Ingeniería en Sistemas

Programación de Redes

Obligatorio II: Sistema de mensajería

Grupo: M6A
Estudiante: Martin Graña 178499

Indice

Alcance

El sistema desarrollado de mensajería fue implementado como una aplicación de consola, tanto para el lado del cliente como el servidor. Manteniendo la estructura de la primera entrega.

Los cambios nuevos implicaron la creación de dos nuevas aplicaciones de consola.
Una para levantar el servicio WCF que contiene las operaciones CRUD para el mantenimiento de perfiles de usuario. Y la otra que va a ir mostrando en la consola los logs de las operaciones que realizan los clientes sobre el servidor, utilizando MSMQ.

Por ultimo, tenemos una aplicación de Windows Forms para poder acceder a las funcionalidades de mantenimiento de usuarios.

El sistema cuenta con la implementación de todos los requerimientos pautados por letra.

Descripción del sistema

Arquitectura y componentes:

Dentro de los cambios para esta entrega. Cree una clase Context, la cual es singleton y contiene las listas con los objetos del sistema de mensajería (clientes activos, usuarios registrados, etc).

```
public class Context
{
    public static Context instance;

    private Dictionary<Socket, UserProfile> authorizedClients;
    private List<KeyValuePair<Socket, Socket>> clientAndMessenger;
    private List<UserProfile> storedUserProfiles;
    private Dictionary<Socket, Thread> activeClientThreads;

    static readonly object _lockAuthorizedClients = new object();
    static readonly object _lockClientAndMessenger = new object();
    static readonly object _lockStoredProfiles = new object();
    static readonly object _lockActiveClients = new object();

    1 reference | 0 changes | 0 authors, 0 changes
    private Context()
    {
        LoadPersistenceStructures();
        LoadUserProfiles();
        activeClientThreads = new Dictionary<Socket, Thread>();
        clientAndMessenger = new List<KeyValuePair<Socket, Socket>>();
    }

    2 references | 0 changes | 0 authors, 0 changes
    public static Context GetInstance()
    {
        if (instance == null)
            instance = new Context();
        return instance;
    }
```

Desde el servidor de mensajería se obtiene la instancia de dicha clase para realizar consultas/ modificaciones al recibir consultas de clientes. Y es usada a su vez, por la implementación de una interfaz llamada IUserRepository, la cual se registra como servicio por Remoting para realizar operaciones CRUD en usuarios.

```
4 references | 0 changes | 0 authors, 0 changes
public interface IUserRepository
{
    2 references | 0 changes | 0 authors, 0 changes
    UserProfile GetUserProfile(string username);
    2 references | 0 changes | 0 authors, 0 changes
    void CreateUserProfile(string username, string password);
    2 references | 0 changes | 0 authors, 0 changes
    void ModifyUserProfile(string profile, string newUserName, string newPassword);
    2 references | 0 changes | 0 authors, 0 changes
    void DeleteUserProfile(string username);
}
```

El Remoting se lleva a cabo en el ServerMessenger/Server.cs y WSHostMessenger/ WSUserProfiles.cs de la siguiente manera:

En Server.cs

```
1 reference | 0 changes | 0 authors, 0 changes
private static void RegisterRemotingUserRepository()
{
    TcpServerChannel channel = new TcpServerChannel(7777);
    ChannelServices.RegisterChannel(channel, false);
    RemotingConfiguration.RegisterWellKnownServiceType(typeof(UserRepository), "Users", WellKnownObjectMode.SingleCall);
}
```

En WSUserProfiles.cs

```
[ServiceBehavior(IncludeExceptionDetailInFaults = true)]
2 references | 0 changes | 0 authors, 0 changes
public class WSUserProfiles : IWSUserProfile
{
    private IUserRepository UserRepository;
    0 references | 0 changes | 0 authors, 0 changes
    public WSUserProfiles()
    {
        ActivateRemotingUserRepository();
    }

    1 reference | 0 changes | 0 authors, 0 changes
    private void ActivateRemotingUserRepository()
    {
        ConfigurationManager.RefreshSection("appSettings");
        string remotingIp = ConfigurationManager.AppSettings["RemotingIp"];
        TcpClientChannel channel = new TcpClientChannel();
        ChannelServices.RegisterChannel(channel, false);
        UserRepository = (IUserRepository)Activator.GetObject(typeof(IUserRepository), "tcp://"+remotingIp+":7777/Users");
    }
```

