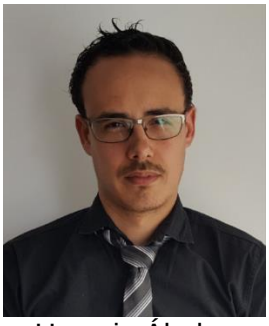
# Universidad Ort

# Programación de Redes

Primera Entrega de Obligatorio


Federico Alonso



Horacio Ábalos


25 de abril de 2021

Link al Repositorio

Contenido

# Declaración de autoría:

Nosotros, Horacio Ábalos y Federico Alonso, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

a) La obra fue producida en su totalidad mientras realizábamos el obligatorio de Programación de Redes;
b) Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
c) Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
d) En la obra, hemos acusado recibo de las ayudas recibidas;
e) Cuando la obra se basa en trabajo realizado juntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
f) Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Alonso Fuentes, Federico Nicolás
25 de abril del 2021

Horacio Abalos
25 de abril del 2021

# Introducción

El presente documento se brinda como especificación técnica y explicación del trabajo realizado por el equipo para la primera entrega del obligatorio de la materia Programación de Redes. La propuesta recibida solicita la creación de un sistema Cliente – Servidor que simule algunas funcionalidades de la red social *Twitter*. Dicha simulación debe contar con una aplicación servidor en la que se manejará la información relacionada a usuarios y sus publicaciones, y otra aplicación que será la contraparte de cliente para dicho servidor y se encargará –fundamentalmente– de la interacción de los usuarios con el sistema.

El objetivo de esta primera entrega es practicar los conocimientos adquiridos en el curso en lo que respecta a manejo de paralelismo, concurrencia y sincronización de *Threads*, manejo de *sockets*, manejo de *streams* e implementación y diseño de un protocolo propio para el envío de información.

# Alcance

La solución propuesta deberá comprender la realización de dos aplicaciones de consola, una para cliente y otra para servidor, las mismas se comunicarán mediante el uso de *sockets* y deberán poseer una comunicación fluida y constante. La aplicación servidor deberá estar preparada para el manejo de múltiples clientes conectados de forma simultánea. Además, se deberá incluir la funcionalidad de transmisión de imágenes desde el cliente hacia el servidor.

Dada la existencia de múltiples clientes, el sistema poseerá una funcionalidad para seguimiento de usuarios. Esta funcionalidad brindará un servicio de notificaciones desde el cliente al servidor que, dada la ocurrencia de una acción desencadenante, proceda al envío de las referidas notificaciones correspondientes a los clientes conectados y suscriptos a la misma.

Esta solución no manejará persistencia de datos, pero sí se brindará una funcionalidad del lado del servidor para una carga de datos de prueba a fin de enriquecer la experiencia de usuario al momento de verificar su funcionamiento.

# Descripción de la Arquitectura

La solución presentada está compuesta por 3 proyectos de Visual Studio:

## Servidor

Proyecto aplicación de consola, el cual oficia de servidor de la solución. Es quien controla los datos de la instancia y acepta múltiples conexiones de aplicaciones clientes, maneja su recurrencia, las notificaciones, y sincroniza a los mismos.

Cuenta con una interfaz de administración de consola, que ofrece la posibilidad a su usuario de realizar tareas de administración propias de la aplicación, como ser ver usuario, bloquearlos, autorizarlos, buscar chips por palabras, entre otras.

Depende del proyecto Servicios, que es una librería de clases que brinda métodos de uso común.

Este proyecto no posee ni expone métodos o clases públicas ya que, dada la arquitectura solicitada por la propuesta, potencialmente puede ser ejecutado en una computadora diferente a la aplicación cliente.

Este proyecto cuenta con un archivo de configuración *App.config*, modificable según la necesidad de los parámetros necesarios para su funcionamiento. En el se pueden configurar, la dirección IP del servidor, el puerto en el que permanecerá recibiendo comunicaciones y el Backlog para registrar las ocurrencias del sistema

## Cliente

Es un proyecto aplicación de consola, el cual oficia de instancia de cliente. Permite a los clientes interactuar con el servidor a través de la solicitud de los datos necesarios para completar la acción según la tarea que se quiera desempeñar. A modo de ejemplo, permite: Alta de usuario, conexión y desconexión al servidor, *login*, y diversas opciones adicionales para usuarios registrados en el sistema y que hayan presentado credenciales validad al momento de ingresar al mismo.

Al igual que *Servidor*, depende del proyecto *Servicios* que oficia de librería de clases con métodos de uso común para fomentar el reúso de código común.

Por otra parte, tampoco posee ni expone métodos o clases públicas, ya que, gracias al protocolo de comunicación que existe entre ambas aplicaciones, puede ser ejecutado –potencialmente– en un entorno diferente al *Servidor*

Cuenta con un archivo de configuración *App.config* que permite la fácil configuración y modificación de la dirección IP del servidor, el puerto del servidor y la dirección IP local.

## Servicios

Proyecto del tipo librería de clases, el cual sirve mediante métodos, clases e interfaces públicas a las otras dos aplicaciones, cliente y servidor.

Este proyecto tiene la particularidad de no depender de otros proyectos ya que el motivo de su existencia es el reúso de código.

## Diagrama de componentes

Si consideramos su objetivo principal, sin especificar que gran parte de los recursos que integran el componente Servicios podrían ser encapsulados aisladamente[1], los componentes de esta solución son fundamentalmente 3 como se muestra en la imagen 1:

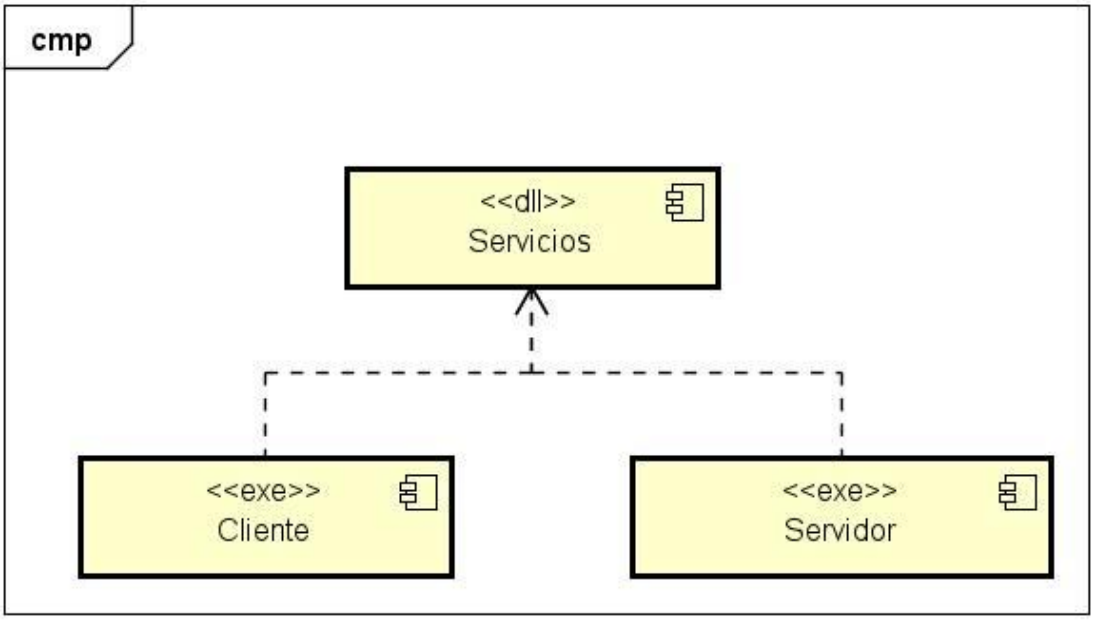



*Ilustración 1 - Diagrama de Componentes*

[1] Podríamos alinearnos con el principio de reúso común de paquetes y encapsular a varias clases de este componente individualmente conforme la funcionalidad que permitan realizar en conjunto. Independientemente de lo anterior, creemos que la solución presentada escaló en complejidad considerando la cantidad de clases final, por lo que entendemos más adecuado utilizar un principio de reúso común de todos estos elementos más flexible y hacerlo en conjunto, intentando así ser transparente en este sentido. Esto no implica que, por motivos de prolijidad y claridad en el proyecto, la agrupación lógica de los elementos está realizada por carpetas.

## Diagrama de Despliegue

Los componentes anteriores pueden ser –potencialmente– desplegados en nodos completamente distintos, es decir, entornos de ejecución completamente diferentes que se comuniquen a través del protocolo TCP/IP.

De esta forma, la imagen 2 muestra cómo podría ser considerado un despliegue maximizando esta posibilidad:

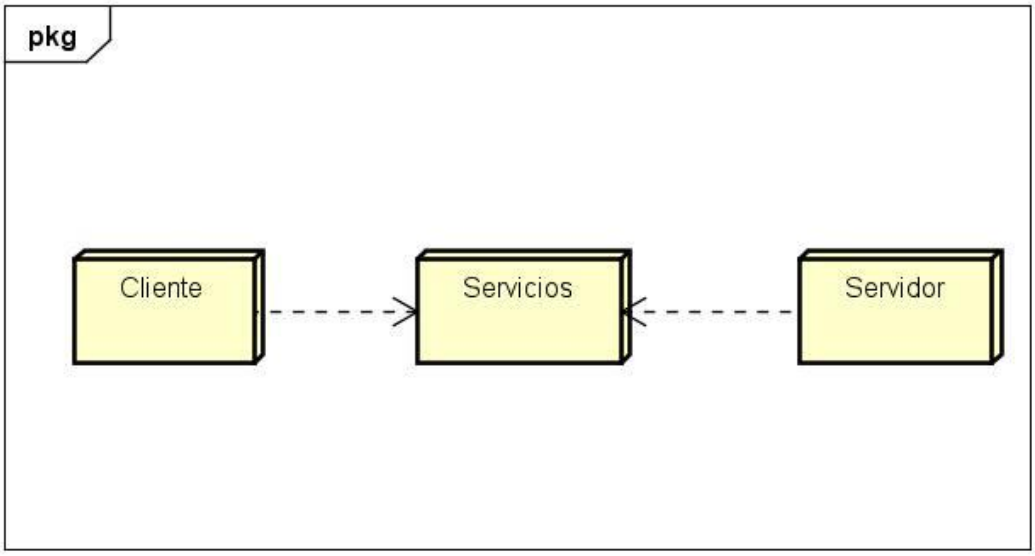



*Ilustración 2 - Diagrama de despliegue*

En el diagrama anterior se observa claramente como los nodos que ejecutan la aplicación cliente y servidor no dependen directamente uno del otro, sino que utilizan el código en común extraído a *Servicios*.

