Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

An Extension of MS Windows to Support the Development of Distributed Object-Oriented Applications

Heng-Ching Lin and Chih-Ping Chu

Institute of Information Engineering
National Cheng Kung University
Tainan, Taiwan 701, R.O.C.
chucp@server2.iie.ncku.edu.tw

Abstract

MS Windows is the most widely used operating environment with graphics user interface for personal computers. However, in MS Windows there exists no appropriate, high-level support for developing distributed windows applications. In this paper we propose an expansion to the MS Windows environment to support the development of distributed windows applications across heterogeneous platforms. The extended version of MS Windows (we call it EMS Windows) adopts Client/Server architecture with the features of network transparency and independent execution-platform applications. In addition, a class library supporting the development of distributed, object-oriented windows applications is also built into EMS Windows. Based on EMS Windows, we found that developing a distributed windows application is very fast and efficient.

1. Introduction

Since the computer network has become popular, application systems have advanced from operating in a stand-alone computer to working in network-connected computer clusters. Applications which used to be executed on mainframes have been downsized to redevelop on distributed computing platforms. People at various sites can operate and share resources with each other, eliminating time and space restrictions. On the other hand, ever since MS Windows operating environments were released on personal computers, more and more people have gradually come to believe that the windows interface is a more user-friendly interface and may replace the command mode interface of the DOS environment. Many applications have thus been developed with windows interfaces in MS Windows environment. In addition, the advantages of object-oriented programming - information hiding, more structure, easy to understand, reuse, and maintain, etc., and the advancement of object-oriented languages have enabled many applications to be constructed on the basis of objects. Due to these facts distributed, object-oriented windows applications will be the mainstream of application software in the near future. However, currently it is still rather difficult to develop distributed applications in an MS Windows environment, especially for a novice in networking programming. Therefore, an extension of MS Windows (we call it EMS Windows) to support the development of distributed, object-oriented window applications with the features of network transparency and independent execution-platform applications is considered necessary.

In Section 2 some background information is introduced. Section 3 describes how to design and implement EMS Windows. Section 4 presents applications implemented in EMS Windows, showing also the relationship between EMS Windows and X Window. Finally, Section 5 draws conclusions and outlines future research directions.

2. Background

EMS Windows, an extension of MS Windows, adopts the client/server architecture supporting distributed applications executed across different windows environments such as X Window, the Apple's Macintosh, and other windows systems. We will first briefly provide some background information about the client/server architecture, the MS Windows System and the X Window System in this section.

2.1 Client/Server architecture

A distributed application contains software programs and data resources scattered across independent computers connected through a communication network. Coordination models establishes logical roles - along with associated behaviors for applications that assume roles. One coordination model widely used in a distributed

system is the client/server mode. A program, the *client* requests an operation or service that some other application, the *server*, provides. Upon receiving a client request, the server performs the requested service and returns the results. A client interface specifies the individual services or operations supported by the server. Clients can only request services that conform to the client interface of the given server.

To design a model for a development environment supporting applications with client/server coordination several key issues must be considered carefully [Adler, 1]:

- direct or indirect services

The service mode provided by the server can be classified as direct or indirect. A direct service implies the client communicates with the server immediately, while an indirect service means the client merely communicates with an agent and the agent communicates with the actual server. Both modes have their application considerations including execution-platforms, efficiency, extendability, coding complexity, etc.

- explicit or implicit requests

Explicit requests refer to when a client needs to state clearly the service and the server it is requesting. An implicit request means that a client needs to state simply the service it is requesting but not the server. The server providing that specific service will be found and bound by the supporting environment.

- blocking or non-blocking

A client after submitting a service request may be blocked in order to wait for the results, or it may continue to do its own job without being blocked.

In EMS Windows environment the supported client/server mode adopts direct services, explicit requests, and non-blocking scheme.

2.2 MS Windows system

Programming an application in MS Windows usually includes the following steps:

1. Register a window class and assign message-handling procedures.
2. Create a window and expose the window.
3. Get messages and dispatch them to message-handling procedures for processing.