Donde WSUserProfiles va a ser utilizado por el servicio WCF

```
class Service
{
    0 references | 0 changes | 0 authors, 0 changes
    static void Main(string[] args)
    {
        ServiceHost messengerServiceHost = null;
        try
        {
            ConfigurationManager.RefreshSection("appSettings");
            string serviceIp = ConfigurationManager.AppSettings["ServiceIp"];

            Uri httpBaseAddress = new Uri("http://"+ serviceIp + ":8719/MessengerService");

            messengerServiceHost = new ServiceHost(typeof(WSHostMessenger.WSUserProfiles), httpBaseAddress);
            messengerServiceHost.AddServiceEndpoint(typeof(WSHostMessenger.IWSUserProfile), new WSHttpBinding(), "");

            ServiceMetadataBehavior serviceBehavior = new ServiceMetadataBehavior();
            serviceBehavior.HttpGetEnabled = true;
            messengerServiceHost.Description.Behaviors.Add(serviceBehavior);

            messengerServiceHost.Open();
            Console.WriteLine("Service is live now at: {0}", httpBaseAddress);
            Console.ReadKey();
        }
```

El manejo de los logs de MSMQ se encarga la clase LogView. Que una vez que se ejecuta, crea un MessageQueue privado llamado ‘networkMessenger’ si es que este no existe. Y ejecuta un evento asincrónico para ir mostrando los logs a medida que los recibe y los consume de la queue.

```
class LogView
{
    private static string queueName;
    private static MessageQueue msgQueue;

    0 references | 0 changes | 0 authors, 0 changes
    static void Main(string[] args)
    {
        queueName = @".\private$\networkMessenger";
        InitMessageQueue();
        Console.WriteLine("> Server aplication log online");
        Console.WriteLine("> Receiving messages ...");

        msgQueue.PeekCompleted += new PeekCompletedEventHandler(MessageHasBeenReceived);
        msgQueue.BeginPeek();
        Console.ReadKey();
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private static void InitMessageQueue()
    {
        if (!MessageQueue.Exists(queueName))
            msgQueue = MessageQueue.Create(queueName);
        else
            msgQueue = new MessageQueue(queueName);
        msgQueue.Formatter = new XmlMessageFormatter(new Type[] { typeof(String) });
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private static void MessageHasBeenReceived(object sender, PeekCompletedEventArgs e)
    {
        var msg = msgQueue.EndPeek(e.AsyncResult);
        Console.WriteLine((string)msg.Body);
        msgQueue.ReceiveById(msg.Id);
        msgQueue.BeginPeek();
    }
```

Para enviar los mensajes a la MessageQueue definida, donde se esta ejecutando el programa. La aplicación de servidor de mensajería lo hace a través de la clase AppLog por un canal TCP con la Ip de donde esta corriendo el LogView.

```
3 references | 0 changes | 0 authors, 0 changes
public class AppLog
{
    private static MessageQueue msgQueue;
    1 reference | 0 changes | 0 authors, 0 changes
    public AppLog(string remoteIp)
    {
        string queueName = @"FormatName:DIRECT=TCP:"+remoteIp+@"\private$\networkMessenger";
        msgQueue = new MessageQueue(queueName);
    }

    23 references | 0 changes | 0 authors, 0 changes
    public void SendDetailedMessage(string operation, string user, string detail)
    {
        string messageData = "Operation: " + operation + " - User: " + user + " - Details: " + detail;
        Message msg = new Message(messageData);
        msgQueue.Send(msg);
    }
```

Por ultimo. El funcionamiento de la subida de archivos es el siguiente.
El cliente al seleccionar un archivo valido, genera un paquete con el nombre y el tamaño del mismo y la petición de upload al servidor.
El servidor entonces ahora sabe cuantos bytes va a tener que leer y con que nombre y extension lo va a almacenar.
Una vez que ocurre esto, el cliente recibe una respuesta y el archivo que tenia pendiente por subir comienza a ser enviado en pequeños fragmentos.

En el orden explicado:
El cliente realiza

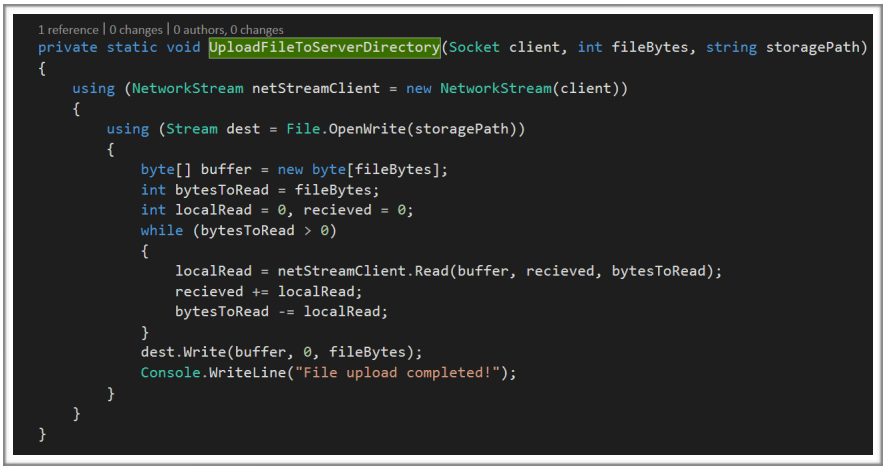
```
1 reference | 0 changes | 0 authors, 0 changes
private static void SendFileRequestToServer(ChatProtocol request)
{
    if (!File.Exists(request.Payload))
        throw new Exception("Error: Invalid file path for upload");

    using(FileStream fileStream = new FileStream(request.Payload, FileMode.Open, FileAccess.Read))
    {
        var name = Path.GetFileName(request.Payload);
        var length = fileStream.Length;
        var command = request.Command;
        var packageName = chatManager.CreateRequestProtocol(command, name + "#" + length);
        SendPackage(packageName);
    }
    fileToSend = request.Payload;
}
```

