The objective of this lab is to become familiar with network programming using the .NET Remoting API, which is representative of other Distributed Objects APIs as well.

In this lab, you will build the client and server components of a chat application. Your chat server will act as a broker that connects up the two clients interested in chatting with each other. Once this connection has been established, all chat interactions take place directly between the clients, without involving the server. The interaction between a client and the server, and between two clients is accomplished using remote object invocations supported by the .NET Remoting API. You will only need to use the subset of the API that was discussed in Lectures 8-10.

Preliminaries:

The discussion below refers to the starter files and sample .NET remoting application that I have made available as a zipped-folder Lab3.zip on each of the development servers at the following location:

D:\VSDev\Public\vijayk\Lab3.zip

This folder expands into a Visual Studio.NET solution, which contains six projects: Remotable-Client, Remotable-Server, Client, Client-ObjRef, and Server contain the files for the sample application discussed in class. The ChatClient project contains the designs of the required Windows forms and examples of some GUI code, which should hopefully be all you will need to learn for completing this lab. Note that the controls on these forms have not been completely configured: you will need to do whatever is required.

Unlike Labs 1 and 2, where the interfaces between the client and server components of the distributed application were relatively concretely specified, this lab offers you substantial leeway in orchestrating the interactions. The only requirement is that the following sequence of operations be supported (which therefore serves as an operational specification):

1. The chat server, a console application, should start up by reading a list of user IDs from a file. While a full-fledged chat application would require a mechanism for new users to create accounts, in this lab, you should assume that the only users that will need to use the chat facility are pre-registered with the server.

2. The chat client, a Windows Forms-based application, should start up as demonstrated in the supplied ChatClient project. Specifically, a Debug form is created, which in turn pops up a Main Window. The other forms that are created by the supplied code are for illustration purposes only, and your final lab should follow the sequence of steps described below.

3. The Main Window offers an option for the user to Sign In by supplying an ID. This should be one of the IDs that the server is aware of. Clicking on the Sign In button should result in an interaction with the server program to (a) verify that the ID being used is in fact valid; (b) register one or more callback objects with the server that will be used as described in Steps 4 and 5 below; and (c) receive a list of known user IDs with their current sign-in status. Each user is either signed-in or not. [Again, a full-fledged chat application would maintain a contact-list per user and allow multiple visibility choices]. The list of user IDs should be rendered at the client using the DataGrid control in the Main Window form. See the Lab 2 code for help on setting up and using the DataGrid control.

Once the user has signed in, the Sign Out button should be enabled. Clicking on it should result in an interaction with the server program to sign the user out.
4. Whenever a user’s sign-in status changes (either because the user has signed in or out), the server should invoke a callback object for all previously signed-in users to inform them about this change. This invocation should end up appropriately changing the DataGrid control view in the corresponding client’s Main Window.

5. When the user selects a DataGrid row of a signed-in user, the Chat button at the lower-left of the Main Window form should be enabled. Clicking on this button would initiate a chat session with the selected user. The supplied code contains two forms, TextView and ScribbleView, which should give you an idea of the kind of interaction that is expected. A basic chat session would involve each client popping up a TextView window, and then sending messages to each other (the buttons on the TextView form should be self-explanatory). A more advanced chat session would include a ScribbleView window popping up at each client permitting them to also exchange free-form drawings.

Before the TextView window is opened, your client program should send a request to the other client verifying that the other user is interested in chatting. You can handle this verification by popping up a MessageBox at the other end, and waiting for the user to either accept or decline the invitation. Once the above has been verified, both ends can open up their TextView windows and proceed with the chat session. The session terminates whenever either of the users clicks on the Quit button of TextView.

Note that the above should **not involve any communication with the server**. In other words, the return value(s) of the registration request should contain sufficient information (i.e., remote object references) to permit the above interaction.

