Universidad ORT Uruguay
Facultad de Ingeniería

Documentación Obligatorio 3
Programación de Redes

Federico Iglesias - 244831
Marcelo Pérez - 227362

2021

Índice general

1 Repositorio

Link al repositorio GitHub con el código fuente (rama master):

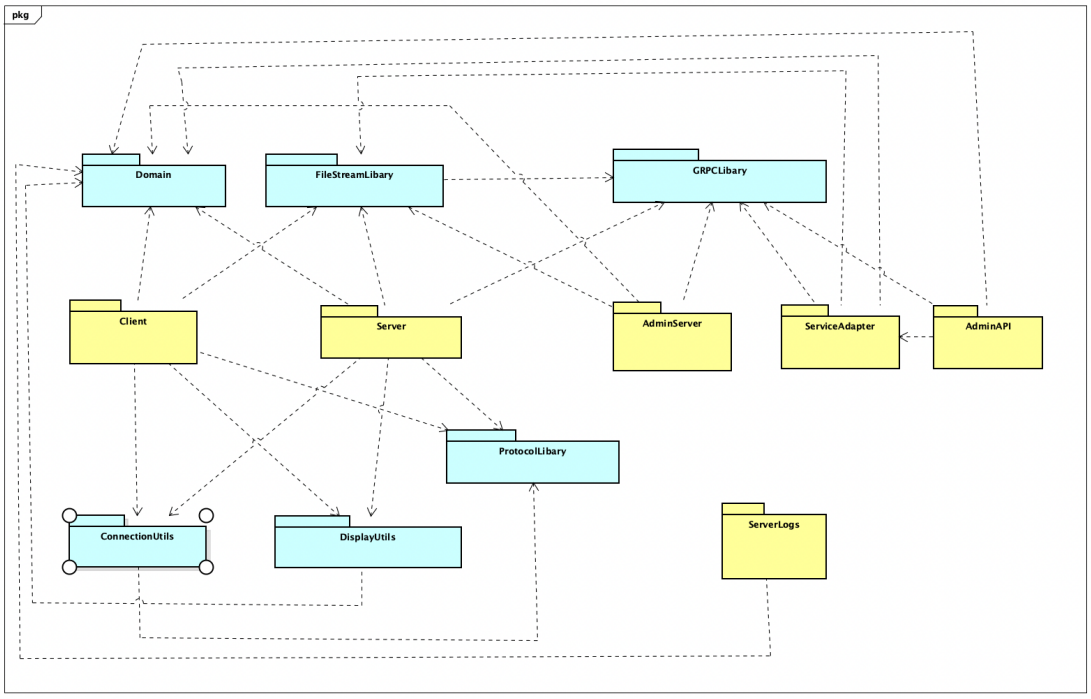
`https://github.com/mperezjodal/Redes-Obl1-Perez-Iglesias.git`

2 Arquitectura

Nuestra solución cuenta con un total de 12 paquetes:

- **Client** (cliente TCP)
- **Server** (servidor TCP)
- **AdminServer** (servidor administrativo implementado con GRPC)
- **AdminAPI** (API que brinda los servicios de AdminServer)
- **ServerLogs** (API que brinda los servicios de servidor de Logs implementado con MOM RabitMQ)
- **GRPCLibary**
- **ServiceAdapter**
- **Domain**
- **ProtocolLibary**
- **ConnectionUtils**
- **DisplayUtils**
- **FileStreamLibary**

Se incluye el diagrama de paquetes:



Client y **Server** corresponden a las aplicaciones de consola de cliente TCP y servidor TCP.

AdminServer corresponde al nuevo servidor administrativo, que se comunica mediante *GRPC*. A su vez, **AdminAPI** contiene la *REST API* que brinda los servicios de *AdminServer*.

También, tenemos **ServerLogs** que contiene el servidor de logs implementado con *MOM RabitMQ*, y al mismo tiempo incluye una *REST API* que permite hacer una búsqueda de estos logs.

GRPCLibary es la librería donde se encuentra el archivo `.proto` que declara los métodos implementados por *ServerAdmin*. A su vez, contiene los distintos *Models* utilizados y una clase para ensamblar los mismos.

ServiceAdapter es la clase intermedia entre *AdminAPI* y *ServerAdmin*, tiene una conexión al servidor *GRPC* y es utilizada por *AdminAPI* para interactuar con *ServerAdmin*.

Como su nombre pretende denotar, el paquete **Domain** declara las clases que serán utilizadas por el sistema: GameSystem, User, Game, Review, y algunas cla-

ses auxiliares como CustomEncoder que se utiliza para parsear los objetos y convertirlos en strings que serán enviados entre los componentes Cliente y Servidor. Además, incluye clases de excepciones personalizadas.

