Universidad ORT Uruguay

Facultad de Ingeniería Bernard Wand Polak

Programación de Redes

Documentación Obligatorio 1

Sebastian Zolkwer - 222841

Alex Piczenik - 231093

Fecha: 30.09.2021
Grupo: M6B

Índice

Protocolo

La implementación del protocolo está pautada de forma que este se pueda utilizar por ambas partes del sistema, el/los clientes y el servidor. A continuación detallaremos las principales decisiones tomadas al momento de implementarlo. El envío se realiza mediante un objeto de tipo Header que contiene en su definición los siguientes atributos:

1. Información del tipo de envío request o response (REQ/RES) dependiendo del tipo de mensaje.
2. Número de método. En el caso de recibir un header de respuesta, el número debe ser un 0 en caso de error o un 1 de haberse concretado la operación. Por otro lado, en un request, el número varía entre 1 y 13 permitiéndonos definir una funcionalidad distinta para cada uno de los indicadores utilizados. A modo de aclaración, el usuario únicamente puede seleccionar desde su menú un número del 1 al 10, sin embargo

de manera interna subdividimos determinados request en más de uno provocando que las posibles opciones de número de método existan entre 1 y 13.

3. El tercer aspecto incluido en el header es el largo total de la información a mandar posteriormente.

En el momento en el que creamos una nueva instancia del objeto Header, los campos se codifican mediante Encoding.UTF8.GetBytes(). Los largos máximos para cada uno de los atributos están predefinidos y son:

1. Req / Res: 3
2. Número de método: 2
3. Largo total: 4

```
public class Header
{
    private byte[] direction;
    private byte[] method;
    private byte[] dataLength;

    private string _direction;
    private int _method;
    private int _dataLength;

    2 references
    public Header(string status, int _method, int datalength)
    {
        direction = Encoding.UTF8.GetBytes(status);
        string stringMethod = _method.ToString("D2");
        method = Encoding.UTF8.GetBytes(stringMethod);
        string stringData = datalength.ToString("D4");
        dataLength = Encoding.UTF8.GetBytes(stringData);
    }
```

Desde el lado receptor, en primer lugar se reserva la memoria necesaria para guardar los datos a recibir conociendo previamente los largos máximos disponibles. Luego, se recibe y decodifica para poder obtener el header correspondiente.

```
public static Header ReceiveAndDecodeFixData(Socket handler, Header header)
{
    int headerLength = Request.Length + MethodLength + DataLength;
    byte[] data = ReceiveData(handler, headerLength);
    header.DecodeData(data);
    return header;
}
```

En el siguiente paso, se utiliza un nuevo método encargado de recibir y decodificar la información enviada, desde donde se utiliza el método ReceiveData que explicaremos a continuación.

```
public static string RecieveAndDecodeVariableData(Socket socket, int length)
{
    byte[] value = ReceiveData(socket, length);
    string response = Encoding.UTF8.GetString(value);
    return response;
}
```

Desde el método ReceiveData nos encargamos de recibir tanta información como el largo máximo enviado por el header en un comienzo. Con el fin de asegurarnos que no existieron pérdidas en el proceso, se itera en función de dos variables que administran la cantidad de data recibida y la restante.

```
public static byte[] ReceiveData(Socket socket, int length)
{
    int offset = 0;
    byte[] response = new byte[length];
    while (offset < length)
    {
        int received = socket.Receive(response, offset, length - offset, SocketFlags.None);
        if (received == 0)
        {
            throw new SocketException();
        }
        offset += received;
    }
    return response;
}
```

Por otro lado, en caso de recibir una imagen, en lugar de acudir a la función ReceiveData se llama directamente a una correspondiente llamada ReceiveFile.

A modo de comentario, en caso de querer enviar una colección de datos, se parsea a string concatenando cada elemento con “-” o “!”. De modo que para enviar un juego con su correspondiente nombre, género y descripción se hace del siguiente modo:

```
$"{titleGame}-{rate}-{gender}",
```

Análogamente, para enviar una imagen junto a su nombre, título, y tamaño:

```
string fileData = fileName + "!" + title + "!" + fileSize;
```

