Process Communication through Proxy Pipe

Shuping Chen
Department of Computer Science
September 17, 1999

Major Professor: Dr. Timothy Budd

Report completed in partial fulfillment of the requirements for
the Master of Science at Oregon State University

Process Communication through Proxy Pipe

Shuping Chen
Department of Computer Science
September 17, 1999

Major Professor: Dr. Timothy Budd

Report completed in partial fulfillment of the requirements for
the Master of Science at Oregon State University

Table of Contents

1. Introduction

2. Requirement Analysis

3. Class Design

3.1 From Use Case to CRC Cards
3.2 From CRC Cards to Sequence Diagram
3.3 From Sequence Diagram to State Machine Diagram
3.4 Structure of the Real Problem
3.5 Class ControlProxy and ControlProxyServer
3.6 Relationship of the Involved Classes

4. Implementation Details

5. Conclusions

6. Future Directions

References

Abstract

Proxy Pipe is designed to be a bus in the transport layer of the communication between the two different processes. Its major function is to transfer bytes. When the client process tries to send a command to the server process, it will talk to a proxy as if it were talking to a remote object. The proxy then translates the command into bytes and then asks the server Proxy Pipe to write the bytes to the client Proxy Pipe. The client Proxy Pipe the reads the bytes and translates them into the command. Proxy Pipe defines a user-friendly interface and it encapsulates all the related NT system calls. Proxy Pipe will be implemented in Microsoft Visual C++. We will present the requirement analysis and design part by using UML (Unified Modeling Language). Design patterns like proxy, state and observer will also be applied.

Special Thanks

I would like to thank Dr. Budd and my other committee members, Dr. Burton (my major advisor in math) and Dr. Quinn, for their patience and feedback. I would also like to give special thanks to Steve McBride who is my manager when I was doing this project in Electroglas, one of the biggest company in inspection products.

1. Introduction

In the computer-engineering field, sometimes it is necessary to separate one application program into at least two different processes. As an example, in the Wafer Bump Inspection Machine software system, there are two major processes:

- Control.exe - To control the movement of a Machine. The machine has devices such as Robot, Pre-Aligner, OCR (Optical Code Reader) and Camera, etc.
- Image.exe - To check if the bump on the wafer is in good condition by its image.

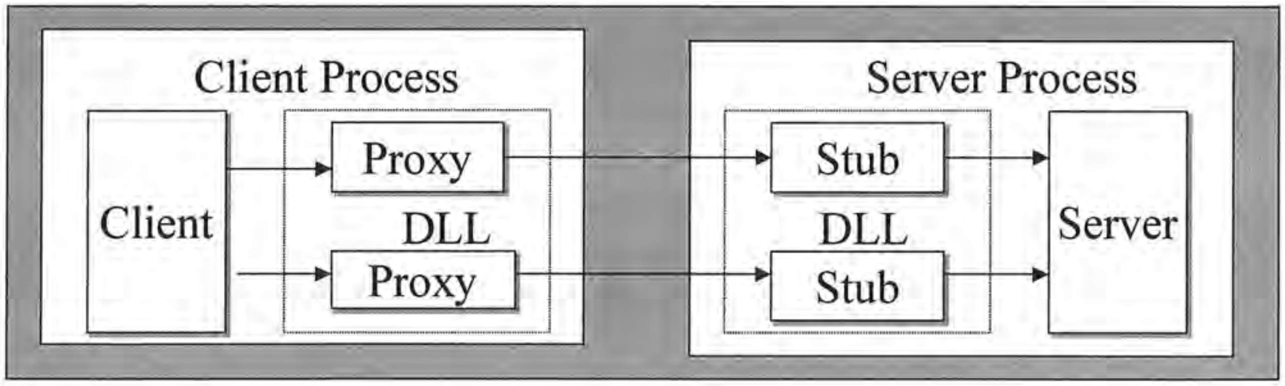
It is very possible to have two computers in two different rooms. One computer is installed with Control.exe and the other has Image.exe. Two different teams can work on the two processes separately. These make it necessary for the two processes to be separated. However, these two processes need to talk with each other. For example, Image.exe needs to tell the Machine in Control.exe to adjust the position of a Camera in order to focus best. One possible solution would be that the process Image.exe creates a pointer/object of class Machine that is in a different process from that in Control.exe. Then send message to this object to move focus. However due to the memory protection, two different processes take their own private CPU memory space. Pointer in one memory space would not be able to point to a different memory space. This would make sure that when one process is running, it would not affect the other. For example, we could run a game and Microsoft Word at the same time in the same computer without affecting each other. However this gives hard time for the two different processes to communicate with each other.

A Proxy Pipe is used to work as a bus in the transport layer of the communication between the two different processes. The Proxy Pipe will play roles of both server and client. Windows NT has some built-in system calls that do the process communication. Those system calls are hard to find for the users that are not familiar with NT system functions. And some function calls like Connect() will not return instantaneously. User will have to create a thread for each blocking call. With multiple threads, a race condition would easily happen. A race condition exists when the success of your operation depends on which of two threads finishes its task first, and the two threads operate independently of one another. Based on the Named Pipe of Windows NT, Proxy Pipe overcomes the above difficulties in the following ways. First, it provides the Proxy pattern so that the calling application thinks it is talking directly to the object in the different memory space, though in realty it is talking to a proxy. A proxy is an object which represents the intended object, but in a different memory space. Second, Proxy Pipe is Object Oriented. It defines a high level interface so that the user doesn't have to know the names of NT system calls. Finally, Proxy Pipe controls its behavior though a state machine so that the least threads will be needed.

The purpose of this project is to allow the Image process to send commands to the Control process through a proxy pattern. In the Image process we will create a proxy object to which the Image process can send commands. This proxy object will then send message across the different processes/memory spaces by using the Proxy Pipe in the transport layer. According to the book *Design Patterns*, Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Proxy would have all the necessary functions that the object has. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space.
2. A **virtual proxy** creates expensive objects on demand.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - Counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers).
   - Loading a persistent object into memory when it's first referenced
   - Checking that the real object is locked before it's accessed to ensure that no other object can change it.

The proxy that we use in our project would be a remote proxy. Another application of remote proxy is the marshaling in COM (the Component Object Model). Marshaling is the process by which parameters are sent across apartment, process, or machine boundaries. In COM, interface-specific proxies carry out marshaling. A proxy runs in the address space of the client and looks exactly like the interface it represents. The proxy accepts function calls from the client, then packages up the parameters to transmit them to a corresponding stub in the receiver's address space. The stub receives the parameters from the proxy, unmarshals them, and makes the necessary calls against the server's interface in its own process. It then takes the result of the calls and passes them back to the proxy, so that it, in turn, can pass them on to the client. The proxy and the stub allow the client and server code to be written without regard to their relative locations. In the following diagram, the server and client are each in separate processes and require the intervention of in-process proxies and stubs to carry out the cross-process calls transparently.



The rest of the paper is organized to match the phases of object-oriented development in this computer software project. It will focus on requirement analysis, design, implementation and testing. The paper will start with requirement analysis through the use cases technique. Next, class design is presented from a sequence diagram to a state machine. In the design part we will unwrap the problem layer by layer so that the relationship of the all the classes involved in this project will be clear. This can also help test each component of the project solidly. Then we will implement a synchronization mechanism called multiple lock. Finally, the paper will document the project by attaching all the class specifications.

2. Requirement Analysis

The phases of the software development are requirement analysis, design, implementation and testing. First let's figure out the detailed requirements. A key tool in this phase is the development of use cases. Use cases describe how the system will be used. A use case is a description of an interaction between the system we are building and a person or another system. We call the object taking part in the interaction the actor.

Let's develop a few use cases for the system Proxy Pipe:

Actor: The customers of Proxy Pipe - both server and client

For each of the actors, we should consider the following questions:

What tasks does the customer want the ProxyPipe to perform?
The server would want the ProxyPipe to declare itself as a server Proxy Pipe. The Client would want ProxyPipe to declare itself as a client Proxy Pipe.
Both server and client want to read and write bytes through the Proxy Pipe.
Proxy Pipe should be able to shutdown itself.
What information must the actor provide to the system?
This actor must provide pipe names for both client end and server end. They have to be same for the connection to complete. The actor must also provide the character buffer to store the data that has been read from the ProxyPipe and the character array to be written to the ProxyPipe
Are there events the actor must tell the system about?
No.
Does the actor need to be informed when something happens?
The actor needs to be informed when the ProxyPipe state machine changes its state.
Does the actor help initialize or shutdown the system?
No

Resulting use cases:

1.The sever/client provides the pipe name for the ProxyPipe to initialize itself.

