2006

Distributed Objects in C#

Xiaoyun Xie

Rochester Institute of Technology

Department of Computer Science

Master of Science Project

Distributed Objects System in C#

Submitted By: Xie, Xiaoyun (Sherry)

Date: February 2004

Chairman: Dr. Axel T. Schreiner
Reader: Dr. Hans-Peter Bischof
Observer: Dr. James Heliotis

ABSTRACT

Today more and more programs run over a collection of autonomous computers linked by a network and are designed to produce an integrated computing facility. Java Distributed Objects (JDO) proposed by Dr. Axel T. Schreiner [1] builds an infrastructure which allows distributed program components to communicate over a network in a transparent, reliable, efficient, and generic way.

JDO was originally intended as a teaching device to assess design parameters for distributed objects. This project focuses on porting JDO, which is implemented in Java on Sun's JDK, to C# on Microsoft's .NET. On one hand, it builds an infrastructure in C# that simplifies the construction of distributed programs by hiding the distributed nature of remote objects. On the other hand, it generates insights into the differences between two platforms, namely, Java on Sun and C# on .NET, in the distributed objects area. This document illustrates the architectural design of the C# Distributed Objects system and compares programming technologies, which are required by this system design, in Java and C#.

Table of Contents

ABSTRACT 3
1 Introduction 7
1.1 Motivation and Goal 7
1.2 Composition 8
2 Background 9
2.1 Terminologies 9
2.1.1 Remote Procedure Call 9
2.1.2 Nested Remote Procedure Call 10
2.1.3 Distributed Objects 11
2.2 Systems Overview 12
2.2.1 Proxy Pattern 12
2.2.2 Systems Comparisons 13
3 System Specification 15
3.1 System Architectural Overview 15
3.2 Workflow Overview 17
3.2.1 CDO 18
3.2.2 ACDO 19
4 Analysis of Key Technologies 21
4.1 Interface 21
4.2 Threads 22
4.3 Sockets 23
4.4 Object Stream 24
4.5 Dynamic Class Loading 27
4.6 Identity Hash Table 29
5 Usage and Sample Applications 31
5.1 Time Service 31
5.2 CPU Service 32
5.3 Chat Room Service 33
6 Summary 35
7 References 37

1 Introduction

1.1 Motivation and Goal

Applications consisting of complex components running on various machines across a network are very common today. In the so-called client/server model, client and server programs run on different hosts in a networked system. A client program sends requests and receives answers from a server; the server program that receives requests sends out replies after performing associated computing tasks. In a modern object-oriented distributed application, objects need to be able to communicate with one another over a network.

There are various approaches to remote object communication: Java/Remote Method Invocation (Java/RMI) from JavaSoft, Common Object Request Broker Architecture (CORBA) from OMG, and .NET Remoting from Microsoft are good examples. They each aim to extend an object-oriented system by distributing objects to different processes and hosts, allowing each component to interoperate as a unified whole [paraphrased from 2, Introduction section].

There are two types of clients which will be mentioned frequently in the rest of this report. One is the client program as in the conventional concept of the client/server application model; the other is the client of a distributed objects system, that is, the programmer who utilizes the system to develop applications. To differentiate between these two types of clients, from now on, the latter will be called the "client programmer".

None of those technologies is transparent to client programmers: they more or less know about the remote nature of the services. User transparency is an important parameter in software design. It generally simplifies the client programmers' programming work, making user applications more organized and easier to maintain.

The purpose of this project is to port a distribution methodology from Java to C#. Dr. Axel T. Schreiner's Java Distributed Objects system is the Java version of this architecture (information about JDO can be found in [1]). JDO was intended as a teaching device to assess design parameters for distributed objects. This distribution methodology builds a middle layer that is completely transparent to end users. Remote service objects look local to client applications, even though they can be distributed to different machines across a network.

The ported system is implemented in C# under .NET. Although .NET offers a strong high-level remoting infrastructure, this project uses lower-level network programming technologies to build its own remoting solution. Building the solution largely from scratch allows experimentation on various design issues of distributed objects. Porting from Java to C# also allows one to compare the related technologies from the two different platforms.

1.2 Composition

The project consists of several parts. Firstly, two system packages with different architectural design are provided, namely the C# Distributed Objects (CDO) system and the Asynchronous C# Distributed Objects (ACDO) system. Then three sample applications are included, demonstrating the usage of the two system packages.

This document will explain the technical details of the project. Chapter 2 provides some background knowledge of distributed objects systems. Chapter 3 analyzes the system architecture of the C# Distributed Objects system, demonstrating how this mechanism enables distributed components to communicate over a network. Chapter 4 analyzes the programming technologies required by this particular system architecture, focusing on how they are done in Java, and how they could be solved in C#. It also discusses comparisons of Java and C# on various issues. Chapter 5 briefly introduces the three sample applications which demonstrate how the C# Distributed Objects system is tested and utilized. Lastly, a brief summary on various porting issues are presented in Chapter 6.

Readers should have a working knowledge of network programming (such as TCP/IP, sockets, streams, etc.) and some basic understanding of Java as well as C# and the .NET platform.

2 Background

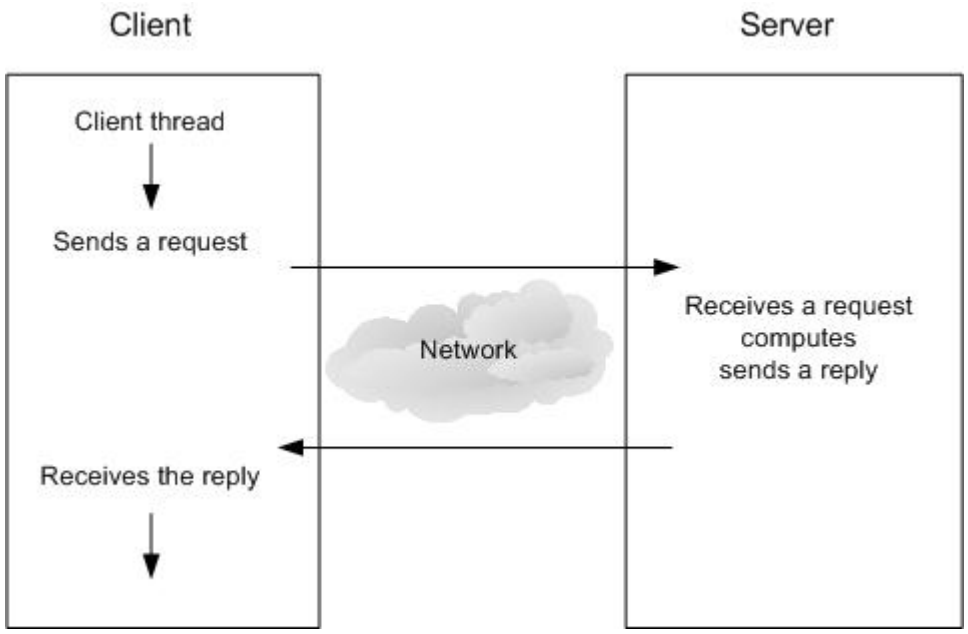
This chapter reviews some information about remoting. First, terminologies of a few technologies related to distributed computing are introduced. Those concepts are the basis of the C# Distributed Objects system and are referenced when discussing the technical details of the C# D.O. system in later chapters. Then, a proxy pattern is explained, from which readers shall gain a high-level understanding of the general design of distributed objects systems. Last, the C# D.O. system is compared with several other existing remoting systems; features of each system and the differences among them are discussed.

2.1 Terminologies

2.1.1 Remote Procedure Call

RPC allows a client to make a function call to access the server on a remote system without knowledge of the lower-level network information. RPC uses a synchronous, request-reply (sometimes referred to as "call/wait") protocol which involves blocking the client until the server fulfills its request [paraphrased from 3].

