Universidad ORT Uruguay

Facultad de ingeniería

Obligatorio

Documentación

Programación de redes

Diego Baccino 212995

Andrés Tomás 199834

Profesor: Luis Barrague

Año 2020

Repositorio: DiegoBaccino1/Obligatorio-Programacion-de-redes (github.com)

Declaración de autoría

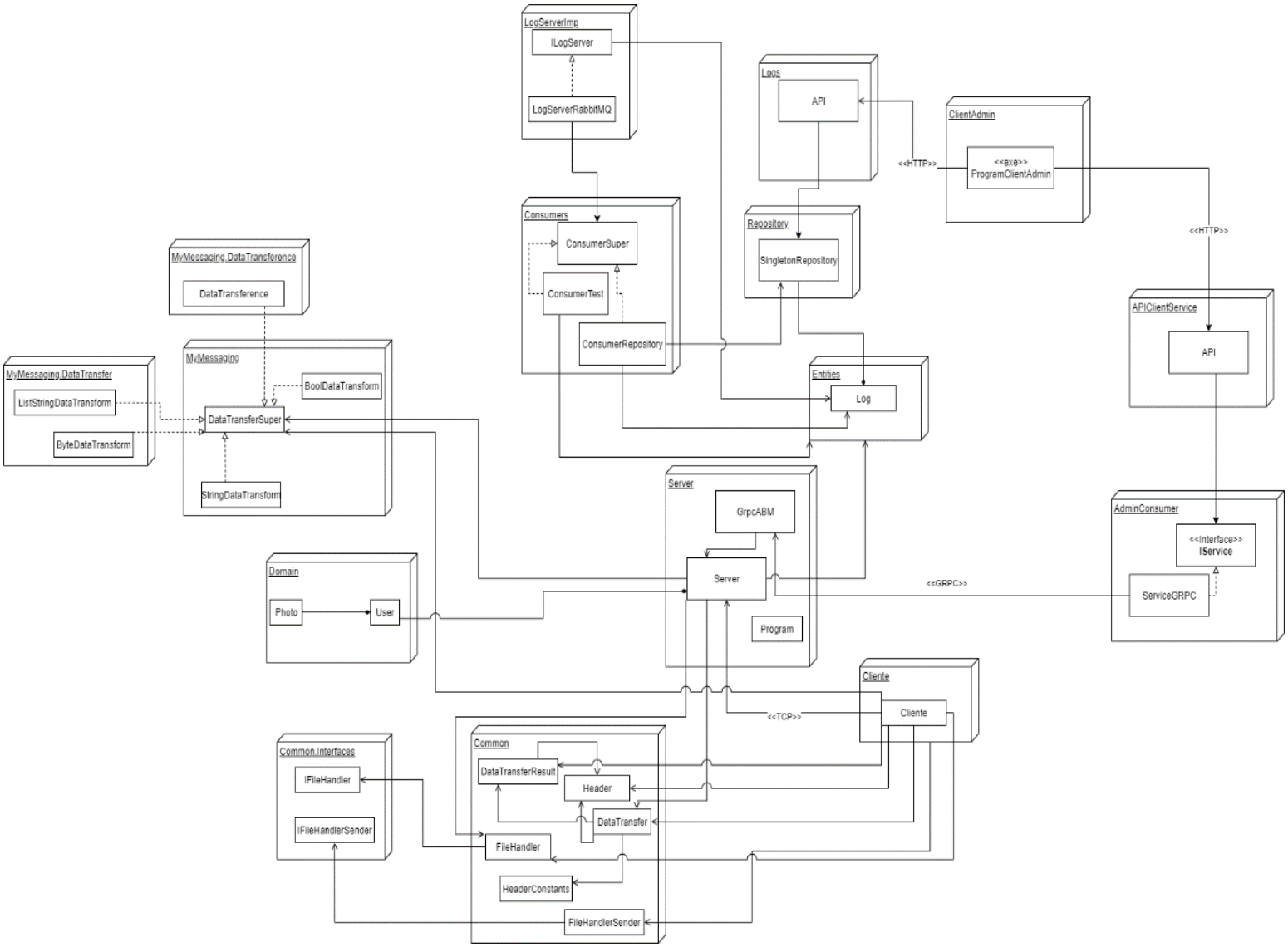
Nosotros, Diego Baccino y Andrés Tomás, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano.

La obra fue producida en su totalidad mientras realizábamos el obligatorio para programación de redes.