2.The server writes a string to the ProxyPipe

3.The client reads the string from the ProxyPipe

4.The server writes a character array with '\0' in the middle to the ProxyPipe

5.The client reads the character array with '\0' in the middle the ProxyPipe

6.The server writes a number to the ProxyPipe

7.The client reads the number from the ProxyPipe

8.The client writes a string to the ProxyPipe

9.The server reads the string from the ProxyPipe

10.The client writes a character array with '\0' in the middle to the ProxyPipe

11. The server reads the character array with '\0' in the middle from the ProxyPipe

12.Server/client shutdown

If we take a second look at our list of use cases, we can aggregate the similar ones. For example, use cases 2, 4 and 6 can be consolidated into "The server writes bytes to the ProxyPipe". Use case 3, 5 and 7 can be consolidated into "The client reads bytes from the ProxyPipe". Use case 8, 10 and 12 can be consolidated into "The client writes bytes to the ProxyPipe". Use case 9, 11 and 13 can be consolidated into "The server reads the bytes from the ProxyPipe".

The use cases can be simplified into:

1.The sever/client provides the pipe name for the ProxyPipe to initialize itself.

2.The server writes bytes to the ProxyPipe

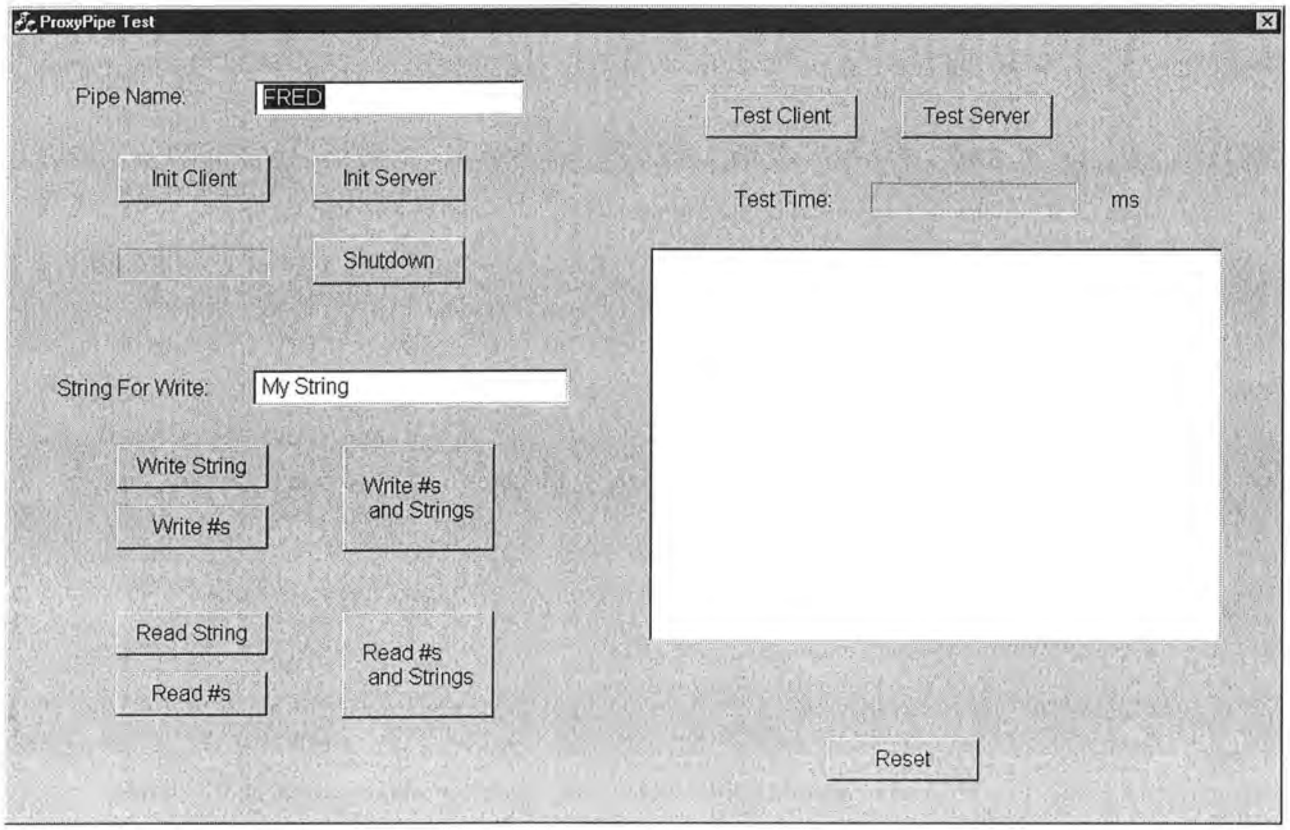
3.The client reads the bytes from the ProxyPipe

4.The client writes bytes to the ProxyPipe

5.The server reads the bytes from the ProxyPipe

6.Server/client shutdown

Now we can get a draft user interface. This will be a set of screen shots. We still keep the different cases for bytes in order to do a better testing.



Many use cases provide a flow of events called a scenario. For example, we have the use case "The server sends bytes to the client". The scenario might read

The server tells the ProxyPipe to send the bytes. Then the ProxyPipe calls the right NT system call to do the work.

As part of exploring the use case, we will represent some of the interactions between the actors and the system in an interaction diagram in the next part.

3. Class Design

For each use case, we will identify those objects that are needed, and we will flesh out the functionality of the objects, based on the behavior dictated by the use case. We will build at least one sequence diagram for every identified use case.

When we go through the process of requirement analysis, we focus on creating the use cases that identify how the product will be used. Along the way we identify the interactions and associations of objects in the problem domain. In design, we turn attention to the solution. We will have to ask ourselves the following questions:

What are the objects in the design?
What are their attributes and capabilities?

How will they interact?

We will try to decide what the properties and methods of our objects are and what their parameters and return values of these methods will be and so forth. We will try to sketch out the public interface for all of our classes. In short, we will have created the object model that serves as the blueprint for our implementation.

3.1 From Use Case to CRC Cards

One way to get started with determining the responsibility and fundamental associations of a class is by using a technique called CRC cards. CRC cards are simply index cards on which we write the name of each class. CRC stands for Class-Responsibility-Collaboration, and these cards will track the responsibility of each class. We will also note the collaborators - the other classes to which each class delegates responsibility.

Across the top we write the class name and possibly its derived classes and its super classes. On the rest of the card we use the left two thirds for Responsibilities and the right one third for collaborations. On the back of the card, we write a short definition of the class.

Now it is the time to flesh out the use case scenario. To make the connection between the server and the client, we can imagine that both client and server ProxyPipe object will talk to the NT system, then the two ends of NT system makes the connection. Two NT systems are necessary especially when the client and server stay in two different machines. So server, client, 2 ProxyPipe objects and 2 NT Systems will be involved. Here is a sequence of things that could happen: (Here ProxyPipe1 & ProxyPipe2 are the two ends of the named pipe.)

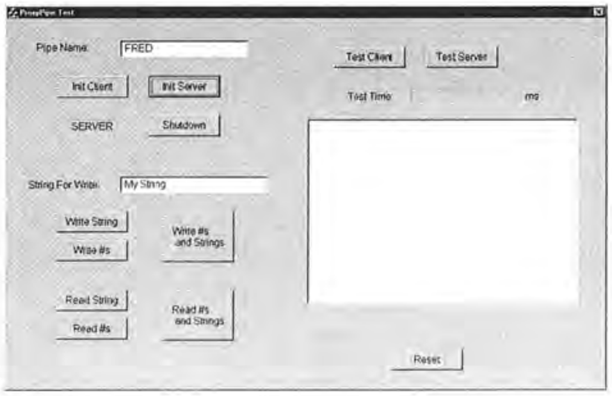
Server tells ProxyPipe1 to initialize itself to represent a server
ProxyPipe1 creates a handle to the server end of a named pipe instance
ProxyPipe1 tries to connect with the client end of the named pipe instance, if no client available it will wait
Client tells ProxyPipe2 to initialize itself to represent a client
ProxyPipe2 saves the NamedPipe name
Client tells ProxyPipe2 to write/send a character array to the Named Pipe
ProxyPipe2 opens the client side of a named pipe and tries to connect with the server end
After the connection completes, ProxyPipe2 can write the character array to the named pipe
Server tells ProxyPipe1 to read/receive the character array from write end of the named pipe
Client tells ProxyPipe2 to shutdown. That is to disconnect the client end from the named pipe
Server tells ProxyPipe1 to try writing; it will get an error message saying that the client end is not available

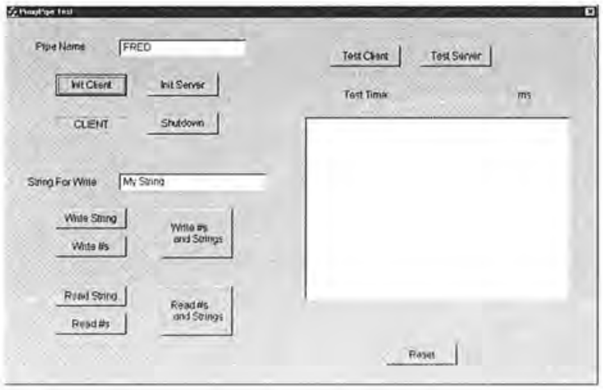
. . .