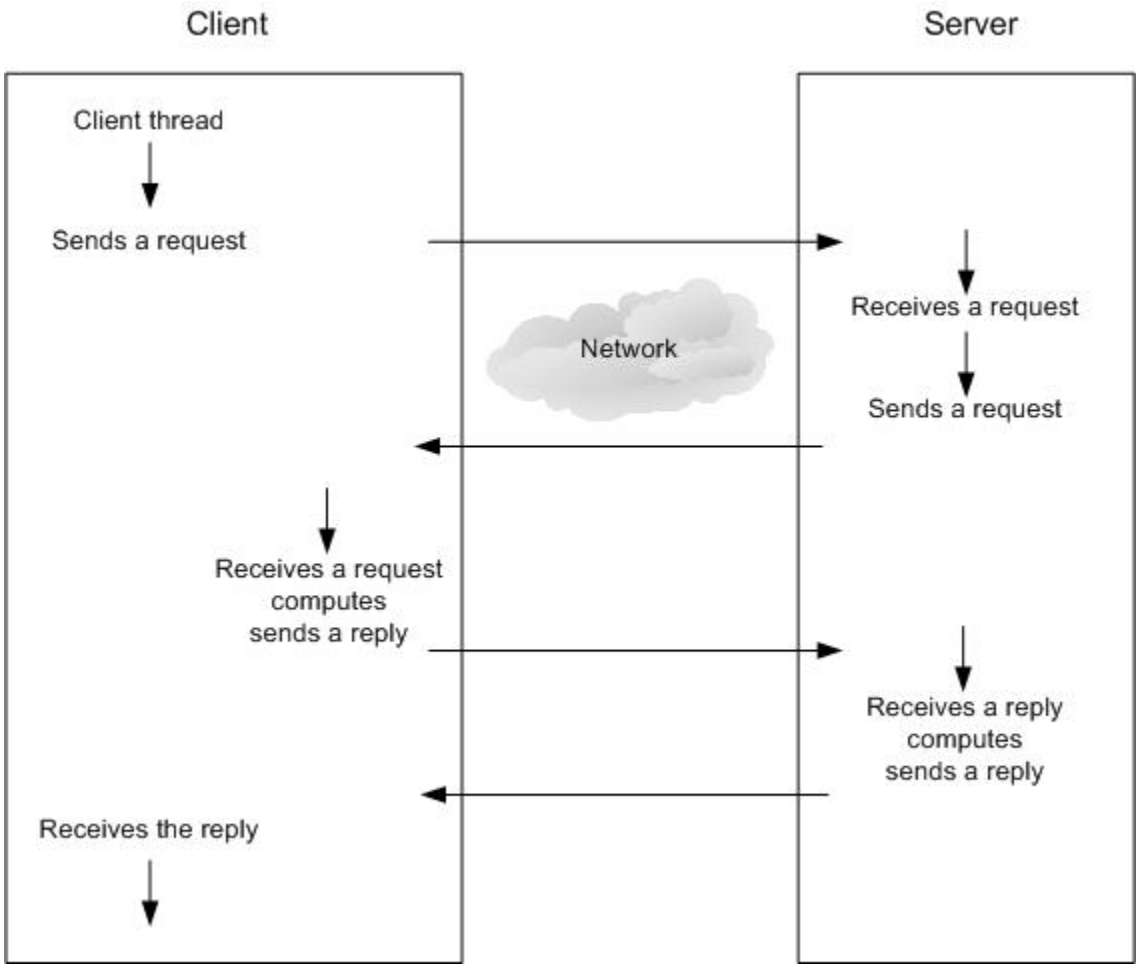
The following diagram shows the communication mode of RPC.



A client issues a request and waits for a server's response before continuing its own processing (more detailed information about RPC can be found in [4]).

2.1.2 Nested Remote Procedure Call

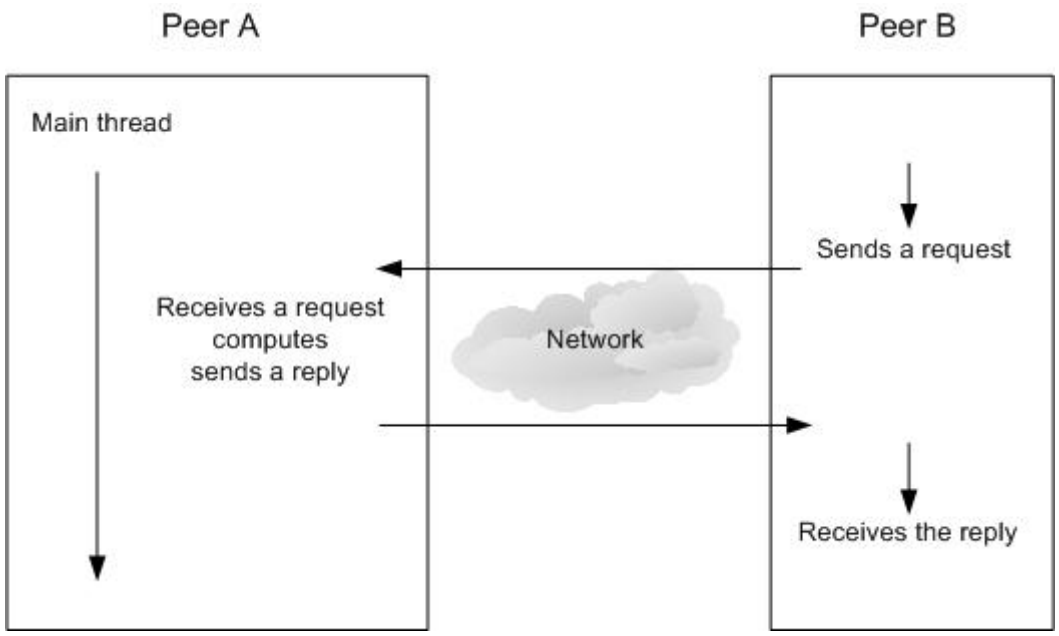
Nested Remote Procedure Call (NRPC) is similar to RPC except that a server can talk back to a client during a conversation. For example, during a server's computation for a client request, it may send a request to the client to ask for further information or help. The following diagram shows the communication mode of NRPC.



NRPC increases the flexibility of a system by allowing more complicated conversations between clients and servers: while the client thread is waiting for the reply to a RPC, it can process a callback from the server.

2.1.3 Distributed Objects

A distributed objects (DO) architecture allows asynchronous communication between objects. There is no distinct difference between a client and a server in this architecture since either one can start a conversation by sending out a request. Frequently DO is referred to as peer-to-peer communication. The following diagram shows an example of DO communication mode.



In the above example, peer B starts a conversation by sending a request to peer A; peer A responds without interfering with the normal main thread. Thus, DO requires multithreading.

A typical example of where this architecture is needed would be a chat room scenario. Multiple chat room clients can log into a chat room and talk to each other. A client may send a package out and receive a package at the same time. An incoming package may arrive at any time. Thus, more than one thread is needed; RPC or NRPC would not work in this situation.

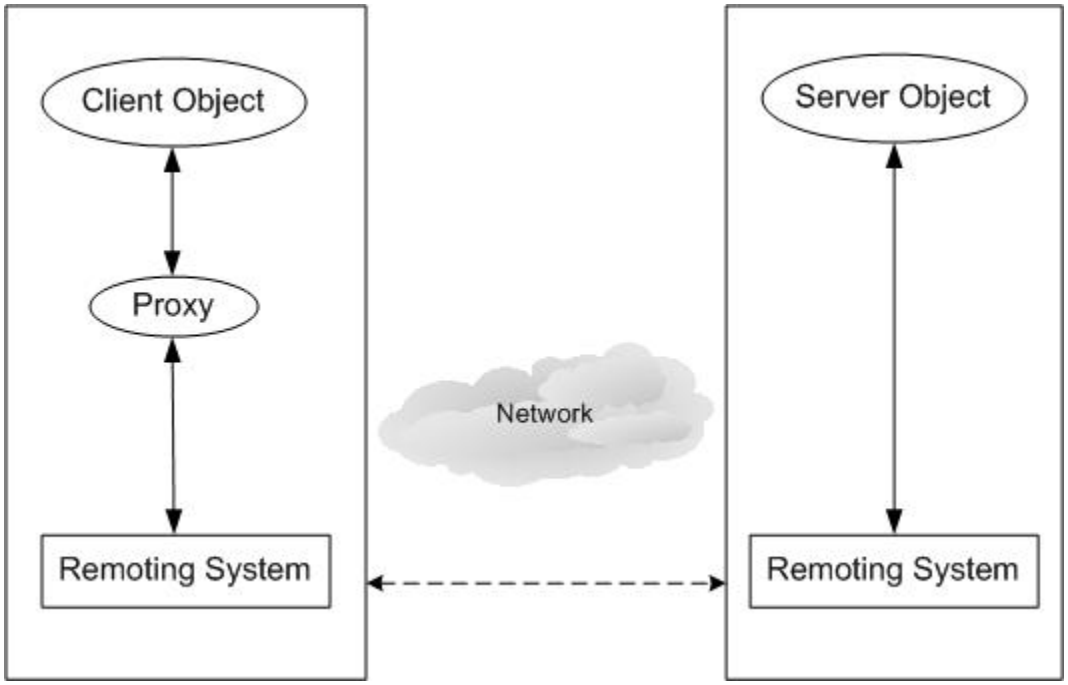
2.2 Systems Overview

2.2.1 Proxy Pattern

The general goal of a distributed object system is to keep the semantics of a service the same whether or not the client and the server reside in the same address space. Using object proxies to communicate between a client and a server object is the common design.