Índice

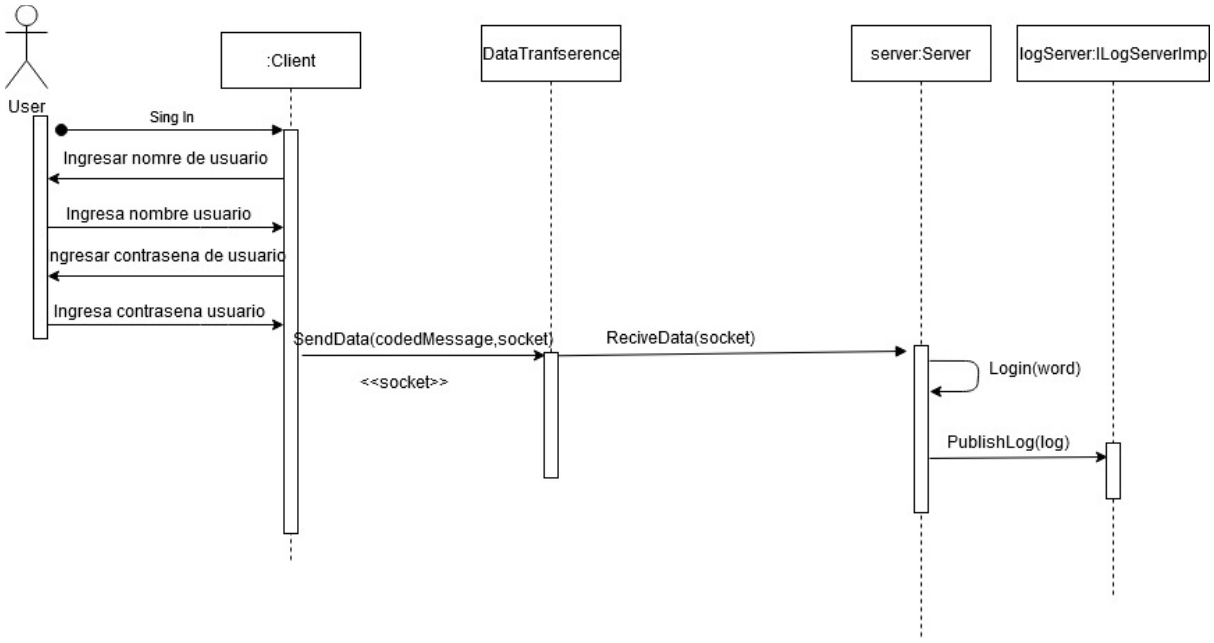
Descripción del trabajo

La solución esta compuesta por 15 proyectos, tanto de dominio y utilidades (ya sea para enviar archivos, definir el protocolo, etc.) como las diferentes aplicaciones para su modo de uso. Ya que tiene distintos modos de uso. El primero sería server de instafoto -cliente, para este modo se deberán ejecutar los proyectos GrpcABM ya que es el que tiene la iniciación del server y también el servicio de grpc. Se inician ambos desde el mismo proyecto porque como el servicio de grpc utiliza funcionalidades expuestas por el server, por tanto, si el server no esta activo no tiene sentido exponer el servicio de grpc. Y El cliente establecerá una conexión mediante socket y tcp hacia el server. Para esto se definió el siguiente protocolo; Los primeros 3 caracteres indicaran si es una petición (REQ) o una respuesta (RES), los siguientes 2 indican el comando a ejecutar siendo uno de los siguientes 1-SignUp (Creación del usuario), 2-Login (cuando se inicia con un usuario que no existe se cierra el cliente, es un bug), 3-UploadFile (Subir foto), 4-ListUsers (Listado de usuarios, solo el nombre), 5-ViewComments, 6- , 7- , 8-Desconectar. A continuación, se muestra un diagrama de componentes