A window is always created on the basis of a window class. The window class identifies the window procedure that processes messages for that window. The window class defines the window procedure and some other characteristics of the windows that are created based on that class. When a window is created, additional characteristics that are unique to that window may also be defined. Like X Window, MS Windows is also event-driven. In MS Windows events are called *messages*. There are two kinds of message: *queued messages* and *nonqueued messages*. The queued messages are those that are placed in an application's message queue by Windows and retrieved and dispatched in the application's message loop. The nonqueued messages are sent to the window directly when Windows calls the window procedure. The queued messages are primarily those that result from user input, while the nonqueued messages result from the Windows system (e.g., events for creating windows). Usually the window procedure processes some special kinds of message and the others are sent to a default window procedure for processing.

2.3 X Window system

The X Window System, having been developed on top of operating systems, is a development environment with the feature of network transparency, hardware independence and text and graphics computing window interfaces, supporting the development of centralized and distributed applications [Jones, 3].

As shown in Figure 1, the basic architecture of the X Window System is a client/server model. There is a server managing all input and output devices in each host machine. The server notifies clients of the occurrence of events and handles requests from clients. Clients and servers do not have to be in the same machine. An application in X Window System communicates with a server by a display connection. A display connection is a logical network connection between an application and a server. X Windows System provides programmers network transparency by means of display connection.

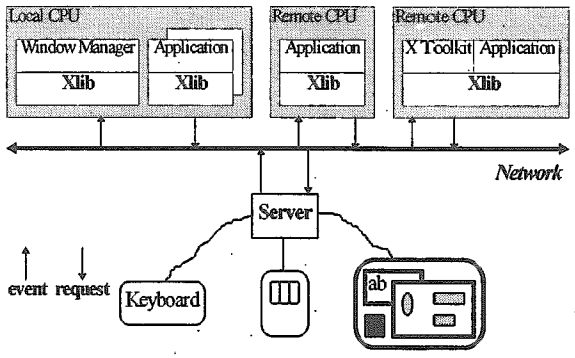

Fig. 1 Network Transparency of X Windows

Event-driven is another property of X Window applications. The X protocol specifies what makes up each packet of information transferred between the server and the application in both directions. There are four types of

packets transferred via the protocol: *requests*, *replies*, *events* and *errors*. A protocol *request* is generated by an application and sent to the server and the others are sent from the server to an application. The event is the most important packet because all X applications are event-driven. When the program starts, it first connects with a server. Then, after creating windows, the program repeatedly waits for an event. When a user operates I/O equipment, an event will be generated and sent to the application by the server. Then the application will undertake appropriate processing after receiving the event.

3. Design and implementation of the EMS Windows

The goals of EMS Windows are to support the development of distributed applications in procedural and object-oriented languages and to provide an environment with network transparency and execution-platform independence.

3.1 System architecture

As shown in Figure 2, EMS Windows is developed with a client/server architecture. It provides programmers an interface between an application and the network. An application (client) can communicate with the server either by passing messages to objects or by calling functions. Objects consist of member functions. The server either calls an appropriate MS Windows application program interface (API) or X Library (X Lib) to be processed, depending on the execution platform. The MS Windows API will talk with the MS Windows System and the X Lib will talk with the X server. The results will be returned to the application. The network protocol is TCP/IP.

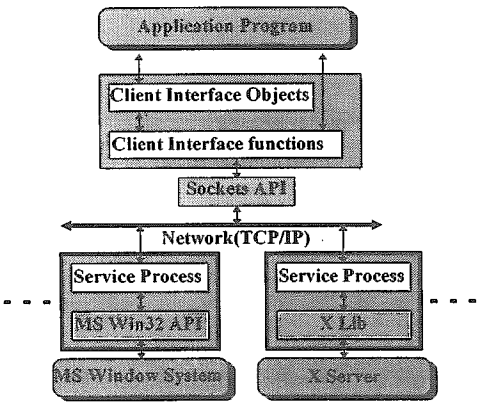

Fig. 2 System Architecture

We first need to execute a daemon process on each computer that wants to be a server. Figure 3 shows the operation flow of the daemon process in EMS Windows. The algorithm for the daemon process is:

1. Initialize windows' sockets.
2. Wait for a connection request from a client.
3. Establish connection with the client.
4. Produce a server for the client, and then go back to step 2.