A distributed objects system uses proxy objects to create the impression that the server object is in the client's process. Proxies are stand-in objects that present themselves as some other objects. When a client creates an instance of the remote type, the distributed object infrastructure creates a proxy object that looks exactly like the remote type to the client. The client calls a method on that proxy, and the distributed objects system receives the call, routes it to the server process, invokes the server object, and returns the return value to the client proxy, which returns the result to the client. [Paraphrased from 5]

The following diagram illustrates the architecture of a general proxy pattern.



How remote calls are conveyed between clients and server objects, and how a proxy and a server object are matched are major concerns of a distributed objects system design.

2.2.2 Systems Comparisons

Note that we do not intend to replace the existing strong remoting technologies mentioned in section 1.1 with techniques explored by the small-scale effort outlined here. JDO was originally intended as a teaching device to assess design parameters for distributed objects.

The following table compares the C# Distributed Objects system with RMI, CORBA, and .NET remoting, demonstrating the features and differences of these technologies.

	Operating System	Programming Language	Transparency
CORBA	multiple	several	no
RMI	multiple	Java	medium
.NET Remoting	Windows	(CLR)	medium
C# D.O. system	Windows	(CLR)	almost

CORBA can be programmed in many languages and run under multiple operating systems. RMI supports only Java and can run under various operating systems. .NET Remoting and the C# D.O. system run on Windows and can be used in multiple languages on the .NET platform which are supported by the Common Language Runtime (more information about CLR can be found in [6, page 18]).

Transparency describes to what degree the user application is aware of the distribution of service objects. CORBA is essentially a protocol; a CORBA application developer knows all the details of the system. Thus, CORBA is not transparent at all.

RMI hides the communication between distributed objects, but a RMI application developer is still well aware of the remote nature of the service objects, since every remote object has to implement a specific remote interface and throw remote exceptions.

.NET remoting is on a level similar to RMI since a remotable object has to be derived from *MarshalByRefObject* and a programmer has to write configuration files for both the client and server applications.

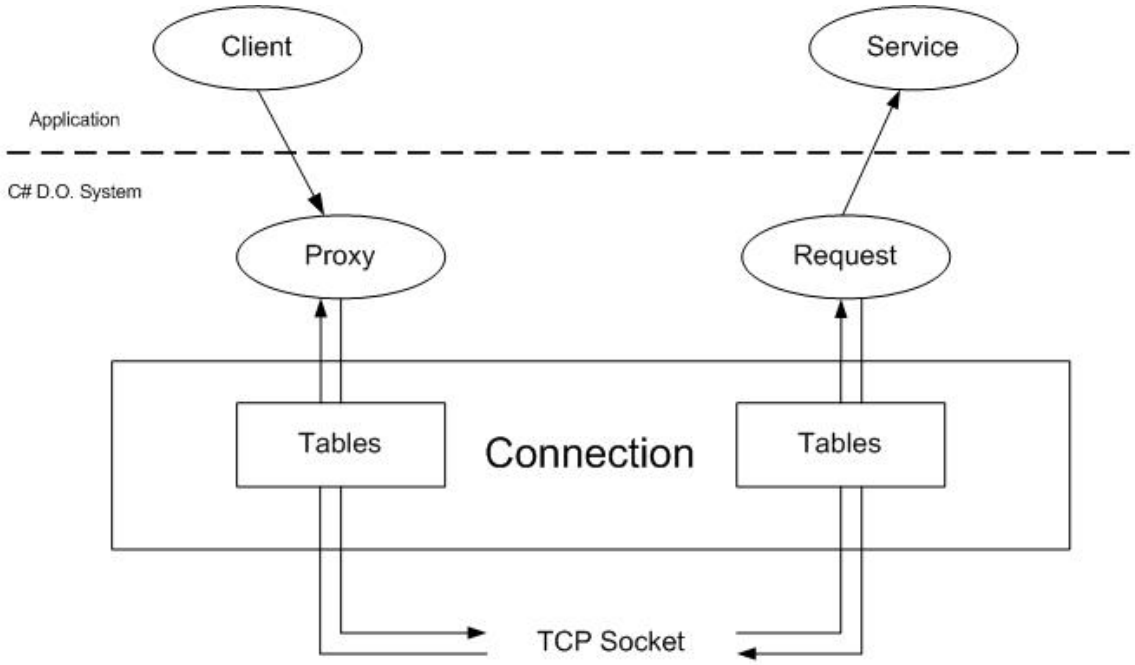
In the C# D.O. system most of the communication work is done silently and a developer implements the client and server objects just as usual. In this particular implementation of the C# D.O. system, a client does have to know the host name and port number of the server. However, a naming service could be implemented to avoid it; that is to say, a client can obtain the address of a remote service object by providing a logical name.

3 System Specification

This chapter starts with an analysis of the C# Distributed Objects system architecture, followed by a detailed explanation of workflows of the two system packages, CDO and ACDO.

3.1 System Architectural Overview

Based on the proxy pattern that is described in section 2.2.1, the following diagram shows the basic architecture and the control/data flow of the C# Distributed Objects system.



The client program and the server program reside on different end systems connected by a computer network. Services are represented by proxies. The client is aware of the service object's interface, which ensures that the methods of the remote object are invoked properly. The *Connection* is essentially the Transport-Layer of the system. It is built on top of sockets, utilizing a TCP connection between the client and the server to do the lower-level communication work. When the client wants to access a remote service, it invokes the specific proxy. The proxy sends a request out through the *Connection*. The request arrives at the service's side, and somehow the appropriate service object is invoked. A result value is wrapped in a reply and sent back to the proxy via the *Connection*. The proxy retrieves the return value and sends it up to the client.

A request wraps a method call, i.e. it should include the receiver, the method description, and arguments. Since a *Connection* can serve multiple objects, the name of the receiver should be specific enough to locate the appropriate service object. Therefore,

identifications are needed to uniquely identify objects, and mapping tables are needed to manage identifications.

A service can send a client messages that may include service objects. Instead of sending a service object over a *Connection*, its ID is sent. A mapping table that converts service objects to their IDs is needed. Service objects are represented by proxies at the client side, thus a mapping table that converts IDs to proxies is also necessary.

If a proxy is included in a message as an argument, it does not make sense to send the proxy over a network; instead, its ID is sent. The ID can be stored in the proxy as part of its state and retrieved when needed. When this ID reaches the service's side, it should be converted into the service object the proxy represents. Thus, a table that converts IDs to services is needed.

At the recipient's side, an ID for the service object should be resolved into a proxy and an ID for the proxy should be resolved into a service object. "Resolve" means that given an ID, the appropriate object is able to be found. To avoid confusion, two ID classes are created, namely, *ServiceId* and *ProxyId*. There should be a one-to-one relationship between *ServiceId*s and *ProxyId*s, so that a service object and a proxy can be correctly mapped.

The *Connection* layer in the C# D.O. system contains three identification tables, *ServiceById*, *IdByService*, and *ProxyById*. *ServiceById* maps *ServiceId*s to service objects. *IdByService* maps service objects to *ServiceId*s. *ProxyById* maps *ProxyId*s to proxies.

When a service object is sent out, its *ServiceId* is sent instead by looking up the *IdByService* table. After arriving at the client side, it is converted into a proxy by looking up the *ProxyById* table. Note *ProxyById* maps *ProxyId*s to proxies, but the one-to-one relationship between a *ProxyId* and a *ServiceId* will make sure the appropriate proxy is located.