ProtocolLibrary define la clase Header y las constantes que se utilizan para regular el envío de datos entre las aplicaciones TCP, como será explicado detalladamente más adelante. Aquí se definen también los CommandConstants que serán enviados en el header de cada envío de datos y funciones de codificación de información.

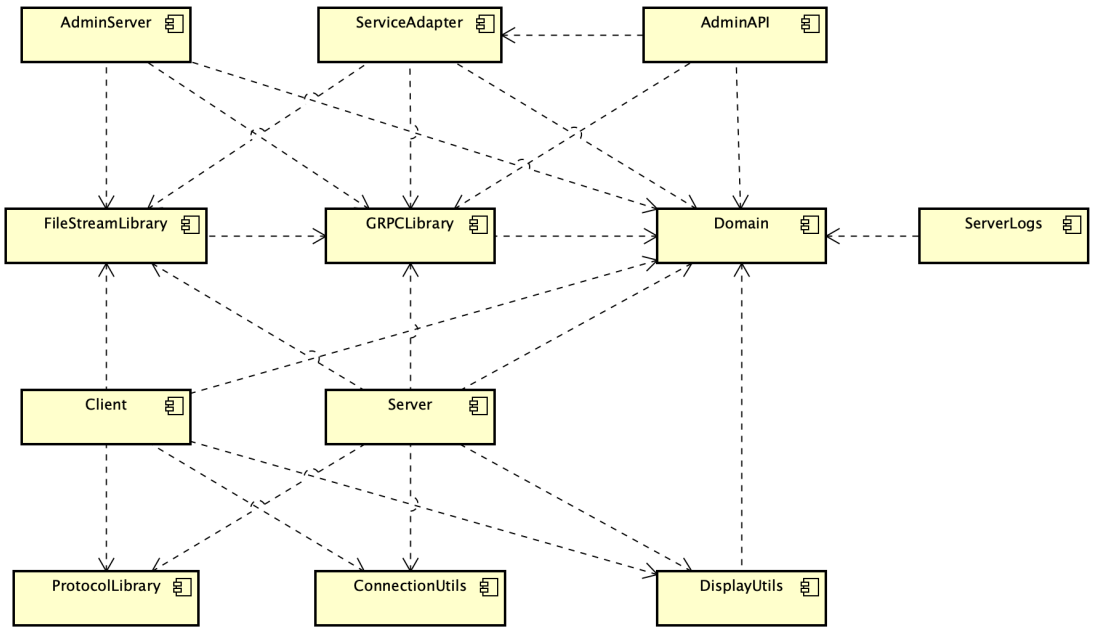
ConnectionUtils se utiliza para extraer métodos de envío y recepción de datos entre las aplicaciones TCP.

DisplayUtils contiene métodos para dialogar por consola con los usuarios de las aplicaciones, tanto *Client* como *Server* lo utilizan mostrar información y recibir inputs.

FileStreamLibrary es la librería encargada del envío de archivos, utilizada para enviar las carátulas de los juegos. Cuenta con funciones para enviar archivos por *TCP* o por *GRPC*.

3 Diseño de los componentes y decisiones de diseño

Se incluye el diagrama de componentes de la solución:



En cuanto al alcance de la aplicación, se ha cumplido con los requerimientos del sistema, permitiendo tener varios clientes conectados al servidor que pueden realizar operaciones respetando la mutua exclusión en todo momento.

El dominio de nuestra solución cuenta con la clase *User*, que utilizan los clientes de consola para conectarse al servidor y mediante la cual podrán adquirir juegos. El login está implementado a modo de permitir manejar concurrencia, y evita que dos usuarios tengan el mismo nombre. Los usuarios no son persistidos, es decir que una vez que se cierre el *AdminServer*, se deberá crear otro usuario para volver a conectarse.

Luego, la clase *Game* contiene los atributos de los juegos y su lista de *Reviews*, que son objetos de otra clase que representa las calificaciones con un rating entre 1 y 10 y un comentario.

La clase *GameSystem* contiene las listas de usuarios, juegos y juegos en modificación, con más detalles a continuación.

También hay otras clases que tienen el fin de facilitar operaciones como las de codificación y decodificación de información, que permiten pasar los objetos a strings que serán enviados entre los componentes.

La aplicación esta diseñada para soportar la conexión de múltiples clientes y sus solicitudes de forma concurrente, para esto se utiliza la clase *Task* que permite manejar múltiples hilos. El *Server* crea un hilo por cada conexión con un cliente.