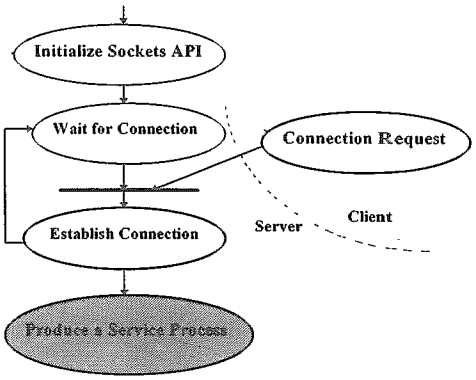

Fig. 3 Execution Flow of the Daemon Process

After producing a service process, the daemon process continues waiting for the next connection request from another client. Meanwhile, the serving process begins to serve the client as shown in Figure 4. The operation flow of the serving process is:

1. Wait for a service request.
2. Provide the client with a suitable service according to the service identification, then return the results to the client and go back to state 1.

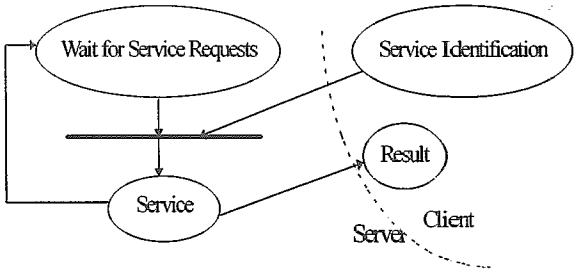

Fig. 4 The operation flow of the serving process

3.2 Detailed design

By referring to [3,14] the coding paradigm for writing an application in EMS Windows is established as follows:

1. Establish a display connection.
2. Register a window class.
3. Create a window and expose the window.
4. Get messages and dispatch them to the message

processing procedure to handle them.

In this section, based on [5-11], we will describe how to design the interface functions and the interface objects to meet the expected goals.

Establish a display connection

As we have described in Section 3.1, first, we need to start up a daemon in the server node. In the EMS Windows system, the daemon process is SERROOT (i.e., Server Root). SERROOT does not provide any service to clients but only produces a serving process for the client requesting connection. The serving process will handle all requests from the connected client. SERROOT will tell the serving process its client's display number. As shown in Figure 5, each client has an individual serving process in each connected computer, and the serving process will handle its requests. Applications use *OpenDisplay()* to connect with a server in either execution model. When the connection is closed, the serving process will be killed.

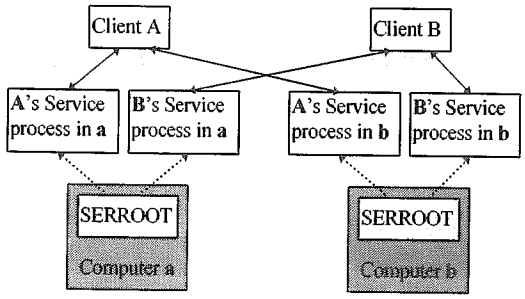

Fig. 5 Clients VS. Servers

Register window classes

In conventional windows programming, before creating a window, the window class has to be registered in advance. In EMS Windows, it is necessary to register a window class before a window is created. By means of the display connection a client notifies a server it wants to register a specific window class.

The function - *MInitialWindow()* is defined to register a window class. This function needs eight parameters in addition to the server's identification. The eight parameters are specified as follows:

- *Style* Specifies the configuration of the window.
- *cbClsExtra* Defines a certain number of extra bytes that will be appended at the end of the window class structure and reserved for special purposes.
- *cbWndExtra* The number of extra bytes added to the end of each newly created window structure.
- *hIcon* Defines the icon assigned to the window class.
- *hCursor* The handle for a cursor type.
- *hMenu* A long pointer of a character string ending in "\0". This character string assigns a menu resource for the window class.
- *lpszClassName* A long pointer to a character string ending in "\0", that contains the name of the window class.
- *hbrBackground* A handle for a background brush.

The conventional coding method for windows applications needs to fill in the variable fields of WNDCLASS and then call the function - *RegisterClass()* to register the window class. WNDCLASS, in addition to the eight field shown above, has two extra fields. In the EMS system, these two fields are filled in by the server. The names and the reasons are explained below.

- ***lpfnWndProc*** A pointer of the window function assigned to the window class

Traditionally, each window class has a window (control) function. When a window belonging to some class receives an event, the windows system will call its window function. In EMS Windows the server will get events to happen in the window on the server node and notify clients to handle them. Thus, a function that can record events belonging to user-defined classes of windows is needed. The function is assigned to all user-defined classes to the window function by the server. Then servers can record events and notify clients.

