Master’s Project Proposal: Distributed Objects in C#

Xiaoyun(Sherry) Xie

February 2003
ABSTRACT

With the continuing advancement of network technologies, our information society has entered such an era that “network is computer”. More and more programs run over a collection of autonomous computers linked by a network and are designed to produce an integrated computing facility. Concerned on supporting the remoting and communication needs between client and server programs or between distributed peers, different companies have offered different solutions, such as JavaSoft’s Java Remote Method Invocation (JAVA/RMI), Microsoft’s Distributed Component Object Model (DCOM), and OMG’s Common Object Request Broker Architecture (CORBA). Those technologies have their own special features, and of course, pros and cons.

This project is focused on implementing a mechanism based on Dr. Axel T. Schreiner’s idea of Java Distributed Objects (JDO), which builds an infrastructure that allows the distributed program components to communicate over a network in a reliable, efficient and generic way. The goal of the mechanism is to hide the distributed nature of remote objects, i.e. the distributing and remoting part shall be transparent to clients. This infrastructure utilizes proxies to represent remote services, and can serve multiple objects in a parallel way. Dr. Schreiner’s JDO was originally intended as a teaching device to assess design parameters for distributed objects. In particular, this project will port the above infrastructure to C# and shall generate insights into implementing remoting on the .NET platform.
1 Overview

Nowadays network programming plays such an important role in application development that it is even hard to find an integrated program all running in a single process on a single machine. The type of program, consisting of complex components running in processes across a network, is very common today. Responsibility distribution has become the trend. In the so-called client/server model, client and server programs run on either end of a networked system. A client program sends requests and receives answers from the server; the server program which receives the requests sends out replies after performing associated computing tasks. Since client and server programs typically run on different computers (this kind of client/server application is defined as distributed application), objects need to be able to talk with one another over the network.

There are various answers to this need from different companies, the three major and popular ones are JAVA/Remote Method Invocation (JAVA/RMI) from JavaSoft, Distributed Component Object Model (DCOM) from Microsoft, and Common Object Requests Broker Architecture (CORBA) from OMG. They all aim to extend an object-oriented programming system by distributing objects to different processes and hosts. Note that neither of those technologies is transparent to users; clients more or less know about the remote nature of the services. User transparency is an important parameter in software design, it could simplify the end user’s programming work, make user applications more organized and easier to maintain. The purpose of this project is to implement a distribution pattern that is totally transparent to end users. Furthermore, in a traditional Client/Server program, only client sends requests to server, normally it’s not possible for a server to question a client. Though in many situations, a server may wish to request further information or help from the client. In the Distributed Objects system, clients and servers are enabled to talk back, i.e. the server and client can exchange roles freely. We don’t expect to replace the existing strong remoting technologies by such a small-scale project, but we would like to discover aspects and options of Distributed Objects system design and communication programming on .NET.
2 Project Specification and System Architecture

2.1 System Architecture Overview

The above diagram shows the architecture and the control/data flow of the Distributed Objects system. The client and the server reside on different end systems connected by computer networks. Objects on the server offer services, and are represented by proxies on the client side. The client is aware of the service object’s interface, which ensures that the methods of the remote object are invoked in a correct way. Each service object has a corresponding proxy on the client side, and the client invokes the object through its proxy. The Connection is essentially the Transport-Layer of the system, it keeps tables of identifications, such as the ServiceID, which is a unique id that identifies each service object, and the ProxyID, which distinguishes each proxy. When the client wants to access a remote service, it invokes the specific proxy -- a method of the proxy sends out a request to the corresponding service object through the Connection, and returns the received information to the client as the return value. When the Connection sends out a request, each object inside the request is replaced by its id. At the server end, the Connection receives requests, and each id in a request is replaced by the corresponding object. A request contains the requesting information plus the requester itself, i.e. the proxy. The one-on-one relationship between a proxy and a service object makes sure the appropriate service object receives the request. The reply or request for further information to the original request also contains this requester information, which makes sure the original requester gets the feedback.
2.2 Major Functional Modules of the System

2.2.1 Message

All information passed over a Connection takes the form of Message. Message is the common interface, and different messages are distinguished by their classes. Two abstract classes, Request and Reply, implementing the Message interface, are the roots of all other method-specific message classes. The following diagram shows the class hierarchy.

![Class Hierarchy Diagram]

To transport messages over the network, we use the BinaryFormatter in C# to serialize message objects into the network stream, and deserialize them at the receiver end.

