to the caller. In this project, defined a general *PlotterException*, which contains *message* field for detail descriptions on which errors happen and *exceptionName* field for what kind of error happens.

The IDL interface provides a description of the functionality that will be provided by an object. An interface definition typically specifies the operations and the parameters of each operation. The syntax for IDL operation or method is as follows:

```
<method_return_type> <method_name> (<parameter_direction>
                              <parameter_type>  <parameter_name>, (1-n))
```

There are three choices for parameter direction; 'in' for parameter passed from client to the called object, 'out' for parameter passed from the called object to client, and 'inout' for both directions.

The *PlotterServer* interface in this project defines six operations, each operation will raise CORBA exception *PlotterException* if an error occurs when the server processes the operation. The six operations are:

- *checkExpr*: The plotter server provides a service to parse expression through this operation, it accepts an expression as a string and checks the expression against the parser's grammar. If anything in the expression does not conform to the grammar, the predefined *PlotterException* with appropriate information will be thrown.

- *SetVariable*: It allows the client to set the independent variable in the expression, including the variable name, start and end values. If the name is not in the expression, an exception will be thrown.
- *SetParameter*: It allows the client to set the expression's parameter, including its name, start value, end value and the step value. If the name is not in the expression, an exception will be thrown.
- *SetConstant*: It allows the client to set the expression's constant, including its name and value. If the name is not in the expression, an exception will be thrown.
- *evaluate*: Client can call this operation to evaluate the expression based on its variable, parameter and constant settings. The 'in' argument in this operation 'step' is used to tell server what the step value of the variable used to evaluate the expression. The 'out' argument 'res' is the result of the evaluation and send back from server to client.
- *clear*: Client call this method to clean up old expression settings on the server object and ready for parsing and evaluating new expression.

5.3 Server Implementation

The server implementation has two major parts; (1) Create the server mainline. This is the program entry point and is typically used to initialize the system, create objects offered for service, and connect them to the ORB so that they may handle requests. (2) Implement the object.

5.3.1 Creating server mainline

The implementation of server mainline *PlotterServer.cpp* in this project has following steps:

- Initialize the ORB

  As stated in previous chapter, the ORB provides a communication link between client requests and object implementations. Each application must initialize the ORB before communicating with it. Here is the code for this:

```
// Initialize the ORB.
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

- Create and setup the POA

  In basic term, the POA determine which *servant* should be invoked when a client request is received, and then invokes that servant. A servant is a programming object that provides the implementation of an abstract object. Each ORB supplies one POA (called the root POA). Sometimes we need create additional POAs and configure them with different behavior. We first create a root POA as in the following code.

```
// get a reference to the root POA
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
```

  Then a child POA 'myPOA' is created from the root POA and has a persistent lifespan policy. The POA Managers for the root POA is used to control the state of this myPOA.

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
                                PortableServer::PERSISTENT);

// get the POA Manager
PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

// Create myPOA with the right policies
PortableServer::POA_var myPOA = rootPOA->create_POA(
                    "plotter_agent_poa", poa_manager, policies);
```

- Creating and activating a servant

  We create an instance of the servant class *PlotterServerImpl* (this is the actual object implementation and will be discussed in the next section) and give a name as its object ID. Then activate the servant with its ID on *myPOA*.

```
// Create the servant
PlotterServerImpl plotterServant;

// Decide on the ID for the servant
PortableServer::ObjectId_var plotterId =
             PortableServer::string_to_ObjectId("PlotterServer");

// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(plotterId, &plotterServant);
```

- Active the POA Manager

  Activating the POA Manager associated *myPOA* will change the state of POA Manager from holding state to active state, so that the clients request can be processed.

```
// Activate the POA Manager
poa_manager->activate();
```

- Registing with the Naming Service

  First we obtain a reference to an initial context of a Naming Service via the ORB's bootstrap mechanisms. Then register our object into Naming Service by binding its name with its object reference.

```
CORBA::Object_var reference
                 = myPOA->servant_to_reference(&plotterServant);

CosNaming::NamingContext_var rootContext
              = CosNaming::NamingContext::_narrow(
                    orb->resolve_initial_references("NameService"));

CosNaming::Name name;
name.length(1);
name[0].id = "my_plotter";
name[0].kind = "";
rootContext->rebind(name, reference);
```