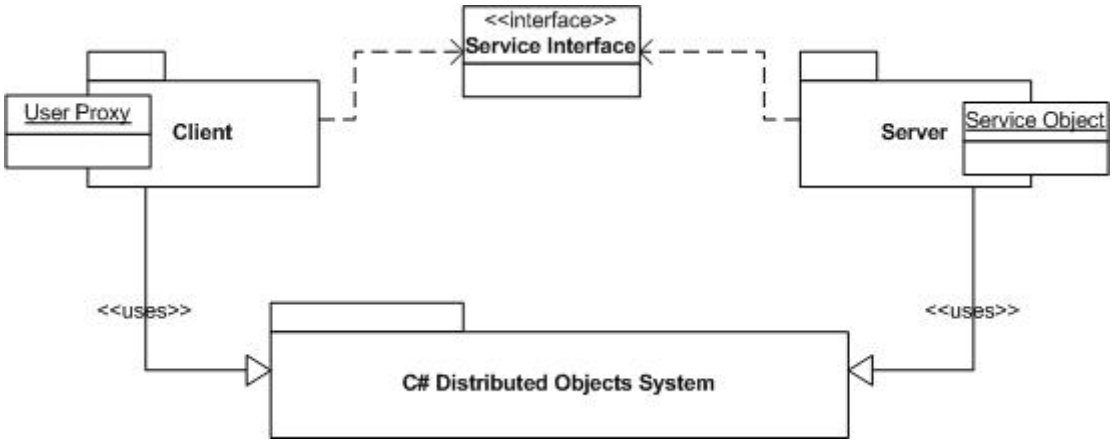
A server sends a service object to a client when the client first connects to the server. The service object is replaced by its *ServiceId*. When it arrives at the client side, the *Connection* finds out that it does not have any information on this *ServiceId* in its identification table. It realizes this is the first time such a service object has shown up, so it dynamically loads a proxy, and returns it to the client. From the client's viewpoint, it obtains a proxy from the server. So from now on, this process will simply be referred to as the server sending the client a proxy.

When a proxy object is sent out, its *ProxyId* is sent instead by retrieving the ID information from the proxy. After arriving at the service's side, it is converted into a service object by looking up the *ServiceById* table. Similarly, though *ServiceById* maps *ServiceId*s to services, the one-to-one relationship between a *ProxyId* and a *ServiceId* will make sure the appropriate service object is located.

The replacement of objects and their IDs occurs in the *Connection* layer and it is done by user-defined surrogates. See section 4.4 for technical information on surrogates.

3.2 Workflow Overview

The following diagram shows the structure of a distributed application using the C# Distributed Objects system.



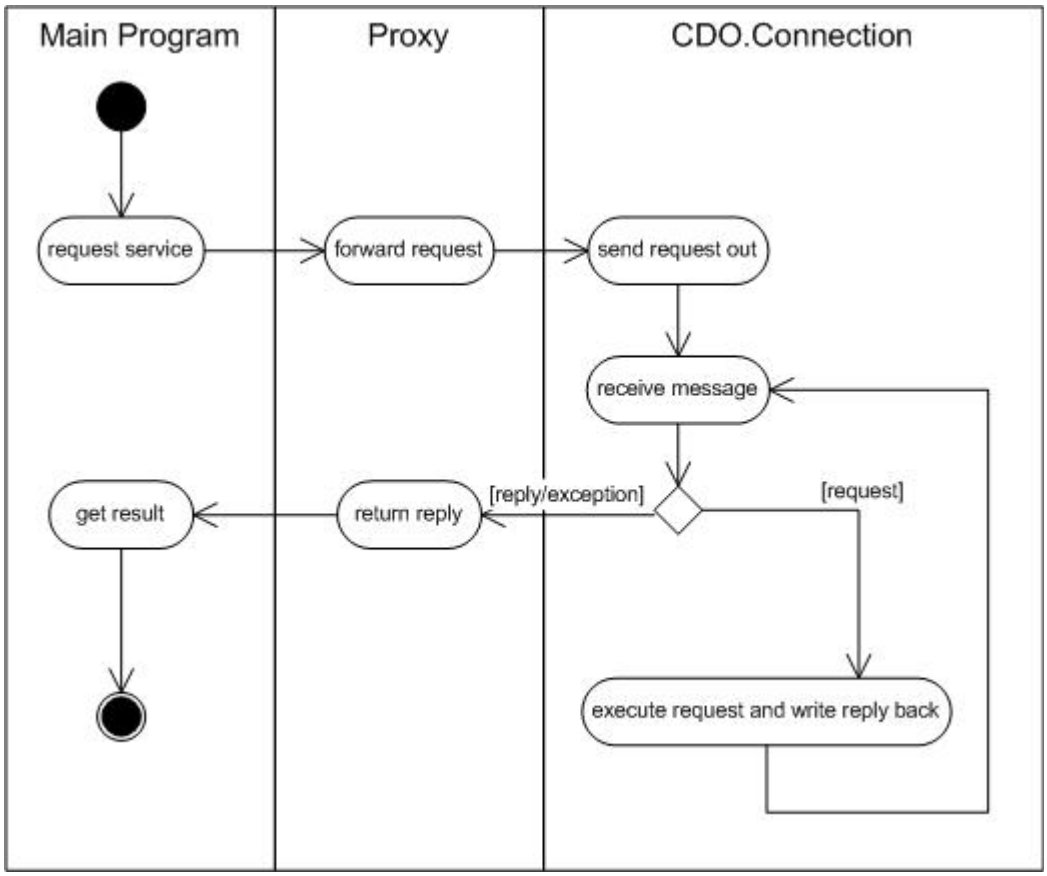
A service object on the server implements a specific interface and offers the service. The service object is represented by a proxy on the client side which implements the same interface. The client takes the proxy as if it were the service object. The client is aware of the interface and invokes the proxy according to the interface. If the user requests the remote service, the proxy sends a request to the C# D.O. system, and receives the result from it later. All the communication over the network is done by the C# D.O. system; the remote service object appears local to the client programmer.

The following sections will illustrate the In/Out flow and other details of CDO and ACDO.

3.2.1 CDO

CDO uses the traditional client/server structure. A client sends a request to a server, and gets the result from the server; the server monitors incoming packets and waits for the client's request passively.

CDO is essentially an implementation of NRPC (Nested Remote Procedure Call, see section 2.1.2). The following activity diagram shows the workflow of the client side of CDO. The structure of the server side is rather simple. For the workflow of the server side, please refer to the diagram in section 2.1.2.



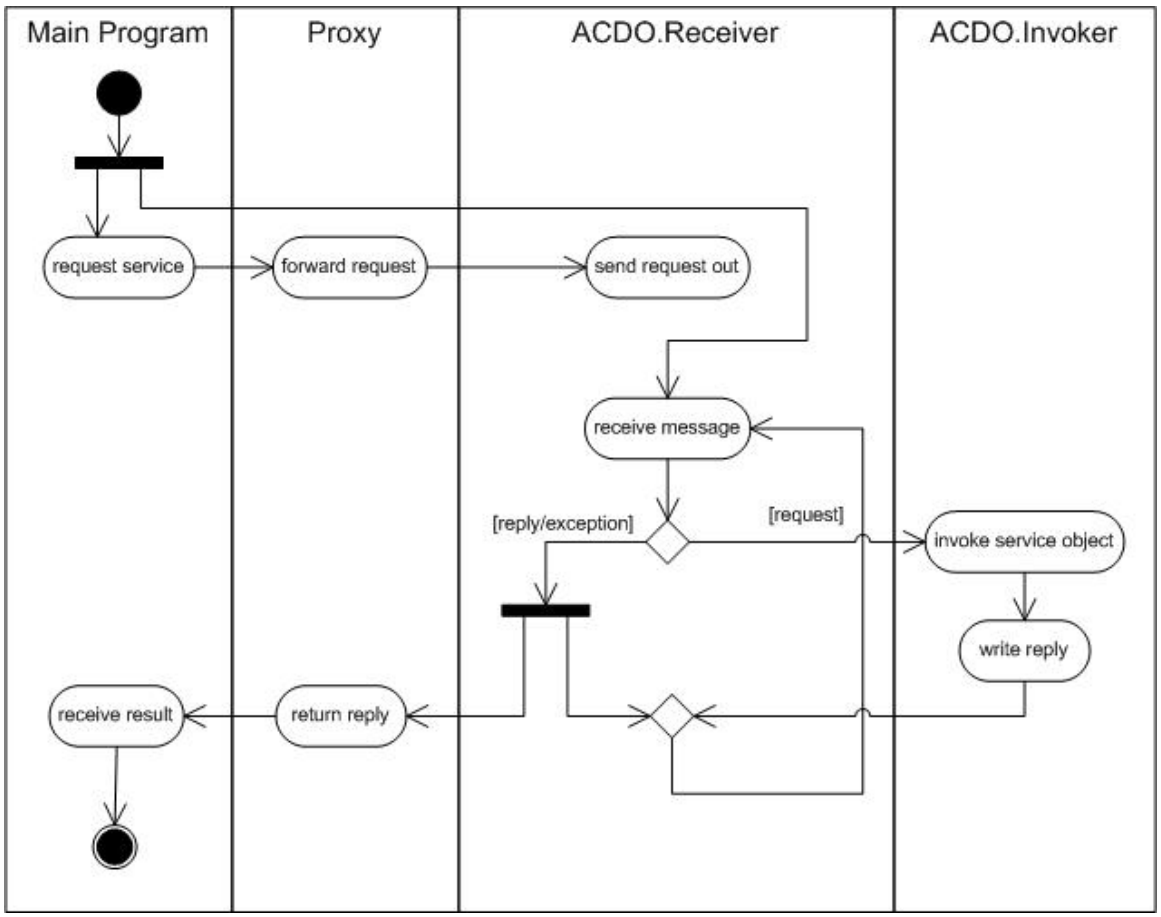
This shows that all communications between a client and a server are fully nested, or synchronous. The client does not expect to receive anything from the server unless it sends out a request first. In this case, a single client thread is enough. The main thread invokes CDO via function calls; results are sent back to the client as return values.