En cuanto al manejo de la concurrencia en relación a los juegos del sistema, se decidió guardar en el *GameSystem* una lista de juegos en modificación. Entonces, cuando un usuario desea modificar los datos de un juego, solo se le permite si el juego no se encuentra en dicha lista. En ese caso, antes de hacer los cambios se ingresa el juego a la lista de juegos en modificación y una vez finalizadas las modificaciones, se elimina el juego de la lista de modo que quede accesible para operaciones nuevamente.

La lista antedicha no solo evita que dos clientes modifiquen un mismo juego a la vez, sino que también evita que otro usuario elimine o ingrese una calificación de los juegos en modificación, ya que estas operaciones se harán sobre objetos que están siendo modificados. Asimismo, si un cliente está realizando ingresando una calificación de un juego y el mismo es eliminado por otro cliente entre tanto se ingresa esta calificación, la calificación queda sin efecto alguno en el sistema y se retorna el menú principal.

Las validaciones anteriores funcionan para todas las formas de interacción con el sistema. Por ejemplo, si un cliente de consola se encuentra modificando un juego, otro cliente no podrá hacerlo a través de la REST API.

Se utilizaron `locks` en las partes del código del *AdminServer* que ejecutan el get, add, delete, modify sobre la lista de juegos, para evitar que estos se ejecuten a la vez desde distintos hilos de clients.

Otros chequeos del sistema incluyen el no poder ingresar dos veces a un mismo juego, el de no ingresar una calificación de un juego que haya sido eliminado después de empezar pero antes de terminar de ingresar la calificación, el permiso de modificar o eliminar usuarios por parte del *ServerAdmin* únicamente cuando estos no tengan una sesión abierta, etc.

Cuando un usuario opta por ver la lista de juegos y una vez visualizándola se elimina uno de sus juegos, el usuario sigue viendo la misma lista, incluyendo los juegos eliminados, pero no se le permitirá hacer ninguna operación sobre ella y apenas vuelva a seleccionar la opción de ver la lista, esta estará actualizada.

En cuanto al manejo de los covers de los juegos, estos son almacenados directamente en *AdminServer* cuando se ingresan por las aplicaciones de consola o por medio de la *REST API*. Cuando se quiere visualizar los detalles de un juego desde la aplicación de consola de *Server*: en caso de que el juego tenga un cover, este se le envía por *GRPC* y se almacena en el *Server*. Cuando sucede lo propio desde la aplicación de consola de *Client*, en primer lugar el *Server* pide el cover a *AdminServer*, lo guarda y en segundo lugar lo envía por *TCP* al *Client*.

Otra decisión importante de diseño fue en la conexión entre *AdminAPI* y *AdminServer*, para la cual se utilizó una inyección de dependencias, utilizando el *Adapter* del paquete *ServiceAdapter* para manejar, validar y transformar los objetos entre la *REST API* y el *AdminServer*.

A lo largo de la aplicación se implementó el envío de datos haciendo uso de funciones `async Task` (excepto en el caso de RabbitMQ asincrónico), invocándolas con `await` y propagando así los `async Task`.

4 Mecanismos de comunicación de los componentes

4.1 TCP

El intercambio de datos entre el cliente y el servidor se da mediante *TcpListener* y *TcpClient* utilizando un protocolo a medida.

Para el envío de los objetos entre el cliente y el servidor se implementaron métodos que los convierten en strings, para que luego nuestro paquete de protocolo pueda enviarlos correctamente a través de bytes. Estos métodos y sus reversiones se encuentran en las clases correspondientes del dominio.

Los clientes envían solicitudes al servidor y el mismo les responde siguiendo el protocolo:

1 byte para el Header que puede ser una request o response.

2 bytes para el Command, que es un código asociado a un pedido o respuesta.

4 bytes para el largo de los datos.

Datos con largo variable identificado anteriormente.

El protocolo antedicho fue el que consideramos que se ajustaba más a los requisitos del sistema.

A su vez, los juegos pueden tener una carátula, y para implementar el intercambio de archivos entre las partes se utilizaron los *NetworkStreams* de la clase *TcpListener* y *TcpClient*. Las carátulas pueden ser ingresadas cuando el server publica un juego, o cuando un cliente publica o modifica uno, y estas se almacenan en la carpeta del *ServerAdmin*. Cuando el *Client* solicita ver el detalle de un juego se le envía la carátula desde el *Server* a través de un *NetworkStreams* y estas se almacenan en la carpeta *Client*.

