OBLIGATORIO Programación de Redes

2021

Universidad ORT Uruguay
Profesor: Luis Barragué Martínez

Iñaki Etchegaray - 241072
Matías González - 219329

Índice

Abstract

El objetivo de esta entrega es la creación de una aplicación con arquitectura cliente servidor.

La aplicación comprende de dos ejecutables:

- Server
- Client

Para el funcionamiento correcto de las aplicaciones, es necesario para acceder a la funcionalidad del cliente tener iniciado el servidor. Los clientes esperarán hasta conseguir una conexión aceptable con el servidor.

El Sistema Vapor es un sistema multiusuario que permite la creación y persistencia de un catálogo de juegos sobre la plataforma. Uno cuando entra solo debe de indicar su nombre de usuario, si el usuario no existe este se crea automáticamente. No se requiere de una contraseña. Por supuesto, no se permite que haya dos usuarios con el mismo nombre ni que dos clientes distintos puedan acceder al mismo usuario.

La aplicación servidor solo permite el cierre del mismo y no mucho más. El funcionamiento del mismo se limita mucho más a la gestión de múltiples conexiones con usuarios, el manejo de la base de datos y de concurrencia sobre los múltiples procesos que gestionan a los clientes.

La aplicación cliente trae mucha más funcionalidad. Posee un UI simple pero efectivo mediante consola que acepta caracteres entrantes.

El cliente puede publicar juegos con su información pertinente, subir una carátula para el mismo, puede editar y borrar sus juegos publicados. Se pueden ver todos los juegos subidos al sistema y buscar en base a parámetros como el género. Se puede acceder a la información de los juegos de otros usuarios y dejar una reseña de sus juegos con una clasificación del 1 al 5, consecuentemente también se puede acceder a cada una de estas reseñas y ver el puntaje promedio del juego en base a las mismas.

Por último, un usuario puede adquirir un juego, aunque este hecho aún no tiene mucha implicancia en la funcionalidad.

A continuación, explicaremos los detalles detrás del trabajo incluyendo la arquitectura y las principales decisiones de diseño de la misma.

Antes de iniciar, vale la pena mencionar que este proyecto se encuentra versionado en un repositorio en github.

El link al mismo: https://github.com/SleepyWolfy/Obligatorio-1-PR

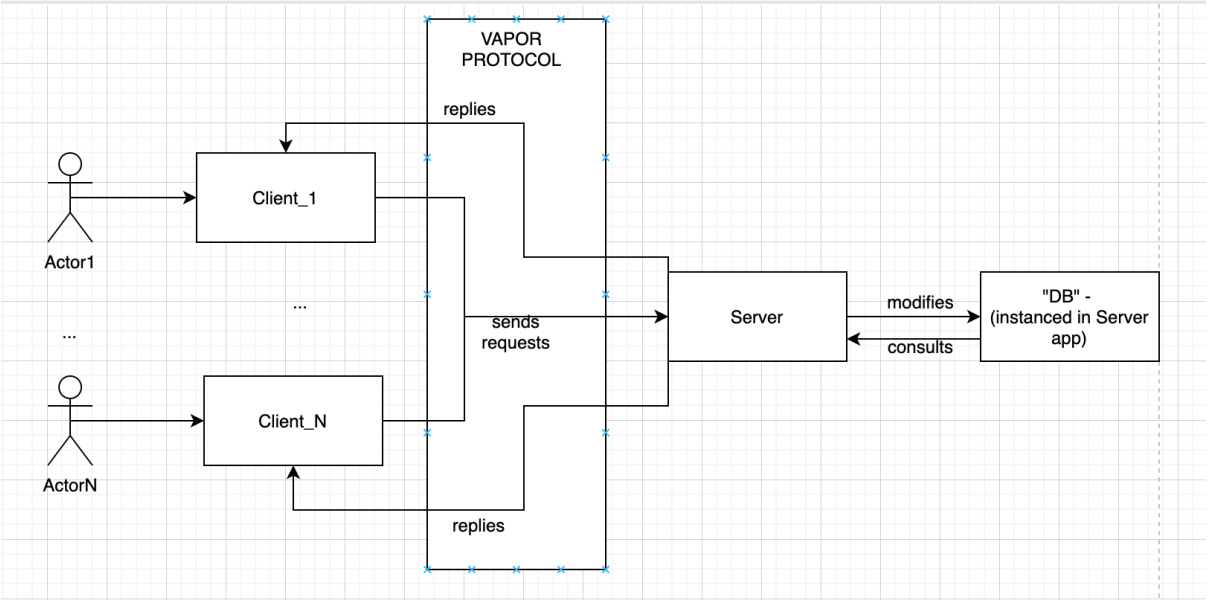
Descripción de Arquitectura

A muy alto nivel la aplicación implementa una Arquitectura Cliente Servidor el cual, como mencionamos previamente, implica dos aplicaciones de software: un servidor y un cliente. El servidor, como dice su nombre, sirve a los clientes que acceden a su funcionalidad. Varios clientes pueden conectarse a un mismo servidor, este maneja las conexiones y las peticiones de todos al mismo tiempo. La idea es que el sistema sea descentralizado y que funcione para clientes conectados sobre la red.

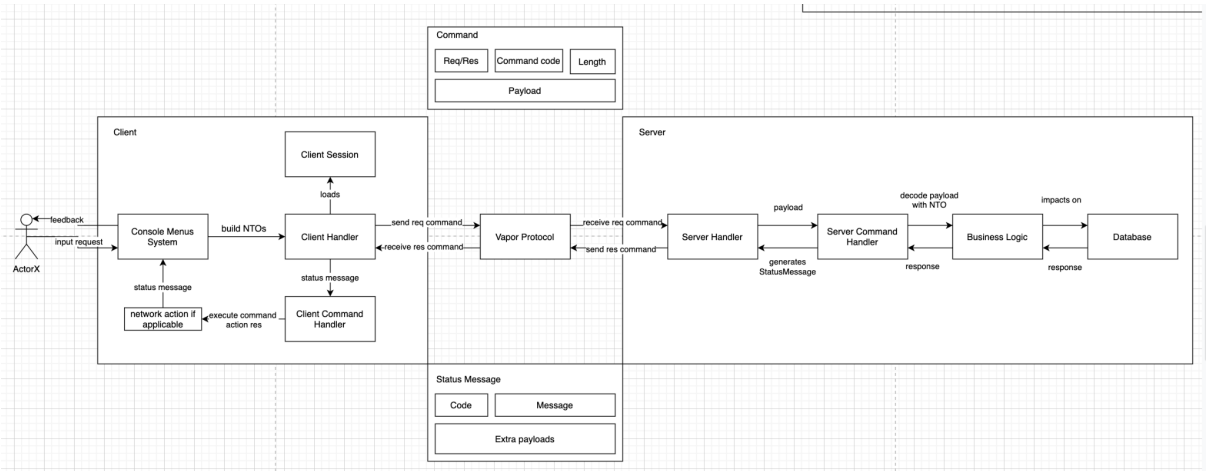
El servidor, en este obligatorio, es una única unidad que persiste toda la información pertinente sobre los juegos que los clientes suben, para el alcance de este primer obligatorio no se pedía una persistencia real, por lo cual nosotros recurrimos a una base de datos falsa en memoria del lado del servidor.

Como el servidor y los clientes deben de comunicarse entre ellos a través de la red, es necesario codificar la información para enviarla en forma de un paquete de información transferible. Luego, por otro lado, es necesario decodificar esta información. Para poder codificar y decodificar los mensajes para que se logren entender ambas aplicaciones sobre la red, necesitamos definir un protocolo de comunicación.