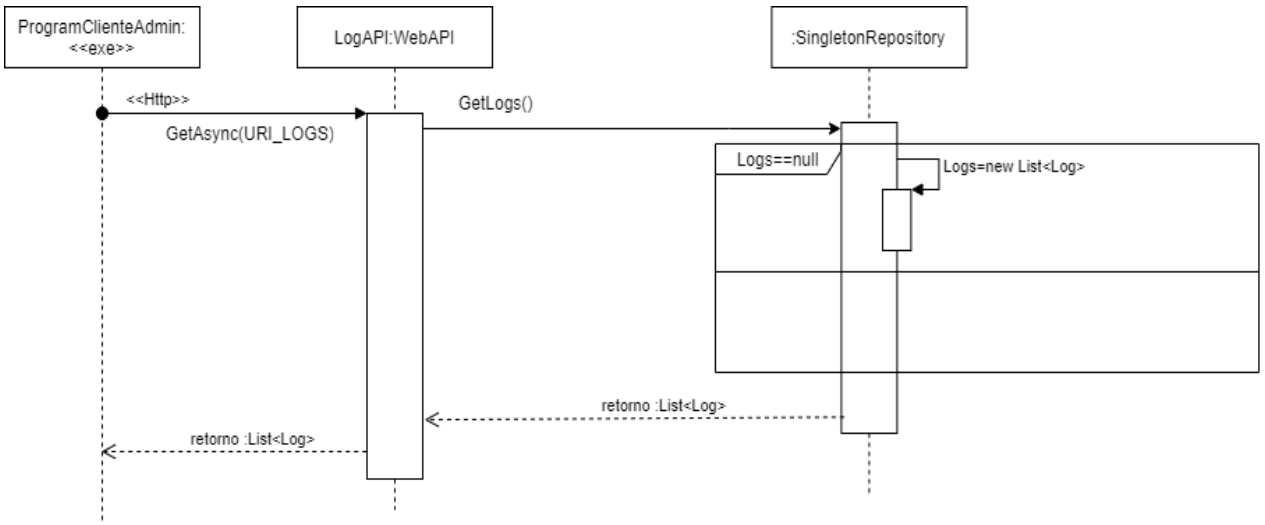
En este modo de uso el cliente podrá dar de alta un usuario indicando nombre de usuario y contraseña, el nombre debe ser único. También se podrá hacer un login indicando un usuario y una contraseña de un usuario ingresado en el sistema y que no este “logueado”. Si alguno de estos casos falla no se podrá hacer el login. Una vez “logueado” el usuario podrá subir una foto al servidor indicando la ruta de esta, ver todos los usuarios, agregar un comentario a una foto propia indicando el comentario y el nombre de la foto, ver los comentarios de una foto indicando el nombre de usuario y el nombre de la foto y ver sus fotos (se lista el nombre de las fotos) y desconectarse del servidor (de manera gracefull). Estas operaciones generaran logs tanto si fueron operaciones exitosas como si no. Es Del

lado del server se podrá desconectar dando de baja todas las conexiones con los clientes. A continuación, se muestra un diagrama de secuencia de crear un usuario desde el cliente, es decir el modo cliente-servidor