- Entering the dispatch loop

The last step of coding the server mainline is to enter a dispatch loop by calling run() on the ORB to wait for incoming invocations.

```
// Wait for incoming requests
orb->run();
```

5.3.2 Implementing the Object

This part of development involves actually implementing the interface defined in the project IDL specification. It includes the implementation of all six operations in that interface. Since in this project's design phase, we decided to reuse the existing parser component and chose CORBA as our middleware because it is good at wrapping existing application, it only takes a little effort to implement out *PlotterServer* object. Basically our *PlotterServerImpl* class wraps the existing *ParseVal* class. To do this, we create a new instance of *ParseVal* class in the constructor of our *PlotterServerImpl* class and use the *ParseVal's* methods in the implementation of our six operations. The *ParseVal* class offers following service:

- *void parse (char *expression, bool & ok)*: It parses the given expression (the string argument *expression,* and set its bool argument *ok* to *true* if the operation is successful or *false* if it failed.
- *bool set (char *name, double value)*: It sets *name* variable in the expression with given value *value.*
- *double eval ()*: It evaluate the expression with current values of variables.

It is noted that the *ParseVal* class treated all letters in the expression as variables. Our application should distinguish them as constants, parameter and independent variable. This is the reason that our object interface has separate operations for the different role of letters in the expression. The remaining implementation for these operations is straightforward. For example, for operation *evaluate,* we first need call *ParseVal's set* method multiple times to set constants, parameter and variable. Then based on the expression's settings we need to calculate all the points across the entire ranges of the variable and the parameter and call *ParseVal's eval* method for each point to get the expression's value. Finally we need to populate our predefined data structure *Results* with these values, and the operation return it to the client.

5.4 Client Implementation

PlotterClient is implemented as a Java application. It has two major parts: (1) a CORBA client part whose responsibility is to talk with CORBA ORB in order to find the server object and invoke methods on it. (2) a GUI client part whose responsibility is to provide GUI elements in order to interact with user and finally to plot curves on the screen.

5.4.1 CORBA Client Implementation

As in the CORBA server implementation, the first step is to initialize the ORB in order to start communicating with it.

```
// Initialize the ORB.
orb = org.omg.CORBA.ORB.init(args, null);
```

Then the client tries to obtain the server object reference. In this project, we use the Visibroker's Naming Service to help the client to get the object reference. To do so, we need first to get the root context of the Naming Service and then use its *resolve* method to get the object reference by providing it the object name.

```
// Obtain the root context.
org.omg.CORBA.Object rootObj =
                    orb.resolve_initial_references("NameService");
NamingContextExt root =NamingContextExtHelper.narrow(rootObj);

// Locate a plotter server through the Naming Service.
org.omg.CORBA.Object myObj = root.resolve(root.to_name("my_plotter"));
aPlotterServer = Plotter.PlotterServerHelper.narrow(myObj);
```

After the point that the reference of the server object is obtained, the client can invoke all methods provided by the server object as if the server is resided on the client machine. The actual location, the detail implementation of the server and the platform that the server running on is all shielded by the CORBA middleware from the client developer.

5.4.2 GUI Client Implementation

PlotterClient is implemented as a Java application. It has a main *JPanel* container, which contains three GUI elements: a menu bar (*mb*) on the top, a display area (*rectangleArea*) on the middle and a label (*label*) on the bottom. The *label* is a *JLabel* object, and is used to show two pieces of information: one for the coordinates of a point in the display area when the user clicks it and another for the expression of a curve when user clicks on the curve. The *mb* is a *JMenuBar* object, which contains a group of *JMenu* objects. Each *JMenu* handles one category of the application menu, which in turns contains one or more menu items *JMenuItem* Objects. The *rectangleArea* is an instance of

RectangleArea class that inherits the *JFC* class *JPanel*. The *RectangleArea* class has two responsibilities: getting server object's reference with aid of CORBA (for details, see section 5.4.1) and creating a display area with the graph drawn on it.