By examining the use cases in last part, we can find there are three major classes that will be involved: Client, Sever, and ProxyPipe. ProxyPipe communicates with NT system by the NT calls. We would not really see the NT system in our application. There is no need to have the NT system as a class. Since we are focused on the process communication part, we are only interested in those functions in client and server that have to deal with crossing the

ProxyPipe. Client could send any commands to the server as long as the command belongs to the server's interface, and vice versa. Sending any command is just writing bytes to the ProxyPipe. So we will not worry about the details of different commands for either server or client first. We will use Write/Read to start with. In this case the interface for the client and server is really simple. We will concentrate on class ProxyPipe.

From the detailed use case above, we noticed that server and client have the same interface. They both initialize themselves first and then do read or write to the other. Finally they can both shutdown themselves. So for right now, to be simple enough, we will make only one class for both server and client. This is like a single person is playing two different roles in a movie. This class is a prototype used to test the functionality of the class ProxyPipe. It will also handle the Graphical User Interface.





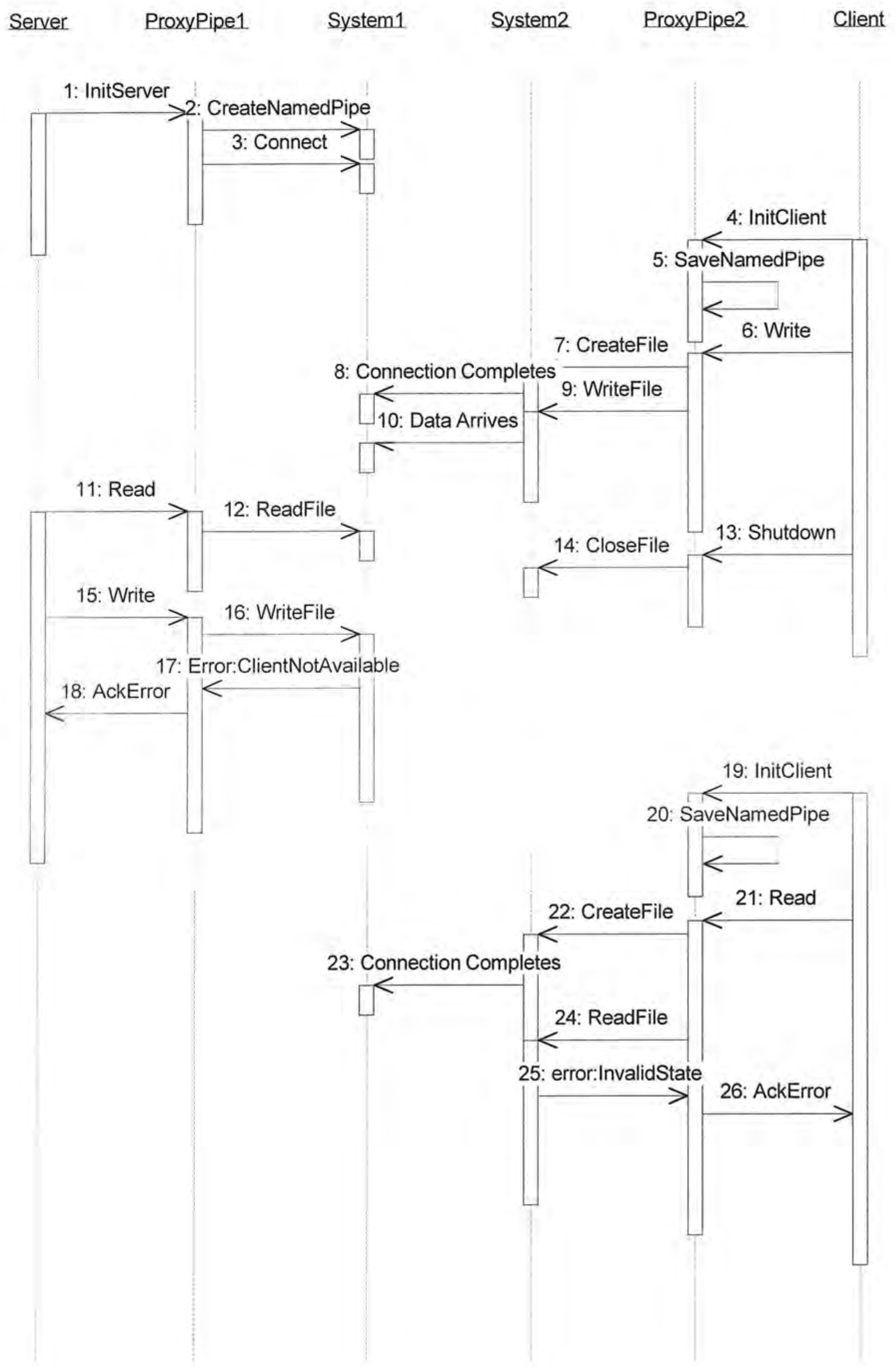
We will share one CRC card for Client/Server class and ProxyPipe class:

Class: Client/Server	**SuperClass:**
Responsibilities	Collaborations
Init its role as client or server	ProxyPipe
Write bytes to the ProxyPipe	
Read bytes from the ProxyPipe	
Shutdown	

Class: ProxyPipe	**SuperClass: StateMachine**
Responsibilities	Collaborations
Init the client or server end	
of the named pipe	NT system
Write byes to the named pipe	Client/Sever
Read bytes from the named pipe	
Disconnect the client or server end	

3.2 From CRC Cards to Sequence Diagram

Once the responsibilities are understood, it is time to turn our attention to how these responsibilities will be played out in object interactions. The path from CRC cards to sequence diagrams is often quite straightforward. We have assigned a responsibility to a particular class and that class will then interact with other classes to fulfill that responsibility. The sequence captures these interactions. Look at the following diagram.

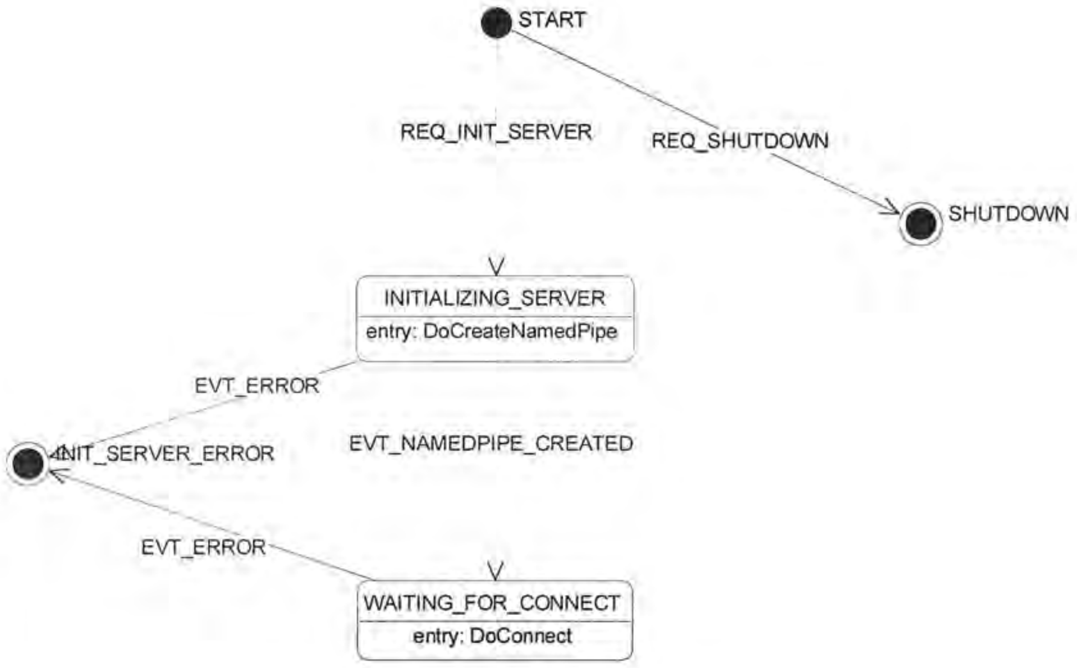

Server
ProxyPipe1
System1
System2
ProxyPipe2
Client
1: InitServer
2: CreateNamedPipe
3: Connect
4: InitClient
5: SaveNamedPipe
6: Write
7: CreateFile
8: Connection Completes
9: WriteFile
10: Data Arrives
11: Read
12: ReadFile
13: Shutdown
14: CloseFile
15: Write
16: WriteFile
17: Error:ClientNotAvailable
18: AckError
19: InitClient
20: SaveNamedPipe
21: Read
22: CreateFile
23: Connection Completes
24: ReadFile
25: error:InvalidState
26: AckError