Servicios se presenta como un nodo individual pero también puede ser propio tanto del componente *Cliente* como *Servidor* –es decir, repetir este código en ambos componentes– de manera que sean solamente 2: *Cliente* y *Servidor*.

Alternativamente, *Servicios* puede ser incluido en ambos componentes utilizando *reflexion* o consumiendo estos servicios de alguna otra manera.

## Diagrama de paquetes

A continuación, la imagen 3, ilustra internamente cada uno de los paquetes presentados.


*Ilustración 3 - Diagrama de paquetes*

# Componentes de la Solución

## Aplicación Cliente

La imagen 4 introduce una referencia de las dependencias internas del componente Cliente.

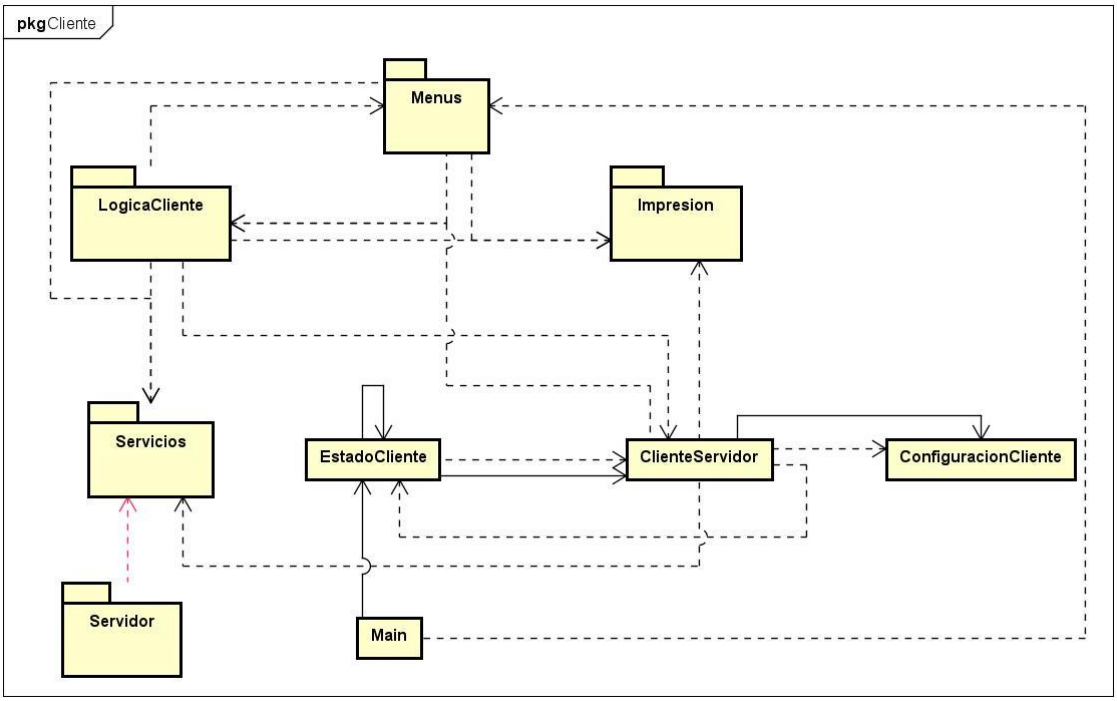



*Ilustración 4 - Cliente, Diagrama de clases*

Por motivos de simplificación, consideramos relevante interpretar cada una de las carpetas internas del paquete como un sub-paquete utilizado por una clase interna, de este modo se abstrae la presentación de dependencias que de otra manera dificultan el entendimiento.

Podemos observar que la dependencia interna, aunque se mantiene dentro de lo esperaba, es bastante elevada, por lo tanto, estamos ante uno de los puntos donde podríamos evolucionar el desacople introduciendo interfaces en cada una de las divisiones lógicas. Esto no fue realizado para esta primera entrega por motivos exclusivos de tiempo, ya que consideramos que es una gran posibilidad de evolución para la siguiente entrega. Independientemente de lo anterior, los cambios a realizar no implicarían muchas horas de esfuerzo y está dentro del alcance que esperamos para la siguiente entrega: permitir la extensibilidad de alguna función.

Retomando la presentación de la arquitectura, entendemos que la simplificación realizada al presentar la solución encapsulando las dependencias en carpetas conlleva ocultar cierta información, por lo tanto, presentamos una imagen de cada una de las divisiones lógicas internas del paquete[2][3].

[2] Es pertinente la misma información para la presentación del componente Servidor.

[3] La presentación del componente servicios se realiza de manera solamente genérica sin la visión interna de cada una de las divisiones lógicas.

Impresion

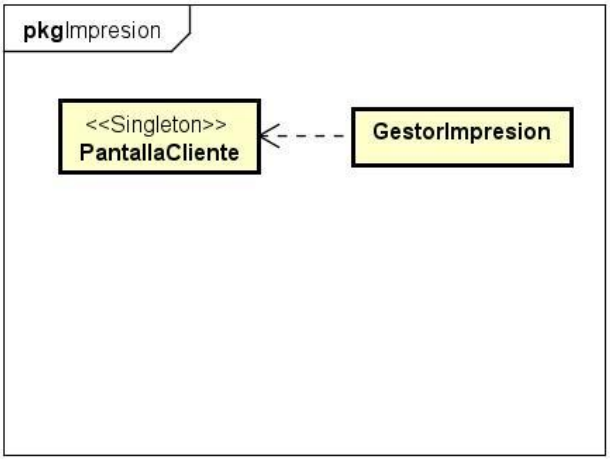



*Ilustración 5 - Impresion, Diagrama de clases*

Esta agrupación lógica tan simple encierra la idea de considerar la impresión en pantalla como una interfaz de usuario por la cual los hilos concurrentes estarán compitiendo. De este modo fue conveniente que la *PantallaCliente* tenga la responsabilidad de imprimir y leer de consola. Al seguir el patrón *Singleton* solamente existe un objeto con esta capacidad, instancia sobre la cual los hilos tendrán mutua exclusión en su acceso.[4]

Para gestionar el acceso a la clase desde diferentes lugares, se creó la clase *GestorImpresion* que tiene como responsabilidad fundamental el acceso a la instancia y a métodos de escritura y lectura.

[4] Existe una rama en el repositorio llamada *RetiroAsociaciónFotoPerfilInicial* en la que se utilizan Monitores para el acceso a la pantalla. Esta solución no permitió solucionar un problema de acceso al recurso generado por el hecho del menú estar esperando un ingreso por consola. Por este motivo decidimos implementar una solución que se explica mas adelante.

## LogicaCliente

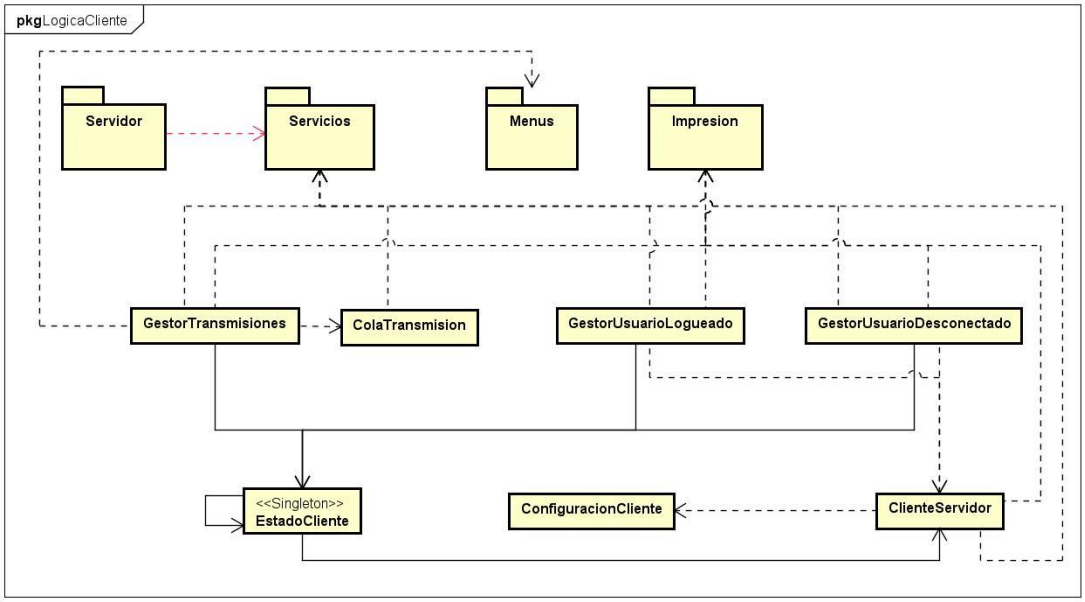



*Ilustración 6 - LogicaCliente, Diagrama de clases*

La imagen 6 muestra cómo desde la parte central del componente Cliente, no existe relación alguna con el componente Servidor, reforzando el concepto de potenciales entornos de ejecución agnósticos uno del otro.