Nuestro protocolo, llamado Vapor Protocol, se encarga de codificar y decodificar los mensajes para luego traducirlos en funcionalidad en el código. El siguiente esquema resume la vista general de la arquitectura:

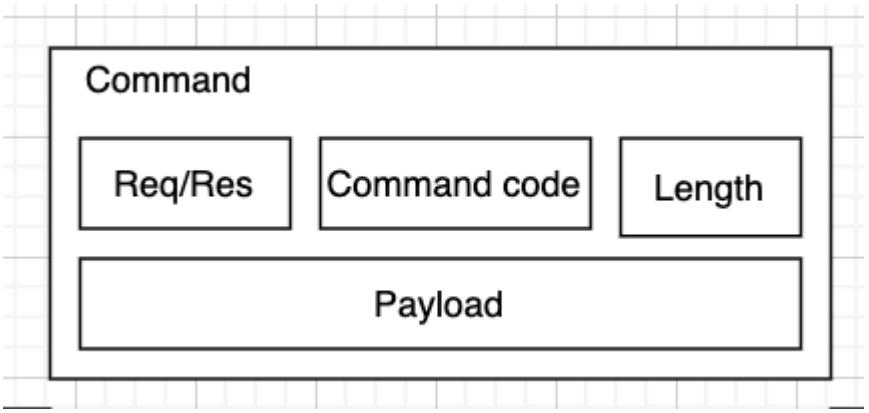


Bajando un poco más a nuestra arquitectura en específico, nuestra especificación de la Arquitectura Servidor Cliente se podría resumir rápidamente en el siguiente gráfico:



El diagrama muestra la interacción del usuario con el sistema y como su mensaje es procesado típicamente por la aplicación. Este diagrama es un flujo que asume la conexión ya establecida y por el lado del servidor representa el proceso manejado por los threads delegados a la gestión de clientes.

Los clientes y el servidor interactúan mediante el Vapor Protocol intercambiando comandos. Los comandos son paquetes con la siguiente información:



El diseño del mismo y los significados de su información será explicado en la sección de diseño del protocolo.

A nivel de código, los comandos son una interfaz que se encargan de realizar la acción definida por el comando del lado del servidor o cliente. Su interfaz es la siguiente:

```
public interface ICommand
{
    1 reference
    string Command {get;}

    1 reference
    string ActionReq(byte[] payload);

    1 reference
    VaporStatusResponse ActionRes(byte[] reqPayload);
```

La property Command indica cual es el código del comando, la función ActionReq recibe el payload y ejecuta la acción del comando del lado del servidor (se lee la acción en reacción de que el comando es un request de un cliente), luego la función ActionRes es el mismo concepto, pero en respuesta de que el comando es una respuesta del servidor. En consecuencia, ActionReq se corre solamente en la ejecución de un comando en server y ActionRes en cliente.

Existe una clase llamada CommandHandler que maneja la ejecución de un comando y además es el que las crea (porque lo que llega es un código), esta clase tiene una versión para el Cliente (Client Command Handler) y una versión para el Servidor (Server Command Handler) por algunas pequeñas diferencias al ejecutar el comando.

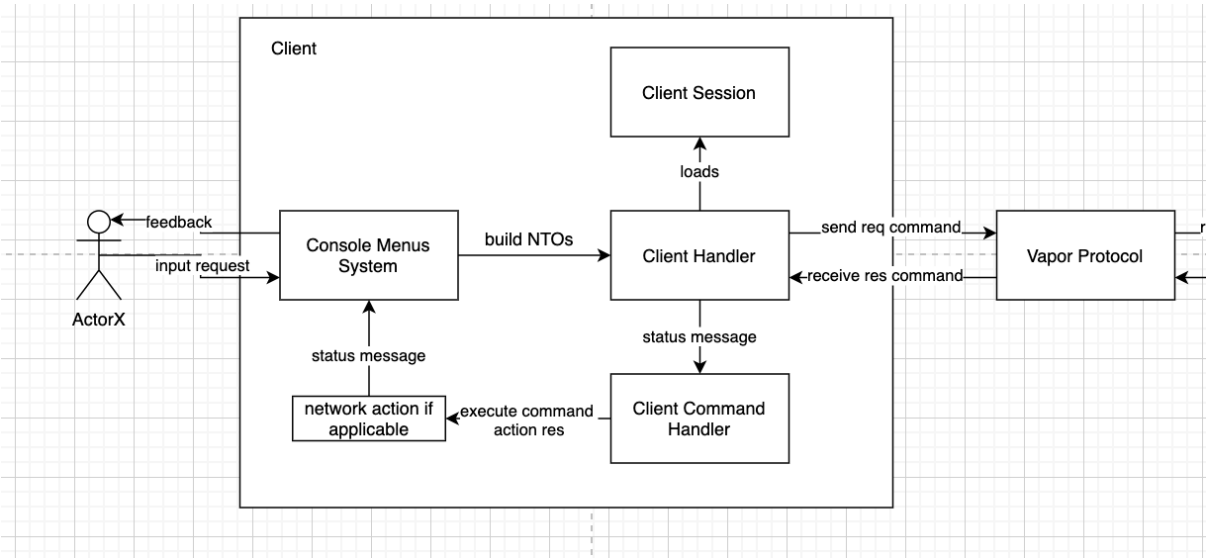
Los comandos se encargan de acceder a la funcionalidad en Business, son los únicos que interactúan con business (exceptuando un caso en servidor: cuando alguien se desconecta inesperadamente, hay que hacer un logout), separando la capa de lógica del servidor-cliente de la lógica de negocio. También se encargan de armar las respuestas con los códigos de estado, y se asisten de unos objetos llamados NTOs en interpretar los byte arrays que recibe, más sobre los NTOs en la parte de diseño del protocolo.

Es importante mencionar que al armar los comandos, su diseño y ejecución fueron pensados como funciones atómicas a través de la red. Es decir, que una funcionalidad de la aplicación se traduzca a una ejecución de un comando y que solo con la ejecución de ese comando se cumpla esa funcionalidad. A efectos de esto, como se ve en el diagrama general, la ejecución de un comando nunca va a desatar la ejecución de otro comando.

Esto ayuda a que el flow de la aplicación sea más fácil de entender y, a su vez, en caso de que haya algún error, este no se propague a la ejecución de otros comandos. Incluso, cuando cerramos el servidor, el mismo espera a que todos los clientes hayan terminado de ejecutar sus comandos, asegurando consistencia en el estado de la base de datos (aunque esta en realidad es falsa).

Cliente

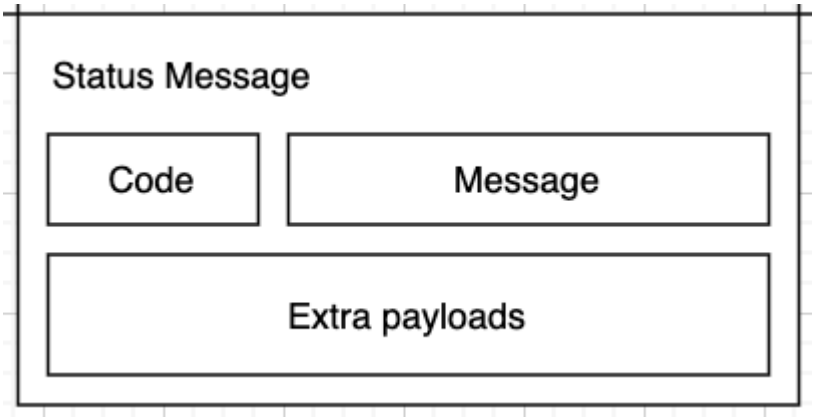
Viendo más de cerca la seccion del cliente:



Un actor X interactúa con el sistema de menús de consola, este le pide input al cual el actor responde. El sistema de consolas toma el input y los transforma en unos objetos cuyo concepto creamos llamados NTOs (Network Transfer Objects), detallaremos más sobre ellos en la parte del diseño del protocolo.

El menú delega al Client Handler, el cual es una fachada al sistema de comandos, el cual se encarga de desplegar el protocolo y a través de él enviar el comando introducido por el actor a través de la red hacia el servidor.