- ***hInstance*** The handle for the instance of the application for which the window class is being registered

From a user's viewpoint this function is an application which registers the window class. But an application just tells the server information about the class, and the server actually registers it. So, it is correct that the handle for the instance of the application is assigned by the server.

Expose windows

In the creation of windows, the parameters needed by the function - *MCreateWindow()* are similar to the ones used by EMS Windows. There are two different points that are the necessary for the server's identification and the lack of a handle for the instance of the application. We will know which server needs to create a window by means of the server identification. The reason for the lack of a handle is the same as in registering window classes. After the server receives all information, it will create the specified window for the client. In addition, the system also supports a system-defined windows class - *Edit* - in EMS Windows. Because the class has a defined window function, the service process will change the pointer from an edit window function to the window function provided by EMS Windows, which will record events originally sent to the edit window function. The application will then be able to know the events in the edit window, but it will need to use *CallEditFunction()* to call the original edit

window function, if you want to keep the original operation of the edit window.

If the window's type is set as WS_VISIBLE in window creation, the window will be exposed on the screen after it has been successfully created. If we do not set the window's type, we will need to call the function - *MShowWindow()* to expose the window on the screen after it has been created. Parameters needed by this function are similar to the corresponding function in EMS Windows, but in creating a window it needs a server's identification to notify the correct server.

Event Process

In MS Windows, there are two kinds of message. One is a *queued message*, and the other is a *nonqueued message*. In EMS Windows, the classification and handling methods of events are different with respect to the viewpoints of the users and the system. We will describe them here.

Figure 6 shows the event queue from the user's viewpoint. From user's (programmer's) viewpoint, all events are in the event queue. When an event takes place, it will be entered in the queue to wait for processing. Clients use the function - *MGetEvent(Display Dpy, MEvent *event)* to read an event from the queue of the server whose identification is Dpy. Because this function will block the program before it gets an event, sometimes we can check to see if there are waiting events before we read an event. We can use *MLookupEvent()* to check it. Clients also can use *MSendEvent()* to duplicate an event to a specified server; the event will be inserted between the duplicated event and other events. They also can use *AppendEvent()* to produce an event at a specified server that appears as a general event.

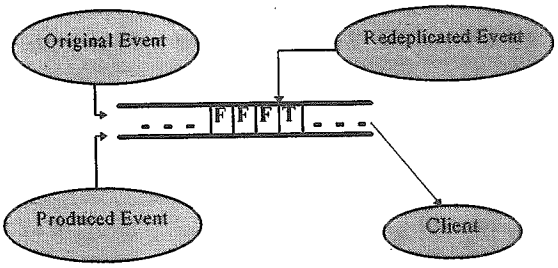

Fig. 6 The User's Viewpoint to the Event Queue

The fields of the event structure (MEvent) is as follows:

- *dpy* (server's identification) Specifies the issuance of the event.
- *hwnd* Specifies the handle for the window on which the event takes place.
- *type* Describes the event type ex. WM_RBUTTONDOWN.
- *wParam* Short extra information about a special event.
- *lParam* Long extra information about a special event.
- *Send_Type* Specifies if the event is a reduplicated one. If it is reduplicated, this field will be set to TRUE; otherwise, it is FALSE.

Figure 7 shows the appearance of the queue in terms of the system. Two linked lists make up the queue looked at by clients. The responsibility of list B is to record reduplicated events by clients. List A records the other events. When a client asks if an event exists or it reads an event, events in list B will be handled first, and the events in list A will be handled until B is null. By the way, we can make a queue that can distinguish if an event is reduplicated. A reduplicated event is handled first and sets *Send_Type* to TRUE. This property allows programmers to be able to write a synchronous application easily.

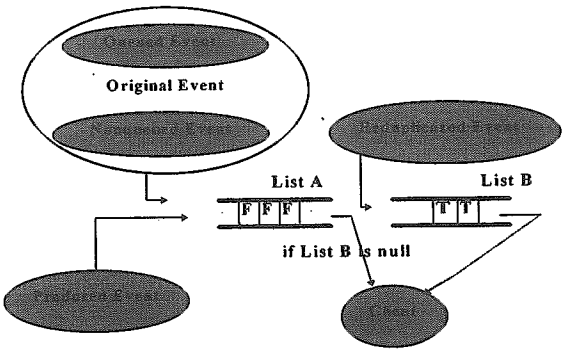

Fig. 7 The Event Queue from the System's Viewpoint

Figure 8 shows the implemented client interface in EMS Windows. Next, we will introduce the objects in EMS Windows.