## Menus

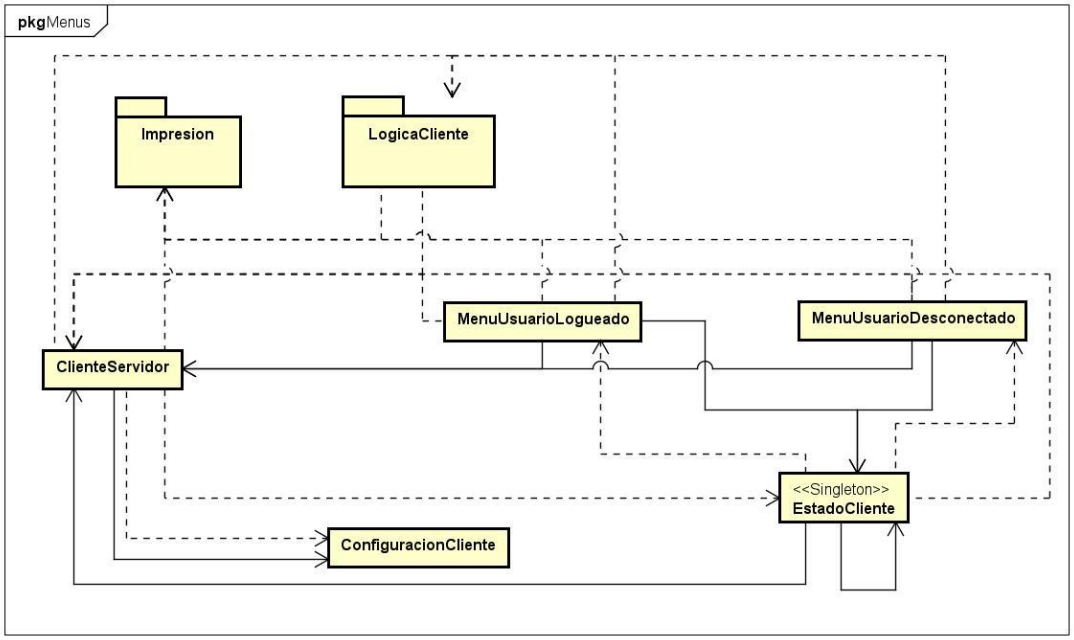



*Ilustración 7 - Menus, Diagrama de clases*

La imagen 7 tiene por elementos centrales a ambos menús de usuario: *MenuUsuarioLogueado* y *MenuUsusarioDesconectado.*

Dado lo nemotécnico de sus nombres no es necesario mencionar su función principal, pero sí que cada uno tiene responsabilidades complementarias a las del otro. Para completar las acciones ofrecidas, ambos menús delegan parte del trabajo a gestores (en este caso encapsulados en la carpeta *LogicaCliente*), y cuando es necesario imprimir en pantalla se utiliza la asociación anterior obteniendo el acceso al objeto singleton *PantallaCliente* y operando con él a través del *GestorImpresion* (encapsulado por la carpeta *Impresion*).

Ambos menús tienen acceso a un objeto también singleton *EstadoCliente* utilizado para llevar el seguimiento del estado del cliente y guardar cierta información relevante para el funcionamiento de la aplicación; por ejemplo, la clase *ClienteSeervidor* que contiene todo lo necesario para establecer la comunicación con el servidor. De esta forma, se consigue la delegación por composición resultando en una solución más entendible.

## Aplicación Servidor

La imagen 8 introduce una referencia de las dependencias internas del componente Servidor.

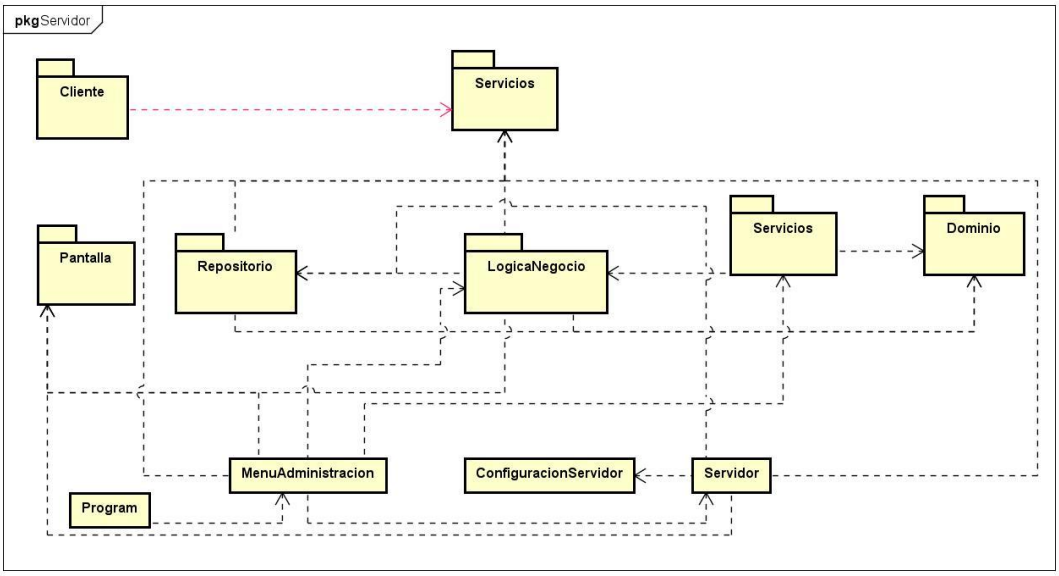



*Ilustración 8 - Servidor, Diagrama de clases*

La imagen anterior muestra un componente con un nivel de acoplamiento semejante al de la aplicación Cliente.

Podemos realizar casi todas las mismas observaciones que fueron realizadas para el componente anterior. Especialmente aquello referido a la inclusión de interfaces a cada una de las divisiones lógicas para estar mas alineados con el principio de diseño OCP. De acuerdo con lo ya expuesto, ambos integrantes consideramos este aspecto constituye un elemento fundamental a mejorar.

Desde la perspectiva del servidor, se observa como el cliente solamente comparte referencias con Servicios y no se acopla directamente a Servidor.

## Domino

De acuerdo con la imagen 9, Domino solamente constituye un encapsulamiento de las clases más representativas de la solución para un mejor entendimiento.

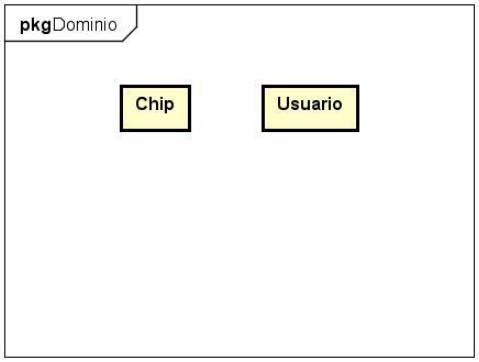



*Ilustración 9 - Dominio, Diagrama de clases*

Se observan dos clases sin dependencia ni relacionamiento, esto se debe a que son el centro de la solución y no poseen referencias a ninguna otra. La simpleza de esta carpeta está sustentada en la intención de no generar clases del dominio por lo que en esta entrega no tiene relevancia.

Hubiese sido posible generar una clase Foto para encapsular la información de los archivos, pero, en primera instancia, preferimos solamente manejar la metadata necesaria para la transferencia del archivo en una clase del componente Servicios. Es altamente probable que una modificación de los requerimientos que solicite almacenar más información (por ejemplo, considerar la fecha de carga de la foto), derive en una clase de dominio para dicho componente.

## LogicaNegocio

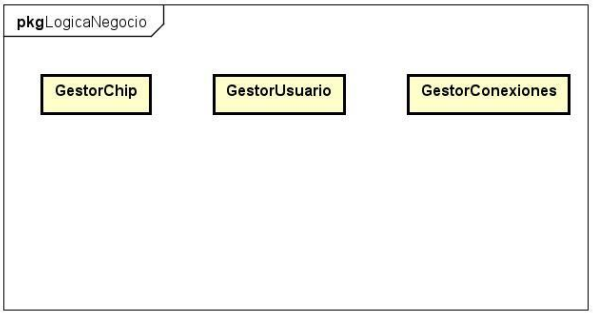



*Ilustración 10 - LogicaNegocio, Diagrama de clases*

El objetivo de incluir la imagen 10 en esta presentación no es mostrar las dependencias de las clases que componen esta agrupación con respecto a las del exterior, aunque exista una abstracción que las encapsule, en la imagen 8 se aprecian estas dependencias y son compartidas por todas ellas.

Quisimos incluirla para mostrar como las lógicas de cada uno de los elementos que consideramos relevantes en esta solución para orquestar su funcionamiento se mantienen sin referenciarse. Esto indica que las responsabilidades de la lógica fueron divididas en línea con la alta cohesión.

## Repositorio

En la misma línea de lo anterior, y respetando la delegación del acceso a memoria, cada uno de los gestores tiene que obtener la instancia correspondiente de del repositorio para completar la acción de guardar un objeto en memoria (dinámica). La imagen 11 muestra como los repositorios también se mantienen sin referenciarse.