3.2.2 ACDO

CDO, described in the last section, should be able to handle normal distributed applications. However, it cannot support the chat room scenario described at the end of section 2.1.3, since conversations in a chat room application are asynchronous.

ACDO is the advanced form of CDO. It is the “complete” version of CDO. ACDO can handle everything in CDO; in addition, it meets the need of asynchronous communication.

ACDO is an implementation of distributed objects (see section 2.1.3). The following activity diagram shows the workflow of a peer that utilizes the ACDO system.



In ACDO, CDO.Connection is expanded into ACDO.Receiver and ACDO.Invoker with multithreading used. The main thread runs as usual. If access to a remote service is needed, the main thread makes a function call to ACDO, which then sends a request to the remote service object. The main thread waits until it receives notification that the reply has arrived. ACDO dedicates a thread called *receiver* to reading all the incoming packets from the network. If *receiver* receives a request, it invokes another thread to execute the request. A separate thread to handle the execution is implemented since we

do not want the execution to block the reading of any other incoming packet. If *receiver* receives a reply or an exception, it notifies the sender.

Since sending and receiving messages are in two threads in ACDO, there might be multiple requestors waiting for replies. It is necessary to set up a mechanism to pair off replies with requests. Requests and replies need some kind of identification, and there must be a way to match a reply with a request by their identifications. ACDO assigns each outgoing request a unique number, and the reply carries the same number as its requestor. When a reply arrives, its number is checked, and it will be sent to the request with the same number.

The number matching mechanism is also used in CDO. However, CDO only supports nested conversation, there is always only one request waiting for a reply. Thus, the number matching is just a safety check in CDO.

4 Analysis of Key Technologies

Based on the system structure analysis in chapter 3, several technologies are critical for implementing the system. How C# and .NET solve these requirements, if they do offer solutions, is essential to successfully porting the system from Java to C#. It also decides how the C# implementation of the system differs from the Java version. The following sections each focus on one of the technologies, explaining why it is required in the system, illustrating the possible solutions in C#, and comparing how it is done in Java to C#.

4.1 Interface

The goal of the C# Distributed Objects system is to hide the entire communication part of a distributed application, and make the remote service objects appear local to the clients. At the server side, the service object implements the interface and offers the service. At the client end, the only available information about the remote object is its service interface, which ensures that the client invokes the method of a service object in an appropriate way. A proxy resides at the client side, representing the server object. The proxy implements the same interface, thus the client can treat the proxy as if it were the service object; and the proxy is local.

Like Java, C# fully supports interfaces. An interface provides a specification rather than an implementation for its members. It is a promise that the classes that implement it will provide the required methods in the specified way.

Here is a simple interface that defines a single method:

```
public interface ITime
{
	string GetTime();
}
```

In C#, it is a common practice to put an *I* as the first letter of the interface identifier. *I* stands for interface, indicating it is an interface type.

The following code snippet shows how to implement an interface:

```
public class TimeService : ITime, IService
{
	public string GetTime ()
	{
		return DateTime.Now.ToString();
	}
}
```

A class can implement more than one interface. In the above example, class *TimeService* implements two interfaces, *ITime* and *IService*.

IService is a marking interface in the C# D.O. system for all service objects so that the user-defined surrogates can recognize them during deserialization. Surrogates will be explained in section 4.4. The alternative way to do this is to make a Service attribute, and tag each service object class with the customized attribute. Since Java only offers the

interface solution, interfaces are used in the C# D.O. system to keep analogous to JDO.

In general, C# supports interfaces in the same way that Java does.

4.2 Threads

Usually multithreading is desirable because of the architecture of a server. In general, multithreading increases the speed of a server's response and improves system efficiency by enabling a server to serve multiple requests in parallel. However, in principle, a server can serve clients in a sequential way, which blocks other clients while serving the current client.

The fundamental reason that multithreading is required in the C# D.O. system is the need for asynchronous communication. Synchronous communication means the conversation between the client and server is fully nested. A client sends out a request, waits for the response, and receives the response from the server. If the response asks for more information, the client sends out the required information and waits for the response. The conversation goes back and forth like this, until the client gets the final reply. The client expects all the incoming packets. The client will only receive information when it is waiting for it. However, in the chat room scenario described in section 2.1.3, a chat room client may receive packets at any time; a conversation between a chat room client and the chat room server may or may not be started by the client. This is called asynchronous communication, and an additional thread dedicated to monitoring the incoming messages is needed.

The Visual Studio .NET platform provides convenient classes and interfaces to enable multithreaded programming. With the help of the *System.Threading* namespace, users can create, manage and kill threads.

A process can create a new thread to execute a portion of the program code associated with the process. The constructor of the *Thread* class is as follows:

```
public Thread ( ThreadStart start );
```

Parameter *start* is a *ThreadStart* delegate that references the method to be invoked when this thread begins executing.

The following code snippet demonstrates how to create and run a new thread in C#:

```
using System;
using System.Threading;

class Tcp
{
	protected readonly TcpListener listener;

	public Tcp ( int port )
	{
		this.listener = new TcpListener ( port );
	}

	public void ListenerWork ( )
```

```
        {
                // do whatever the program wants to do
                …
        }

        public static void Main ( string[] args )
        {
                Tcp tcp = new Tcp ( Convert.ToInt32 ( args[0] ));
                Thread newTcp = new Thread (
                        new ThreadStart ( tcp.ListenerWork ));
                newTcp.Start( );
        }
}
```

In the *Main* function a new thread object *newTcp* is constructed by passing it a *ThreadStart* delegate that wraps the method that specifies where to start execution, which in this case is *ListenerWork()*. Then the new thread's *Start()* function gets called, and the thread begins to run.

Another important class associated with multithreaded programming is *System.Threading.Monitor*. The *Monitor* class provides tools that enable thread synchronization, that is, to coordinate multiple threads' access to shared resources. The *Monitor* class allows users to use any reference-type instance as a monitor. The *Enter* method obtains a lock on an object. If the object is unavailable at the moment, *Enter* waits until the lock is released. The *Exit* method releases a lock on an object. The *Wait* method allows a thread holding a lock to temporarily release the lock and block itself while waiting for another thread to notify it. The *Pulse* method allows a thread holding a lock to wake up a blocked thread as soon as it releases its lock. [Information gathered from 8]

The following table compares the usage of threads in Java and C#.

	Java [7]	**C#** [8]
To create a thread	Implements *Runable* interface or subclasses *Thread* class	*ThreadStart* (see above)
To synchronize threads	*synchronized* keyword *wait(), notify(), notifyAll()* of the *Object* class	*Monitor* class

4.3 Sockets

As mentioned in section 3.1, the *Connection* layer of the C# D.O. system is built on top of sockets, utilizing a TCP connection between the client and the server to do the lower-level communication work.

The *System.Net.Sockets* namespace contains a set of useful classes that provides TCP/UDP programming support. The C# D.O. system mainly uses the *TcpListener* and *TcpClient* classes. The server side program uses *TcpListener*. A *TcpListener* listens to incoming requests, creating *TcpClient* instances that respond to the connection requests.