3.3 From Sequence Diagram to State Machine Diagram

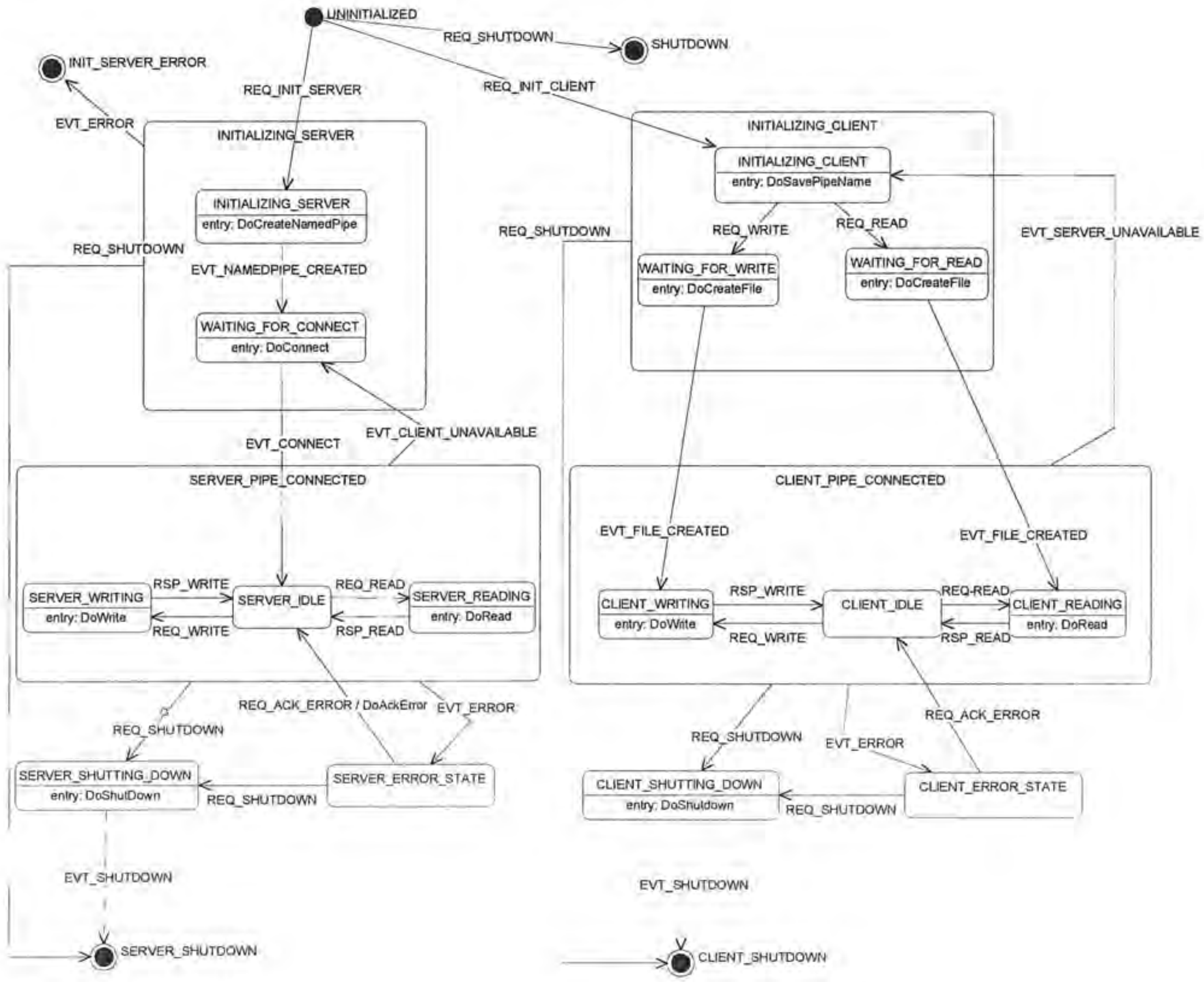
As we come to understand the interactions among the objects, we will need to understand the various possible states of each individual object. The state of an object is the current set of values for each of its member variables. We can model these states in a state transition diagram.

Every state diagram begins with a single start state and ends with zero or more end states. States are represented as rounded rectangles; each state has an identifying name. States are linked via unidirectional connections called transitions. An event is used as a trigger to go from one state to another.

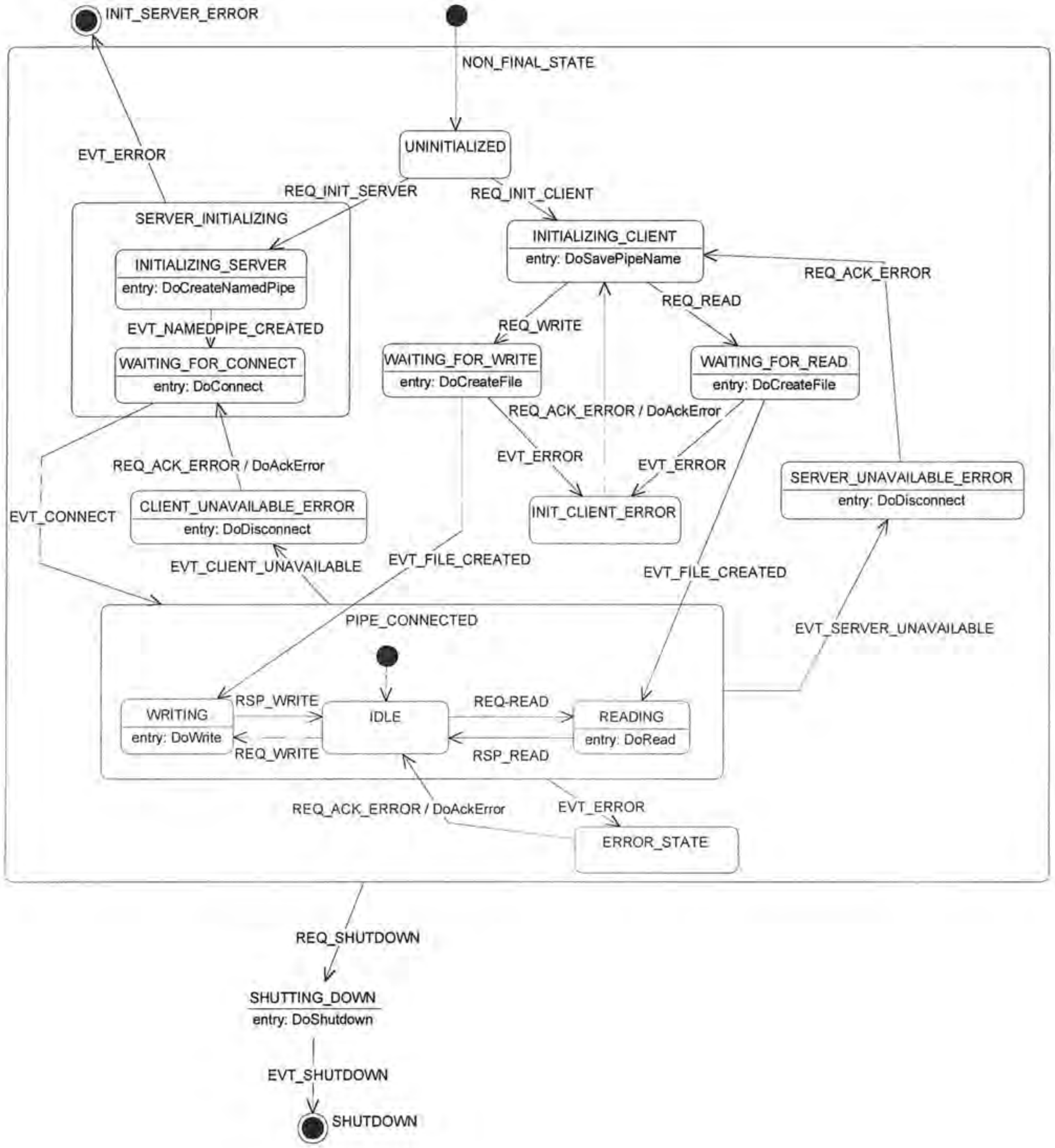
Let's start with state diagram for ProxyPipe. The state diagram for ProxyPipe to initialize it to be the Server end is shown below