6. The ScribbleView window should be opened only if both clients are capable of supporting an advanced chat session, as determined by passing an appropriate command-line argument when starting the client program. While client capabilities can be communicated explicitly (e.g., as an extra parameter during the registration process, which is subsequently communicated as part of the callback to previously signed-in clients), you are required to communicate this capability by encoding it as part of the type information. In other words, a client receiving an object reference for another client should be able to inspect it (as in our sample application) to determine whether it can support a ScribbleView session or not. If it can, a different proxy should be instantiated, otherwise not. [Encoding client capabilities as part of the type information permits client functionality to evolve separately from server functionality, which does not need to change everytime clients incorporate an additional feature]

Read through the ScribbleView code to fully understand how it works. Each client maintains two data structures: one for its own scribbles, and the other as a copy for the chat partner’s scribbles. The latter presents a remote object interface (which is registered as a callback) to permit it to be updated everytime a user draws a new stroke. The supplied code captures this interaction between two ScribbleView objects that are part of the same application domain: you should be able to reuse most of the code but will need to make sure that remote objects are exported and imported appropriately.

7. Your ChatClient should be capable of multiple simultaneous chat sessions, so you will need to make sure that your remote objects are defined appropriately.

**What You Need To Do:**

1. Following the structure of the simple .NET Remoting application, create two additional projects (of type “Visual C# Class Library) under the Lab3 solution: ChatRemotable-Client and ChatRemotable-Server. As with the sample application, the former should contain interfaces/classes expected by the ChatClient program, while the latter defines the same for the server program.

Note that your application has three kinds of remote object interactions: (a) between the chat client and the server (for registration: the object resides in the server’s domain); (b) between the chat server and the client (for callback: the object resides in the chat client’s domain); and (c) between two chat clients (for the TextView and ScribbleView sessions: objects reside on both clients). defining the RPC protocol,
while the latter defines the server functionality. Thus, the ChatRemotable-Client and ChatRemotable-Server projects will end up including three kinds of classes: those whose interface is exposed in one while the implementation is in another (and vice-versa), and those whose interface and implementation are both in the ChatRemotable-Client project.

Make sure that you modify the default name of the assembly that will be produced as a result of building these two projects (right-click on the project name in the Solution Explorer panel and select Configuration options). The name of the assembly in each case should be the same, otherwise the remoting infrastructure will not be able to match the metadata of the interface and implementation classes.

2. Create an additional project (of type “Visual C# Console Application”), ChatServer, to hold the server code. Add a reference to the ChatRemotable-Server project. Similarly, in the existing ChatClient project, add a reference to the ChatRemotable-Client project.

3. Provide the required code in each of the Chat* projects. I would advocate following a staged approach, where you first verify that one part of the overall application works completely before proceeding to the next one. Here is a suggested breakdown of tasks:

   a. Client registration at server (involves a single remote object call from the client to the server)
   b. Server callback at client (involves the client passing an object reference to the server as part of the registration process, and the server using this reference later to invoke a method on the client).
   c. Client request for a chat session (involves the client receiving an object reference as a return value from the server, and then using it to invoke a method on another client).
   d. Chat sessions using only the TextView form (involves exchange of object references to each other’s TextView objects and use of these references to pass messages back-and-forth)
   e. Chat sessions using both TextView and ScribbleView forms (involves type inspection of object references, dynamic creation of proxies, and exchange/use of object references within the ScribbleView structure).

When grading your lab, we will be looking at which of the above milestones you have met, so I strongly encourage you to follow this suggestion.

4. A portion of your grade will be determined by how well you handle “race conditions”. Consider for instance what might happen if two clients register with the server at the same time: is it possible that one of them will miss the notification from the server that the other has signed in? Similarly, what happens when two clients request chat sessions with each other at exactly the same time: do you pop up two sets of TextView forms, or just one? You will find it useful to take a look at the C#/.NET lock and Monitor constructs for this purpose.

5. Handle errors and exceptions as gracefully as you can (e.g., by catching the appropriate exception and printing out an error message or informing the user about the error using a MessageBox).

Submit the lab folder information and writeup using the web-based submission form on the development server that you have an account on.