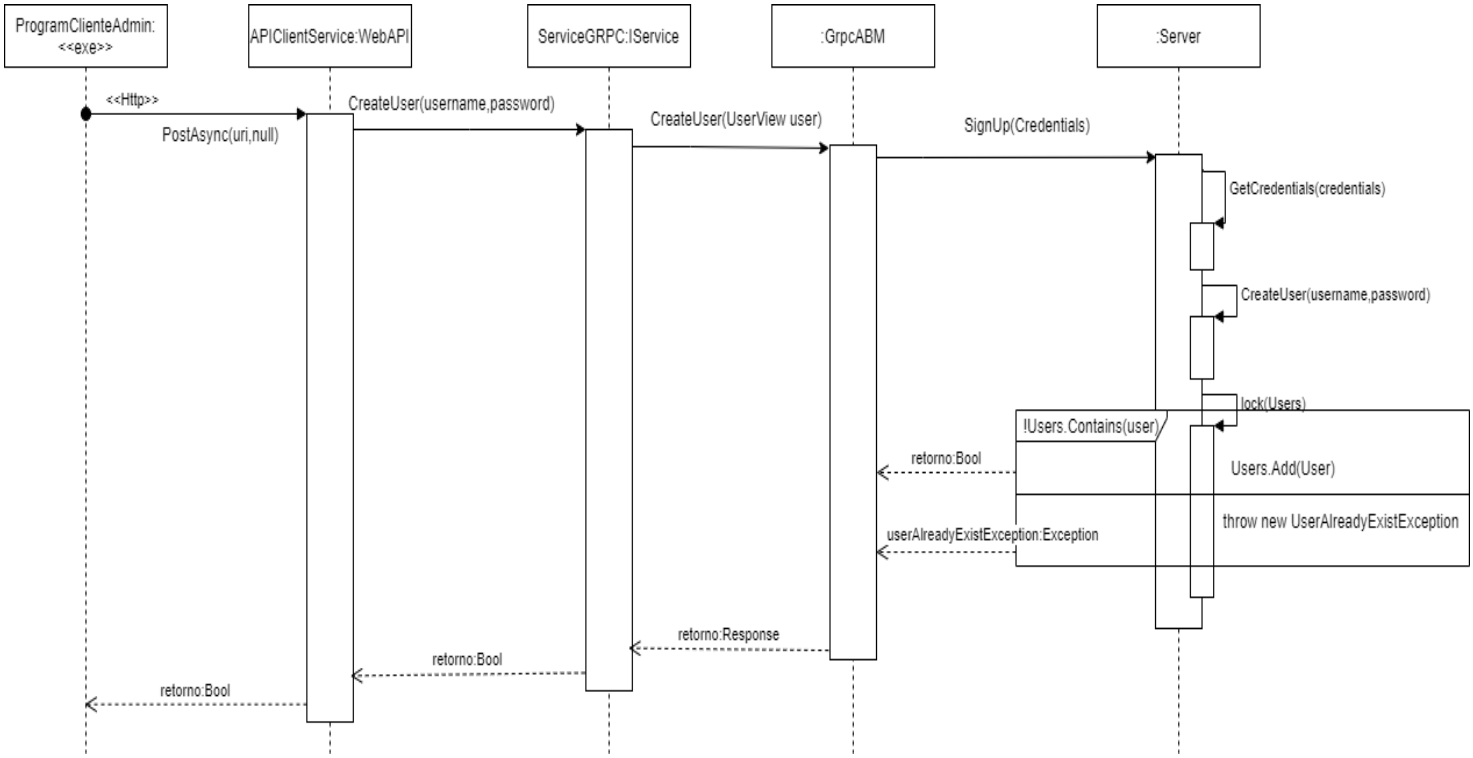


Otro modo de uso es el de cliente administrador con consulta de logs o servicio de grpc. Estas webAPI son independientes. Por tanto, se podrán ejecutar en simultaneo o por separado

Para el primer modo se deberá ejecutar ClientAdmin el cual ejecutará una consola y la web api de nombre LogsApi. En este modo el Cliente mediante la consola podrá elegir si quiere consultar todos los logs, los logs exitosos o los logs de error. En caso de que la api no se haya iniciado no se podrá hacer la request y se notificará al usuario.



Para el segundo modo se deberá ejecutar ClientAdmin y el proyecto GrpcABM y APIClientService, como mencionamos antes este proyecto inicializará en un hilo el server para continuar con la ejecución y poder iniciar el servicio de grpc y la API se comunicará con



el servicio. En este modo el Cliente mediante la consola podrá elegir si quiere crear un usuario indicando usuario y contraseña, de esta manera se hace una petición POST hacia la webApi que se comunica con el servicio de grpc. Con las operación de modificar se debe ingresar el usuario a modificar y las nuevas credenciales y se hará la request de igual manera. Por último, con la operación de borrar se deberá indicar únicamente el nombre del usuario que se desea borrar. Para estas operaciones se notificará al usuario mediante un bool si fue posible realizar la operación. Pero en caso de borrar si se pudo borrar se retornará un “true” y si no también, ya que el objetivo de la operación deja a la aplicación en el mismo estado. Es decir que si no se encontraba deja a los usuarios ya ingresados menos el objetivo y si se encontraba también ya que lo borra. En caso de que la api no se haya iniciado no se podrá hacer la request y se notificará al usuario. A continuación, se detalla un diagrama de sequencia para la funcionalidad alta de usuario desde el cliente administrador.

Diseño

Para la solución se dividió la aplicación en varios proyectos para intentar hacerlo lo más mantenible posible. Menos en las clases de “Server” y “Cliente”, estas quedaron muy acopladas y con muchas responsabilidades, no respetando el principio de SRP y haciendo que un cambio pueda tener un gran impacto. Pero también en cuanto a la cohesión de paquetes se buscó seguir los siguientes patrones, REP (Reuse Equivalence Principle), ya que el código reusado está en un mismo paquete, es decir que para reusar código se incluye la dll. Por ejemplo, en los proyectos mencionados anteriormente de server y cliente utilizan el paquete de MyMessaging para mandar y recibir datos mediante los sockets. También Los principios de CCP (Common Closure Principle) y CRP (Common Reuse Principle), ya que las clases que cambian juntas y se usan juntas están dentro del mismo paquete, por tanto CCP se respeta, ay que cuando una clase cambia afecta a clases dentro de ese paquete y CRP se respeta porque cuando se reúsa una clase de un paquete se debería poder usar todas, por ejemplo en caso del paquete de LogServerImp, si se usan las clases del paquete se usan juntas y si cambian el impacto es dentro de ese paquete.