Las carátulas pueden ser modificadas como cualquier otro atributo de los juegos, y también pueden haber varios juegos con la misma carátula sin que se generen conflictos.

En cuanto al manejo de errores, tanto cuando se hace *exit* desde el *Server* como desde el *Client*, el sistema maneja diferentes exceptions. Si el *Server* ejecuta `exit` sin ningún cliente conectado, simplemente se cierra el *TcpListener*. En el caso de

existir clientes conectados, cuando el cliente quiera intercambiar datos se lanza una *SocketException* que el sistema maneja y se le avisa al *Client* que el *Server* cerró la conexión.

4.2 GRPC

Entre el *Server* mencionado anteriormente y *AdminServer* (encargado de guardar y mantener las listas con los datos del sistema), la comunicación se da a través de *GRPC*. para eso se creó el paquete *GRPCLibrary* que cuenta con la definición de los *protos* utilizados para el intercambio de datos.

Además de *Server* y *AdminServer*, también el paquete *FileStreamLibrary* contiene una referencia a *GRPCLibrary*, ya que incluimos en él funciones para enviar archivos por *GRPC*, dividiéndolo en partes como se hizo para *TCP*.

Por último, *AdminAPI* también intercambia datos con *AdminServer* a través de *GRPC*, y para esto se creó un paquete *ServiceAdapter* con una interfaz *IAdapter* y una clase *Adapter*, en la cuál se definen métodos que se utilizan para transformar y validar los datos recibidos por los endpoints y posteriormente enviarlos a través de *GRPC* al *AdminServer*.

Adicionalmente, se definieron excepciones personalizadas en el paquete *Domain* para poder comunicar fallas o validaciones no cumplidas a través de GRPC. Por ejemplo, cuando se quiere crear un nuevo usuario pero se encuentra que ya existe uno con el mismo nombre se ejecuta lo siguiente:

```
if (existingUser != null)
{
    return Task.FromException<UserModel>(new AlreadyExistsException("Usuario ya existe"));
}
```

4.3 RabbitMQ

El paquete *AdminServer* tiene una conexión con un canal para publicar mensajes a un servidor asincrónico *RabbitMQ*, y así genera un *LogEntry* cada vez que se ejecuta una acción relevante:

La clase *LogEntry* está definida en el dominio y almacena juego, usuario, fecha y tipo de acción.

Por otro lado, *ServerLogs* consume los logs que se encuentren en el servidor de *RabbitMQ*.

4.4 REST API

El proyecto cuenta con dos paquetes *WebApi*: *AdminAPI* y *ServerLogs*. Estos tienen endpoints desde los cuales se pueden ejecutar una serie de acciones.

AdminAPI cuenta con tres controllers, uno para manejar las operaciones ABM de juegos, otro de usuarios, y un último para la adquisición de juegos por parte de usuarios. Los controllers de ABM cuentan con funciones para los verbos POST (alta), DELETE (baja), PUT (modificación). POST y PUT reciben desde el Body el nuevo objeto, PUT y DELETE reciben desde la ruta el objeto a ser modificado (nombre en caso de usuario y título en caso de juego). El controller encargado de las adquisiciones utiliza los verbos POST, DELETE, para crear o eliminar asociaciones usuario - juego respectivamente; recibe el nombre de usuario por la ruta y un string con el título del juego por el Body.

ServerLogs solamente cuenta con una función de GET, que tiene la opción de recibir parámetros para poder filtrar la búsqueda por juego, usuario, fechas desde y fecha hasta.

Como retorno de todos estos endpoints, además de un mensaje con los datos relevantes se recibe un *HTTP response status code*. Este último es 400 (Bad Request) en caso de que haya un problema con la petición del usuario, 200 (Ok) en caso de éxito, o 202 en particular cuando se crea un juego o usuario nuevo.

En el repositorio de la solución se incluye una colección de Postman con ejemplos de casos de uso.

5 Funcionamiento de la aplicación

5.1 Aplicaciones de consola

El sistema cuenta con dos aplicaciones de consola, un **Client** y un **Server**. A su vez, cuenta con un servidor GRPC: **AdminServer** y 2 REST APIs, **AdminAPI** que es cliente de **AdminServer** y **ServerLogs** que expone los servicios de Logs.

Para utilizar el sistema, siempre tiene que estar corriendo **AdminServer** y el **servidor de rabbitmq**. Tanto **Server** como **AdminAPI** se puden conectar a **AdminServer**.

