Universidad ORT Uruguay Facultad de Ingeniería

Programación de Redes Obligatorio 3

Federica Blanco - 227670
Alfredo Eyheralde - 248577

Índice

Introducción

A continuación se detalla la documentación para el obligatorio 3. Este obligatorio consiste en una mejora realizada sobre el preexistente realizado en los obligatorios anteriores por lo que se asumirá que el lector ya leyó las anteriores documentaciones y por ende se mencionan sólo los cambios hechos para esta entrega particular.
Los cambios que se detallarán aquí son la nueva arquitectura de la solución así como la forma en la que se utilizaron las tecnologías especificadas en la letra. Estas nuevas tecnologías son GRPC, RabbitMQ y Web APIs REST.

Repositorio de GitHub

Release ServerGrpc:
Obligatorio\ServerGrpc\bin\Release\net6.0\ServerGrpc.exe
No confundir con otro ejecutable de nombre Server**Grcp**.exe

Release ServerAdmin:
Obligatorio\ServerAdmin\bin\Release\net6.0\ServerAdmin.exe

Release ServerLog:
Obligatorio\ServerLog\bin\Release\net6.0\ServerLog.exe

Release Client:
Obligatorio\Client\bin\Release\netcoreapp6.0\Client.exe

Cambios en la instalación

Para poder instalar los nuevos servicios es necesario colocar el mismo archivo de configuración App.Config que se utilizaba para el paquete Server. Esto debido a que el Server ahora se lanzará desde la aplicación ServerGrpc utilizando Tasks para poder tener en paralelo un servicio Grpc junto al servicio ya brindado anteriormente por el servidor utilizando TcpListener.

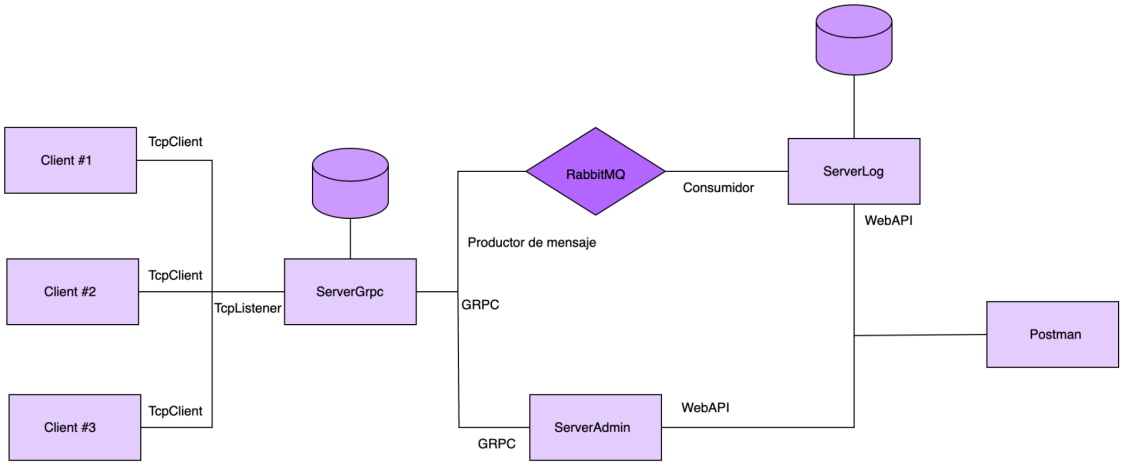
Uso

Para correr el sistema, los pasos a seguir son pocos.
Primero se debe levantar el ServerGrpc lo cual actuará como nuestro servidor principal. A este se podrán conectar clientes como lo hacían anteriormente. Esta nueva entrega le suma dos nuevos servidores. Uno de Logs llamado ServerLog el cual es una web API que internamente consume mensajes de una cola de mensajes RabbitMQ hosteada en CloudAMQP a la cual le llegan mensajes desde ServerGrpc.
El segundo nuevo servidor es un servidor administrativo que expone otra web API (ServerAdmin) e internamente también es un cliente Grpc el cual se conecta con la parte de servidor grpc dentro de ServerGrpc.

Mecanismo de concurrencia

Para las listas, decidimos utilizar locks para controlar la concurrencia, entonces al insertar, eliminar o editar los elementos de estas listas, lo que hacemos es bloquear la lista hasta que se haya terminado para poder proteger la integridad de los datos y que no se generen problemas con las mismas. Para hacerlo, hicimos que las lista sean privadas y para realizar acciones sobre las listas, se tenga que usar los métodos de la clase que contiene la lista, así controlamos que siempre se use el lock al modificar la lista.

Arquitectura



En el diagrama anterior se puede observar un vistazo general de cómo es la arquitectura de nuestra aplicación. Los roles de cada nuevo componente fueron descritos de forma general en la introducción pero ahora se procederá a describir cada uno más en profundidad.

ServerGrpc

Este proyecto contiene toda la parte relacionada al servicio de servidor grpc y a esto se le suma el viejo proyecto Server mediante la clase Server. De este modo, este proyecto lanza en paralelo un servicio Grpc y nuestro servidor a base de TcpListener. Cumple con todas las funcionalidades del Server anterior pero además tiene con el servicio Grpc se conecta con el ServerAdmin.
El proyecto tiene como subpaquetes un Proto con un *.proto* por cada entidad (photo,profile,user), un paquete Services que contiene un service también por cada entidad, y otro Properties. Este último tiene la configuración para levantar el servidor Kestrel y el paquete Protos tiene los archivos de ProtoBuffers que utilizara nuestro servicio Grpc. Además, contiene un subpaquete Clases, que es el mismo usado por el viejo Server pero se le agregar la class LogModel (con el modelo de los logs implementados en esta entrega) y un LogPublisher (clase con el método encargado de publicar un mensaje al ServerLog). Finalmente, agregamos un subpaquete Models con DTOs para los métodos de la api.