Java graphic programming is event driven, the program interacts with the user and the events drive the execution of the program. An event can be generated by external user actions or by the operating system. The GUI component on which the event is generated is called a source object. In this project, all our menu item objects are source objects. A predefined event *ActionEvent* will be generated when the user makes selection on one of the menu items. An object interested in the event receives the event. Such an object is called a "listener". To become a listener, the object must be registered as a listener by the source object. The source object notifies all the registered listeners when the event occurs. Upon receiving the notification, the Java runtime system invokes the event-handling method on the listener object to respond to the event. It is the listener object's responsibility to implement the event-handling methods for a correct response. Our *PlotterClient* is a listener object, it is required for *PlotterClient* to implement all event-handling methods for each menu item object used in this application. In this project there are two major categories of event-handling methods, depending on their tasks. One type of event-handling method (such as *HandleExpression*, *HandleVariable* etc.) is responsible to popup a dialog box to accept the user's input. A dialog box is implemented by an instance of a class that inherits the *JDialog* class. Each of these classes may contain a set of GUI elements, such as text fields (*JTextField*), buttons (*JButton*), or radio buttons (*JRadioButton*), depending on its detail usage. These classes also store a reference to the owner *PlotterClient* object, via which it may send validated user inputs back to the *PlotterClient*. Another type of event-handling methods (such as *HandlePlot*) is to invoke methods in *rectangleArea* object, which in turn communicates with the server via CORBA and plots graphs on the display area. In order to show the coordinates of a point on the display area when the user clicks on it and the expression of the user-selected curve, we implement *rectangleArea* as another listener object to respond to the mouse click event on the display area.

In *RectangleArea* class's constructor, we make the connection to the *PlotterServer* via CORBA by getting the server object reference (see detail code in section 5.4.1). After the object reference is obtained, we can use it to get the services provided by server. The *RectangleArea* implements methods, such as *parseExpr*, *setVariable*, and *plotCurve*, which call corresponding methods in the server to get the expression parsed and evaluated. These methods are invoked by the *PlotterClient* object's event-handling method *HandlePlot* when the user selects the *plot* menu item.

5.5 Deployment

Application deployment is an important phase in the distributed application. VisiBroker is also used in the deployment phase. It requires the ORB be installed on the machine that runs the server application as well as the machines that run the client programs. This section will give detail on our plotter project's deployment process. We assume this deployment is done on Windows NT platform; the deployment is similar for other platforms.

5.5.1 Generate Skeleton and Stub

To use VisiBroker run-time libraries, we need to set the run-time environment for it. For server machine where we use C++ language, we need to include VisiBroker's *bin* sub-directory in the PATH environment. For client machine where we run Java application, we need to set CLASSPATH to include VisiBroker's *lib* sub-directory.

First we use VisiBroker's C++ IDL compiler *idl2cpp* to compile our project's interface IDL (*Plotter.idl*):

```
        idl2cpp -src_suffix cpp Plotter.idl
```

It generates C++ stub and skeleton files *Plotter_c.*hh, *Plotter_c.cpp*, *Plotter_s.hh* and *Plotter_s.cpp*.

Then we use VisiBroker's Java IDL compiler *idl2java* to compile the same IDL file:

```
        idl2java -no_comments Plotter.idl
```

It creates Java stub and skeleton files, such as *PlotterServer.java*, ant *PlotterServerPOA.java*

5.5.2 Compile the Server

Compile *Plotter_s.cpp, Plotter_c.cpp, PlotterServer.cpp* (implemented as part of this project) and *Parseval.cpp* (the existing Parser/Evaluation implementation) and link together to create the server program *Plotter.exe*. The Appendix includes a detailed makefile.

5.5.3 Compile the Client

Compile all Java files we implemented along with the generated client Java stub files by executing the following command:

```
        vbjc -d . -g *.java
```

This creates our Java client class *PlotterClient.class*.