Una vez que el servidor procesa nuestra respuesta, nos devuelve un mensaje que procesamos por el Client Command Handler en un objeto que llamamos Vapor Status Message. El mismo trae un código que nos dice si todo salió bien o si algo salió mal. El sistema de menús y el Client Handler reaccionan a estas situaciones de la manera que sea necesaria. El paquete de status message se puede describir con el siguiente diagrama:



La especificación de este status message será explicada en la sección de diseño del protocolo.

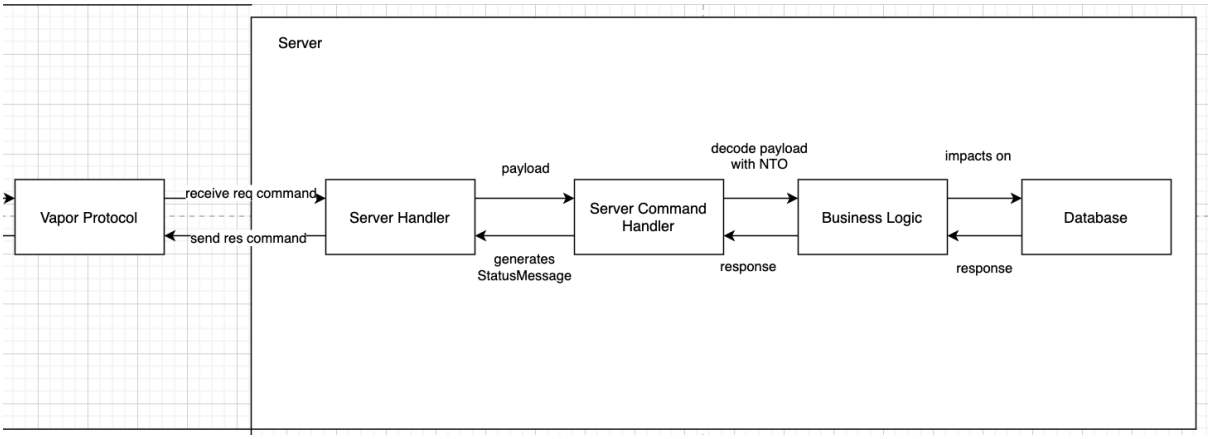
Algunas partes de la respuesta del mensaje serán guardadas en la sesión del cliente (como un juego seleccionado, o la información de la cuenta del cliente). De camino a la

lógica de los menús, el Client Handler puede realizar alguna acción de conexión (por ejemplo, en el caso de pedir una desconexión, desconectar), esta acción se encuentra en esta parte del flujo y en client handler porque consideramos que el tiene la responsabilidad de manejar la conexión con el servidor.

Finalmente el feedback de la request llegará nuevamente al actor quien continuará interaccionando con el sistema.

Server

Luego tenemos al server:



El servidor otorga threads a cada uno de los clientes que se conectan al mismo. En este thread, el servidor simplemente espera a recibir una petición de un cliente. El cliente envía su petición en forma de un comando (previamente mencionado), el servidor la recibe y se la pasa al Command Handler, este traduce la petición a una instancia de una clase comando dependiendo de cual comando se trata. Esta clase comando se encarga de decodificar el payload con ayuda del NTO y de decirle que hacer a la capa de negocio, pasándole la información necesaria decodificada.

Luego la business logic se encarga de impactar sobre la base de datos de la manera que sea necesaria de acuerdo al comando especificado. Una vez que la base de datos responde con un positivo o un error, este es atrapado en el comando nuevamente el cual se encarga de armar un paquete de respuesta de acuerdo al resultado. Si la respuesta fue buena enviará un código OK, de lo contrario (dependiendo de la excepción lanzada) enviará un código de ERROR. Adicionalmente, esta respuesta puede traer un payload extra en el caso de que el cliente haya pedido información del servidor.

Finalmente, el servidor envía el comando res devuelta al cliente con la información pertinente, el resto depende del cliente.

Diseño del Protocolo

Vista General

El protocolo implementado en la aplicación (el cual llamamos Vapor Protocol) es un protocolo basado en caracteres el cual está dividido en dos partes. La primera parte es el protocolo de comandos, este protocolo define la codificación y decodificación de los comandos enviados entre los clientes y el servidor.

La segunda parte es el protocolo de transferencia de archivos. Parecido a como hace FTP, nosotros tenemos un header aparte para la transferencia de archivos entre cliente servidor o servidor cliente.

Esta decisión se tomó conscientemente porque no nos parecía que tuviera sentido definir tres comandos (uno para indicar envío de imagen, otro para indicar que llega una pieza de la imagen y uno último que indica la finalización del envío de la imagen) y hacer que el paquete de envío de una imagen pase por todo el framework de los comandos. Esto causaba que nuestro código se volviera demasiado complicado con casos especiales entre otros, y preferimos apegarnos al principio de diseño KISS de mantener las cosas simples. Además la optimización no es buena y nos ayuda a cumplir con que los comandos mantengan la atomicidad mencionada en la parte de arquitectura.

Protocolo de Comandos

Lo siguiente es la estructura básica del protocolo de comandos:

Petición Comando Largo-Payload Payload

La siguiente tabla define cada entrada:

Input	**Dominio**	**Largo Máximo**	**Descripción**
Petición	{“Req”, “Res”}	3	Indica el tipo de petición de comando. Si se trata de un REQ, entonces es un request del cliente al servidor. Si se trata de un RES, entonces es una response del servidor al cliente.
Comando	“00” al “99”	2	Un código identificador con un comando definido.
Largo-Payload	enteros	4	El largo del payload.
Payload	caracteres	9999 (Largo-Payload)	La información que el comando trae y cree pertinente al caso.

Network Transfer Objects

Esta parte del protocolo trae un payload genérico de caracteres que puede variar enormemente en forma. El payload puede traer un juego, o información de un usuario para hacer un login, puede traer todo tipos de listas como de juegos o de reviews, también puede traer un query de búsqueda de juegos. Esta cantidad de posibilidades abrumadora nos forzó a implementar un sistema de sub-protocolos para el payload de caracteres que sepa cómo codificar y decodificar un cierto payload.

Para ello definimos unos objetos llamados Network Transfer Objects (sigla NTO), los mismos son unos objetos que implementan la siguiente interfaz:

```
public interface INetworkTransferObject<T>
{
    9 references
    string Encode();
    18 references
    T Decode(string toDecode);
    11 references
    void Load(T obj);
}
```

Son objetos que actúan como wrappers de un objeto de tipo T y que cargan con información extra que define cómo se codifica y decodifica un objeto en específico. Para bajar a tierra el concepto, miremos el siguiente ejemplo:

```
public class GameSearchQuery
{
    4 references
    public string Title { get; set; }
    4 references
    public string Genre {get; set;}
    5 references
    public int Score {get; set;}

    1 reference
    public GameSearchQuery(){}
    1 reference
    public GameSearchQuery(string title, string genre, int score)
    {
        Title = title;
        Genre = genre;
        Score = score;
    }
}
```

El código anterior define la clase GameSearchQuery, esta clase representa la búsqueda que podemos hacer de un juego en el sistema. La información pertinente es de Titulo, Genero y un Score promedio. Para poder enviar un GameSearchQuery por la red, tenemos que crear un sub-protocolo que define cómo codificar y decodificar un GameSearchQuery que viene en un payload de un comando:

```
public class GameSearchQueryNetworkTransferObject : INetworkTransferObject<GameSearchQuery>
```

Creamos el NTO e implementamos las funciones de Encode y Decode, para este caso definimos la siguiente estructura:

Largo-Título Título Largo-Genero Genero Largo-Score Score

Todas las entradas son de dominio de caracteres, incluso los largos que vienen en un formato con ceros a la izquierda (por ejemplo, si el largo es 3, entonces viene “0003”), esto nos permite generalizar los métodos de extracción de las entradas. Los largos varían y están definidos por NTO. En este caso, el largo del título de un juego lo limitamos a un largo fijo de 2 (es decir, el título puede tener largo de “00” a “99”), cada entrada varía acordemente por NTO.