To have the named pipe connected, we will have to initialize the client. After both server and client are initialized, they might not be connected yet. The connection only happens when client tries to Read or Write. This is because we only want the connection completed when it is needed. If an error occurs in the server before the pipe connected, it is not recoverable. State INITIALIZING_SERVER AND WAITING_FOR_CONNECT both have a path to go to this error state. They can be described as being in the INITIALIZING_SERVER super state. Similarly, we can apply the super state for other states to simplify the transition. Let's first separate client and server and consider the following state transition diagram:



We also noticed that in the diagram above the super state, SERVER_PIPE_CONNECTED and CLIENT_PIPE_CONNECTED have the same kind of structure, states and transition. So can combine them into one as showed in the following. However the INITIALIZING_SERVER AND INITIALIZING_CLIENT can not be combined into one since INITIALIZING_CLIENT has three states and INITIALIZING_SERVER has two states and their transitions are different too. INITIALIZING_SERVER goes to the INIT_SERVER_ERROR, which is not recoverable and INITIALIZING_CLIENT goes to INIT_CLIENT_ERROR, which is recoverable.



3.4 The Structure of the Real Problem

Now let's take a second look at the user of ProxyPipe-Client, Server, and ProxyPipe itself in our real problem.

We have two different process, Control.exe and Image.exe, which possibly are installed in different machine. We will take Image.exe as the client process and Control.exe as the server process.

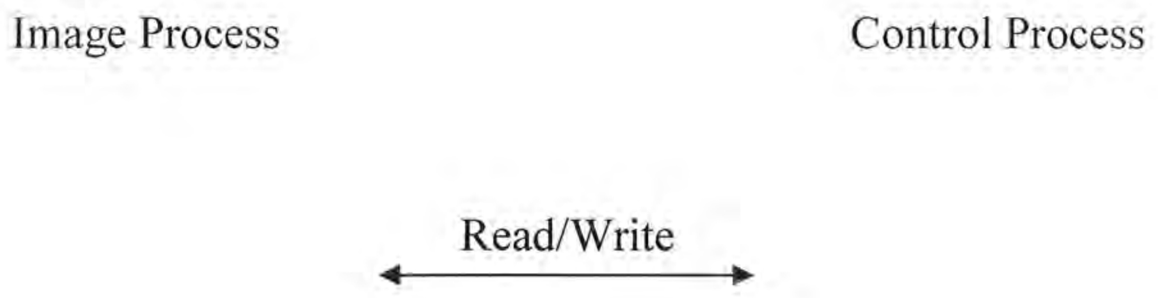
Here are the three objects that are involved in our real problem:

- ControlProxy object - The proxy for the Machine object of Control.exe process. It stays in the Image.exe process. The Image.exe process, the client, can send messages to ControlProxy as if it were sending message to real Machine object. These messages include FocusMove(), FocusJog() and FocusGetState(), etc. The interface of class ControlProxy should be exactly the part of class Machine interface that is involved in the process communication.
- ControlProxyServer object - The server that listens to the message sent from the ControlProxy and tells the Machine to respond to the message. It stays in the Control.exe process. Since the server always keeps listening, we need a listener thread.
- ProxyPipe Object - The transport bus between the ControlProxy object and the ControlProxyServer object. The same ProxyPipe class will play roles of both server and client. Both ControlProxy and ControlProxyServer classes are user of ProxyPipe object. That is, they both have an instance of ProxyPipe as their member variable.

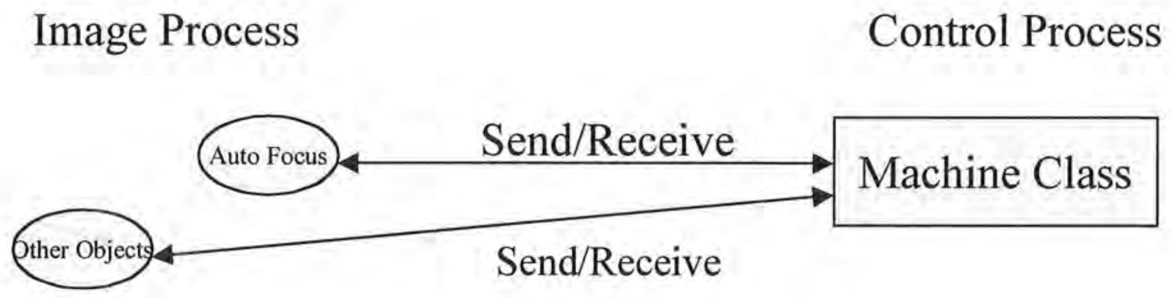
The ControlProxy class and the ControlProxyServer class work together to allow other processes to send commands to the Control.exe process.

To understand the relationship of objects mentioned above, let's divide them into the following different layers, which go from more abstract to more concrete levels.