Siguiendo con los principios de paquetes, se intentó buscar que el acoplamiento sea el menor posible. Para esto se buscó seguir los Principios de ADP (Acyclic Dependencies Principle), SDP (Stable Dependencies Principle) y SAP (Stable Abstraction Principle). Para el primer principio se ve en el diagrama anterior que no hay dependencias cíclicas. Para los otros dos principios debemos analizar las métricas de abstracción y estabilidad.

La abstracción de los paquetes los paquetes MyMessaging, Server, Cliente y Common tienden a 0, pues poseen más clases concretas que interfaces, ya que en un paquete se define el contrato en caso de interfaz y/o comportamiento en caso de clase abstracta pero como hay varias implementaciones tiende a ser un paquete concreto y poco abstracto. Pero aun así si dependen de otro paquete, el otro paquete es más abstracto. Por otro lado, los paquetes AdminConsumer, LogServer, Consumers son paquetes más abstractos ya que, aunque siguiendo el mismo criterio de tener la interfaz y la implementación en el mismo paquete como no hay muchas implementaciones. Por tanto, la abstracción es media (en torno a 0.5). Pero paquetes como Entities que solo definen una entidad que va a ser usada

por otros paquetes la abstracción es de 1 pero su estabilidad es de 0, por tanto, estaría en la línea media de la gráfica Abstracción-Estabilidad, ya que Abstracción + Estabilidad=1.

En cuanto a la estabilidad los paquetes siempre tienden a depender de otros paquetes más estables que ellos. Excepto por los paquetes de server y

En cuanto al diseño de las clases para la api del servicio de abm de usuario se inyecta la dependencia del servicio de grpc, de esta forma logramos cumplir con el principio de OCP, ya que si se quiere hacer una nueva implementación del servicio solo se debe cambiar en la configuración de la api, pero no cambia el comportamiento de la lógica de los controller. Y así también se logra desacoplar la api de la lógica.

Siguiendo con el OCP buscamos siempre intentar definir una abstracción y usarla, de forma análoga a como se usa la inyección de dependencias en la api. Por ejemplo, para transferir datos desde el cliente al server se crea una instancia de DataTransferSuper que va a ser la encargada de decodificar el mensaje que llega como bytes. Así podemos enviar y recibir distintos tipos de datos según lo necesite el programa y cambiarlo en tiempo de ejecución y que de esta manera no se esté codificando “a mano” el mensaje si no que se use una librería. Si se quiere crear un nuevo tipo de dato para enviar se debe implementar esa clase abstracta e instanciarla cuando se quiera enviar. Para esta idea también se intentó aplicar el concepto de State Method para cambiarla en tiempo de ejecución y Template Method para definir un comportamiento común y que la implementación sea la mínima indispensable.

También se buscó seguir el principio de SRP (Single Responsability Principle) y que las clases tengan solo un motivo para cambiar, ya que así se desacopla el sistema y los cambios no deberían tener tanto impacto. Por ejemplo, la clase DataTransferece es la encargada de hacer el envío y recibir mediante los sockets y DataTransferSuper la encargada de hacer la codificación y decodificación del objeto. De esta manera si la codificación cambia no impacta en el envío de datos y viceversa. De igual manera con el consumo de los logs y el server, si se quieren consumir de una manera distinta se puede implementar, en nuestro caso para verificar que el consumo sea correcto se implementó un ConsumerTest que hereda de ConsumerSuper pero que imprime por pantalla los logs, como en este caso no nos interesaba que haga el ack de los logs también hicimos un método que seteara el ack y

dejarlo a criterio de la implementación. De igual manera para los logs, si se quisiera hacer una implementación en azure por ejemplo se podría hacer sin cambiar las demás clases, ya que del consumo de logs se encarga otra clase. Por último, aplicamos el patrón de Singleton en el repositorio de los Logs ya que el repositorio no puede vivir dentro del scope de consumir porque se destruye en cuanto se termina la operación. Por tanto, necesitábamos un punto de acceso global.

En cuanto a los patrones GRASP en las funcionalidades de grpc service (abm) y logs se buscó aplicar la ley de demeter y no depender de implementaciones y para eso aplicamos polimorfismo. Como mencionamos antes intentamos mantener el acoplamiento bajo y la cohesión alta, quizá esta no se llevó a cabo del todo bien. Intentamos aplicar el patrón experto principalmente en clases como Header que define el protocolo, por tanto, la clase Header tendrá las operaciones referentes al protocolo.