Las clases del sistema se asisten de los NTOs a la hora de armar los paquetes para enviar por la red. De esta manera tenemos un sistema que nos permite extender el protocolo indefinidamente sin tener que cambiar código (cumple Open-Closed de SOLID), soportado por una interfaz que el sistema conoce. También es fácil de codificar ya que en la misma clase podemos seguir fácilmente la decodificación teniendo la codificación del objeto en un método justo arriba.

Protocolo de Transferencia de Archivos

Este protocolo se utiliza principalmente para enviar y recibir la imagen de carátula de los juegos, pero podría utilizarse para cualquier tipo de archivo binario. El protocolo de transferencia de archivos se define con la siguiente estructura:

Confirmado Largo-Nombre-Archivo Tamaño-Bytes-Archivo Nombre-Archivo

Input	**Dominio**	**Largo Máximo**	**Descripción**
Confirmado	{“Y”, “N”}	1	Indica si el envío del archivo fue confirmado. Esto es “Y” en los casos felices, pero si algo sale mal del otro lado, el enviador le avisa al receptor con un “N” así no queda esperando por un archivo que nunca va a llegar.
Largo-Nombre-Archivo	entero	2	El largo del nombre del archivo.
Tamaño-Bytes-Archivo	entero	8	El tamaño en bytes del archivo.
Nombre-Archivo	caracteres	99 (Largo-Nombre-Archivo)	El nombre del archivo.

Luego de enviado el comando, el recipiente recibe parte por parte las piezas del archivo, estas partes tienen un tamaño máximo estipulado por el protocolo (32 kB). El recipiente luego arma el archivo en el directorio que tenga seteado en el archivo de configuración.

Vapor Status Message

El comando de respuesta de un servidor viene en un cierto formato que definimos en una especie de NTO especial llamado Vapor Status Message. Funcionalmente es un NTO, pero es especial ya que todas las respuestas del servidor al cliente vendrán en este formato.

La estructura de un Vapor Status Message es la siguiente:
Code Message Extra-Payloads

Input	**Dominio**	**Largo Máximo**	**Descripción**
Code	{“10”,“20”, “40”, “41”, “50”}	2	Código de estado de la petición que vuelve al cliente. Este código le indica al cliente si todo salió bien del lado del servidor o si hubo algún error.
Message	caracteres	9999 (lo que entra en el payload de command)	Un mensaje que puede ser impreso del otro lado.
Extra-Payloads	caracteres	variable, definido por el NTO interno	Payloads extra que puede traer la respuesta que sea de interés al cliente y su petición. Estos payloads extra son NTOs.

Tabla de Comandos Definidos

A continuación, desplegamos una tabla con todos los comandos que el protocolo implementa. Esta tabla fue creada al principio del proyecto y fue muy útil al armar la aplicación. La tabla es exhaustiva y menciona todas las acciones de cada comando.

COMANDO	**NOMBRE**	**CLIENTE**	**SERVIDOR**
01	Login	**Acción:** Envía un nombre de usuario. **Si es rechazado:** Vuelve al login	**Acción:** Loguea al usuario. Si el usuario no existe lo crea y luego lo loguea. **Rechaza si:**

		Si es aceptado: Ingresa a la aplicación. Guarda en su Client Session su nombre de usuario.	Ocurre algo inesperado.
02	Lista de Juegos	**Acción:** Selecciona opción en menú. No envía nada. **Si es rechazado:** Vuelve al menú principal. **Si es aceptado:** Imprime sólo los títulos de los juegos.	**Acción:** Devuelve la lista entera de juegos en la base de datos **Rechaza si:** Ocurre algo inesperado.
03	Adquirir Juego	**Acción:** Envía nombre de usuario e ID de un juego a adquirir. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** No hace nada.	**Acción:** Asocia en la base de datos que el usuario recibido tiene ese juego. **Rechaza si:** Ya tiene el juego o si no existe tal juego.
04	Publicar Juego	**Acción:** Envía los datos de un juego y el nombre usuario. Luego de enviados estos datos, comienza a enviar una carátula. **Si es rechazado:** Vuelve al menú principal. **Si es aceptado:** Menciona que el juego fue publicado. Vuelve al menú principal.	**Acción:** Agrega un juego a la base de datos con los datos existentes. Asocia al usuario como creador. Luego espera a recibir una imagen. Si la imagen no llega, el juego queda publicado pero sin imagen. **Rechaza si:** Ya existe un juego con el mismo título. No proporciono un título.
05	Publicar Review de un Juego	**Acción:** Envía su nombre de usuario, y el ID del juego al que le está haciendo la review, accediendo a	**Acción:** Publica una review asociada al juego enviado y al usuario enviado. Si la review ya existía de ese

		su sesión de usuario. Además una descripción y un puntaje. El puntaje se envía ya cumpliendo el formato **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** Vuelve al menú del juego.	usuario para ese juego, se reemplaza. La razón es para evitar falsos puntajes promedio. **Rechaza si:** No existe el juego.
06	Selección de un juego.	**Acción:** Envía el nombre de un juego. **Si es rechazado:** Vuelve al menú principal. **Si es aceptado:** Accede a un menú del juego. Guarda en su Client Session el nombre del juego al que accedió.	**Acción:** Le devuelve al cliente la información de que existe el juego. **Rechaza si:** No existe el juego.
07	Buscar Juegos	**Acción:** Envía un título, un género y un puntaje. **Si es rechazado:** Vuelve al menú principal. **Si es aceptado:** Muestra la lista de los nombres de los juegos retornados. Esta lista puede ser vacía.	**Acción:** Le devuelve al cliente una lista de juegos que coinciden con todos los parámetros ingresados. El título coincide si contiene un substring de lo enviado. Igualmente con género. El puntaje coincide si es igual o mayor al enviado. **Rechaza si:** Pasa algo inesperado.
08	Modificar un juego	**Acción:** Envía el nombre de usuario, un juego nuevo y el ID del juego. Luego envía una carátula. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:**	**Acción:** Modifica al juego cuyo ID coincide con el seleccionado con los datos nuevos. Recibe una carátula y la reemplaza la carátula ya existente. Si la imagen no llega, el juego queda publicado pero sin imagen. **Rechaza si:** Ya existe un juego con el

		Vuelve al menú del juego.	mismo título. No proporciono un título. No sos el creador del juego.
09	Borrar un juego	**Acción:** Envía el nombre de usuario y el ID del juego a borrar. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** Vuelve al menú del juego.	**Acción:** Borra el juego de la base de datos. Desasocia de todos los usuarios que hayan adquirido el juego. **Rechaza si:** No existe el juego. No proporciono un título. No sos el creador del juego.
10	Ver la calificación de un juego	**Acción:** Envía el ID del juego a consultar. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** Imprime la lista de reviews y el puntaje. El puntaje puede ser NaN y la lista vacía.	**Acción:** Devuelve el promedio de todos los puntajes de todos los reviews del juego asociado. Además retorna una lista con todos los nombres de usuario que dejaron un review. **Rechaza si:** No existe el juego.
11	Ver si soy dueño de un juego	**Acción:** Envía el nombre de usuario y el ID del juego. **Si es rechazado:** Accede a un menú de juego con menos funcionalidades. **Si es aceptado:** Accede a un menú de juego con más funcionalidades.	**Acción:** Devuelve si el usuario es dueño del juego. **Rechaza si:** No existe el juego.
12	Ver detalle de un Review	**Acción:** Envía el ID del juego y el nombre del usuario que hizo el review. **Si es rechazado:** Vuelve al menú del juego.	**Acción:** Le devuelve al cliente el review del usuario con nombre del mismo, el puntaje y la descripción dada. **Rechaza si:** No existe el juego. No existe el review hecho por

		Si es aceptado: Despliega el review con la información pertinente.	ese usuario.
13	Ver detalle de un juego	**Acción:** Envía el ID del juego. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** Despliega toda la información de un juego.	**Acción:** Le devuelve al cliente un detalle exhaustivo del juego. Le proporciona todos los datos, incluyendo la lista de reviews y su puntaje promedio. También pregunta si el usuario quiere descargar la imagen. **Rechaza si:** No existe el juego.
14	Descargar carátula	**Acción:** Envía el ID de un juego. Luego espera a recibir una carátula. **Si es rechazado:** Vuelve al menú del juego. **Si es aceptado:** Vuelve al menú del juego.pertinente.	**Acción:** Comienza el envío de una carátula. **Rechaza si:** No existe el juego. No existe carátula para ese juego.
15	Abandonar la aplicación	**Acción:** Envía que se va a desconectar. Envía el nombre de usuario. Luego se desconecta. **Si es rechazado:** Se desconecta. **Si es aceptado:** Se desconecta.	**Acción:** Hacer el logout del usuario del cliente. **Rechaza si:** Sucede algo inesperado.