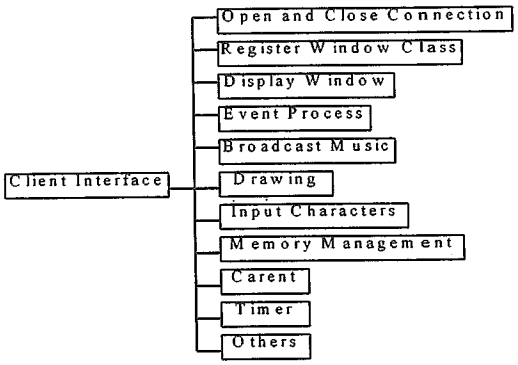

Fig. 8 Client Interface

3.3 Design of objects

To enable programmers to be able to develop their applications in an object-oriented programming mode, in EMS Windows the interface objects are also built. As shown in Figure 9, the objects currently provided can be classified as *connection objects*, *window objects*, *image objects*, *CD object* and *wave object*.

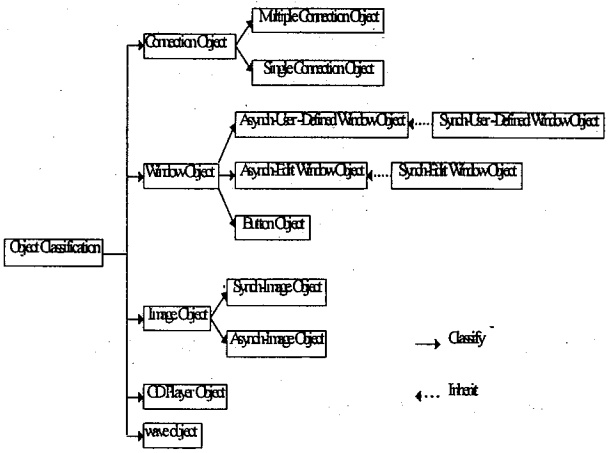

Fig. 9 Classification of objects

Connection objects are divided into multiple and single connection objects. There are three operating methods on the object. They are:

1. Open connection.
2. Read events.
3. Close connection.

When programmers start an object, they need to pass the messages of the computers' addresses to it. Fig. 10 shows a connection object and the messages passed-in and passed-out.

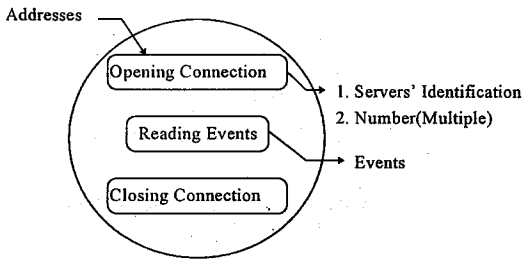

Fig. 10 A Connection Object and the Messages Passed

Window objects are divided into user-defined window objects, edit window objects and button objects. User-defined window objects and edit window objects both can be synchronized and asynchronized. Synchronized means that the windows will have the same events and asynchronized means that they are independent. Thus, the synchronous object inherits from the asynchronous one and modifies the event processing. Their operating methods for all window objects are the same. They are:

1. Initialize window.
2. Display window.
3. Handle events.
4. Close window.

Fig. 11 shows a window object and the messages passed-in and passed-out.

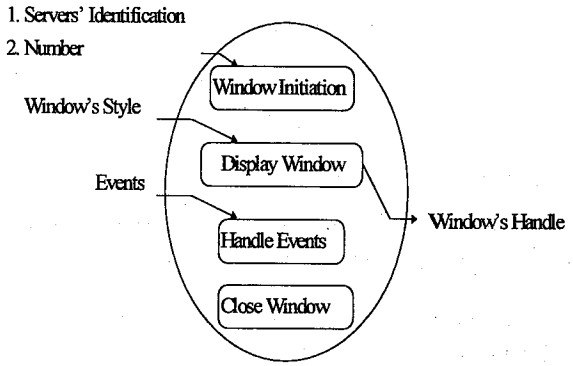

Fig. 11 A Window Object and the Messages Passed

However, the operating logic for each of the methods has a few differences. That is, when starting window objects, programmers need to pass two messages - servers' identification and the number of hosts to be connected - to the objects. For user-defined window objects the methods 'window initialization' need to handle the registration of window class to each of the connected hosts. For the other two window objects this job is not required as they are defined window classes in MS Windows system. In event processing, the operations for synchronous user-defined and edit window objects are different from those of other window objects. That is, an event will be duplicated on all other connected hosts before it is processed by a method. This mechanism will make the synchronization behaviors of applications be achieved implicitly.