ServerAdmin

Este proyecto hace referencia al Server administrativo. Este expone un Api REST mediante la cual se pueden realizar las distintas operaciones requeridas por la letra del obligatorio 3. La API recibe solicitudes a los endpoints /Photo, /Profile y /User con los distintos verbos HTTP para poder diferenciar las distintas operaciones.
Los localhosts en cual se hicieron la request fueron obtenidos de correr dicho servidor, y tomarlo de la consola de la línea “Now listening on”. En el zip se adjunta la colección de Postman. Cabe destacar que el administrador no puede editar ningún email de ningún usuario, ingresa el email para avisar a qué usuario quiere editar su nombre y/o password.
Este ServerAdmin se conecta a ServerGrpc mediante Grpc. Es por esto que este web api también tiene un subpaquete Protos con todos los protos por entidad, pero esta vez en la configuración del proyecto el archivo se puso en modo Client para que el proceso de build genere los archivos pertinentes al cliente Grpc y no el Server.

ServerLog

Para el Servidor de Logs se hizo uso del servicio de RabbitMq. En el servidor ServerGrpc se implementa un método para enviar mensajes a RabbitMq, y a través del mismo canal se implementó la recepción de los mismos en el servidor de logs.
Dicho proyecto también expone una API REST mediante la cual uno puede ver los logs de los eventos del sistema así como filtrarlos mediante query params especificando el filtro a aplicar y los valores para cada filtro. Los mismos son: por el

email del usuario logueado, fecha siguiendo el patrón MM/dd/yyyy y por evento (registro, login…).

```
public MQService() {

    // Conexión con RabbitMQ local:
    var factory = new ConnectionFactory() { HostName = "localhost" }; // Defino la conexion

     var connection = factory.CreateConnection();
     var channel = connection.CreateModel();

    channel.QueueDeclare(queue: "log", // en el canal, definimos la Queue de la conexion
        durable: false,
        exclusive: false,
        autoDelete: false,
        arguments: null);

    //Defino el mecanismo de consumo
    var consumer = new EventingBasicConsumer(channel);
    //Defino el evento que sera invocado cuando llegue un mensaje
    consumer.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        Console.WriteLine(" [x] Received {0}", message);
        LogModel log = JsonSerializer.Deserialize<LogModel>(message);

        var data = DataAccess.GetInstance();      AlfredoEyheralde, 24 hours ago • rabbit agregado …
        data.AddLog(log);
    };

    //"PRENDO" el consumo de mensajes
    channel.BasicConsume(queue: "log",
        autoAck: true,
        consumer: consumer);


}
```

```
namespace ServerGrpc.Clases
{
    8 references | AlfredoEyheralde, 19 hours ago | 1 author (AlfredoEyheralde)
    public class LogPublisher
    {
        8 references
        public static void Message(IModel channel, string userEmail, string eventDone)
        {
            var log = new LogModel
            {
                Date = DateTime.Now,
                UserEmail = userEmail,
                Event = eventDone
            };

            string messsage = JsonSerializer.Serialize(log);
            var body = Encoding.UTF8.GetBytes(messsage);
            channel.BasicPublish(exchange: "",
                routingKey: "log",
                basicProperties: null,
                body: body);
        }
    }
}
```

Para que el proyecto corra la WebApi, mientras que recibe los mensajes de RabbitMq, se implementó un servicio MQService, en el que se realizó la lógica para recibir los mensajes de la cola y guardarlos en el sistema, y en el Program se instanció este servicio.

```
using ServerLog.Service;

var builder = WebApplication.CreateBuilder(args);

var mq = new MQService();

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

El listado de logs se guarda bajo una clase DataAccess, construida bajo el patrón singleton y que se encarga de manejar la concurrencia sobre la lista. Así delegamos a esta clase la persistencia de los logs recibidos en el background service.

Tipos de eventos válidos para el filtrado:

1. Login
2. Register
3. ListarUsuarios
4. ListarUsuarioEspecifico
5. CrearPerfilLaboral
6. SubirFoto
7. LeerChat
8. EnviarChat

RabbitMQ

Por último, el paquete RabbitMq centraliza todo lo necesario para trabajar con colas de mensajes tanto para nuestro ServerGrpc y también para ServerLog. Este paquete expone la interfaz IMQHelper y su implementación RabbitMQHelper las cuales se utilizan para mandar y recibir mensajes utilizando la cola de mensajes especificada en la clase Queue (log en nuestro caso). Además posee la clase

LogModel.cs la cual sirve como clase para representar los logs dentro de nuestra solución.

Funcionamiento de la aplicación

No encontramos ningún problema con nuestra aplicación, aunque creemos que pueden haber casos bordes que no pudimos probar y pueden llegar a haber algún problema que no tuvimos en cuenta.
En cuanto a los logs, decidimos loguear cuando el usuario interactúa con una funcionalidad, más allá de su éxito con la misma. Es decir, que se registra si busco un perfil por más de que la búsqueda haya sido exitosa.
Además, en la subida y descarga de imágenes, en cuanto al NuevoServer éstas quedan dentro del paquete y si es un Client estas se guardan en el root de todo el sistema.