A *TcpClient* connects to a remote host, hides the details of the underlying socket, and provides simple methods for sending and receiving data over a network.

The following sample code demonstrates how *TcpListener* and *TcpClient* work:

```
using System.Net.Socket;

TcpListener listener = new TcpListener ( portNumber );
TcpClient client = listener.AcceptTcpClient( );
```

An object *listener* of the *TcpListener* type is constructed by passing it the port number to which it is supposed to listen. Then, the *AcceptTcpClient()* function is called on *listener*. The *AcceptTcpClient()* function blocks the current thread until an incoming connection request is received; then a *TcpClient* type object is returned, with which users can send and receive data.

The client side code is simple:

```
TcpClient client = new TcpClient(host, port);
```

An object *client* of the *TcpClient* type is constructed and connected by passing it the server's address.

The following table compares the class names of TCP sockets in Java and C#.

	Java [7]	**C#** [8]
Server side	*ServerSocket* class	*TcpListener* class
Client side	*Socket* class	*TcpClient* class

4.4 Object Stream

A distributed objects system depends on the ability to transport objects over a network.

In Java, this is simple. Java has the built-in *ObjectInputStream* and *ObjectOutputStream*, which contain a *readObject* and a *writeObject* method, respectively. Objects implementing the *Serializable* interface or the *Externalizable* interface can be transmitted over a network by calling *writeObject* and *readObject*.

There is no object stream class in C#. C# uses serialization formatters to serialize objects over a network. The .NET framework offers three serialization formatters: binary formatter, SOAP formatter, and XML serializer. The XML serializer "serializes only public fields of an object and does not preserve type fidelity [9]". The binary formatter and the SOAP formatter provide the same functionality, but there are bugs in the SOAP formatter class. Therefore, the C# D.O. system uses binary formatters.

In C#, all objects that need to be serialized must be marked with the *Serializable* attribute. If an object implements the *ISerializable* interface, it can control its own serialization process; otherwise, the default serialization policy applies.

In Java, the *Serializable* interface is the counterpart of the C# *Serializable* attribute, and the *Externalizable* interface is the counterpart of the C# *ISerializable* interface.

The .NET framework also allows selective serialization by using the *NonSerialized* attribute, which comes in handy when the object to be transmitted contains some fields that should not be serialized. The counterpart in Java is the *transient* keyword.

As discussed in section 3.1, when serializing an outgoing message, any service object should be replaced by its *ServiceId*, and any proxy object should be replaced by its *ProxyId*. When deserializing an incoming message, any *ProxyId* should be replaced by a service object, and any *ServiceId* should be replaced by a proxy object. In C#, this object replacement is accomplished by implementing user-defined surrogates and a surrogate selector for C# formatters. In the Java version of the D.O. system, this is done by the object stream's *replaceObject()* method. The difference is that a Java object stream works on objects, while a C# formatter works on classes. A Java object stream captures each object and further decides whether an object should be replaced. A C# formatter considers an object for replacement because it belongs to a specific class.

The C# user-defined surrogate mechanism works as follows:

A customized surrogate is provided for each object class that needs to be replaced. A user-defined surrogate implements the *ISerializationSurrogate* interface, and contains two methods, namely, *GetObjectData* and *SetObjectData*. *GetObjectData()* populates the provided *SerializationInfo* with the data needed to serialize the object. *SetObjectData()* populates the object using the information from the provided *SerializationInfo*. Please refer to [8] for more information.

A user-defined surrogate selector assists formatters in the selection of the serialization surrogate to delegate the serialization or deserialization process [paraphrased from 8]. In order to serialize/deserialize a message, a surrogate selector is created, user-defined surrogates are added to the selector, a binary formatter is created based on the selector, and the *Serialize()/Deserialize()* method of the formatter is invoked.

In the C# Distributed Objects system, two user-defined surrogates and a surrogate selector are implemented. The two surrogates are *ProxyToService* and *ServiceToProxy*.

Take the *ProxyToService* as an example:

```
class ProxyToService: ISerializationSurrogate
{
	// turns proxy into ProxyId
	public void GetObjectData(object obj,
		SerializationInfo info StreamingContext context)
	{
		info.SetType(typeof(WireProxyId));

		//Proxy.Replace() returns the proxy's ProxyId
		info.AddValue("id", ((Proxy)obj).Replace(
			(Connection)context.Context));
	}

	// turns ProxyId into service
	public object SetObjectData(object obj, SerializationInfo info,
		StreamingContext context, ISurrogateSelector selector)
```

```
        {
                ProxyId id = (ProxyId)info.GetValue("id", typeof(ProxyId));

                //ProxyId.Resolve() searches the ServiceById table
                //and returns a service object
                object result = id.Resolve((Connection)context.Context);

                if (result == null)
                        throw new ApplicationException(
                                "No Service Object available" + id);
                else return result;
        }
}
```

Note that in *GetObjectData()*, theoretically the type set in *info.SetType()* should be *ProxyId*. However, a *ProxyId* object itself is supposed to be further serialized. It will be a problem for the recipient to retrieve the desired information. To avoid this, an auxiliary class *WireProxyId* is provided. *WireProxyId* is an empty class, only its type is needed, and it only exists during the serialization process.

ServiceToProxy is similar to *ProxyToService* in nature. A *WireServiceId* is provided for *ServiceToProxy*.

The following code snippet shows how to use a user-defined surrogate selector.

```
StreamingContext context = new StreamingContext(
        StreamingContextStates.All, this);

SurrogateSelector selector = new SurrogateSelector();

selector.Add(typeof(IService), new ServiceToProxy());

selector.Add(typeof(Proxy), new ProxyToService());

BinaryFormatter s = new BinaryFormatter(selector, context);

s.Serialize(outgoing, message);
```

The deserialization process is similar.

Note this mechanism only works correctly in .NET framework 1.1; there are bugs in the serialization library classes in .NET framework 1.0.

A Java object stream remembers objects across calls. “Within an *ObjectOutputStream* the first reference to any object results in the object being serialized and the assignment of a handle. Subsequent references to that object record only the handle [10].” A C# formatter does not record such information across calls.

This makes a big difference when replacing objects. For instance, when replacing a service object with its *ServiceId*, the Java version of the Distributed Objects system looks up the service object in the *IdByService* table only once, because after the first round the streams have learned and remembered the mapping. The C# version of the Distributed Objects system must look up the service object whenever it needs to be replaced.

When replacing a *ServiceId* with a proxy, it is important to map a *ServiceId* always to the same proxy. In the Java version of the Distributed Objects system, at the first time a

ServiceId is shipped to the client side, the object stream loads a proxy to replace it. After that, the object stream remembers the mapping, and automatically replaces this *ServiceId* with the same proxy when the *ServiceId* comes again. However, since a C# formatter does not remember such information across calls, to avoid loading a new proxy every time a *ServiceId* comes, the C# Distributed Objects system must be programmed to look up the *ProxyById* table to decide whether to load a new proxy or use an existing proxy.

The following table compares a Java object stream with a C# formatter.

	Java object stream [7]	**C# formatter** [8]
Default serialization	*Serializable* interface	*Serializable* attribute
Self-controlled serialization	*Externalizable* interface	*ISerializable* interface
Selective serialization	*transient* keyword	*NonSerialized* attribute
To replace objects	*replaceObject()*	User-defined surrogates
Sensitive level	Object-sensitive	Class-sensitive
Information kept across calls	Remembers objects across calls	Does not remember objects across calls

4.5 Dynamic Class Loading

As discussed in section 3.1, a remote service object is represented by a proxy at the client side. Both the proxy and the service object implement the same interface, so that a client can use the proxy as if it were the service object and invoke the method on it.

A client should have the class code of the interface at compilation; otherwise, it could not invoke the method. A client should be able to load a proxy in its own memory space at runtime when it acquires an object reference of the proxy. To load a proxy, the client needs the proxy class file. However, a proxy is supposed to be invisible to a client, and the client is not responsible for the implementation of the proxy class. It would be greatly convenient and flexible if the delivery of a proxy class file could be delayed until runtime. For example, the proxy class code "resides on the server's host (or perhaps another location), and can be downloaded to the client on demand [11]".

Java Virtual Machines have built-in class loaders that are responsible for loading system classes, installed extension classes, and classes on the class path [12]. In addition, Java allows user-defined class loaders to be used in place of the built-in class loader. A user-defined class loader subclasses the *ClassLoader* class and can load classes from some alternate source, such as the Internet.