Image objects are divided into synchronous and asynchronous. A synchronous object will show the same image to different servers, and an asynchronous object will show different images to different servers. There are four methods for operating an image object. They are:

1. Initialize object.
2. Show an image.
3. Repaint an image.
4. Clear image data.

Fig. 12 shows an image object and the messages passed into and passed out of the object.

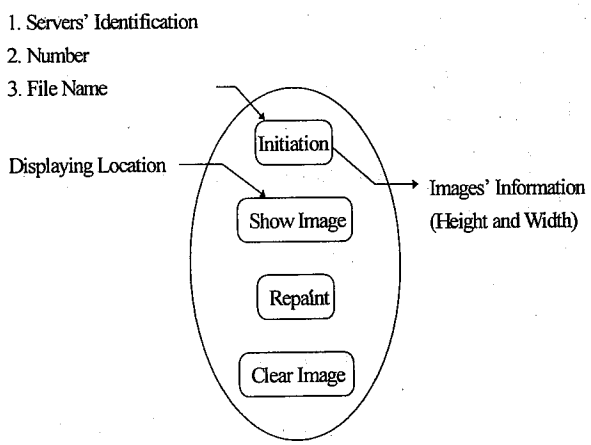

Fig. 12 An Image Object and the Messages Passed-in and Passed-out

A ***CD object*** has several operating methods:

1. Open CD.
2. Operate (play, pause, and chapter-selection, etc.) CD.
3. Close CD.

When starting a CD object, programmers pass a server's identification and number of computers to be connected to it. When ending an object, CD will be closed. Fig. 13 shows a CD object and the messages passed-in.

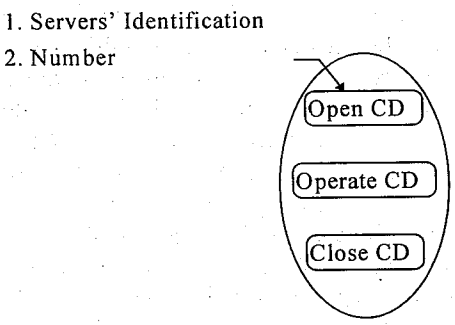

Fig. 13 A CD Object and the Messages Passed-in

A ***Wave object*** is shown in Figure 14. The several methods for operating a wave object are as follows:

1. Open audiowave input/output device.
2. Record and broadcast.

Fig. 14 also shows the required messages passed into the Wave object for different operations provided.

4. Application based on EMS Windows

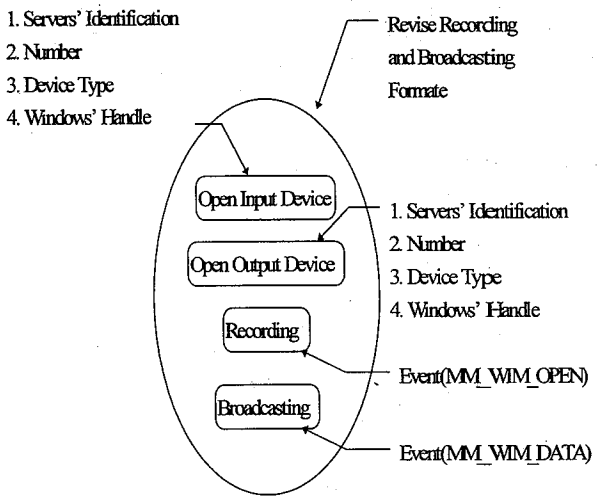

Fig. 14 A Wave Object and its Passed Message

4.1 Homogeneous platforms

An application was implemented in Borland C++ under MS Windows NT System [12,13,15]. It allows users to co-edit a text, co-watch an identical image and co-listen to music from a CD player. Figure 15 shows the initial appearance of the application. We can see the same windows as Fig. 15 in each connected computer. There are three buttons - *white board, CD player* and *image* - in the initial window. After a user clicks white board, CD