5.5.4 Start VisiBroker Smart Agent, Naming Service and OAD

First we start the VisiBroker Smart Agent (described in section 3.3) on the server machine or client machine or any machine connected in the network:

```
        osagent -C
```

Then we start the VisiBroker Naming Service (described in section 3.24) Server (named MY_NS) on any machine that is in the network with the machine running osagent:

```
        nameserv.exe MY_NS -J-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=21111
```

Finally we start the VisiBroker Object Activation Demon (OAD) (described in section 3.3) on server machine:

```
        oad.exe -verbose -VBJprop JDKrenameBug
```

5.5.5 Register Server and Run Client

We first register the server with OAD that the server can be started when client sends request on it.

```
        oadutil reg -poa /plotter_agent_poa -cpp $(MAKEDIR)\Plotter.exe
```

Then we register the server with Naming Service:

```
Plotter register -ORBInitRef
NameService=iioploc://$(COMPUTERNAME):$(NAMESERV_PORT)/NameService
```

Finally we can start the client:

```
vbj -DSVCnameroot=MY_NS MyClient.PlotterClient
```

5.5.6 Activating the Plotter Server

Figure 17 shows the project's deployment configuration and what happens when client sends request to the server. Let's summarize our deployment process first. At beginning, we start the osagent, and then start the Naming Service Server. When the NS server is started, it automatically registered itself to osagent such that the osagent knows where can find NS server. We also start the OAD in order to launch the Plotter server on demand. After that we register the Plotter sever into Naming Service with server's logical name 'my-plotter', the Naming Service make a link from the name to the Plotter server's implementation. Finally we register the server to the OAD, the OAD acts on behalf of its registered implementation by telling that this implementation is actually active within the OAD itself.

In figure 17 following sequence events happen when the plotter client issues a request to the plotter server in our above deployment configuration.

1. Before client issues a request, it first call *resolve_initial_refrernce('NamingService')* to get the object reference of the Naming Service.
2. Since the osagent has the information about the Naming Service, it returns the object reference of the Naming Service.

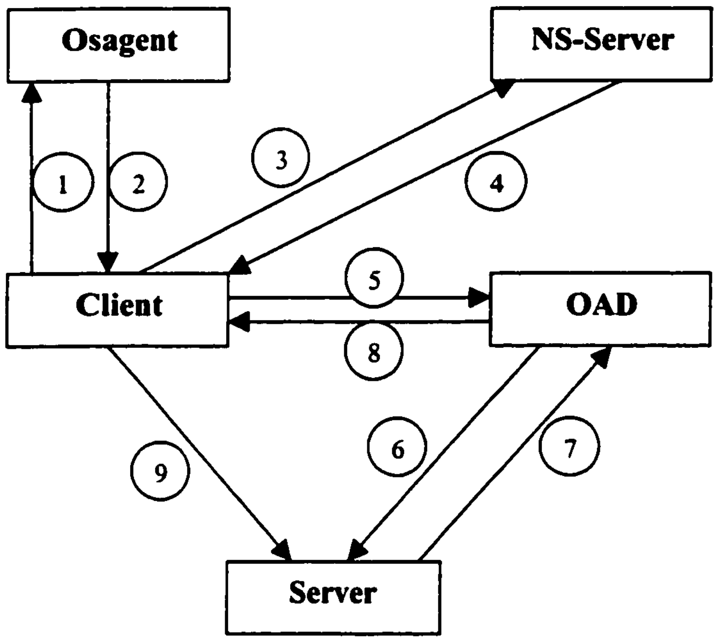


Figure 17. Activating the plotter server.

3. The client then calls the resolve method on the Naming Service to find out the plotter server's object reference from its logical name.
4. The client then calls the resolve method on the Naming Service to find out the plotter server's object reference from its logical name.
5. The Naming Service returns the plotter server's object reference.
6. Since the OAD acts on behave of the plotter server's implementation, when client issues a request, it goes to OAD.
7. When OAD receives the request, it activates the server.
8. The server's POA notifies the OAD that a new implementation is available.
9. Once the server object is available, the OAD sends a reply message to the client.