Mecanismos de Concurrencia

Debido a cómo está armada la arquitectura del sistema, sabemos que los recursos compartidos que los clientes afectan están todos centralizados en un lugar: Database. Por lo tanto toda la concurrencia del sistema se encuentra implementada en los mecanismos de la base de datos en memoria y en su acceso ya que es el único recurso compartido que logramos identificar del sistema. Para eso hicimos tres cosas:

Primero, la clase Database implementa el patrón Singleton ya que debemos asegurarnos que su instancia sea única en todo el sistema por lado del servidor. Su instanciación se encuentra protegida por un lock:

```
public static Database Instance
{
    get
    {
        if(_instance == null)
        {
            lock(_instanceLock)
            {
                if(_instance == null)
                {
                    _instance = new Database();
                }
            }
        }
        return _instance;
    }
}
```

Este código permite que la instancia esté protegida y que se minimicen los tiempos de acceso con el chequeo externo de si la instancia es null.

Segundo, las listas de objetos que poseen la base de datos deben ser listas Thread-Safe. Para ello definimos una clase llamada ThreadSafeList, la misma es un wrapper alrededor de una lista genérica interna que implementa las funciones de una lista normal pero utilizando un lock internamente en el acceso a esa lista.

```
public class ThreadSafeList<T>
```

```
public void Add(T toAdd)
{
    lock(_lock)
    {
        _internalList.Add(toAdd);
    }
}
```

Un ejemplo de una función de ThreadSafeList y cómo se protege el acceso a la lista interna.

Tercero y último, al acceder a los objetos internos de las listas o al obtener la lista en sí, el código que intenta acceder a esa lista debe de recibir una lista cuyos objetos son todos copias profundas de los originales.

Esto se debe a que si devolvemos una lista con copias llanas, podemos causar problemas de concurrencia ya que un código podría intentar modificar ese objeto lo cual causaría un cambio en el objeto original. Implementando esto, las capas anteriores de lógica están obligadas a ir por los sistemas de concurrencia de la base de datos para impactar sobre ella, protegiendo las listas.

```
public List<T> GetCopyOfInternalList()
{
    List<T> copy = new List<T>();

    lock (_lock)
    {
        _internalList.ForEach(elem => copy.Add(elem.DeepClone<T>()));
    }

    return copy;
}
```

Una cosa que vale la pena mencionar es el mecanismo de concurrencia principal elegido: el lock de un objeto. Nosotros sabemos que esta no es la manera más eficiente de manejar escrituras y lecturas (un algoritmo que consideramos implementar con semáforos), pero también consideramos que esa optimización no se justificaba para el alcance de esta aplicación ya que no tendremos cientos de usuarios trabajando al mismo tiempo. Por lo tanto, decidimos apegarnos al lock que nos asegura la concurrencia y la simpleza a cambio del costo de optimización de lecturas y escrituras.

El día de mañana, si esta aplicación se distribuye, inmediatamente recurriremos (dada esta tecnología de concurrencia) a mecanismos y algoritmos más eficientes (e.g: semáforos y lector-escritor).

Despliegue del Diseño

Server

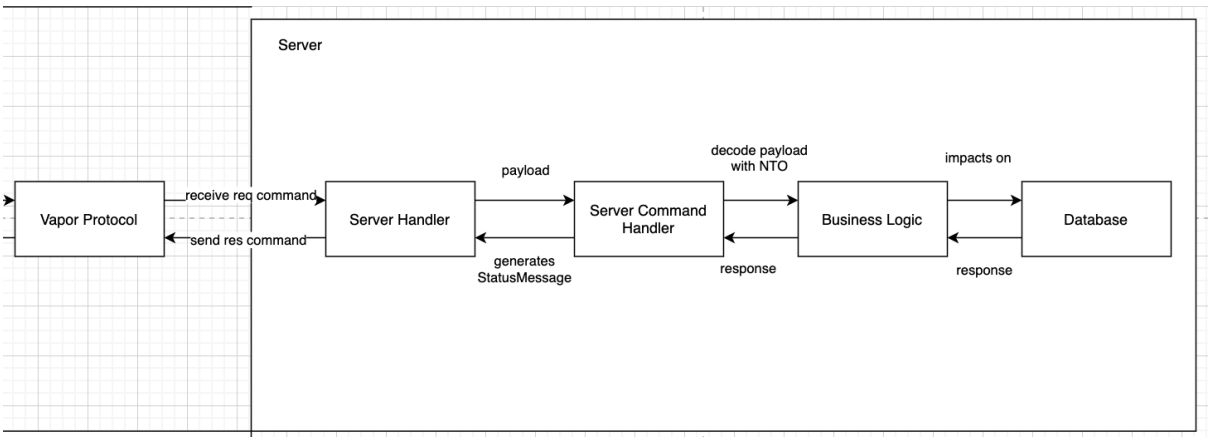
El paquete de Server es el paquete que implementa toda la lógica específica al servidor. Contiene tres threads básicos:

- Menú (Main)
- Listening
- Handle Client

El thread Main lo hacemos llamar Menú porque es aquí donde se corre el menú del servidor. Su única funcionalidad, sin embargo, es el de hacer el cerrado de la aplicación entera y sus conexiones.

El thread Listening es el thread encargado de escuchar por conexiones entrantes de clientes. Una vez que el thread escucha un cliente que quiere conectarse, Listening inicia un Thread nuevo el cual corre la función Handle Client.

Handle Client son todos los threads que manejan las distintas conexiones de los clientes. La función otorgada al thread implementa la siguiente funcionalidad ya mostrada en la parte de Descripción de la Arquitectura:



Todo esto se encuentra implementado en la clase ServerHandler quien administra estos threads.