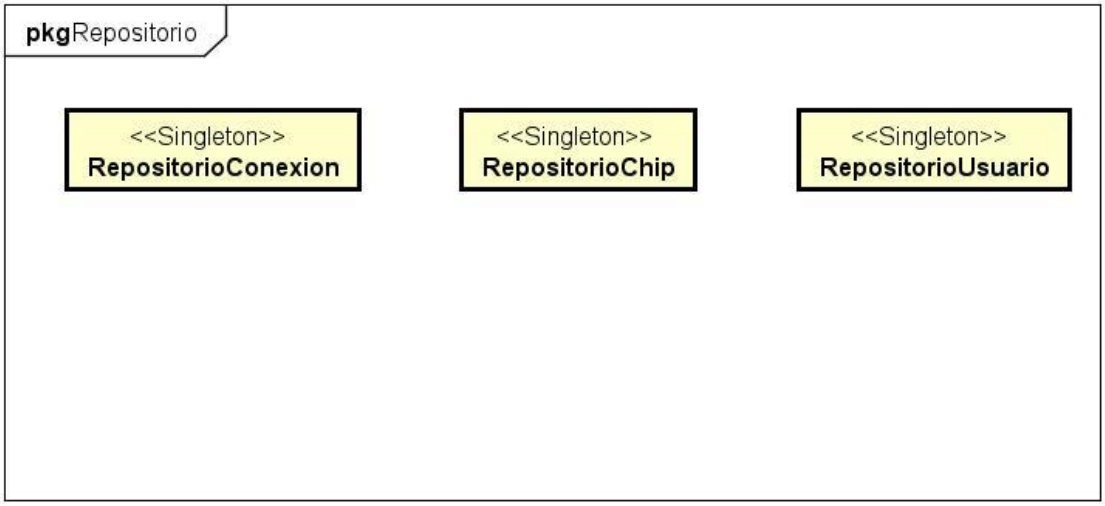



*Ilustración 11 - Repositorio, Diagrama de clases*

Dada la concurrencia de los diferentes hilos de ejecución, cada uno de los repositorios debe ser *singleton thread safe* (al igual que todos los otros elementos *singleton* utilizados) para garantizar que no se produzca un problema en la escritura de la memoria. Cada uno de los repositorios guarda una instancia de un objeto *CandadoInstancia,* sobre el cual se realiza un *lock* al momento de acceder a ella.

## Pantalla

La imagen 12 muestra una implementación análoga a la realizada para el componente cliente expuesta en la imagen 5:

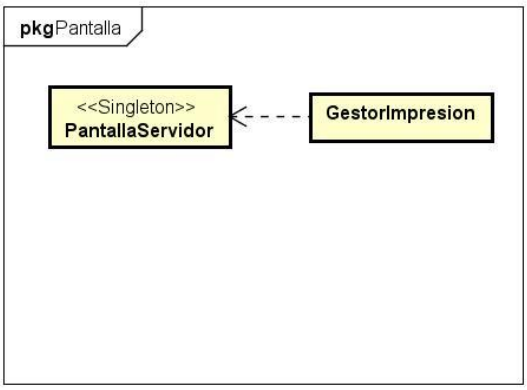



*Ilustración 12 - Pantalla, Diagrama de clases*

## Librería Servicios

La imagen 13 muestra la relación de las clases que integran el componente Servicios. Aunque parezca un componente con un alto acoplamiento, es altamente probable que las dependencias entre estos se segreguen a interfaces, que se agrupen siguiendo los principios de reúso y clausura común, para posteriormente extraerse a proyectos diferentes que estén basados en el protocolo de comunicación o en algún otro servicio brindado.

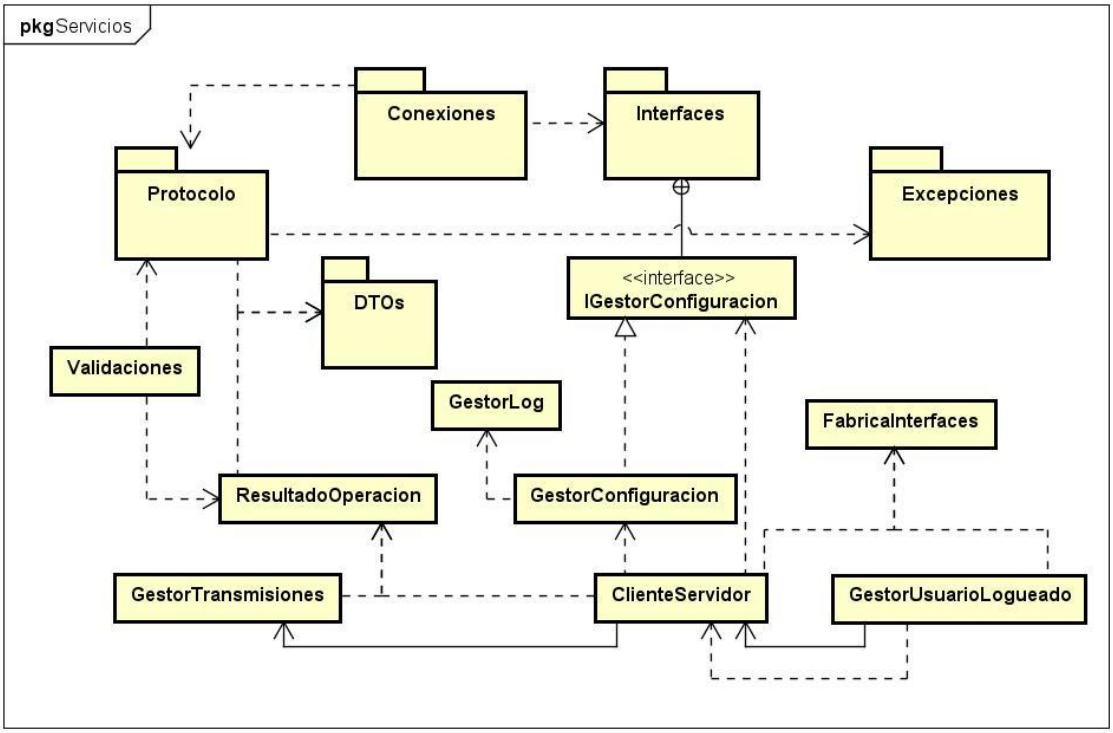



*Ilustración 13 - Servicios, Diagrama de clases*

Considerando que Cliente y Servidor dependen de este paquete integralmente y que desde este paquete no existen referencias hacia ellos, decidimos no incluirlos para mantener el diagrama lo más minimalista posible.

## Decisiones de Diseño Tomadas

Dado que no era un requerimiento explicito, se decide manejar la *persistencia*[5] de la instancia en el paquete de repositorios de la aplicación servidor. Cada uno de los repositorios utilizado para centralizar la responsabilidad del acesso a datos, sigue el patrón *singleton* a efecto de controlar que exista una sola instancia de ellos y que a su vez sea *threat safe*. Para esto último aplicamos un bloqueo de candado sobre el método obtener instancia.

Todos los repositorios de la solución cuentan con las listas de objetos necesarias para la simulación de persistencia implementada. El acceso a dichas listas se realiza utilizando objetos sobre los cuales se realiza un *lock* garantizando así la mutua exclusión; es decir, bloqueándolas una vez alguien accedió con un candado su uso.

Cada una de las aplicaciones cliente que se ejecute se conectan utilizando un hilo de ejecución paralelo al hilo principal, que es el que mantiene el servicio. A modo de ejemplo visual véase: *Ilustración 14 - EncenderServicio, Diagrama de secuencia*.

En la aplicación servidor mantuvimos el estado en la clase *EstadoConexion*. Para garantizar la consistencia de los datos optamos por aplicar el patrón *singleton* en su versión *thread safe*. La responsabilidad de esta calse es guardar el socket del cliente, el usuario conectado, el candado de consola y otros métodos necesarios. Además, por composición delega el envio de los archivos al *SocketHelp*. Entendemos que tanto la conexión como la clase *SocketHelp* son puntos calientes donde la solución debe permitir extensibilidad y evolución, tanto sea para conseguir conectarnos a través de otro mecanismo como para modificar el mecanismo de conexión. Por este motivo decidimos segregar el acoplamiento con interfaces

Experimentamos ciertas dificultades con la impresión en pantalla, por este motivo realizamos algunas implementaciones que no tuvieron resultado (todas pueden ser encontrarse en el repositorio). En líneas más generales, utilizamos un candado para la consola del lado del cliente, pero vimos que se superponían comunicaciones y algunas de ellas no era conveniente cortarlas durante su impresión en pantalla, por esto decidimos realizar un candado de consola que se mantiene en la clase *EstadoConexion* y al que acceden todos los que deseen escribir en la consola, siendo ésta el objeto de mutua exclusión.

Decidimos extraer la mayor cantidad de código no propio de la lógica a la librería de servicios. De esta manera, extrajimos entre otras cosas: métodos de conexión, manejo de sockets y código repetido tanto en cliente como servidor. Todo lo extraído, a efecto de mantener ISP, lo segregamos a través de interfaces. Suponemos que dichos métodos cambiarán en próximas entregas, por lo que se crea una interfaz para ellos. Independientemente de lo anterior, entendemos razonable el nivel de abstracción de dicho componente, aunque podría ser elevado considerablemente.

[5] Este trabajo no incluye persistencia en memoria dura, sino que utiliza memoria dinámica, la utilización de la palabra *persistencia* se indica en cursiva para considerar un significado no literal

Entendemos pertinente crear una fábrica de interfaces, esto se deriva de la intención de modificar la menor cantidad de código posible de las próximas entregas. De esta manera será posible utilizar otra realización de la interfaz simplemente retronando una clase concreta diferente que la implemente.

Creemos que lo mejor para este tipo de trabajos es la implementación del envío de objetos entre cliente y servidor mediante DTOs. En este sentido, se incluye en el componente Servicios una carpeta con una agrupación lógica de los mismos. Así, el cliente es agnóstico del dominio de la solución, y solamente recibe y envía una representación simplificada de la realidad que brinda lo necesario para que su correcto funcionamiento.

En la aplicación cliente se manejan dos hilos de ejecución, uno que queda a la escucha de nuevas transmisiones y otro que interactúa con el usuario. La forma de sincronizar ambos hilos es a través de una cola de transmisiones (clase *ColaTransmision*). En esta implementación, el hilo que escucha va encolando las transmisiones recibidas, mientras que el que interactúa con el usuario va gestionando dichas transmisiones de forma sincrónica. Siendo la zona de mutua exclusión entre ambos hilos la lista de transmisiones que se encuentra en la clase anteriormente nombrada.