10. From this point forward, the client communicates directly with the activated server implementation by reissuing the request using the new object reference.

Chapter 6

6 Conclusion

We designed and implemented a distributed 2-D plotting utility for visualizing mathematical functions in Internet environment.

The utility is designed based on the client/server computing model. The utility's server portion is responsible for parsing and evaluating the mathematical functions; it provides services for multiple clients. The client portion provides user graph interface to end-users; it accepts the user's input and plots graph for them. In this design, the user only need the client portion installed on his/her machine and the server portion is a shared resource for many users. This also means that it is easy to upgrade the utility, user can automatically get the services from the latest version when the server is updated.

The utility has been developed using CORBA technology. Obviously, we have named many advantages of CORBA in the previous chapters. However, this project takes more advantage from that CORBA is good at building a distributed application by wrapping and thus reusing old existing system. The CORBA's language independence makes it easy for us to implement server using C++ and implement client using Java. In server implementation, little effort has been taken to wrap the existing C++ parser/evaluation module by provided a well-defined remote interface. In client side, large amount of existing Java GUI implementation has been reused. The CORBA also make the utility easily be ported to various platforms with its platform independence feature.

The utility has been implemented using component technology and object-oriented programming languages. The components and classes are loosely coupled, which make the system extensible and easy for maintenance.

The utility 's GUI has been designed to be simple and straightforward, which make it easy to learn and intuitive to use.

From this work, I have learned a lot about the client/server model and CORBA technology, I also gained a valuable experience on develop a distributed software system. It leads me go through the whole process to develop a CORBA application, including analyzing, designing, implementing, deploying and testing.

The client portion of this system is implemented as a Java application, it requires to install VisiBroker for Java software in the client machine. In future, it would be quite interesting to implement the client as a Java applet. In this case, the user only need a Web browser and a computer connected to Internet.

Bibliography

1. Anh Phong Tran, *2-D Graph Plotter: A Tool for Plotting Functions*, Master's Major Report, Concordia University, 2000.
2. Meng Cai, *A Plotting Tool for Internet Based on Client/Server Computing Model*, Master's Major Report, Concordia University, 2001.
3. William A. Barrett, Rodney M. Bates, David A. Gustafson, John D. Couch, *Compiler Construction: Theory and Practice*, Science Research Associate Inc., 1979.
4. http://java.sun.com/products/jdk/rmi. Java RMI's home page.
5. http://www.javasoft.com/docs/books/tutorial/uiswing, JFC/Swing tutorial from *Sun Microsystem Inc.*
6. http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html. JNI's home page from *Sun Microsystem Inc.*
7. http://www.sei.cmu.edu/str/descriptions/clientserver.html. *Client/Server Software Architectures – An Overview.*
8. Scott M. Lewandowski, *Frameworks for Component-Based Client/Server Computing*, ACM Computing Surveys, Vol. 30, No. 1, March 1998.
9. James E. Goldman, Phillip T. Rawles, and Julie R. Mariga, *Client/Server Information Systems: A Business-Oriented Approach*, John Wiley & Sons, New York, 1999.
10. http://www.omg.org. Object Management Group's home page.
11. http://www.cs.wustl.edu/~schmidt/corba-overview.html. Overview of CORBA.
12. Doug Pedrick, Jonathan Weedon, Jon Goldberg, and Erik Bleifield, *Programming with VisiBroker*, John Wiley & Sons, 1998.

13. Gerald Brose, Andreas Vogel, and Keith Duddy, *Java Programming with CORBA*, third Edition, OMG Press, 2000.

14. http://www.borland.com/techpubs/visibroker. VisiBroker Product Documentation.