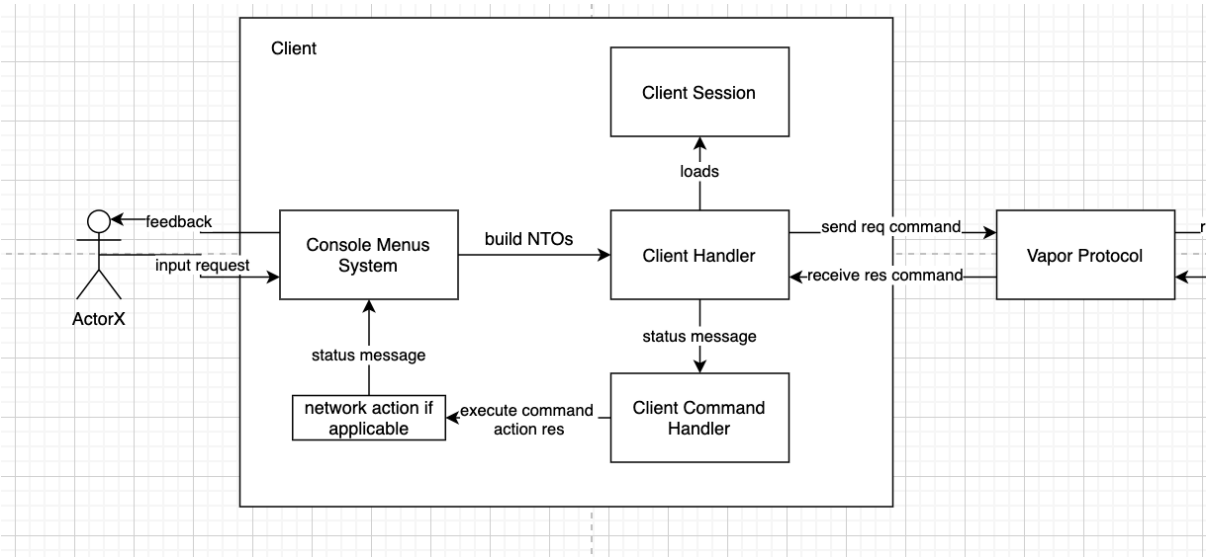
Al cerrar la aplicación, server handler permite mantener la atomicidad de los comandos aplicando una espera si sabe que alguno de sus threads aún está ejecutando un comando:

```
public void CloseServer()
{
    _serverRunning = false;
    if (!AllClientsFinishedExecuting())
    {
        Console.WriteLine("Waiting for clients to finish their commands...");
        System.Threading.Thread.Sleep(MAX_SECONDS_WASTED);
    }
    foreach (TcpClient client in _tcpClients)
    {
        client.Close();
    }
    FakeTcpConnection();
}
```

Esto permite una cerrada conexión de los clientes limpia y decrementa considerablemente la posibilidad de corrupción en la base de datos si, algún día, esta fuera persistida.

Client

Como es de esperar, el paquete de Client implementa la funcionalidad del cliente. El mismo se mantiene en un solo thread que implementa el flujo visto en la parte de Descripción de la Arquitectura:



El mismo se encarga de recibir el input e implementar todo el framework de envio de la información al servidor.

La clase ClientHandler implementa un patrón fachada sobre los comandos, todos sus métodos son mapeables a algún comando definido en la tabla de comandos en la sección de Diseño del Protocolo como se muestra en esta imagen de una parte de la interfaz de ClientHandler:

```
string DeleteGame();

1 reference
string ModifyGame(GameNetworkTransferObject game);

1 reference
string PublishReview(ReviewNetworkTransferObject game);

1 reference
VaporStatusResponse GetGames();

1 reference
VaporStatusResponse GetGameScore();

1 reference
VaporStatusResponse GetGameReview(string username);

1 reference
VaporStatusResponse GetGameDetails();
```

Cada uno de los métodos terminan delegando a un comando, esta delegación se da por los Command Handlers, client tiene el suyo (ClientCommandHandler), éste traduce el código entrante del comando a una instancia del comando requerido y ejecuta el action res:

```
public class ClientCommandHandler : CommandHandler, IClientCommandHandler
{
    1 reference
    public VaporStatusResponse ExecuteCommand(VaporProcessedPacket processedPacket)
    {
        ICommand command = DecideCommand(processedPacket.Command);

        return command.ActionRes(processedPacket.Payload);
    }       Matias González, 2 weeks ago • Created ClientCommandHandler and made a bas
```

```
protected virtual ICommand DecideCommand(string command)
{
    ICommand finalCommand = null;
    switch (command)
    {
        case CommandConstants.COMMAND_LOGIN_CODE:
            finalCommand = new LoginCommand();
            break;
        case CommandConstants.COMMAND_GET_GAMES_CODE:
            finalCommand = new GetGamesCommand();
            break;
        case CommandConstants.COMMAND_SEARCH_GAMES_CODE:
            finalCommand = new SearchGamesCommand();
            break;
```

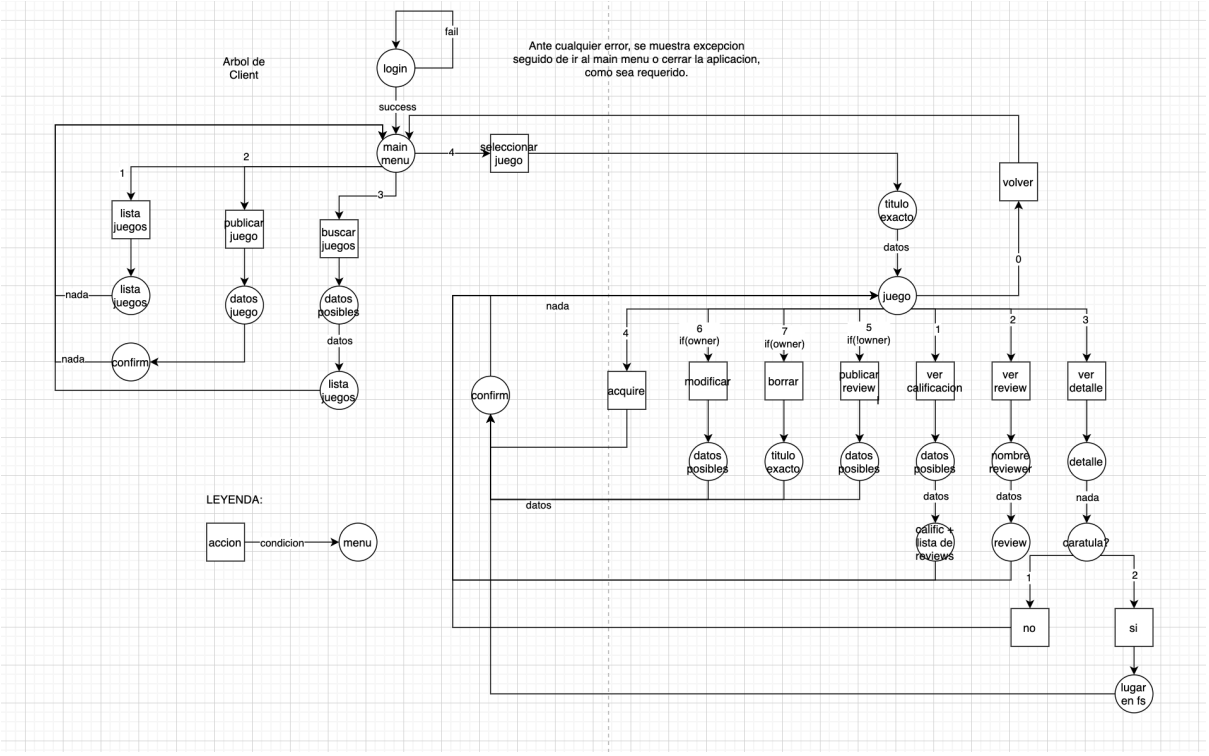
La misma funcionalidad se da para ServerCommandHandler, pero su ejecución del comando difiere un poco por ser contextos distintos.

Por último tenemos ClientSession, esta clase se encarga de guardar en memoria la sesión del cliente. Guarda datos que debe de persistir entre comandos como por ejemplo que usuario se encuentra logueado o cual juego se encuentra seleccionado por el menú en este momento.

Console Menús

El paquete de Console Menús se encarga de ser la UI de la aplicación. Tanto del servidor como del cliente. Su responsabilidad es tomar el input del usuario y transformarlo en objetos utilizables por los Server y Client Handlers, además de imprimir

El servidor lo utiliza solo para exponer la funcionalidad de cerrar las conexiones. Pero el cliente compone un gran árbol de menús que diseñamos previamente:



Este árbol nos ayudó mucho a entender el flow que queríamos con la aplicación y de su construcción.