Inicialmente, la aplicación cliente imprimía en pantalla toda la información recibida del servidor; sin embargo, posteriormente se verificó que, de la información recibida, cierta parte no brindaba nada interesante para el usuario, sino que eran datos recibidos de la otra aplicación o mensajes de error que no tenían nada útil para un usuario. Por este motivo, se decide realizar un gestor de logs. En su versión actual, la aplicación escribe en un archivo log.txt que se guarda en el mismo directorio de la aplicación y muestra información ampliada de los errores, permitiendo que la impresión en pantalla sea más limpia y consiguiendo mensajes de error simplificados a efecto de facilitar el entendimiento al usuario.

El método empleado para sincronizar el usuario logueado con el no logueado es esperar la respuesta del servidor, por lo que se realiza mediante la función:

```
private void EsperarRespuestaLogin()
{
    int contador = 0;
    while(contador < 1000 && estadoCliente._usuario == null)
    {
        GestorTransmisiones.GestionarTodasLasTransmisiones();
        Thread.Sleep(50);
        contador+=50;
    }
    if(estadoCliente._usuario == null)
    {
        GestorImpresion.Imprimir("El login no resultó satisfactorio.\n");
        Menu();
    }
}
```

En la presente función se espera hasta aproximadamente 1 segundo por el usuario para que se autentique contra la aplicación servidor y recibir la respuesta, en caso de no poder, vuelve a mostrar el menú correspondiente al usuario deslogueado. Lo anterior ocurre ya que cuando se realiza el método de *Login* se debe salir del bucle de mostrar el menú de usuario deslogueado, para postrar el de usuario logueado. Ese proceso de espera es el que sincroniza al cliente y al servidor.

Para la respuesta de los chips se decide implementar el *Patrón Composite* en ellos, de tal manera que los mismos chips funcionan como chip y como respuesta. Para ello un chip, estando un chip “respuesta” compuesto por un chip padre. La imagen 14 es un bosquejo de este patrón:

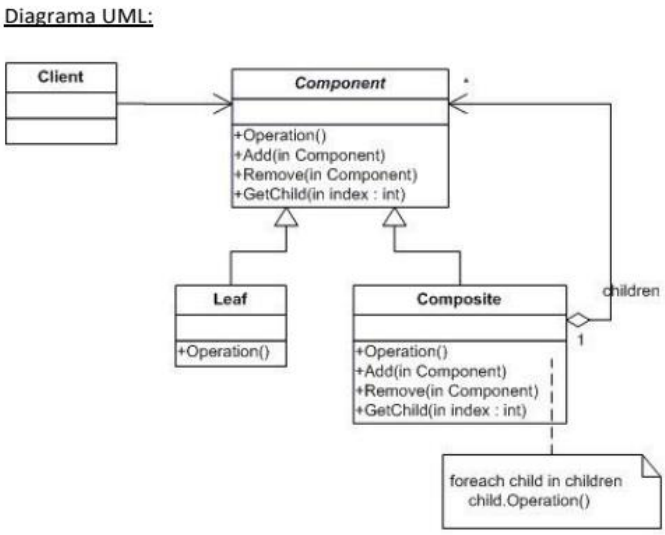



*Ilustración 15 - Composite, Diagrama de clases*

Para los resultados de las transacciones lógicas se decide utilizar un objeto de tipo *RespuestaOperacion*. Este objeto agrupa cierta información relevante para la toma de decisiones a nivel de código; por ejemplo:

**- mensaje** para retornar con información adicional en el caso de un error;
**- código** (intentamos utilizar el estándar de HTTP);
**- tipo**, para automatizar el pasaje a objetos de los datos; y,
**- entidad**, para permitir el pasaje de un objeto genérico.

Con este objeto conseguimos estandarizar el manejo del retorno en la lógica de la solución. Entendemos que la clase en sí está altamente acoplada, pero, a su vez, es una clase que deseamos que se mantenga sin modificaciones que, en caso de ser necesarias, simplemente permite la extensión agregando la *property* correspondiente.

Se decide implementar un pequeño manejo de excepciones relacionada con el envío y recepción de archivos. La intención principal es canalizar en 2 tipos de excepciones todo lo que debe ser soportable por el sistema, estas clases son: *ExcepcionEnManejadorFileStram* y *ExcepcionEnLecturaDeArchivoDuranteEnvio.* Capturando estas excepciones se logra capturar la siguiente lista[6]:

[6] La lista no se presenta en el orden en que las excepciones son capturadas en código, sino que meramente ilustrativa

- `FileNotFoundException`
- `UnauthorizedAccessException`
- `DirectoryNotFoundException`
- `PathTooLongException`
- `FileLoadException`
- `IOException`
- `ArgumentNullException`
- `ArgumentException`
- `NotSupportedException`

De manera complementaria, se utiliza una excepción particular, *ExcepcionEnLecturaDeArchivoDuranteEnvio*. Esta excepción solamente es lanzada en el momento en que la variable *leido sea menor o igual a 0*, lo que representa un error fatal en la comunicación. Al ser lanzada solamente cuando se cumple con esta condición, solo se captura en el menú principal de usuario *logueado* generando la desconexión del servidor.