In the Java version of the Distributed Objects system, a customized class loader, which loads classes from a URL address, is inserted into the object input stream. This enables sending proxy class files to the client at runtime on demand.

.NET brings up the concept of assemblies. "An assembly is a logical package that consists of a manifest of metadata, one or more modules (which essentially are portable executable files), and an optional set of resources, such as a bit map file used by the program [13, page 162]." The *CreateInstance()* method of the *Assembly* class "creates an

instance of a type defined in an assembly by invoking the constructor that best matches the specified arguments. If no arguments are specified, then the constructor that takes no parameters (the default constructor) is invoked [paraphrased from 8]".

The following approach is used to implement dynamic class loading in the C# D.O. system. First, the assembly that contains the desired proxy class is loaded.

```
Assembly assemb = Assembly.LoadFrom ("UserProxy");
```

The static method *LoadFrom* of the *Assembly* class "loads an assembly given its file name or path [8]". The assembly can reside on the local machine or on a network.

It is necessary to create a naming mechanism of the assembly that contains the proxy class. A simple way to do this is to keep each proxy in a separate assembly, and derive the assembly name from the proxy name, which in turn is derived from the service interface name. In the sample applications of this project, all proxies are kept in an assembly named *UserProxy* just for convenience.

Then, the *CreateInstance* method of *Assembly* is called. It locates the specified class from this assembly and creates an instance of it. In the case of the C# D.O. system, the specified class would be the proxy class.

The signature of the *CreateInstance* method is:

```
[Serializable]
[ClassInterface (ClassInterfaceType.AutoDual)]
public object CreateInstance (
       string typeName,
       bool ignoreCase,
       BindingFlags bindingAttr,
       Binder binder,
       object[] args,
       CultureInfo culture,
       Object[] activationAttributes
)
```

typeName is the desired proxy class name. *ignoreCase* indicates if the *typeName* is case sensitive. *args* is an array containing the arguments to be passed to the class constructor. This array of arguments must match in number, order, and type the parameters of the constructor to be invoked. If the default constructor is desired, *args* must be an empty array or a null reference; other arguments are set to null or a default value [paraphrased from 8].

A Java class implicitly loads everything else relative to its own class loader. When loading a proxy, if the proxy refers to other as yet unknown classes, those classes will be implicitly loaded by the proxy's class loader too. When a user-defined class loader loads a class, by convention, it first asks its parent class loader. If the parent class loader cannot handle the job, the child class loader will try to load the class itself. This ensures the system classes are correctly loaded by the default built-in class loaders. However, C# cannot load classes implicitly.

Dynamically loading classes can result in security problems. Java can solve this by the security manager. "The security manager regulates access to sensitive functions, and the class loader makes sure that loaded classes are subject to the security manager and

adhere to the standard Java safety guarantees [11].” .NET can solve this by using security permissions (see [9] for more information). However, the security aspects are not considered in JDO or the C# D.O. system.

4.6 Identity Hash Table

As discussed in section 3.1, the *Connection* layer contains three identification tables to store all the information needed in object replacement. An abstract class *Registry* containing a hash table serves as the base class. The three identification tables, *ProxyById*, *ServiceById* and *IdByService*, derive from *Registry* and use the inherited hash table to record the mapping information.

A hash table checks key identities by calling *Equals()* on the key. However, users can override a service object’s *Equals()* method which is inherited from the *Object* class. It is possible that two service objects redefine *Equals()*, and are considered equal by a hash table. Thus, a regular hash table is not suitable for the *IdByService* table.

Java 1.4 provides an *IdentityHashMap* which can avoid the above issue. .NET does not provide a container class based on identity. In the C# D.O. system, an *OpenHashtable* class that extends *Hashtable* and implements real identity checking is created to solve this potential problem. Furthermore, C# allows operator overloading. Thus, it is inappropriate to check identity by ==. Instead, the static method *ReferenceEquals()* from the *Object* class is used to determine if two objects are the same instance.

5 Usage and Sample Applications

In order to test the C# implementation of the system, three sample applications have been ported. The first two of them use CDO, the third uses ACDO. The executions of the sample applications demonstrate that the C# Distributed Objects system is functional.

To utilize the C# Distributed Objects system:

- The server side program should provide its implementation of the service object.
- Both the client and the server side program should have a copy of the interface of the service.
- An interface-specific proxy for the service object should be generated according to the service interface and referenced by both sides.

Since .NET requires the sender and receiver of the serialization/deserialization process to have the identical assembly file which contain the object that is transmitted over a network, the user proxy is compiled into a separate assembly named *UserProxy* that is referenced by both the server and client programs. Similarly, the service interface should be compiled into a DLL and referenced by both sides.

The server program calls the C# D.O. system to create a socket by indicating the desired port number. A thread is started to monitor the socket and sends a proxy to the client when a connection request is received.

The client program calls the C# D.O. system to make a TCP connection to a server. This call returns a proxy, and then the proxy can be used to communicate with the service.

5.1 Time Service

The Time service example is essentially a RPC (see section 2.1.1) application and utilizes CDO.

The current time can be obtained from a server. The service interface is listed below:

```
public interface ITime
{
	string GetTime();
}
```

The interface is compiled into *CDOTime.DLL*, and is referenced by both the client and the server program.

The corresponding *TimeProxy* implements the *ITime* interface. *TimeProxy* is compiled into *UserProxy.DLL*, and is referenced by both sides as well.

The service object resides on the server, providing the Time service.

```
[Serializable]
public class TimeService : ITime, IService
{
	public string GetTime()
	{
```

```
                return DateTime.Now.ToString();
        }
}
```

To run the server side program, a user needs to indicate a port number. The port number is passed to CDO, which creates a socket and starts a thread to listen to the socket.

Once a client connection request is received, another thread is activated. The new thread sends a time proxy to the client and continues to take care of future communications with this specific client.

To run the client side program, a user needs to indicate the service address. The client program passes the service address to CDO, which sends a request to the indicated address, builds a connection, and receives a time proxy from the server.

Now the client obtains the proxy, and can treat it as if it were the service object. The following line of code shows how a client invokes the proxy *time*'s *GetTime()* method.

```
System.Console.WriteLine(time.GetTime());
```

The proxy *time* then sends a *Request* to the server. The server invokes the *GetTime()* method of the time service object, and writes the result in a *Reply* back to the proxy. The proxy retrieves the current time, and passes it to the client program.

5.2 CPU Service

The CPU service example is essentially a NRPC (see section 2.1.2) application and utilizes CDO.

There are two service objects: the CPU service object that resides on the server side, and the Term service object that resides on the client side.

The CPU service is a simple calculator chip that retrieves integer values and operators from a vector that it fetches from its argument. The CPU interface is as follows:

```
public interface ICpu
{
        /// <summary>
        /// Retrieves Integer array from 'terms' and computes
        /// and stores a[0], a[1], a[2]...
        /// where the a[odd] must be + - *% /
        /// Various exceptions can happen
        /// </summary>
        ICpu Cpu(ITerms terms);

        /// <summary>
        /// Lets cpu deliver result of previous computation
        /// </summary>
        int GetResult(ICpu cpu);
}
```

The Term service simply returns a vector that holds an arithmetic expression.

```
public interface ITerms
```

```
{
        /// <summary>
        /// Provided by the Cpu client
        /// </summary>
        int[] Terms();
}
```

Similar to the Time service example, two proxies are provided here, namely, *TermsProxy* and *CpuProxy*. Each proxy implements a service interface and contains method-specific request and reply inner classes. Note that if an interface contains more than one method, for example *ICpu*, each method will be provided its own method-specific request and reply.

The following paragraph describes a conversation between a CPU client and a CPU server. The communication process may seem redundant and inefficient, but the purpose here is to create more traffic between a client and a server, and test CDO.

The server program starts with calling CDO to create the socket based on the designated port number. The client program builds a connection to the server through CDO and obtains a *CpuProxy*. Invoked by the client, the *CpuProxy* sends a request to the remote service object, passing a Terms service object as the argument. The server side CDO receives the request. When deserializing the request, it finds out that it is the first time such a Terms service object has showed up, so it dynamically loads a *TermsProxy*. CDO invokes the CPU service object, which in turn invokes the *TermsProxy* to obtain an integer vector. Similarly, the *TermsProxy* sends a request back to the client end, and returns a reply to the server. The server program gets the required information and continues to calculate. The result will be kept in the CPU service object. The client calls the *GetResult()* method of the *CpuProxy*. After another round of request/reply communication, the client receives the result.