El sistema se compone de dos interfaces, IMenuManager e IConsoleMenu.

```
public interface IConsoleMenu
{
    1 reference
    IConsoleMenu NextMenu { get; }
    1 reference
    bool RequiresAnswer { get; }
    2 references
    void PrintMenu();

    1 reference
    bool Action(string answer);
}
```

```
public interface IConsoleMenuManager
{
    2 references
    bool Exit { get; }
    2 references
    void ExecuteMenu();
}
```

La clase MenuManager se encarga de manejar el flujo de ejecución de los distintos menús. Las distintas clases que implementen IConsoleMenu serán los menúes en sí. Un loop que se encuentra en el thread de cliente/servidor corre constantemente la función de ejecutar menús de menú manager. Esta función sólo ejecuta el menú del IConsoleMenu y es el que cambia de un menú al otro. También toma input si es indicado por el IConsoleMenu.

Luego IConsoleMenu posee una función que indica que debe imprimir y una accion a un input dado. Cuando creamos una clase de tipo IConsoleMenu, la idea es entender a PrintMenu como la función que imprime el menú principal y luego la acción como la llamada a que hacer con el input dado. Esta acción puede llevarnos a otro menú (con el cual IConsoleMenu settea cambiando la property NextMenu) o puede llamar al ClientHandler, la

fachada al envío de información y ejecución de la funcionalidad. También reacciona a lo que le responda el ClientHandler de la manera que sea adecuada.

Domain, Data Access y Database

Domain es el paquete encargado de definir los objetos que manejamos. Dentro de sí, hay dos subpaquetes, business objects y helper objects. Business Objects representa a los objetos de la lógica de negocio, helper objects son objetos que utilizamos para representar algunos objetos que asisten en el código, un ejemplo de esto es la clase GameSearchQuery quien representa la búsqueda de juegos en la aplicación, pero como no es un objeto persistido, decidimos separarlo de Business Objects. Es utilizado ampliamente por toda la aplicación.

Data Access es una capa que se encarga de acceder a los datos en database. Este paquete ayuda a desacoplar la lógica de negocio y a Database. Se encarga de realizar las operaciones CRUD sobre las listas de la base de datos. Es utilizado solamente por el paquete de business logic.

Database es el paquete que se encarga de guardar los objetos de la lógica de negocio en memoria. Además, es el que realiza control de concurrencia sobre las listas. Más de esto es hablado en la sección de Mecanismos de Concurrencia. Es utilizado solamente por el paquete de DataAccess.

Common

Common es un paquete contenedor que refiere a toda la funcionalidad que es compartida entre Server y Client. Implementa el protocolo, los commands, las clases de configuración y las clases de utilidad

Protocolo y NTO’s

Define todas las clases que implementan el protocolo estipulado, incluyendo las clases que definen aspectos constantes de la aplicación, como los largos máximos estipulados en las tablas presentadas en la parte de Diseño del Protocolo.

La clase VaporProtocol define los métodos de envío de los comandos y de los archivos haciendo uso del NetworkStreamHandler del paquete de NetworkUtilities (mostrado más adelante). VaporProtocol se asiste de VaporHeader el cual define el encabezado del protocolo de comandos y VaporCoverHeader el cual define el encabezado del protocolo de envío de archivos. Las clases de ServerHandler y ClientHandler hacen uso extensivo de VaporProtocol para enviar y recibir la información. Luego está VaporStatusMessage que implementa el mismo VaporStatusMessage que mencionamos en la sección de diseño del protocolo.

Los NTOs son los DTO definidos en la parte de diseño del protocolo. Son clases que contienen información y definen cómo Codificar esa información en un string que luego en VaporProtocol pasaremos a un byte array compatible para el envío por el Network Stream. A su vez también definen cómo decodificar un string cualquiera a una instancia del objeto

que wrappean. Por ejemplo, GameNTO sabe Codificar un Game en un string y cómo decodificar ese string en un objeto de tipo Game.

Por último están las clases estáticas. VaporProtocolHelper implementa clases estáticas que asisten en el protocolo (como el pasado de un int cualquiera a un string con ceros a la izquierda, mencionado en la sección de NTOs en Diseño del Protocolo). StatusCodeConstants define los códigos de error y OK. El enum de ReqResHeader define las constantes de petición del protocolo de comandos. VaporProtocolSpecification implementa todos los largos constantes de los caracteres enviables.

```
public class VaporProtocolSpecification
{
    // COMMAND PROTOCOL
    3 references
    public const int REQ_FIXED_SIZE = 3;
    3 references
    public const int CMD_FIXED_SIZE = 2;
    3 references
    public const int LENGTH_FIXED_SIZE = 4;

    // FILE PROTOCOL
    4 references
    public const int COVER_CONFIRM_FIXED_SIZE = 1;
    4 references
    public const int COVER_FILENAMELENGTH_FIXED_SIZE = 4;
    4 references
    public const int COVER_FILESIZE_FIXED_SIZE = 8;

    // GENERAL PROTOCOL
    3 references
    public const int STATUS_CODE_FIXED_SIZE = 2;
    5 references
    public const int MAX_PACKET_SIZE = 32768;
```

Commands

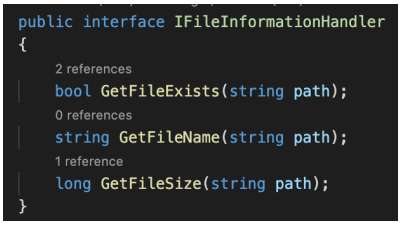
Como mencionamos anteriormente, los Commands son las clases que implementan el sistema de comandos mencionado en la parte de Descripción de la Arquitectura. Todos implementan la interfaz ICommand que se muestra en esa sección.

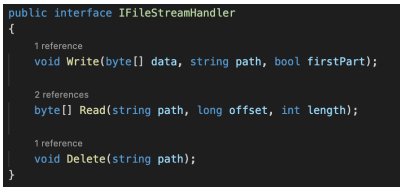
Su responsabilidad es de implementar la funcionalidad descrita en la tabla de comandos en la parte de Diseño del Protocolo, lo hacen mediante un ActionReq (acción Server-Side) y ActionRes (acción Client-Side). Son los responsables de activar a la lógica de negocio por ActionReq y utilizan los NTO para decodificar los mensajes entrantes. Finalmente arman las respuestas con los códigos de estado ya mencionados que se envían al ClientHandler para que sean manejadas de forma adecuada.

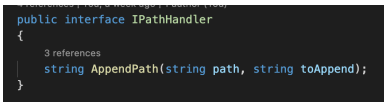
Utilities

Comprenden tres paquetes: FileSystemUtilities, NetworkUtilities y Configuration. Cada uno de ellos proveen clases que wrappean funcionalidades que otorgan utilidad a la aplicación. Son ejemplos del patrón GRASP de Variación Protegida en donde protegemos dependencias del sistema para manejarlas y poder reaccionar en caso del cambio de las mismas.

FileSystemUtilities otorga una interfaz IFileHandler que wrappea las clases que acceden al FileSystem.