2.2.2 Identification

As a Connection can serve multiple proxies and service objects, a unique identification for each proxy and service object is necessary. We create Tag as the fundamental class which provides object containing a unique number. Tag has a static integer data member uniqueTag, which increases by 1 when the constructor Tag() is called. This makes sure every object gets a unique number as its identification. Tag also overrides equals() and hashcode() methods, so that two copies of Tag objects (even if they belong to different subclasses of Tag) will be considered the same as long as they contain the same number.

Abstract class Id which extends Tag is the common base class of ProxyId class and ServiceId class. Id has the abstract method Resolve() which promises the subclass objects can be resolved into the objects they identify.

Finally, ProxyId class and ServiceId class represent proxy object and service
object respectively. The following diagram shows the class hierarchy of all the identification classes.

```
Tag
+uniqueTag : long
+tag : long
+EQuals() : bool
+HashCode() : int

Id
+Resolve() : object

ProxyId

ServiceId
```

When a service is prepared to be exported, a ServiceId is created and entries are made in two tables, serviceById, which maps ids to service objects, and idByService, which maps service objects to their ids. When a proxy is first created for a service object, a ProxyId is created from the ServiceId (the two actually have the same unique number) and stored in the proxy itself.

2.2.3 Connection

A Proxy references the Connection it belongs to, which in turn references input and output streams and identification tables. The Connection is the transport layer of the system, which is built up on the TCP socket. The input and output streams are represented by In and Out classes respectively. All the information transferred over the Connection takes the form of Message (as we discussed in 2.3.1), thus the Connection defines Read( ) and Write( ) methods in terms of messages, and both methods take the actual read and write actions on the In and Out stream. The Out class wraps around the output network stream, and replaces each object contained in outgoing messages with its unique id; the In class wraps around the input network stream, and replaces each id with its corresponding object. Note in C#, unlike Java, there are no separate input and output streams of sockets, one uses GetStream( ) method on a socket and gets the network stream on which both read and write operations are performed. So in the implementation of the system in C#, the In and Out classes would actually refer to the same stream. Still, we’d like to keep those two classes separated for the purpose of clearer system architecture.

The Connection contains several tables, proxyById, serviceById, idByService.
They all are subclasses of Registry, which is built up on a Hashtable. As suggested by
the self-explaining names, proxyById maps ids to proxies, serviceById maps ids to
service objects, and idByService maps service objects to ids. When a proxy or service
object is ready to be used, it registers itself with the Connection by putting entries in
appropriate tables. And the Connection looks up in those tables when needed (such as
the replacement of ids and objects as we mentioned above).

2.3 Key technologies

There are 5 critical technologies required to implement the Distributed Objects system,
which are discussed next:

Threads – required by server architecture, enable server to serve multiple requests in
parallel.
C# provides convenient classes and interfaces to enable multithreaded programming.
With the help of System.Threading namespace, users can create, manage and kill
threads.

Sockets – required by transmission.
C# offers different levels of networking support. High-level access is supported by a
set of library classes that implement generic request/reply architecture. If the
application requires lower-level access to the network, C# library provides classes
dealing with the TCP and UDP. And raw socket access is offered if direct
transport-level access is desired. In this project, middle level networking classes are
used.

Object stream – required by the transport layer.
In C#, a network stream can be extracted from a socket, on which the read and write
operations are performed. There’s no object-stream in C# as in Java, instead one could
serialize an in-memory object into binary format and send that binary stream over the
network.

Interface – required by client proxy, which resides at the client end, representing the
service object.
At the client end, the only available information about the remote object is its
interface, which ensures the client invokes the method of service object appropriately.
C# supports interface which provides a specification rather than an implementation
for its members. It’s a promise that the classes that implement it will provide the
required methods in the specific way.

Class loader – required by dynamic loading and executing classes at runtime.
.NET has no class loader, instead it brings up the concept of assembly, which 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 resource, such as a bit map file used by the program. C# has Assembly Resolver, which locates an assembly, and Assembly Loader, which loads an assembly from a specific location. The two together might fulfill the dynamic loading need of this project.

3 Principal Deliverables

This project will have the following items as the principal deliverables:

- A technical report written in LaTex, about 30-40 pages, which describes the working principle, system architecture and functional modules in detail.
- The source code of C# program implementing the Distributed Objects system.
- The C# source code of sample client and server programs which are used to demonstrate how the system works.

4 References


5 Approximate Schedule

- February 2003 – Complete proposal and approval process.
- April 2003 – Complete program coding.
• May 2003 – Complete report and project defense.