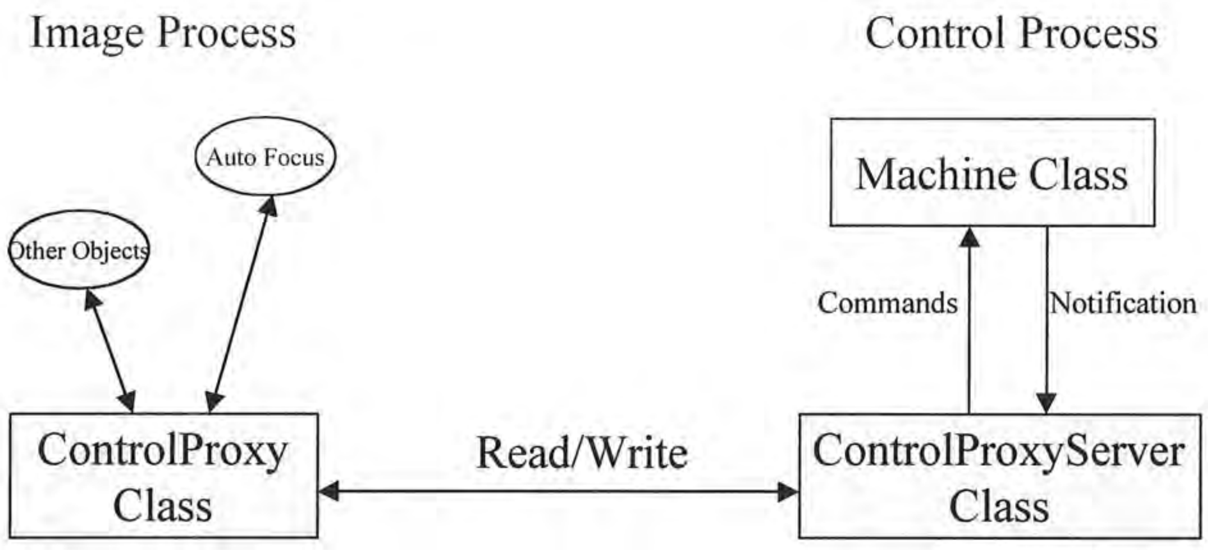
Layer 0: Process Layer



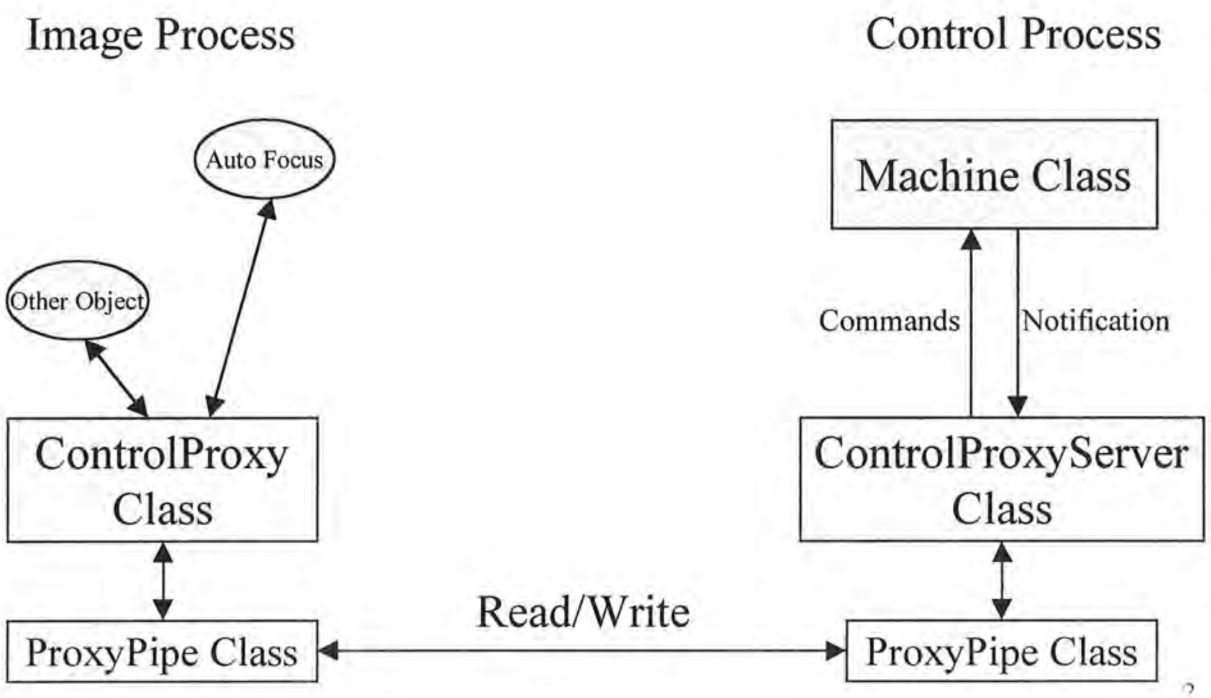
Layer 1: Object - Machine Layer



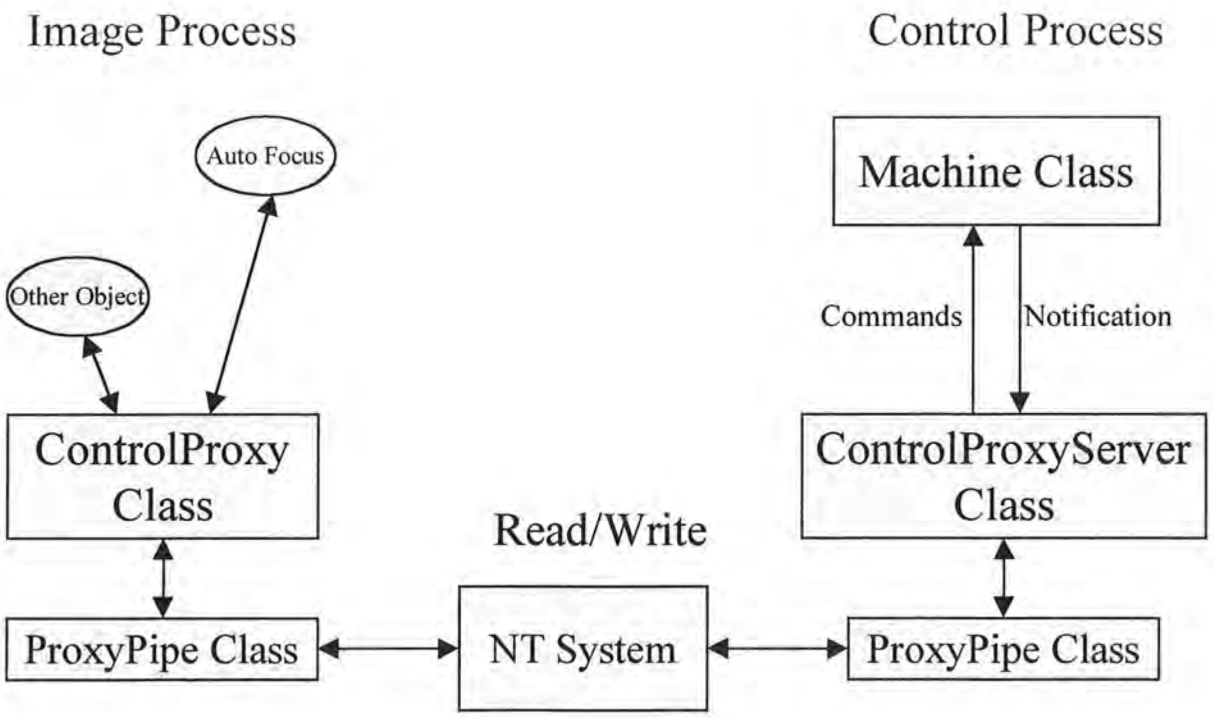
Layer 2: Proxy Layer



Layer 3: Proxy Pipe Layer



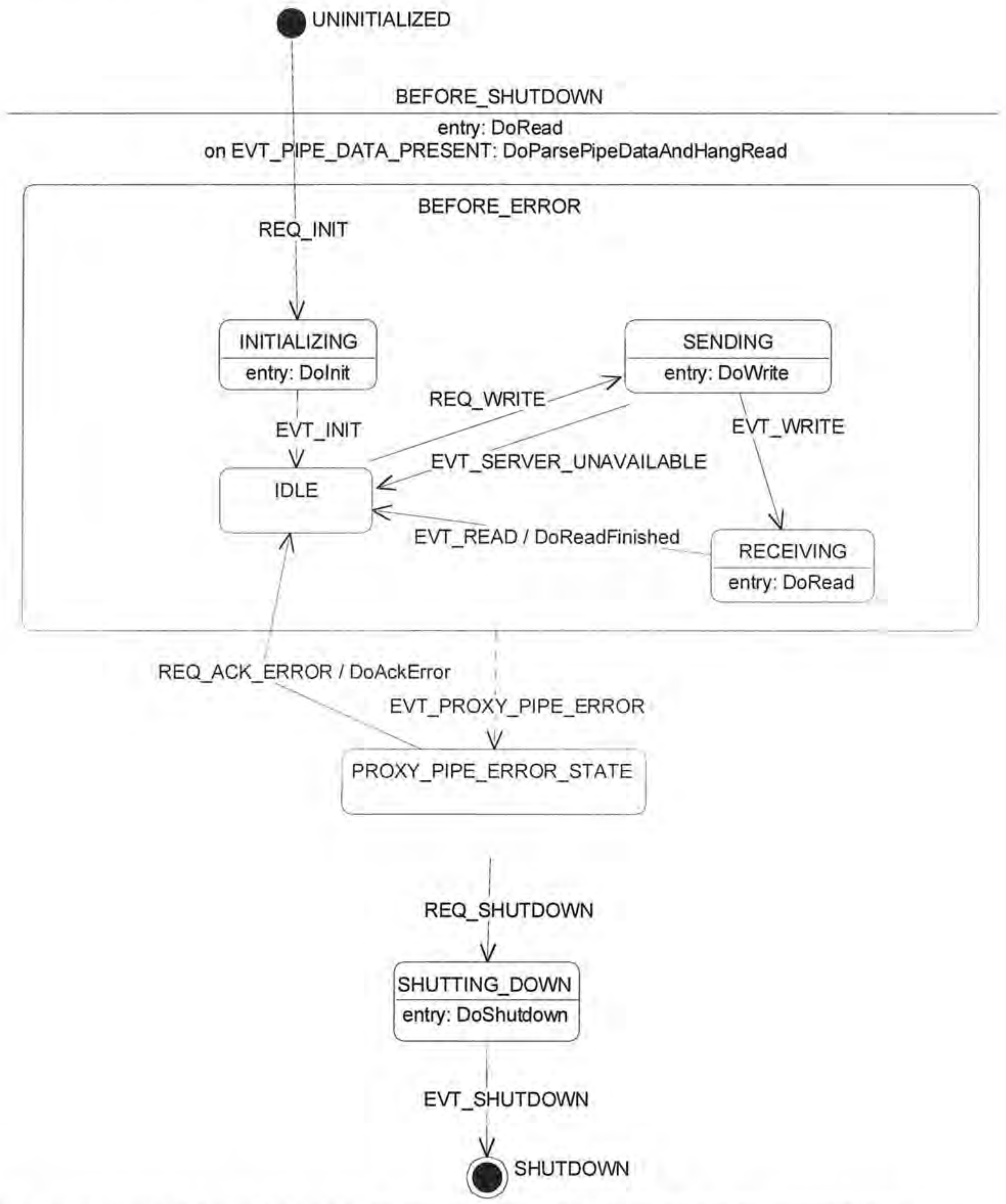
Layer 4: NT system



3.5 Class ControlProxy and ControlProxy Server

We have modeled the states of Class ProxyPipe in a state diagram. Now let's look at the states for ControlProxy. The commands FocusAckError(), FocusJog(), FocusMove(), FocusReportPosition() follow the same pattern of ProxyPipe Write and Read. So its state machine could be abstracted into the following:

Abstract Control Proxy State Machine



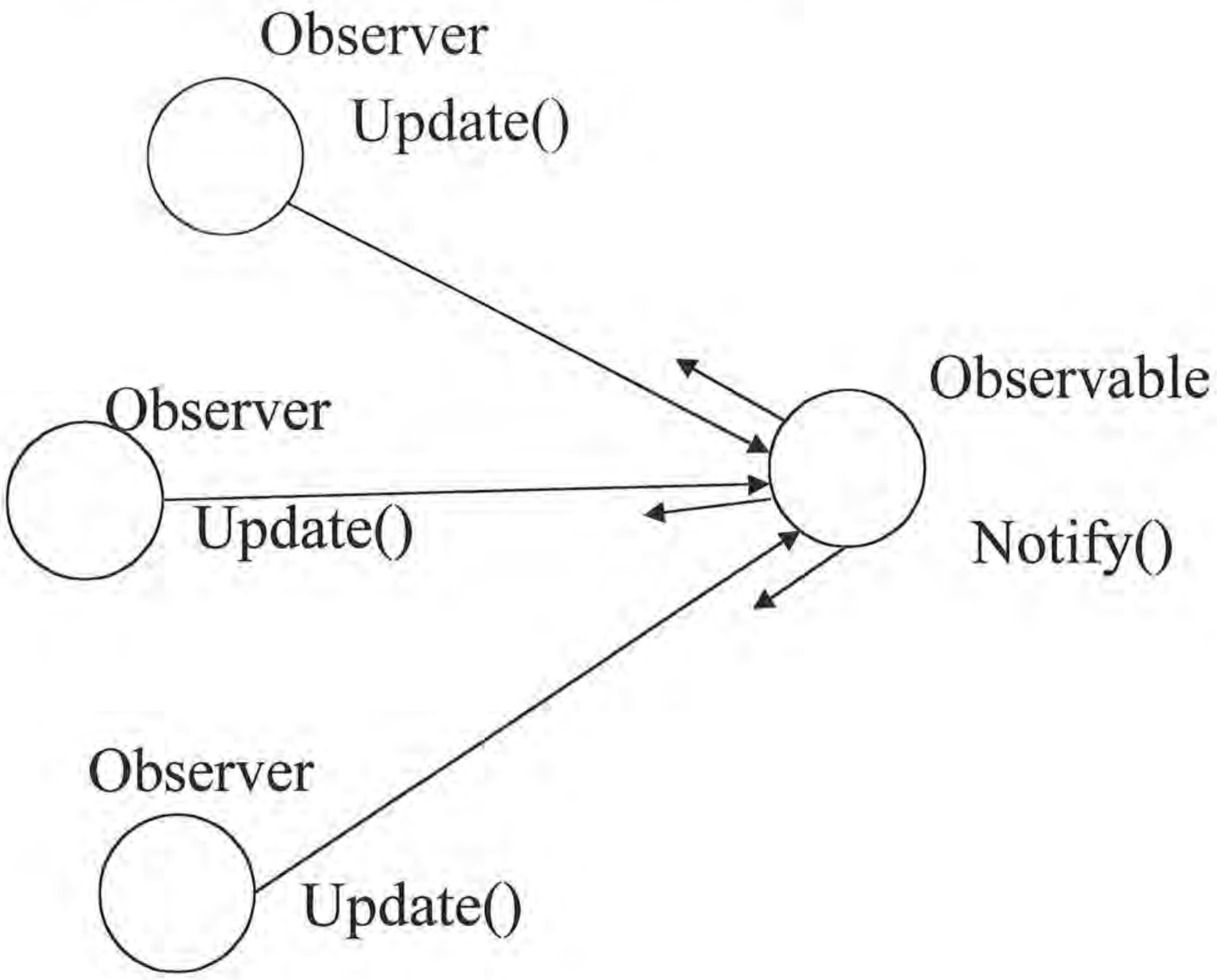
ControlProxyServer does nothing but listen to the commands from ControlProxy and pass the commands to the Machine object. After it is initialized, it will stay in this one state until it is shut down.

The state machines defined by UML are deterministic. Therefore, a state diagram must not leave any room for ambiguous constructs. This means, in particular, that it is always necessary to describe the system's initial state. For a given hierarchical level, there is always one and only one initial state. Conversely, it is always possible to have several final states that each corresponds to a different end condition.

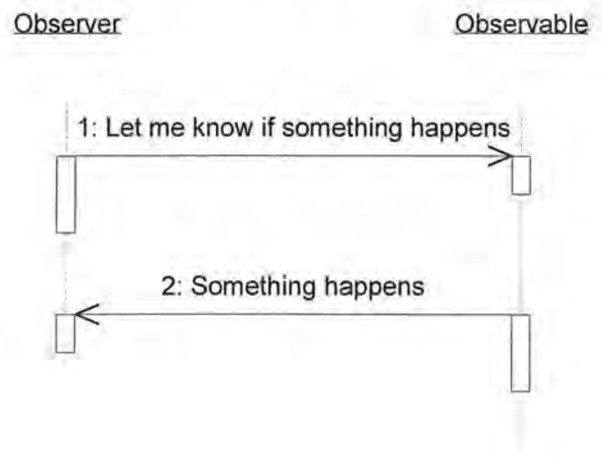
4. Implementation

4.1. Observer

As we discussed above, class ControlProxy and ControlProxyServer both need to include a header file of class ProxyPipe. They both depend on class ProxyPipe. When the ProxyPipe changes its state, ControlProxy and ControlProxyServer both need to be notified and updated automatically. In addition, the ControlProxy cannot start sending commands until the ProxyPipe is ready to transfer bytes. In order to prevent the ControlProxy state machine to receive the invalid input, we need certain way of controlling the startup of the function. We don't want busy waiting since that will need an additional thread. So we use ControlProxy state machine to be an observer of the ProxyPipe. The collaboration diagram for the observer pattern looks like the following:



A simple sequence diagram with one observer would look like this:



There could be more than one observer to one observable object. As an example, we will describe ControlProxy as an observer of class ProxyPipe in the details of code.

First we declare the class ControlProxy in the file ControlProxy.h as a subclass of class Observer:

```
#include "Control\Observer.h"
#include "Control\StateMachine.h"
...
class ControlProxy : public Observer, public StateMachine
{
public:
...
        virtual void Update(class Observable* observable);
...
private:
        class ProxyPipe* proxyPipe;
};
```

Second, we will implement the function Update(). The class ControlProxy updates its states based on the states of class ProxyPipe. In the following code, we are mapping the state of class ProxyPipe into the state of class ControlProxy.

```
void ControlProxy::Update(class Observable* observable)
```

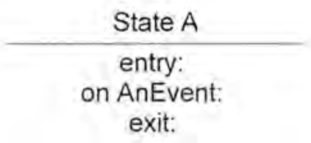
```
{
        switch(proxyPipe->GetState())
        {
        case ProxyPipe::IDLE:
                if(IsValidInput(EVT_READ))
                {
                        Input(EVT_READ);
                }
                if(IsValidInput(EVT_WRITE))
                {
                        Input(EVT_WRITE);
                }
                break;
        case ProxyPipe::INITIALIZING_CLIENT:
                if(IsValidInput(EVT_INIT))
                        Input(EVT_INIT);
                else if(IsValidInput(EVT_SERVER_UNAVAILABLE))
                        Input(EVT_SERVER_UNAVAILABLE);
                break;
        case ProxyPipe::SHUTDOWN:
                if(IsValidInput(EVT_SHUTDOWN))
                        Input(EVT_SHUTDOWN);
                break;
        case ProxyPipe::ERROR_STATE:
        case ProxyPipe::INIT_CLIENT_ERROR:
        case ProxyPipe::INIT_SERVER_ERROR:
                {
                        if(IsValidInput(EVT_PROXY_PIPE_ERROR))
                        {
                                Input(EVT_PROXY_PIPE_ERROR);
                        }
                }
                break;
        default:
                break;
        }
}
```

Finally, we will attach the class ControlProxy to the observer list of class ProxyPipe, which is the observable. We will do the following in the constructor of class ControlProxy:

```
ControlProxy::ControlProxy():
StateMachine("ControlProxy State Machine",UNINITIALIZED,STATE_COUNT,INPUT_COUNT),
Observer(),
...
{
        proxyPipe->GetStateObservable()->Attach(this);
}
```

4.2. Concurrency

In UML, states may also contain actions; these are executed upon entering or exiting a state, or when an event occurs while the object is in the state as shown below.



The action on entry (which is symbolized by the keyword entry:) is executed in an instantaneous and atomic way upon entry into the state. Similarly, the action on exit (symbolized by exit:) is executed upon exiting the state. The action on an internal event (symbolized by the event name followed by :) is executed upon the occurrence of an event that does not lead to another state.