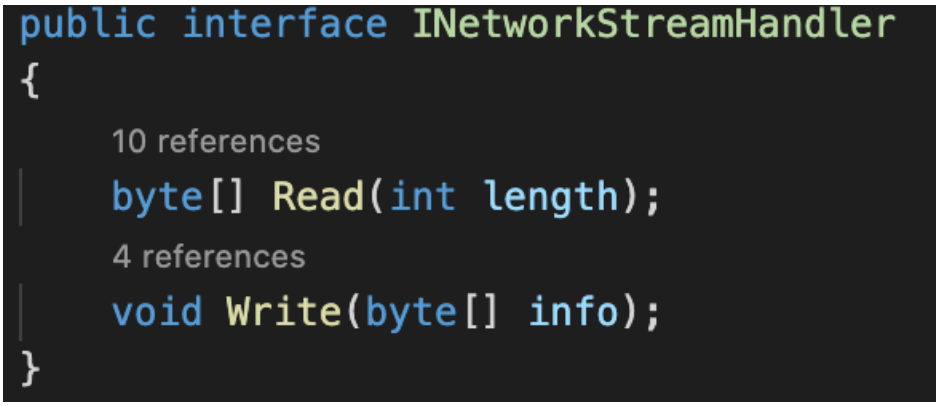






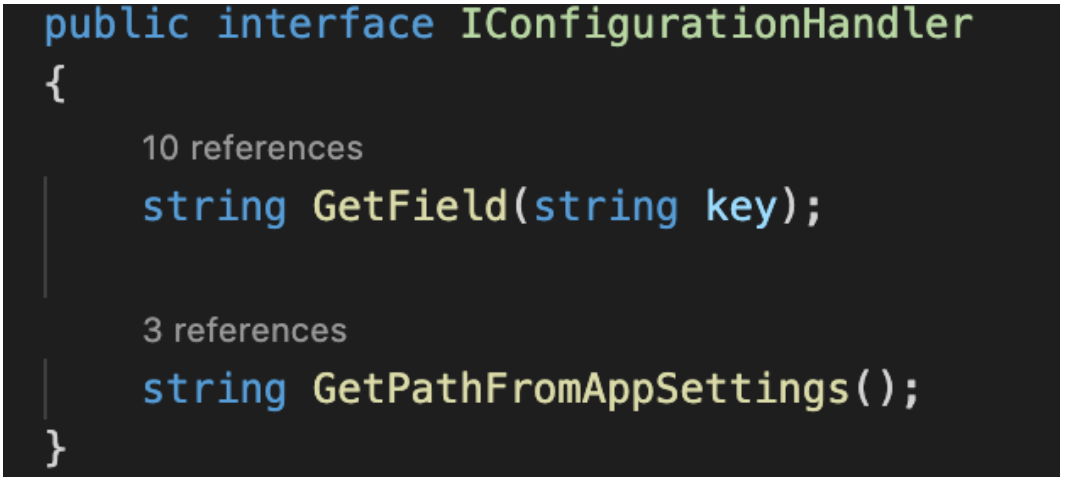
IFileInformationHandler maneja funciones que dan a conocer files en nuestro file system, stream handler maneja el stream de escritura y lectura de los files y path handler maneja al recurso Path de system.

NetworkUtilities otorga interfaces para el manejo del mecanismo de comunicación de network stream. Provee la siguiente interfaz:



La clase que la implementa, NetworkStreamHandler, se encarga de manejar el network stream para enviar y recibir paquetes. Los métodos de envío y recibimiento de la información siguen lo visto en el curso.

Finalmente Configuration se encarga de la clase que maneja la dependencia del acceso al app settings:



Podemos pedir fields del app settings si le proporcionamos una key.

Excepciones

Las excepciones de la aplicación las manejamos por dos tipos:

- Business Exceptions

Son las excepciones originantes de la business logic. Representan errores como que al buscar un User, este no exista o que al intentar agregar un juego se quiera usar el mismo título. Estas excepciones mapean al status code de ERROR_SERVER ya que son errores que pueden surgir solamente en el servidor.

- Connection Exceptions

Son excepciones relacionadas a la conexión de los TcpClients. Manejan casos como cuando un usuario se desconecta de manera imprevista o el servidor se cierra, como también los errores posibles que se puedan dar en el network stream.

El atrapado de excepciones de business se encuentra centralizado en la clase ClientCommandHandler ya que es el que ejecuta todos los comandos. Y como los comandos son el único punto de acceso a la business logic, es allí donde también se traducen al código de error.

```
string response = "";
try
{
    response = command.ActionReq(packet.Payload);
}
catch(BusinessException be)
{
    int statusCode = StatusCodeConstants.ERROR_SERVER;
    response = statusCode.ToString() + be.Message;
}
catch(Exception e)
{
    int statusCode = StatusCodeConstants.ERROR_SERVER;
    response = statusCode.ToString() + $"Something went wrong server-side: {e.Message}";
}
```

El atrapado de excepciones de conexión se encuentran en el NetworkStreamHandler ya que es nuestro mecanismo principal de comunicación entre servidor y cliente. También podemos encontrar atrapado de excepciones de este tipo en el llamado de funciones de VaporProtocol.

Otras Decisiones y Errores Conocidos

Otras Decisiones

Consideraciones con la PK de Juego

Nuestra estructura de juego está compuesta por una PK ID y una AK título. La razón por la cual decidimos que el título no sea la PK es porque es un campo cambiante. Esta decisión nos facilitó muchas cosas.

Nos facilitó la concurrencia, ya que si alguien accede a un juego y su nombre cambia en el camino, al acceder por ID esto no dará un error.

También nos facilitó el manejo de las covers ya que las guardamos en el servidor por ID y no por nombre, entonces al modificar un juego no hay que ir a cambiar el nombre en el file system.

Al Borrar un Juego, sus Reviews quedan en el sistema

Pues nos parece considerable que si un juego es borrado, un usuario aún pueda ver las reviews de ese juego ya no existente. Las reviews podrían considerarse un recuerdo del juego y que tan exitoso fue.

Errores

Como todo sistema, su diseño no es perfecto y al no tener tiempo infinito algunos tropiezos se dieron a conocer demasiado tarde y no los pudimos arreglar. A continuación mencionamos todos los considerados.

Errores de Diseño en Command

Un ejemplo de esto es la respuesta de ActionReq en la interfaz de comando y la existencia de una clase llamada CommandResponse. CommandResponse es, en realidad, el VaporStatusMessage pero del lado del servidor. Nosotros al armar el manejo de comandos no tomamos en cuenta esto y terminamos con un CommandResponse que solo causa problemas y no puede traer ningún tipo de payload extra, complicando la comunicación del Cliente al Servidor. Esto también dio lugar a unas prácticas erróneas en algunos comandos que se contradicen con el diseño e incluso impacta en cosas ajenas como el hecho de que obligamos a los clientes a que proporcionen una carátula al agregar un juego y modificar el juego.

También entendemos que el polimorfismo de ICommand y la existencia del CommandHandler se podría manejar mucho mejor, CommandHandler podemos no necesitarlo y los ICommand podrían manejarse por sí solos. Como también que ICommand no cumple con S de SOLID al tener varias responsabilidades.

Estos errores se mantuvieron ya que su cambio implica un rediseño absoluto de la noción de comando en la aplicación que hubiera tomado demasiado tiempo.

Existe un solo comando que no cumple con nuestra regla de atomicidad, el ver detalles de un juego. Esté activa otro comando de descarga de una carátula y no debería de ser así. Esto se debe a que la carátula se descarga a petición del usuario.

Errores de Diseño con los new

Somos conscientes de que no hacemos un desacoplamiento adecuado con la palabra reservada new. Hubo un intento de desacoplamiento con ConsoleMenus (se puede ver el paquete ConsoleMenusFactory) pero esto no lo pudimos reflejar en el resto de los paquetes. El día de mañana crearemos una factory para cada paquete y los desacoplaremos de la responsabilidad de crear una instancia de una clase.

La Falta de Archivo UML

Para la entrega, nosotros queríamos además otorgar un diagrama de clases UML el cual mostraba todas las clases del sistema. Lamentablemente, por mala gestión del tiempo y por decisión de dejarlo para el final, no pudimos terminarlo a tiempo de una manera que fuera presentable o entendible.

Entendemos que los documentos de diseño existen para comunicar fácilmente los sistemas complejos, y creemos que el resultado apurado que teníamos no iba a cumplir con ese requisito. El aprendizaje está en que tenemos que ir trabajando el UML junto con el sistema y no al final.

Errores de Diseño del Protocolo

Nuestro protocolo de archivos considera los tamaños grandes de los paquetes que pueden entrar y las limitaciones de TCP. Sin embargo, esta consideración no se da en el protocolo de comandos. La consecuencia de esto es que si la aplicación tiene que manejar volúmenes masivos de datos, sean listas enormes de juegos o reviews, simplemente no va a funcionar ya que no consideramos que el paquete pueda venir en partes.

Bugs Conocidos

No se sabe de bugs conocidos en el momento de la creación de este documento.