# Mecanismo de Comunicación entre los Componentes

## Diagramas de secuencia e interacción

### Encender Servicio

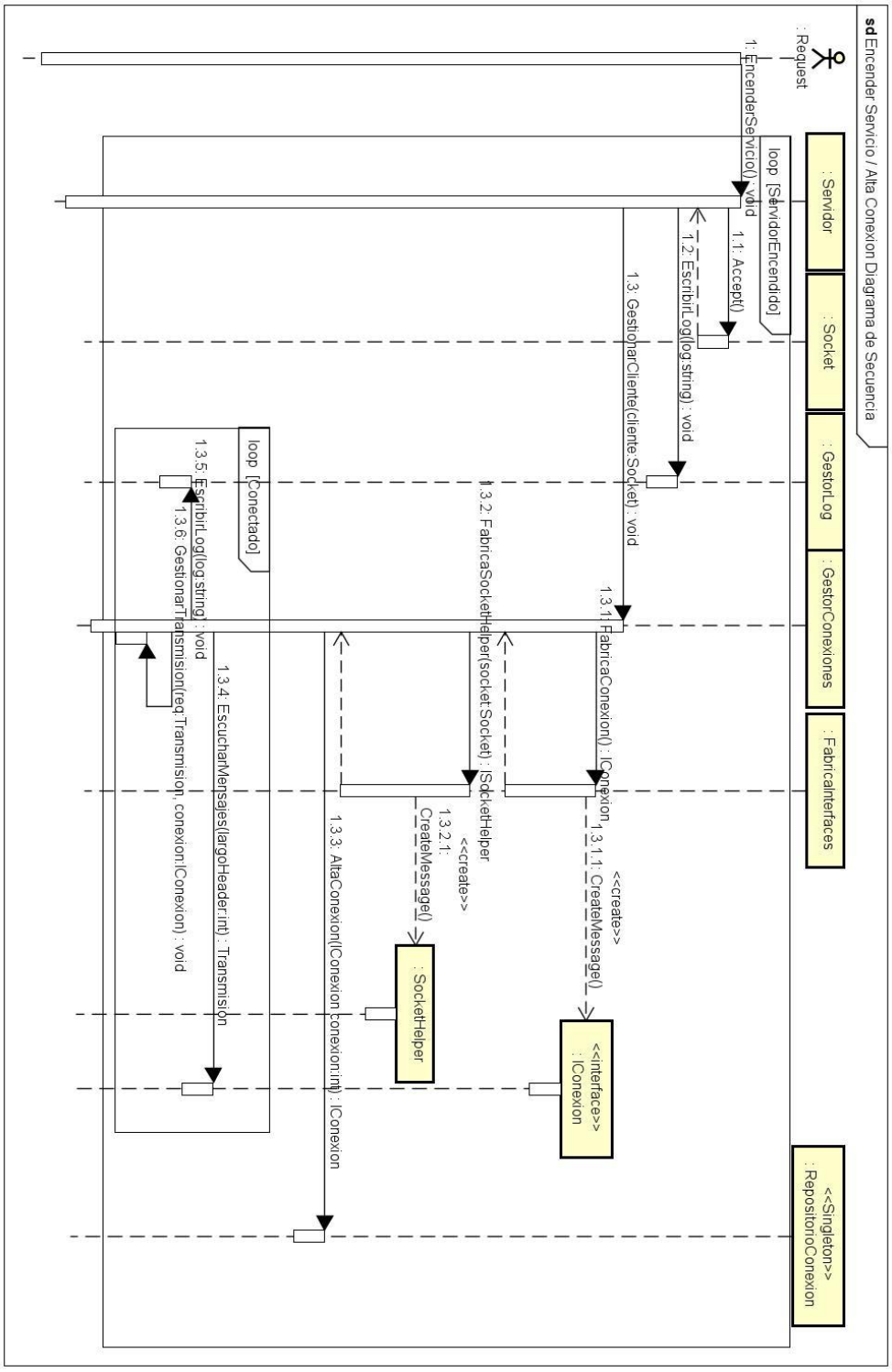



*Ilustración 16 - EncenderServicio, Diagrama de secuencia*

Mostrar Menú Cliente *Deslogueado*

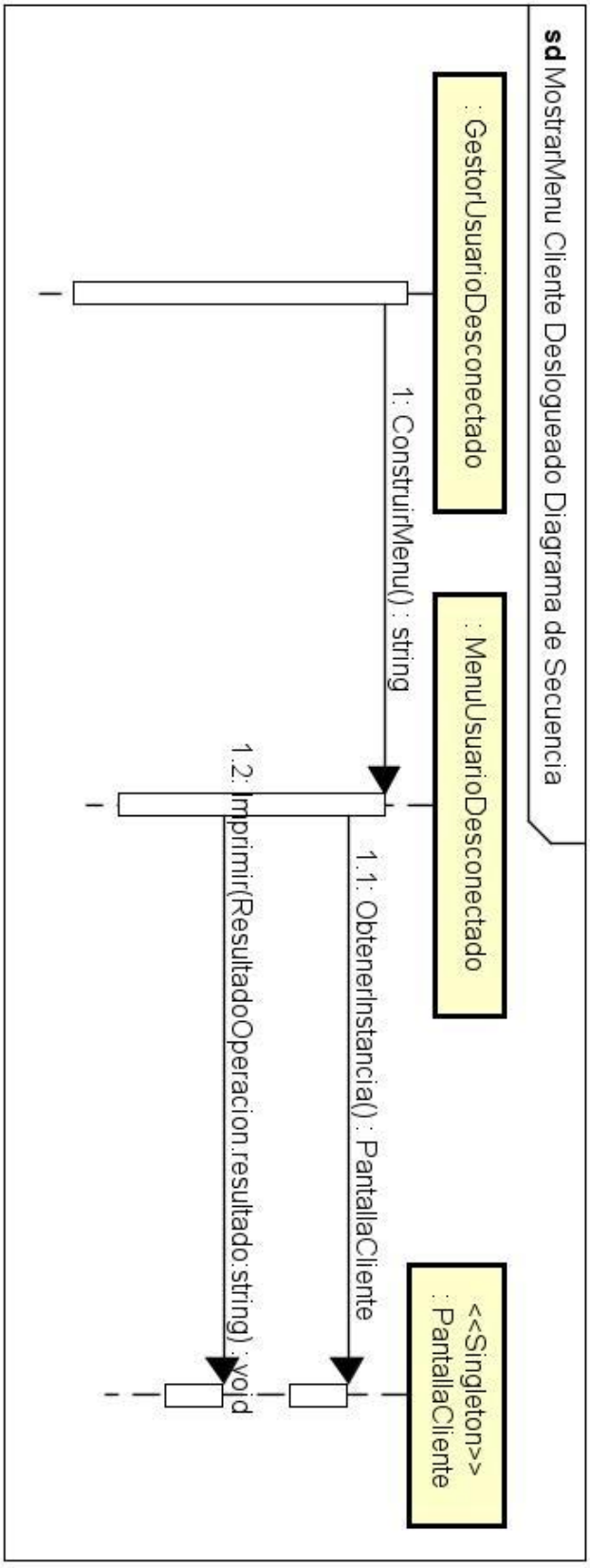



*Ilustración 17 - Mostrar Menu Deslogueado, Diagrama de secuencia*

Mostrar Menú Cliente *Logueado*

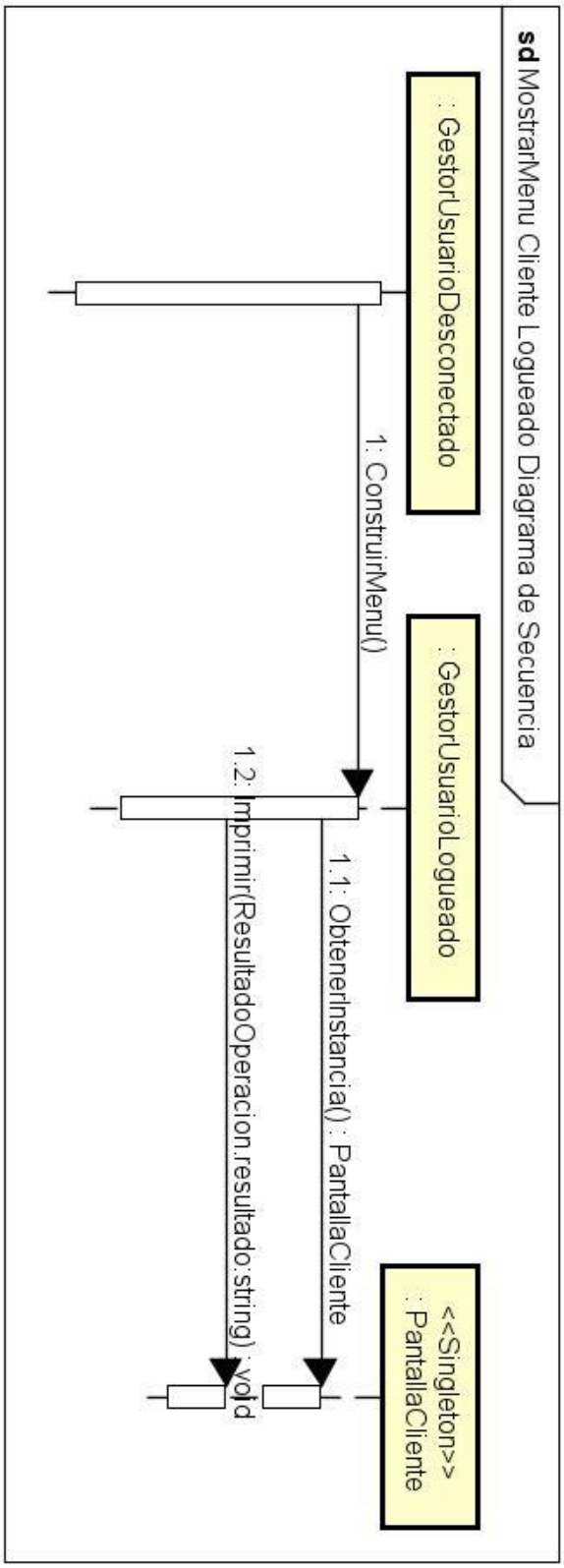



*Ilustración 18 - Mostrar Menu Logueado, Diagrama de secuencia*

Alta de Usuario Con Foto

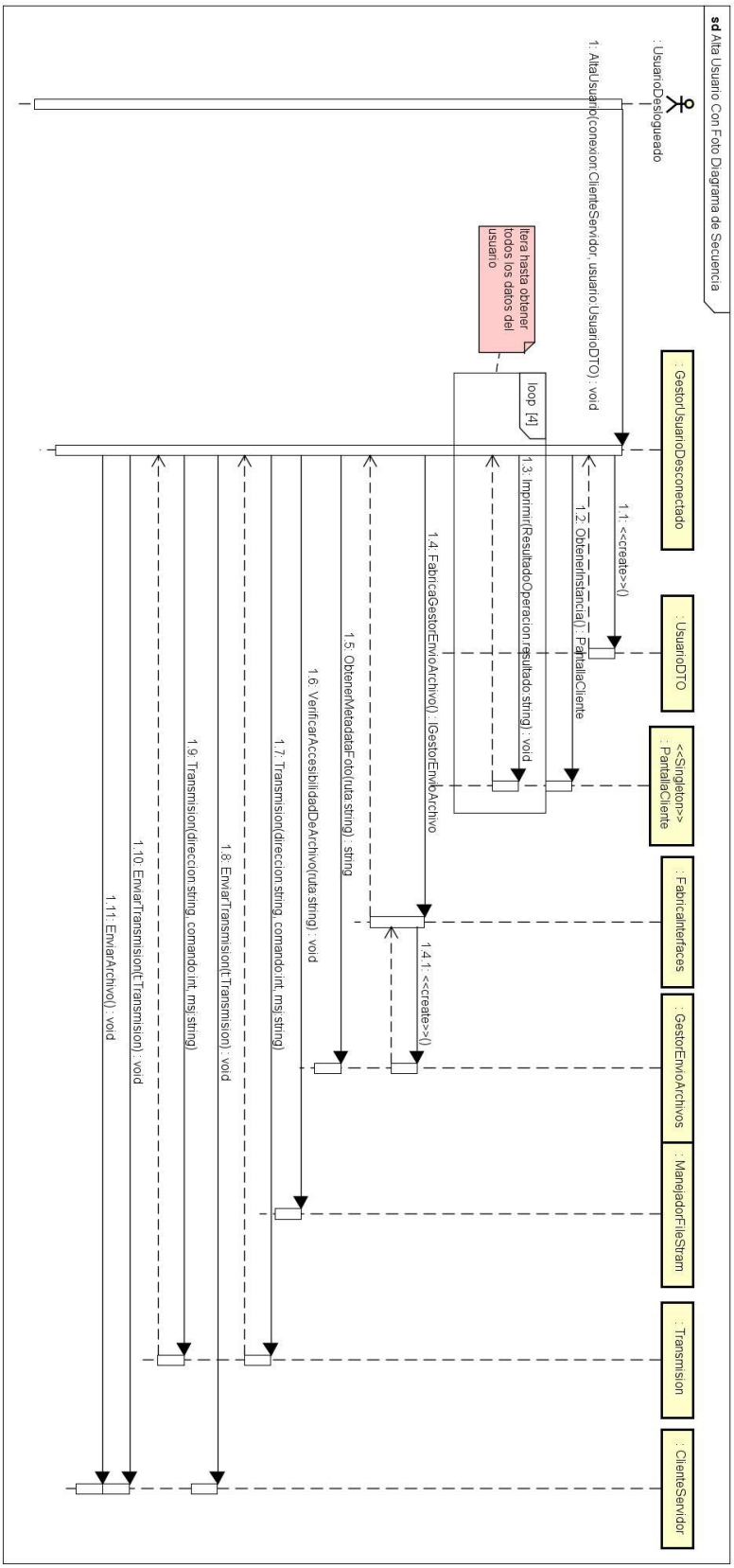



*Ilustración 19 - Alta de usuario con foto, Diagrama de secuencia*

Alta de Usuario continuación

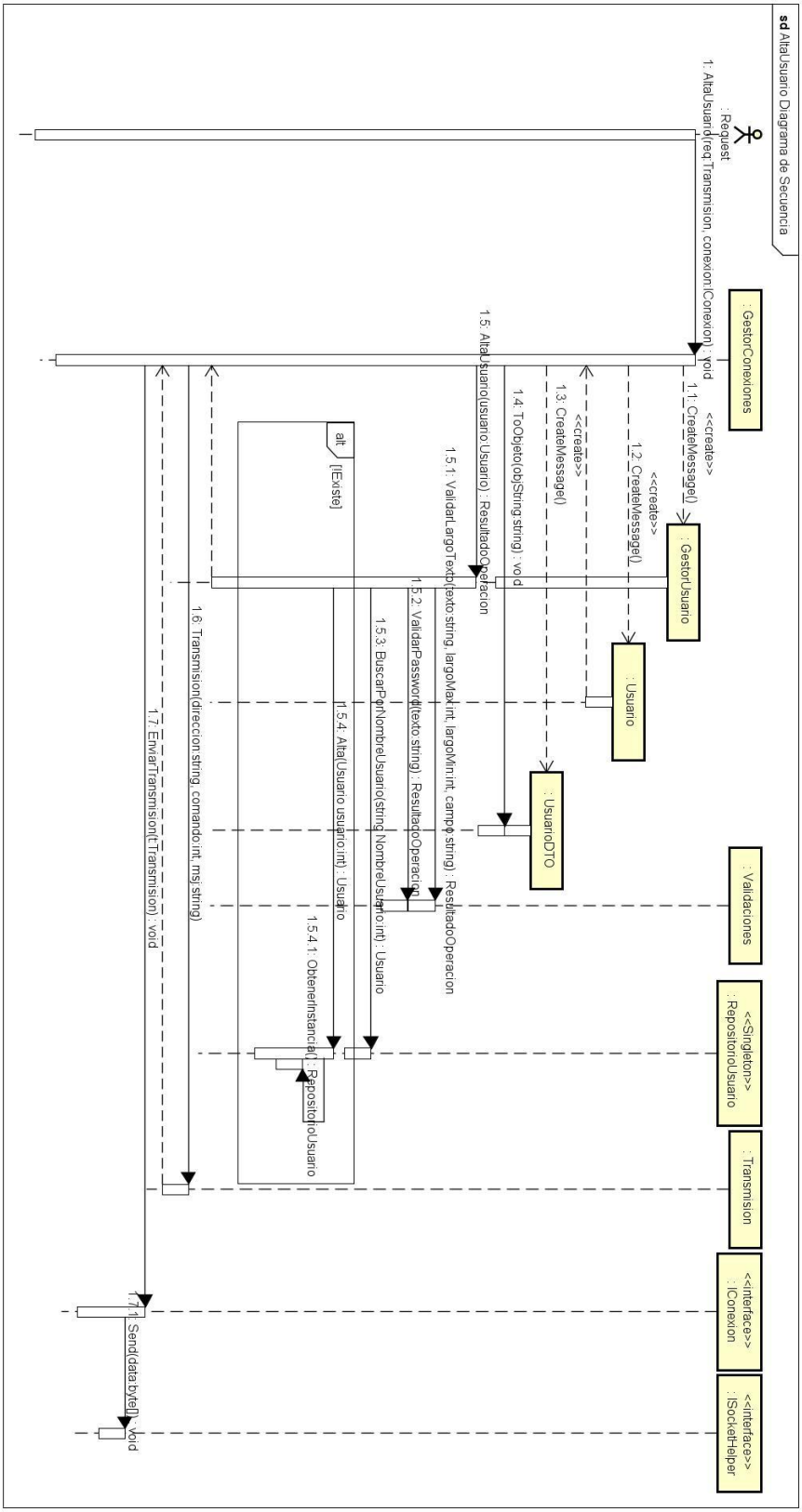



*Ilustración 20 - Alta de usuario (continuación), Diagrama de secuencia*

LogIn Request

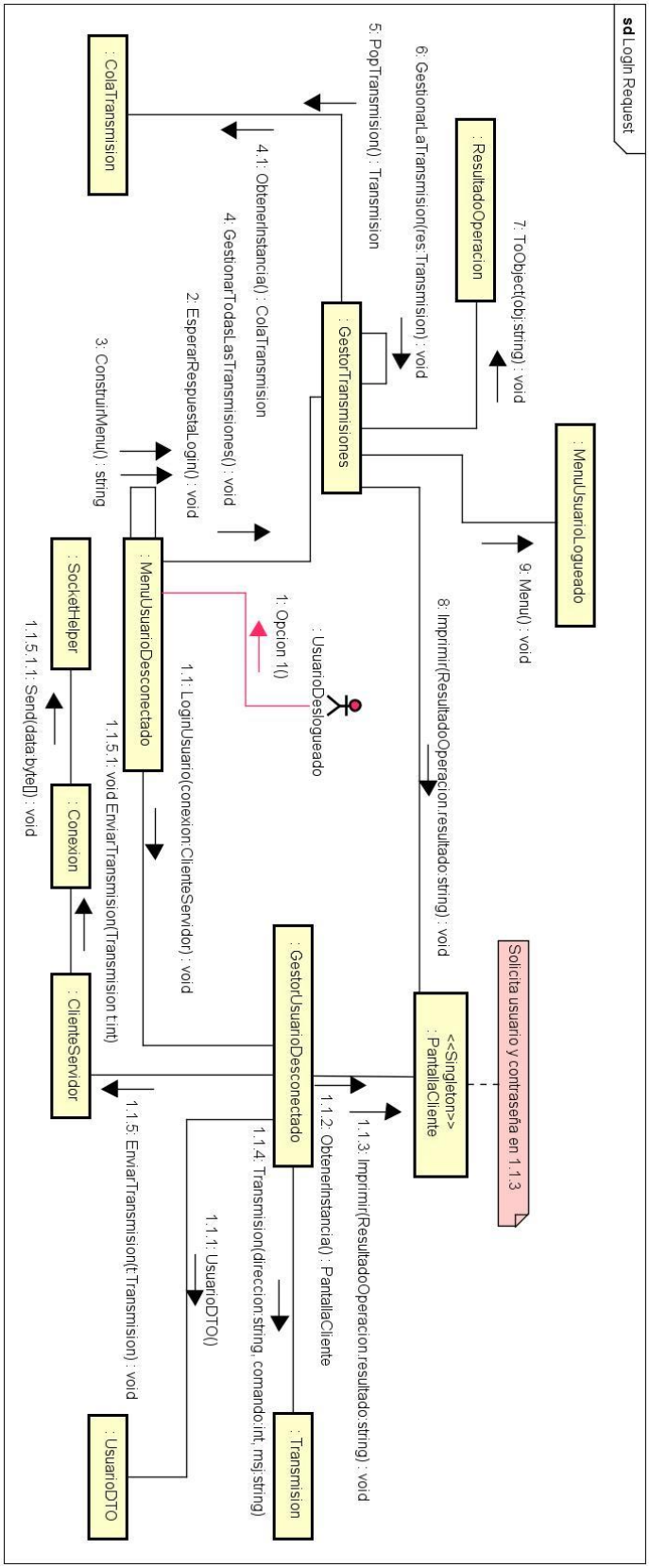



*Ilustración 21 - Login, Diagrama de comunicación*

# Funcionamiento de la aplicación

La solución cuenta con dos aplicaciones, la de cliente y la de servidor, a continuación, se procederá a explicar brevemente el funcionamiento de ambas.