5.3 Chat Room Service

The chat room example is a peer to peer application (see section 2.1.3). Participating clients can join a chat room and send messages to the chat room server. A chat room server keeps track of all participants and sends received messages to each of them. In this scenario, a client may receive incoming messages from a server at any time; both the client and the server can initialize a conversation. Thus, CDO is not sufficient in this situation; instead, ACDO is used.

Although the system structures of CDO and ACDO are quite different, the ways a client programmer uses them are the same. The client programmer just needs to include the appropriate DLL files in a project; the middle tier is completely transparent.

Two services are presented. The client program promises the Chat service, which receives messages from all other participants. The service object *Chatter* implements the *IChat* interface.

```
public interface IChat
{
        /// <summary>
        /// Informs clients the new message
        /// </summary>
```

```
        void Tell(string text);
}
```

The server program provides Room service. The service object *ChatRoomService* implements the *IRoom* interface.

```
public interface IRoom : IChat
{
        /// <summary>
        /// Allows a chatter to join the chat room
        /// </summary>
        void Join(IChat withMe);
}
```

Note that *IRoom* extends *IChat*. The *Join()* method allows a new client to join the chat room. The *Tell()* method, which is inherited from *IChat*, sends a message to all *Chatters*.

RoomProxy and *ChatProxy* are provided as usual.

The following graphical interface interacts with the client user.

The client's main program connects to the server and obtains a *RoomProxy*. Then, it constructs a *Chatter*, calls *RoomProxy.Join()*, and passes the *Chatter* as the argument. The *Chatter* will be turned into a *ChatProxy* when it reaches the server side. When the user enters a new message, *RoomProxy*'s *Tell*() is invoked to send the message to the server. Incoming messages are displayed in the rich text box by *Chatter*'s *Tell*() method.

The server starts with constructing a *ChatRoomService*, which keeps an array of *Person*s. A *Person* is an agent for a *Chatter* on the server, which contains a reference to a *ChatProxy* associated with the *Chatter*. If a new *Chatter* requests to join, the *ChatRoomService* obtains the *ChatProxy* and creates a *Person* for it. A *Person* runs in its own thread, passing messages from the *ChatRoomService* to its *Chatter*. When a *Chatter* sends a message out, the message is sent to the *Chatter*'s *Person*, and this *Person* will attach the new message to all *Persons* in the *ChatRoomService*, including itself. Each *Person* thread reads its own message array-list and invokes the *Tell()* method of its *ChatProxy*. The *ChatProxy* in turn invokes the *Chatter* at the client side to display the new message in the user interface.

6 Summary

The goal of this project is to port JDO to C# and the .NET platform and gain insight into the differences between Java and C# in the distributing technology field.

A distributed objects system depends on the ability to transport objects over a network. In addition, the system design of the Distributed Objects system requires objects replacement during transportation. To achieve this, Java uses object streams while C# uses formatters.

A Java object stream works on the object level. If *enableReplaceObject* is set to true, the *replaceObject()* method of the *ObjectOutputStream* class catches each object, and one can check whether an object should be replaced and implement the replacement in this method.

A C# formatter works on the class level. The classes of objects that need to be replaced are assigned to user-defined surrogates. The type of an object is checked during serialization/deserialization to decide whether it should be replaced.

Furthermore, a Java object stream remembers objects across calls while a C# formatter does not. It is important to always map a service object to the same proxy. Java object streams make this easy. At the first time a *ServiceId* is shipped to the client side, the object stream loads a proxy to replace it. After that, the object stream remembers the mapping, and automatically replaces this *ServiceId* with the same proxy when the *ServiceId* comes again. However, since a C# formatter does not remember objects across calls, to avoid loading a new proxy every time a *ServiceId* comes, the C# Distributed Objects system must be programmed to look up the *ProxyById* table to decide whether to load a new proxy or use an existing proxy.

There are bugs in the serialization library classes in .NET framework 1.0. The surrogate mechanism only works correctly in .NET framework 1.1.

Since C# serialization surrogates recognize classes, objects that need to be replaced should be identified. Java uses interfaces to mark classes. C# offers two options: attributes and interfaces. Interfaces are inherited, i.e. if a class implements an interface, all its subclasses are of that interface type. Attributes are not automatically inherited. Thus, using customized attributes seems to be a safer choice. However, mark up interfaces are adopted in the C# D.O. system just to keep analogous to the JDO version.

Note that the use of mark up interfaces results in decreasing the system transparency, since every service object has to implement the interface. An alternative way to avoid this is to implement a user-defined surrogate that recognizes *Object* and an array that records all objects that may need to be replaced. The surrogate captures each *Object* and checks if it is in the array to decide whether to replace it. However, considering the large overhead of this method, the mark up interfaces seems to be a better design.

.NET requires the sender and receiver of the serialization/deserialization process to have identical assembly files which contain the object that is transmitted over a network. Classes that will be used by both the client program and the server program should be

compiled into an assembly and be referenced by both sides.

In Java, socket streams for input and output are two separate streams; in C#, the *GetStream()* method of a socket returns a *NetStream* object on which both the input and output actions are performed. However, the C# D.O. system still references the incoming and outgoing streams as two separate streams to keep the architecture clear and easy to understand.

Dynamic class loading can delay the delivery of proxy classes until runtime, which greatly increases the system flexibility. “A JVM has a built-in ClassLoader to load the system classes. If one wants to control how other classes are loaded, different ClassLoader objects can be used or even a new ClassLoader can be implemented [1].” C# loads an assembly at runtime by calling *Assembly.LoadFrom()*. Then, the *CreateInstance()* method of *Assembly* creates an instance of a class defined in this assembly.

Java allows setting properties on the command line, which can be used for configuration. Name-and-value pairs can be set from the command line in the “-Dname=value” format. A Java program can retrieve a property value by calling the *System.getProperty()* method, or methods such as *Boolean.getBoolean()*. Please refer to [7] for more information on the above methods. Setting properties from the command line is very helpful when configuration values can only be set at runtime. C# provides configuration files but does not allow setting values on the command line.

Hash tables are used for managing identifications. A hash-table checks key identities by calling *Equals()* on the key. However, users can override a service object’s *Equals()* method which is inherited from the *Object* class. It is possible that two service objects redefine *Equals()*, and are considered equal by a hash table. Java 1.4 provides an *IdentityHashMap* which stores and retrieves key-value pairs based on identity. C# has no counterpart for this. An *OpenHashtable* class derived from *Hashtable*, which implements identity checking of keys, is created for *IdByService* in the C# D.O. system.

In summary, JDO can be ported to C# but it does require some revision of the basic strategies due to different toolsets on the platforms.

7 References

[1] Schreiner, A. Distributed Objects class notes,
http://www.cs.rit.edu/~ats/do-2001-1/home.html

[2] Raj, G. *A Detailed Comparison of CORBA, DCOM and Java/RMI*,
http://gsraj.tripod.com/misc/compare.html

[3] Carnegie Mellon Software Engineering Institute, Software Technology Roadmap
http://www.sei.cmu.edu/str/descriptions/rpc.html

[4] RFC 1050 - RPC: Remote Procedure Call Protocol specification
http://www.faqs.org/rfcs/rfc1050.html

[5] .NET framework SDK documentation -- .NET remoting architecture
http://msdn.microsoft.com/library/default.asp

[6] Deitel H., Deitel, P., Listfield, J., Nieto, T., Yaeger, C., Zlatkina, M. 2002. C# How to Program. Upper Saddle River, NJ: Prentice Hall.

[7] Java 2 Platform API
http://java.sun.com/j2se/1.4.2/docs/api/

[8] .NET framework C# class library
http://msdn.microsoft.com/library/default.asp

[9] MSDN, .NET Framework Developer's Guide

[10] Riggs, R., Waldo, J., Wollrath, A., and Bharat, K., "*Pickling State in the Java System*," Computing Systems, 9(4), pp. 313-329, Fall 1996.

[11] Wollrath, A., Riggs, R., and Waldo, J., "*A Distributed Object Model for the Java System*," USENIX Computing Systems, vol. 9, November/December 1996.

[12] Kaminsky, A., Java Class Loading and Class Loaders
http://www.cs.rit.edu/~ark/lectures/cl/05_01_00.html

[13] Drayton, P., Albahari, B., Neward, T. 2002. C# in a Nutshell: a Desktop Quick Reference. Sebastopol, CA: O'Reilly.