Finalizando con la explicación del protocolo, de forma que no existan “números mágicos” haciendo referencia a métodos específicos, o constantes de tipos string que no dejen en claro su función, decidimos definir una clase estática llamada ProtocolMethods donde definimos una serie de constantes utilizadas durante todo el código.

```
public static class ProtocolMethods
{
    public const int Error = 0;
    public const int Success = 1;
    public const int Create = 1;
    public const int Update = 2;
    public const int Buy = 3;
    public const int Evaluate = 4;
    public const int Search = 5;
    public const int Show = 6;
    public const int ShowAll = 7;
    public const int Reviews = 8;
    public const int Delete = 9;
    public const int Exit = 10;
    public const int SendImage = 11;
    public const int UpdateRoute = 12;
    public const int ReceiveImage = 13;
    public const string Request = "REQ";
    public const string Response = "RES";
}
```

Arquitectura del sistema

Con el objetivo de desacoplar la solución y poder separar las distintas secciones del proyecto, decidimos organizarlo en 5 paquetes (Server, Client, Protocol, File y Bussiness). Procederemos a comentar de que se compone cada uno y sus responsabilidades dentro del proyecto.

Server

En el paquete server contamos con el método main encargado de crear una instancia de serverHandler. En el serverHandler creamos el socket, el Ipendpoint y asociamos el socket al ipEndpoint quedando este en estado de escucha para conexiones. En el constructor mismo se escucha lo que se escribe directamente desde la terminal del server y en el thread se aceptan los correspondientes clientes.
Los dos primeros programas son para que corran en paralelo los dos hilos. Luego abrimos un nuevo thread para cada cliente con el fin de tener más de un cliente trabajando en paralelo. Al momento de crear el thread, utilizamos el método …… ubicado en la clase estática LogicServer. Decidimos implementar la clase HandlerClient de forma estática ya que no consideramos necesario realizar una nueva instancia para cada cliente que desee utilizarla.

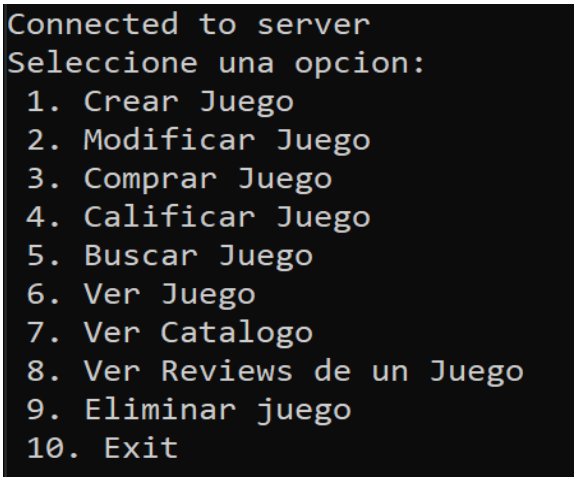
Business

El paquete Business se compone por el dominio y la lógica básica para el funcionamiento del sistema. Por un lado, contiene una carpeta llamada Domain donde definimos las clases Game y Review. Game contiene los atributos necesarios para definir un juego, título, género, sinopsis, calificación promedio, suma de calificación y una lista de reviews, mientras que una Review está formada por una calificación (numérica) y una descripción de la misma.
Por otro lado, en la clase Logic nos enfocamos en definir todas las funcionalidades posibles a realizar con los objetos Game y Review. Para realizar un buen manejo de errores decidimos lanzar las excepciones siempre desde una misma capa del sistema y optamos por la capa de negocios con el fin de poder capturar las excepciones desde un paquete externo a medida que utilizamos las funcionalidades. Adjuntamos foto mostrando métodos definidos en la clase Logic.

```
public static string Add(string information)...
10 references
private static Game GetGame(string title)...
1 reference
public static string Update(string information)...
2 references
public static string GetAll()...
1 reference
public static string Delete(string data)...
1 reference
public static string Evaluate(string information)...
1 reference
public static string Search(string filters)...
1 reference
public static string GetRouteImage(string request)...
1 reference
public static void UpdateRoute(string request)...
1 reference
public static string Show(string data)...
2 references
public static string GetReviews(string data)...
1 reference
public static string Buy(string data, List<Game> boughtGames)...
2 references
private static int GetNumber(string number)...
```

Client

En el paquete Client nos encargamos de inicializar el sistema para que un nuevo cliente pueda comunicarse con el servidor. Desde el main de la clase ClientProgram se inicializa una nueva instancia de ClientHandler desde donde se inicializa un socket y dos Ip, para el cliente y el server utilizando los puertos correspondientes.
Una vez asociado el socket al IpEndpoint del cliente y conectado al IpEndPoint, se llama al método WriteServer de la clase estática LogicClient. Este método despliega en consola el menú de métodos posibles a realizar por el cliente, indicando un número para cada funcionalidad. En función de la respuesta brindada por el client, se solicitan los datos correspondientes para realizar la solicitud e instantáneamente se envía la información mediante los métodos declarados por el protocolo.