We don't want state machine blocked on the entry function since in that case the state machine can't accept another input. In the ProxyPipe state machine above, the state WAITING_FOR_CONNECT has an entry function DoConnect(), which could possibly call the NT function ConnectNamedPipe(). If the client end of ProxyPipe has not been created, then ConnectNamedPipe() would be waiting there until the client end of the ProxyPipe is created. If so, the entry function DoConnect() might not be able to be executed instantaneously. Now when we try to shut down the server end while it is waiting for connection with the client pipe, the ProxyPipe state machine would not change its state to be SHUTDOWN since it is hanging on the ConnectNamedPipe() call. Then we could not shut down the server right away.

The reason for this is that we are only using one thread. Only one thing can happen at a time. The way to solve this problem is to create another thread to do the ConnectNamedPipe() job. And the entry function DoConnect() only needs to notify the other thread that a connect event has occurred.

Similarly, the state READING has an entry function DoRead(). If either the server or the client end of the ProxyPipe try to read bytes while the other end has not write the bytes yet, it will hang on NT system call ReadFile().

We could also let another thread to handle the NT ReadFile() call. The DoRead() entry function will only notify that second thread that a read event has happened.

We could share the same thread to handle both ConnectNamedPipe() and ReadFile() NT system calls. Since ConnectNamedPipe(), ReadFile() and Shutdown() can not happen at the same time, we need an access-control mechanism.

The Microsoft Foundation Classses (MFC) supports four synchronization objects: critical sections, mutexes, semaphores and events. The MFC also supports two types of locks: single locks and multi-locks. The details of the implementation of the locks and even of the synchronization objects are encapsulated in the objects and opaque to the application developer.

In a typical multithreaded application, we need to create thread-safe classes. Thread-safe means the application designed so that no harm can come from being interrupted in mid-operation. To make a class fully thread-safe, first add the appropriate synchronization class to the shared classes as a data member. Next add synchronization calls to the appropriate member functions of each thread-safe class. This means that all member functions that modify the data in the class or access a controlled resource should create either a CSingleLock or CmultiLock object and call that object's Lock function. That is, in order to use the synchronization classes CSemaphore, CMutex, CcriticalSection, and CEvent,, we must create either a CSingleLock or CmultiLock object to wait on and release the synchronization object.

In our application here, we will create a connect event, a read event and a shutdown event for the entry function DoConnect(), DoRead() and DoShutdown() separately to notify the other thread that some event has happened. The connect event , read event and shutdown event are all instances of class CEvent. Here we will use CmultiLock, since there are multiple objects that we could use at a particular time. Here, locks are semaphores that are used to serialize access to critical regions, such as the region where only one of ConnectNamedPipe(), ReadFile() and Shutdown() can be executed.

To explain how we use the MFC synchronization classes CEvent and CmultiLock in more detail, here is what the real code looks like:

First, in class ProxyPipe, we create three instances class CEvent :

```
ProxyPipeImpl : public ProxyPipe
{
...
CEvent *connectEvent;
CEvent *readEvent;
CEvent *shutdownEvent;
…
}
```

Second, implement the entry functions DoConnect(), DoRead() and Do Shutdown() in the following way:

```
Void ProxyPipeImpl::DoConnect()
{
        connectEvent.SetEvent();
}
void ProxyPipeImpl::DoRead()
{
        readEvent.SetEvent();
}
void ProxyPipeImpl::DoShutdown()
{
        shutdownEvent.SetEvent();
}
```

Third, create a second thread by doing the following:

a) Declared the second thread in the header file

```
class ProxyPipeImpl : public ProxyPipe
{
…
private:
        static DWORD __stdcall ProxyPipeThread(void* thisPtr);
        virtual DWORD ProxyPipeThreadFunc();

        CWinThread* proxyPipeThread;
…
}
```

b) Implement the thread function in the cpp file.

```
DWORD __stdcall ProxyPipeImpl::ProxyPipeThread(void* thisPtr)
{
        return ((ProxyPipeImpl *) thisPtr)->ProxyPipeThreadFunc();
}

DWORD ProxyPipeImpl::ProxyPipeThreadFunc()
{
        static const int CONNECT_EVENT = WAIT_OBJECT_0;
static const int READ_EVENT = WAIT_OBJECT_0+1;
static const int SHUTDOWN_EVENT = WAIT_OBJECT_0+2;
CSyncObject* event[3] = {&connectEvent, &readEvent, &shutdownEvent};
CMultiLock lock(event,3);
while(1)
{
        switch(lock.Lock(INFINITE, FALSE))
        {
        case CONNECT_EVENT:
                connectEvent.ResetEvent();
                NT::ConnectNamedPipe(hPipe, &overLappedRead, ntError);
                if(ntError.GetId() == ERROR_SUCCESS ||
                        ntError.GetId()     == ERROR_PIPE_CONNECTED)
                {
                        TRACE("pipe connected\n");
                        error.Set(MachineError::PROXY, NO_ERROR, "NO ERROR");
                }

                else
                {
                        error.Set(MachineError::PROXY, PROXY_PIPE_ERROR ,
GetProxyError());

                }
                break;

        case READ_EVENT:
                error.Clear();
                ReadOverlapped();

                break;

        case SHUTDOWN_EVENT:
                return 0;
                break;

        default:
```

```
                    error.Set(MachineError::PROXY, PROXY_PIPE_ERROR, GetProxyError());
                    assert(false);
                    break;
                }
            }
        }
```

Notice that in the code above we have a class ProxyPipeImpl that extends the class ProxyPipe. Here ProxyPipeIml is one way of implementing class ProxyPipe. Separation of class ProxyPipe and class ProxyPipeImpl will give us flexibility to switch to a different implementation of class ProxyPipe.

4.3. Error Report

Suppose in the middle of the process communication some accident happens in the system. How could the software detect the reason of the problem and report it to the user?

The Windows NT system can return some error code by calling the function GetLastError(). However, the user might not be able to read the error code. So our job now becomes how to translate the NT system error code into the user understandable error message.

The solution is to try each possible case to interrupt the process communication and record the error code returned by GetLastError(). For example, when client tries to read or write bytes to the server before server starts or after server is shut down, GetLastError() will return the error code called ERROR_FILE_NOT_FOUND. We could translate this error code into an error message "Server is unavailable". Here are more examples:

Detected ProxyPipe Error	From NT System
CLIENT_UNAVAILABLE *Start up the client application*	**ERROR_NO_DATA**[232]: The pipe is being closed . *(When the server tries to write after the client is shutdown)* **ERROR_BROKEN_PIPE** [109]: The pipe has been ended. *(when the server tries to read after the client is shutdown)*
BUFFER_TOO_SMALL *Increase the buffer size*	***ERROR_MORE_DATA** [234]: More data is available.* *(When the ProxyPipe class tries to read if the buffer size is smaller than the number of characters in the named pipe.)*
SERVER_BUSY *Only run one client at one time*	**ERROR_PIPE_BUSY** [231]: All pipe instances are busy. *(When a client tries to connect to the server, which has been connected by another client.)*
SERVER_UNAVAILABLE *Start up the server application*	**ERROR_FILE_NOT_FOUND** [2]: The system cannot find the file specified. (*When the client tries to read and write before the server is open* .) **ERROR_PIPE_NOT_CONNECTED** [233]: No process is on the other end of the pipe. (*When the client tries to read and write after the server is shutdown*)

5. Conclusions and Future Directions

ProxyPipe, together with the proxy, make the communication between two processes easier by encapsulating the NT system calls. We can concentrate on sending commands across different memory space rather than dealing with the communication details. ProxyPipe is designed to transfer bytes instead of a string. That is, the ProxyPipe can also transfer the character '\0', which is the ending character for a string.

By using state machine and observer pattern, we can control the start up of certain function without busy waiting. That could avoid using an additional thread.

When the server shut down, the client can detect it by the error message returned when the client was trying to do a read or write. After the server restarts, it can automatically connect to the same client. It is vice versa for the client to shut down and reconnect.

This project has two limitations. First the ProxyPipe will only deal with one server and one client. Second, the ProxyPipe can only handle one read or write at one time. So the future work will include the two points:

1. Develop the ProxyPipe that can deal with multiple servers and multiple clients.
2. Develop the ProxyPipe that can do overlapped read and write at the same time.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software. 1995* Addison-Wesley Professional Computing Series.
2. Pierre-Alain Muller. *Instant UML.* 1997 Wrox Press.
3. Timothy Budd. *An introduction toObject-Oriented Programming, Second edition. 1997 Addison Wesley Longman, Inc.*
4. Jesse Liberty. *Beginning Object-Oriented Analysis and Design with C++.* 1998 Wrox Press.
5. MSDN Library October 1998 release