Ambas aplicaciones cuentan con un archivo propio *App.config,* en el cual se precargan los datos relevantes de la conexión, como ser el host y puerto para conectar ambas instancias. Lo primero que se debe hacer es configurar correctamente estos valores.

## Aplicación Servidor

Lo principal que se debe saber de la aplicación es la misma posee un menú para su gestión. La imagen 22 es una captura durante la ejecución:

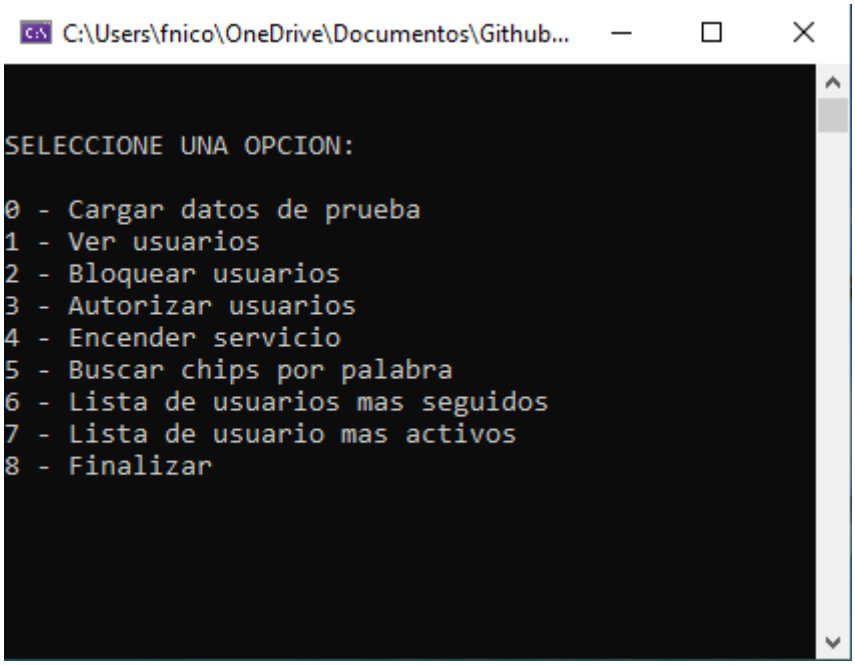



*Ilustración 22 - Menú Administrador*

La aplicación del servidor inicia en el menú anterior, a partir de ahí se dispone de 9 opciones:

0. **Cargar datos de prueba:** Nos cargará una serie de datos para que se puedan probar las funcionalidades sin necesidad de ingresar todo manualmente.
1. **Ver usuario:** Nos muestra una lista con los usuarios de la aplicación, acorde al SRF2.
2. **Bloquear usuarios:** Nos permite bloquear el acceso de un usuario al sistema, además, si el mismo se encuentra conectado, automáticamente lo desconecta (SRF3).
3. **Autorizar usuarios:** Nos permite volver a brindar el acceso a un usuario que se encontraba bloqueado (SRF4).
4. **Encender servicio:** Nos permite poner el servidor a la escucha y recibir los pedidos de los usuarios que lo soliciten (SRF1).