File

El paquete File fue implementado específicamente para ejecutar la funcionalidad encargada de enviar y recibir imágenes entre el cliente y el servidor. Dentro del mismo definimos dos clases: FileHandler y FileStreamhandler . FileHandler cuenta con las funcionalidades necesarias para obtener el largo de un archivo, su nombre y verificar si existe. Por otro lado, en FileStream Handler definimos la funcionalidad para leer y escribir la data de un archivo.

Protocol

Por último, en el paquete Protocol existen tres clases: Protocol, Header y ProtocolMethods.
Si bien el funcionamiento del protocolo fue explicado anteriormente, procederemos a establecer las responsabilidades de las clases dentro del paquete. Con el fin de tener el menor acoplamiento entre clases posible, definimos en la clase protocolo todas las funcionalidades necesarias para lograr enviar y recibir el mensaje desde un lado a otro. De este modo, desde LogicClient o LogicServer únicamente debemos utilizar los métodos del protocolo para interactuar con el texto/imagen, desprendiéndose completamente de la implementación de los mismos. Como también ya mencionamos anteriormente, la clase

ProtocolMethods tiene las distintas constantes a utilizar instanciadas con los valores correspondientes.
Otro de los aspectos a destacar con relación a las justificacion de código en el protocol, es que para enviar o recibir archivos no utilizamos directamente la función Send o Receive del socket, sino que realizamos una iteración basada en lo ya enviado/recibido para volver a solicitar a partir de allí. Adjuntamos extracto del método ReceiveFile:

```
string type = nameFile[nameFile.Length - 1];
while (fileSize > offset)
{
    byte[] buffer;
    if (currentPart != parts)
    {
        buffer = ReceiveData(socket, MaxPacketSize);
        offset += MaxPacketSize;
    }
    else
    {
        int lastPartSize = (int)(fileSize - offset);
        buffer = ReceiveData(socket, lastPartSize);
        offset += lastPartSize;
    }

    if (servidor)
    {
        fileStreamHandler.WriteData("CaratulasServer/" + title + "." + type, buffer);
    }
    else
    {
        fileStreamHandler.WriteData("CaratulasClient/" + title + "." + type, buffer);
    }

    currentPart++;
}
```

Funcionamiento desde el cliente

En primer lugar, desde la consola se indica que se realizó de forma correcta la conexión con el servidor e inmediatamente se expone el menú con las posibles funcionalidades. El cliente debe optar por alguna de ellas mediante un número del 1 al 10. Al elegir, se solicitan distintos campos dependiendo del caso. Una vez ingresados los campos, se obtiene una respuesta correspondiente. Una vez culminada la primera operación indicada por el cliente, se vuelve a desplegar el menú hasta que el cliente seleccione la opción 10 (Exit).

A continuación detallamos cada una de las funcionalidades posibles:

- Crear un juego:
  Se solicita título, género, sinopsis y crítica. Si ya existe otro juego con el mismo título, el juego no se va a poder crear.
- Modificar un juego:
  Se solicita el nombre del juego que se desea modificar. Se solicita nuevo título, género y sinopsis. Si no se desea modificar alguno de estos campos, se deben dejar en blanco presionando enter y continuando con la ejecución.
- Comprar juego:
  Se solicita el nombre de juego que se desea comprar, se procesa la compra si el juego existe en el sistema y no fue comprado por ese cliente previamente.
- Calificar juego:
  Se debe ingresar nombre del juego que se desea calificar, puntuación(número del 1 al 10) y descripción.
- Buscar juego:
  En la búsqueda de un juego se pueden ingresar los siguiente filtros si se desea: título, género y puntuación. Si no se desea se deben dejar en blanco.
- Ver detalles:
  Se solicita nombre del juego y se retorna nombre, género, sinopsis, puntuación en promedio, reviews y se pregunta si quiere descargar la carátula. De querer descargar la carátula se debe responder con alguno de los siguientes comandos (SI, si, Si, sI), en otro caso se interpreta como un “No”.
- Ver catálogo:
  Se retornan todos los juegos del sistema.
- Ver reviews:
  Se solicita nombre de juego y se despliegan todas las reviews existentes para ese juego. El sistema soporta que no hayan reviews previamente por lo que de seleccionar un juego sin reseñas el programa no se caerá.
- Borrar juego:
  Se solicita nombre de juego a eliminar, si existe se borra del sistema
- Salir del sistema:
  Al ingresar el número indicado en el menú, se cierra la terminal para ese cliente

Funcionamiento desde el server

Al existir en la letra mismas funcionalidades a implementar desde el servidor y el cliente, en común acuerdo con los docentes optamos por definir las funcionalidades en común desde el cliente por lo que desde el server únicamente se realizan dos funciones como se muestra en el menú: solicitar catálogo y desconectarse.