Por el lado del **Server** se despliega el Menú Principal que brinda **siete** opciones, junto a la posibilidad de cerrar la conexión mediante el ingreso de la palabra *exit*:

1. Ver juegos y detalles.
2. Publicar juego
3. Publicar calificación de un juego
4. Buscar juegos
5. Insertar usuario
6. Modificar usuario
7. Eliminar usuario

Para seleccionar una funcionalidad, el usuario ingresa el número correspondiente a la misma, de ingresar una opción incorrecta se retorna el Menú Principal.

En *Ver juegos y detalles*, se despliega la lista de juegos y se da la opción de ver el detalle de uno de ellos ingresando el nombre del mismo.

En *Publicar juego*, se permite publicar un juego con todos sus datos correspondientes (el nombre del juego debe ser único) en la lista de juegos del sistema. Si el usuario desea puede insertar una carátula para el juego, para esto, cuando la consola lo pida se debe ingresar una ruta de la foto que se quiere asociar.

A su vez, en *Publicar calificación de un juego* se despliega la lista de juegos y se selecciona el nombre del juego que se quiere calificar.

Además, el sistema permite una búsqueda filtrada sobre los juegos es la lista del sistema, los filtros puede ser los siguientes: Título, Calificación y Categoría. Se selecciona la opción por la cual se quiere filtrar y se despliegan los juegos filtrados.

En *Insertar usuario*, se permite realizar la alta de un usuario, siempre y cuando y ya no exista en el sistema.

En *Modificar usuario*, se permite modificar el nombre de usuario, mientras este no tenga una sesión abierta.

En *Eliminar usuario*, se permite realizar la baja de un usuario, siempre y cuando este no tenga una sesión abierta en el sistema.

Además, el servidor soporta la conexión de varios clientes en forma concurrente, que tienen permitido ejecutar acciones en el servidor mediante la aplicación del Client.

La **aplicación del Client** cuenta con un Menú Principal que brinda **ocho** opciones, como también la posibilidad de cerrar la conexión mediante el ingreso de la palabra *logout*:

1. Publicar juego
2. Modificar juego
3. Eliminar juego
4. Buscar juego
5. Calificar juego
6. Adquirir juego
7. Ver juegos adquiridos
8. Ver juegos y detalles

Para seleccionar una, el usuario ingresa el numero correspondiente a la funcionalidad, de ingresar una opción incorrecta se retorna el Menú Principal.

Modificar juego: despliega la lista de juegos y el usuario selecciona el nombre de aquel juego a modificar, luego se ingresan los nuevos datos o ENTER en los campos que no se quieran modificar.

Eliminar Juego: despliega la lista de juegos y el usuario selecciona el nombre de aquel juego a eliminar del sistema.

Adquirir juego: el usuario cliente puede adquirir un juego que se almacenará en su propia lista de juegos que luego podrá ver en *Ver juegos adquiridos*.

Las funcionalidades no explicadas en detalle tienen el mismo funcionamiento que sus pares en la aplicación del Servidor.

Una consideración para ambas aplicaciones es que una vez dentro de cualquier funcionalidad, se permite retornar al menú presionando ENTER.

5.2 REST APIs

AdminAPI expone los siguientes endpoints:

Juegos:

- **Alta Juego**: POST `/api/games`
  Recibe un Juego y lo agrega a la lista.
- **Baja Juego**: DELETE `/api/games/{gameToDelete}`
  Recibe el nombre de un juego y lo elimina de la lista.
- **Modificación Juego**: PUT `/api/games/{titleGameToModify}`
  Recibe el nombre del juego a modificar y el nuevo juego.

Usuarios:

- **Alta Usuario**: POST `/api/users`
  Recibe el nombre del usuario y lo agrega a la lista.
- **Baja Usuario**: DELETE `/api/users/{username}`
  Recibe el nombre del usuario y lo elimina a la lista.
- **Modificación Usuario**: PUT `/api/games/{userToModify}`
  Recibe el nombre del usuario a modificar y el usuario nuevo.

Adquisiciones:

- **Adquirir Juego por parte de Usuario**: POST `/api/acquisitions/{user}`

  Recibe el nombre del usuario y el juego y lo agrega a los juegos adquiridos de ese usuario

- **Desvincular Juego de Usuario**: POST `/api/acquisitions/{user}`

  Recibe el nombre del usuario y el nombre del juego y lo elimina a los juegos adquiridos de ese usuario

Por útlimo, **ServerLogs** expone un único endpoint:

- **Logs**: GET `/api/logs`

  Tiene como parámetros opcionales, nombre de juego, nombre de usuario, fechaDesde y fechaHasta para el filtrado de logs.

En el repositorio de la solución se incluye una colección de Postman con ejemplos de casos de uso.