El servidor obtiene los datos y responde. A lo que el cliente comienza a enviar el archivo.

```
1 reference | 0 changes | 0 authors, 0 changes
private static void SendFile(string path)
{
    try
    {
        using (Stream source = File.OpenRead(path))
        {
            byte[] buffer = new byte[2048];
            int bytesRead;
            while ((bytesRead = source.Read(buffer, 0, buffer.Length)) > 0)
            {
                netStream.Write(buffer, 0, bytesRead);
            }
        }
    }
    catch (Exception)
    {
        ServerConnectionLost();
    }
}
```

Y por ultimo el servidor lo almacena en su directorio de archivos:



```
1 reference | 0 changes | 0 authors, 0 changes
private static void UploadFileToServerDirectory(Socket client, int fileBytes, string storagePath)
{
    using (NetworkStream netStreamClient = new NetworkStream(client))
    {
        using (Stream dest = File.OpenWrite(storagePath))
        {
            byte[] buffer = new byte[fileBytes];
            int bytesToRead = fileBytes;
            int localRead = 0, recieved = 0;
            while (bytesToRead > 0)
            {
                localRead = netStreamClient.Read(buffer, recieved, bytesToRead);
                recieved += localRead;
                bytesToRead -= localRead;
            }
            dest.Write(buffer, 0, fileBytes);
            Console.WriteLine("File upload completed!");
        }
    }
}
```

Mecanismo de concurrencia:

El mecanismo de concurrencia utilizado para la consistencia de datos al ser un programa multithreaded y orientado a sockets, es el uso de locks en atributos privados de la clase Context que mantiene las listas del sistema y en la clase UserProfile para bloquear el uso de sus atributos cuando intentan modificarlo/acceder varios clientes simultáneos.
Donde cuando un thread requiere el uso de un recurso seriamente reusable, bloquea su uso mientras modifica o lee datos de este.

Guia de uso

Deploy:
Los .exe se encuentran dentro del bin/Release de cada proyecto.

Se deben configurar los parámetros del App.config de ClientMessenger con el ip/puerto del servidor y la ip de la maquina donde se ejecuta el cliente.

Del App.config de ServerMessenger. Se debe indicar la Ip/Puerto de la maquina que va a correr el servidor de mensajeo. Así como modificar el key 'LogsIp' con la ip de la maquina con el ServerMSMQ.

En el proyecto WSHostMessenger en el App.Config, se especifica desde donde se esta corriendo el servicio con la key 'ServiceIp' y 'RemotingIp' con la ip de la maquina que tiene al servidor de mensajeo.

Por ultimo en el WSClientMessenger se modifica el key 'endpoint' del App.config con la url del servicio http://{ip}:8719/MessengerService.

El orden de configuración para que se ejecuten correctamente las aplicaciones:
- ServerMessenger.exe
- WSHostMessenger.exe (modo administrador)
- ServerMSMQ.exe

Posteriormente se pueden iniciar tanto los clientes de mensajeo como los de mantenimiento de usuarios, de forma indiferente.
- WSMessengerClient.exe
- ClientMessenger.exe

Uso de comandos:

00 -> Logout
01Usuario#Password -> Login con user 'Usuario' y password 'Password'
02Usuario#Password -> Registro de nuevo usuario y login del mismo.
03 -> Retorna lista de perfil de usuarios que están online en el servidor.
04 -> Retorna lista de amigos del cliente loggeado.
05Usuario -> Envía una solicitud de amistad al usuario 'Usuario;'
06 -> Retorna lista de solicitudes de amistad pendientes
07Usuario#YES -> Acepta solicitud de amistad pendiente
07Usuario#NO -> Cancela solicitud de amistad
08Usuario#CHAT -> Comienza chat con usuario
08Usuario#END -> Finaliza chat con usuario
09 -> Lista usuarios que dejaron mensajes
09Usuario -> Lista mensajes de Usuario

Nuevos comandos:

10PathDelArchivo(Sin comillas) -> Sube archivo del cliente al servidor
11 -> Lista los archivos del servidor, disponibles para descargar
11NombreDelArchivo -> Se envía al cliente el archivo seleccionado