Manejo de excepciones

Como mencionamos previamente, el objetivo durante todo el desarrollo del proyecto fue lanzar las excepciones desde la capa de Logic, indicando claramente su mensaje de forma que al capturarlas desde las capas superiores mediante try and catch pudiéramos dejar en claro que había provocado su lanzamiento.

Comunicación mediante sockets

La comunicación del cliente y el servidor se realiza mediante sockets tal como lo explicamos anteriormente. Para la implementación de los sockets utilizamos una de las librerías de Sockets de .Net (System.Net.Sockets) la cual provee la clase Socket y facilita el uso de los mismos.

Utilización de threading

Con el fin de poder atender diferentes peticiones entre servidores y clientes, decidimos utilizar threading, sin embargo, para no utilizar más recursos de los necesarios, diagramamos la solución del siguiente modo.
De parte del Server iniciamos dos hilos, el principal, encargado de escuchar lo que se escribe en la terminal del server y el thread para aceptar clientes, donde se abre un nuevo thread por cada cliente iniciado. La función de los thread es poder ejecutar métodos en paralelo. Como aspecto a destacar, el thread para escuchar al cliente lo declaramos background = true tal que al finalizar el hilo principal, este también se cierre.

Mutua exclusividad

Al tener la posibilidad de tener más de un cliente interactuando con la misma lista de juegos, optamos por implementar mutua exclusividad de forma que si un cliente desea modificar la lista, el sistema se asegura que ningún otro esté utilizandola y de así serlo se bloquea permitiendo su modificación. Lo implementamos utilizando lock de la siguiente forma:

```
public static string Add(string information)
{
    string[] data = information.Split("-");
    Game newGame = new Game...;
    lock (gamesLock)
    {
        if (GetGame(data[0]) != null) throw new Exception("Ya existe un juego con ese nombre\n");
        games.Add(newGame);
    }
    return "Fue agregado el juego\n";
}
10 references
```

Manejo de archivos

Al iniciar el sistema, se crea automáticamente tanto del lado del server como del cliente una carpeta ubicada en Directory.getCurrentDirectory(). En estas carpetas es donde se guardarán las imágenes una vez enviadas o recibidas.
Una vez que el cliente decide mediante los comandos de consola crear un nuevo juego, se le solicita la ruta correspondiente de la carátula a enviar. En el código, para realizar correctamente el enviado de la imagen en la data se envía el header y luego la data de la siguiente forma:
fileName + "!" + title + "!" + fileSize codificado.
Desde el lado receptor, primero se decodifica el header, luego decodificamos la data para obtener el largo del archivo, nombre y el título para posteriormente escribirlo en la ruta dependiendo si es servidor o cliente utilizando file stream. Existe una carpeta que se llama

CaratulasServer proyecto\SocketSimpleServer\bin\Release\netcoreapp3.1 y CaratulasClient proyecto\SocketSimpleClient\bin\Release\netcoreapp3.1 con las fotos respectivamente.

Ejecutables

Los ejecutables del server y client se encuentran en:
proyecto\SocketSimpleServer\bin\Release\netcoreapp3.1
proyecto\SocketSimpleClient\bin\Release\netcoreapp3.1
Agregamos además, una carpeta con archivos ejemplo para facilitar el proceso de testing del sistema. Estas imágenes ya fueron probadas y funcionan a la perfección.
En caso de utilizar otros archivos puede llegar a fallar el sistema.

Archivos de configuración

Se implementaron archivos appsetting.json tanto del lado del cliente como del servidor. El objetivo de esto es realizar el sistema más fácil de extender ya que de querer un usuario modificar los puertos, backlog o IpAdress únicamente debe hacerlo desde los appsetting.

Conclusiones

El obligatorio nos permitió terminar de comprender los conceptos trabajados en clase, aplicándolos en las distintas funcionalidades implementadas en el proyecto. Resultó un desafío motivante y creemos que nos permite concientizarnos de lo aprendido.