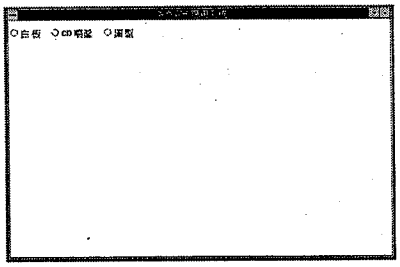

Fig. 15 The Initial Window

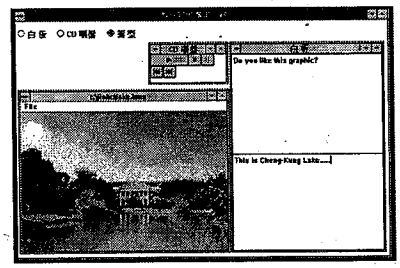

Fig. 16 Program in Execution

player and image buttons, each connected computer will see the same windows as Figure 16. The source code for this application can be found in [Lin, 4].

4.2. Heterogeneous platforms

As EMS Windows provides the same client interface between MS windows and X windows to programmers, programmers do not need to know the remote window environments before they program applications to be executed across different platforms. An application showing the input characters in both MS windows and X window simultaneously was also developed in EMS environment.

According to our experiences, for a programmer familiar with EMS Windows, it will take about one hour and write 50 lines of source code to complete a distributed, synchronous multidisplay white board system. However, to code a similar system in general MS Windows environment, a programmer will spend about one day to one week time and write about 200 lines of source code, depending on his/her network and window programming abilities.

5. Conclusions and future work

MS Windows is a widely adopted operating environment with graphics user interface for personal computers. However, the MS Windows environment does not support the development of distributed applications. In other words, programmers need to deal with network connections, message passing and interprocess coordination in the distributed applications themselves. Therefore, we have expanded the MS Windows environment to adopt a client/server architecture supporting the development of distributed applications. Such an extension includes developing a function library, a class library, and a client/server architecture including a server process. The function library includes a set of interface functions for server node connections, a window class registration, and image, audio and CD processing. Similar functions also exist in the member functions of objects in the class library. The client/server architecture adopts direct services, explicit request, and non-blocking scheme. The server process is actually spawned from a daemon process that is resident on a server node. The server process is in charge of event/request processing. Actually what we are proposing is a model for extending MS Windows to allow it to support distributed applications development. According to our experiences the timing and source code improvement for applications developed in EMS Windows environment is very significant, ranged from 10 to 50 times in timing, depending on the proficiency of network and window programming of a programmer. In the future we plan to implement as many functions and objects as possible to complete a powerful system, integrate the existing PC-based database systems and offer related functions and objects into EMS Windows environment, and provide the supports in fault tolerancy, security, and efficiency. Meanwhile, the serving process in X Window and other systems will also be put into practice.

References

[1] Richard M. Adler, Distributed Coordination Models for Client/Server Computing, IEEE Computer, 1995.
[2] Helen Custer, Inside Windows NT, Microsoft Press, 1993.
[3] Oliver Jones, Introduction to the X Window System, Prentice-Hall International Inc., 1989.
[4] Heng-Ching Lin, "EMS Windows - An Extended MS Windows for Supporting the Development of Distributed Objected-Oriented Window Applications," Master Thesis (in Chinese), National Cheng Kung University, June 1996.
[5] Microsoft Win32 Programmer's Reference V.1, Microsoft Press, 1993.
[6] Microsoft Win32 Programmer's Reference V.2, Microsoft Press, 1993.
[7] Microsoft Win32 Programmer's Reference V.3, Microsoft Press, 1993.
[8] Microsoft Win32 Programmer's Reference V.4, Microsoft Press, 1993.
[9] Microsoft Win32 Programmer's Reference V.5, Microsoft Press, 1993.
[10] Microsoft Win32 Application Programming Interface V.1, Microsoft Press, 1992.
[11] Microsoft Win32 Application Programming Interface V.2 Microsoft Press, 1992.
[12] Microsoft Windows Multimedia Programmer's Reference, Microsoft Press, 1991.
[13] Microsoft Windows Multimedia Programmer's Workbook, Microsoft Press, 1991.
[14] Charles Petzold, Programming Windows 3.1, Microsoft Press,1992.
[15] Peter D. Varhol, Windows NT - Microsoft's New Operating System Strategy, Computer Technology Research Corp., 1993.
[16] Peter Wilken and Dirk Honekamp, Windows System Programming, Abacus, 1991.