15. http://www.borland.com/visibroker/papers/distributed/wp.html#doc2. *Distributed object computing in the Internet age.*

Appendix

A.1 Makefile for the server

```
##################################################################################
# Multi-threaded Windows Visual C++ (nmake) definitions
##################################################################################

##################################################################################
# Platform specific compiler definitions (multi-threaded)
##################################################################################

#DEBUG      = /Z7
STDCC_LIBS = wsock32.lib kernel32.lib user32.lib


##################################################################################
# Default VisiBroker directory location
##################################################################################

!if "$(VBROKERDIR)" == ""
        @echo *** You must set the VBROKERDIR environment variable to the
        @echo *** Root directory where Visibroker ORB is installed.
!else

CC          = @CL -DWIN32 /DVBROKER_V40 /GX /MD
ORBCC       = $(VBROKERDIR)\bin\idl2cpp -src_suffix cpp
LIBDIR      = $(VBROKERDIR)\lib
LIBUSED     = $(LIBDIR)\orb_r.lib $(LIBDIR)\cosnm_r.lib $(LIBDIR)\cosev_r.lib
              $(LIBDIR)\vport_r.lib
CCINCLUDES = -I. -I$(VBROKERDIR)\include -I$(VBROKERDIR)\include\stubs -
              I$(VBROKERDIR)\include\dispatch

##################################################################################
# Compiler flags for debug
##################################################################################

CCFLAGS     = $(CCINCLUDES) $(DEBUG)

##################################################################################
# Standard build rules for .cpp files
##################################################################################

.SUFFIXES: .CPP .obj .h .hh

.CPP.obj:
        $(CC) $(CCFLAGS) -c $<

EXE = Plotter.exe

all: $(EXE)

default: $(EXE)

clean:
        del *.obj
        del *.hh
        del *_c.cpp
        del *_s.cpp
        for %e in ($(EXE)) do del %e
```

```
Plotter_c.cpp: ..\idl\Plotter.idl
	$(ORBCC) ..\idl\Plotter.idl

PLOTTER_OBJ = Plotter_c.obj Plotter_s.obj parseval.obj PlotterServer.obj

Plotter.exe: $(PLOTTER_OBJ)
	$(CC) -o Plotter.exe $(PLOTTER_OBJ) $(LIBNAME) $(LIBUSED) $(STDCC_LIBS)

register:
	oadutil reg -poa /plotter_agent_poa -cpp $(MAKEDIR)\Plotter.exe -e
		COMPUTERNAME=$(COMPUTERNAME)
	Plotter register -ORBInitRef NameService= iioploc://$(COMPUTERNAME):
		$(NAMESERV_PORT)/NameService

unregister:
	oadutil unreg -poa /plotter_agent_poa

!endif
```

A.2 Makefile for the client

```
@echo off
rem Makefile

if "%1"=="" goto idl
if "%1"=="all" goto idl
if "%1"=="idl" goto idl
if "%1"=="java" goto java
if "%1"=="clean" goto clean
if "%1"=="run" goto run
goto end

:idl
@echo on
call idl2java -no_comments ..\idl\Plotter.idl
@echo off
if "%1"=="idl" goto end

:java
@echo on
vbjc -d . -g *.java

@echo off
goto end

:clean
@echo on
del *.class
del Plotter\*.java
del Plotter\*.class
del MyClient\*.class
rmdir Plotter
@echo off
goto end

:run
@echo on
```

```
vbj -DSVCnameroot=MY_NS MyClient.PlotterClient
@echo off
:end
```

A.3 The CORBA IDL of the plotter system

```
#ifndef _plotter_idl_
#define _plotter_idl_

module Plotter
{
	struct ResStruct
	{
		boolean valid;
		double value;
	};

	typedef sequence<ResStruct> ResStructSeq;
	typedef sequence<ResStructSeq> Results;

	exception PlotterException
	{
		string message;
		string exceptionName;
	};

	interface PlotterServer
	{
		// parse the expression
		void checkExpr(in string expr)
			raises(PlotterException);

		// set variable in expression
		void setVariable(in string vname, in double vmin, in double vmax)
			raises(PlotterException);

		// set parameter in expression
		void setParameter(in string pname, in double pfirst,
		                  in double plast, in double pstep)
			raises(PlotterException);

		// set constant in expression
		void setConstant(in string cname, in double cval)
			raises(PlotterException);

		// evaluate the expression
		void evaluate(in double step, out Results res)
			raises(PlotterException);

		// clear
		void clear();
	};
};

#endif
```