5. **Buscar chips por palabras:** Nos permite buscar en la lista de todos los chips si alguno cuenta con la cadena de texto que ingresemos en el cuerpo de este (SRF5).
6. **Lista de usuarios más seguidos:** Nos muestra los 5 usuarios que poseen más seguidores (SRF6).
7. **Listas de usuarios más activos:** Nos muestra la lista de los 5 usuarios que, en un período de tiempo solicitado, han escrito o contestado la mayor cantidad de chips (SRF7).
8. **Finalizar:** Cierra todas las conexiones de los usuarios conectados de forma segura y apaga el servicio y la aplicación.

Usuarios de datos de prueba autogenerados:

| Nombre Real | Nombre Usuario | Contraseña |
|---|---|---|
| Horacio Avalos | havalos | password123456 |
| Federico Alonso | falonso | password123456 |
| Micaela Olivera | molivera | password123456 |
| Teresa Alonso | talonso | password123456 |
| Emiliano Marone | emarona | password123456 |
| Carina Davila | cdavila | password123456 |

## Aplicación Cliente

La aplicación cliente cuenta con dos menús, uno de ellos es para cuando el cliente se encuentra *deslogueado* y la otra es posterior a la realización del *login*.

### Menú *Deslogueado*

La imagen 23 es una captura del menú *deslogueado* durante la ejecución:

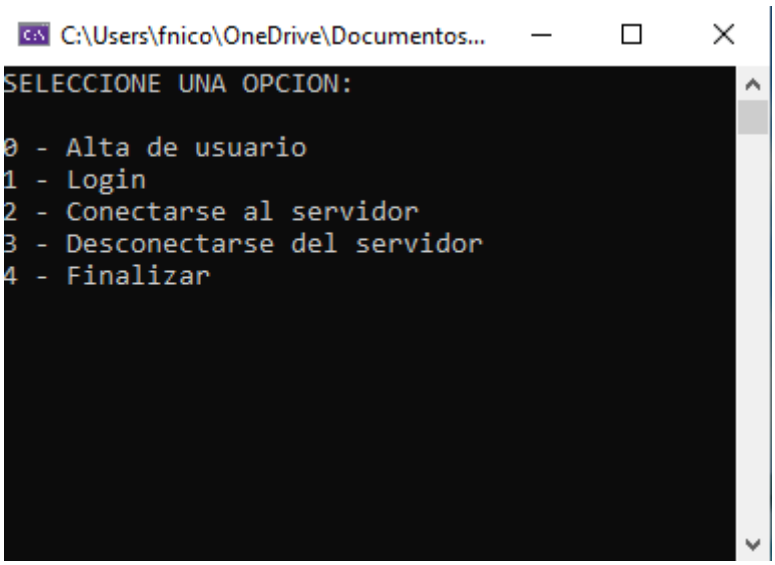



*Ilustración 23 - Menú Usuario Deslogueado*

Un usuario que no se encuentre *logueado* puede realizar cualquiera de estas 5 opciones:

0. **Alta de usuario:** Nos permite mediante el ingreso de un nombre de usuario, nombre real, contraseña y posiblemente una foto, realizar el alta del usuario al sistema (CRF2).
1. ***Login*:** Nos permite ingresar al sistema, posteriormente a ingresar nos mostrará el menú del usuario *logueado* (CRF3).
2. **Conectarse al servidor:** Nos permite establecer una conexión con la aplicación servidor. Esta acción debería hacerse al iniciar la aplicación (CRF1).
3. **Desconectarse del servidor:** Nos permite cerrar la conexión con el servidor realizada en el punto anterior (CRF1).
4. **Finalizar:** Nos permite apagar la aplicación.

Menú *Logueado*

La imagen 24 es una captura del menú de usuario *logueado* durante la ejecución de la aplicación

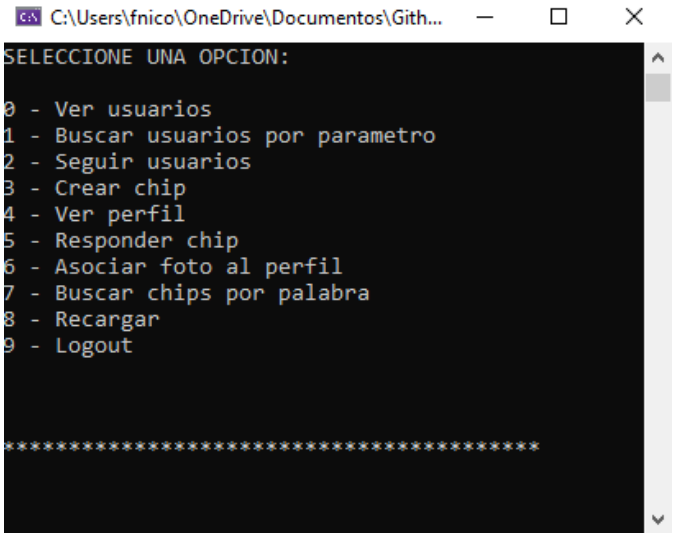



*Ilustración 24 - Menú Usuario Logueado*

Un usuario logueado puede realizar las siguientes opciones:

0. **Ver usuarios:** Nos permite ver los usuarios existentes en el sistema (No era una funcionalidad solicitada pero facilita el uso de la aplicación).
1. **Buscar usuarios por parámetros:** Nos permite hacer una búsqueda de usuario por nombre real y nombre de usuario de forma incluyente o excluyente (CRF4).
2. **Seguir usuarios:** Nos permite seguir a otro usuario del sistema (CRF5).
3. **Crear chip:** Nos permite realizar una publicación de un chip, el mismo debe poseer un cuerpo y hasta 3 fotos (CRF6).
4. **Ver perfil:** Nos pregunta por el usuario que deseemos ver el perfil y nos mostrará el mismo, sus datos de usuario y las publicaciones que haya realizado (CRF8).
5. **Responder chip:** Nos da la posibilidad de responder alguna de las publicaciones o sus respuestas (CRF9).
6. **Asociar foto al perfil:** En caso de no haber creado al usuario con una foto, nos permite asignarle una desde esta opción del menú (CRF2).
7. **Buscar chips por palabras:** Nos da la posibilidad de buscar entre todas las publicaciones si alguna contiene el texto que solicitemos encontrar (No era una funcionalidad solicitada, pero nos resultó interesante realizarla).
8. **Recargar:** El esperar una respuesta del usuario nos hace que quede esperando en esa posición estática hasta que la brinde, esta opción nos permite recargar la pantalla por si el servidor nos envió alguna información.
9. ***Logout*:** Nos permite salir del sistema, posteriormente nos mostrará el menú de usuario *deslogueado*.

Adicionalmente existe la posibilidad de recibir notificaciones. En caso de que alguno de los usuarios que seguimos realice una publicación, el sistema nos notificará y nos brindará la posibilidad de responderla.

## Errores Conocidos

1. No se verifica del lado del servidor que el usuario esté *logueado* para realizar los métodos *logueados*, se supone que no puede acceder a dichos métodos ya que no están habilitados en el menú de consola, pero sabemos que es un cambio para una futura entrega.
2. En ciertas funcionalidades, como por ejemplo responder chip, se recibe por parámetro un *chipDTO*, el *chipDTO* posee imágenes relacionadas, por lo que si se creara otra aplicación cliente podrían mandar una petición con elementos extra a los realmente necesarios, se debe en una futura entrega brindar un DTO específico para cada *endpoint*.
3. En cuando a repetición de código, se podrían manejar por herencia ciertas propiedades de los DTO que son de similar naturaleza, por ejemplo, *loginDTO* es una porción de UsuarioDTO.
4. Se pudo haber buscado una forma de abstraer el *parseo* entre texto y DTO, ya que suponemos que en el futuro se puede llegar a usar algún método de la librería *de .NET* para realizarlo, lo que nos conllevará tocar código en todos los DTO.
5. En la clase *RespuestaOperacion* se posee como atributo una *Entidad*, que es representada por un elemento del tipo *Object*, el cual podría ser implementado usando *Generics*.
6. *Realizar un click de más en el menú para poder ver las respuestas,* la aplicación realiza una especie de *round robin*, gestionando todo lo recibido del cliente cada vez que se muestra el menú. Una vez que se muestra el menú queda trancado el mismo esperando una respuesta del usuario.

## Lecciones Aprendidas

Como lecciones aprendidas tenemos que es muy importante saber qué hilos tenemos en la aplicación, cuáles son los principales, cuáles deben esperar alguna respuesta, y por dónde circula o trabaja cada uno de ellos.

Es imprescindible saber sincronizar todos, y detectar las zonas de mutua exclusión, esto evita tanto problemas del funcionamiento de la aplicación como inconsistencias en su comportamiento.

Aprendimos que muchas veces la solución que mejor funciona es la más simple. Para poder sincronizar el escribir en la pantalla de cliente con los mensajes que llegaban del servidor estuvimos muchos días de estudio e intentos con diferentes perspectivas, resultó ser la solución más simple la de encolar los mensajes y que la pantalla los vea cuando el usuario disponga. Probamos con monitores, semáforos y distintos métodos de espera, siempre existía un caso borde que terminaba rompiendo el correcto funcionar de la aplicación. Finalmente decidimos que es mejor hacer un *click* extra y que todo funcione correctamente, que intentar que quede elegante y se rompa.

Creemos que lo fundamental fue tomar el trabajo con tiempo, cada nueva funcionalidad conlleva un período de aprendizaje y adaptación, y luego de que funciona, a los días se encuentran nuevos errores, o que tal vez el concepto aprendido no era tal cual se pensaba. El hecho de tomar el obligatorio con tiempo nos llevó a implementar una solución que nos resultó a gusto y a intentar implementar acciones extra para poseer de ejemplo para nuevas instancias.

A modo general, entendemos que la solución que entregamos a la propuesta planteada posee aspectos de mejora a ser evaluados e incluidos dentro del nuevo alcance de la segunda entrega.