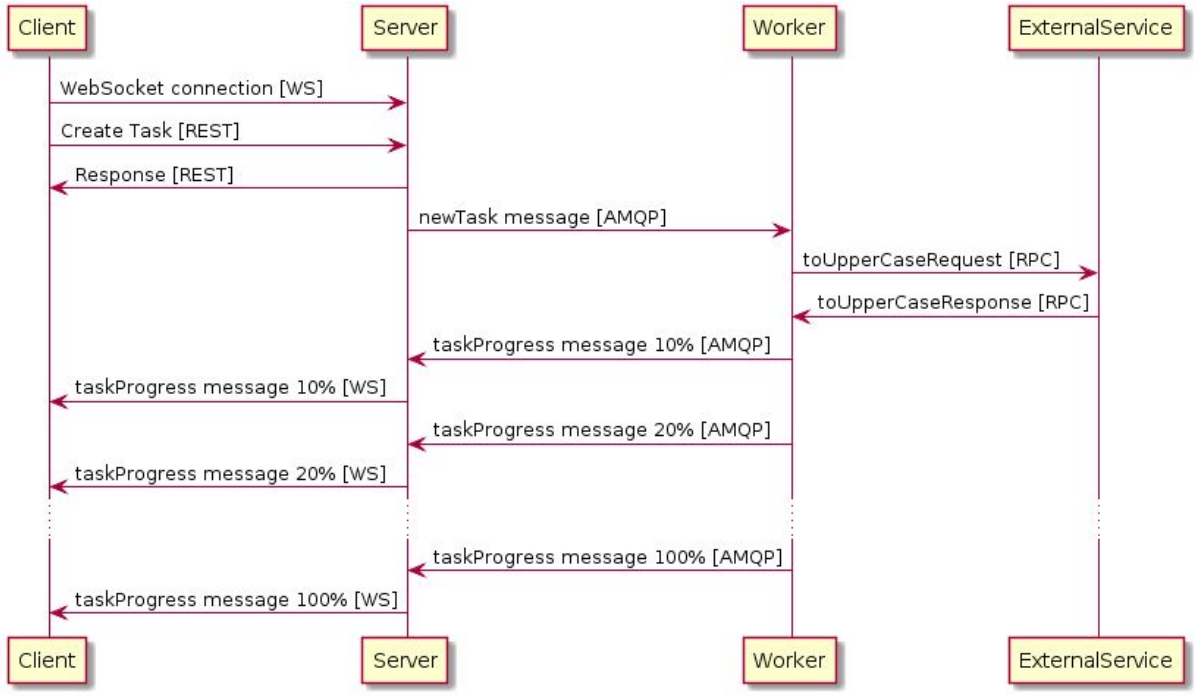
Práctica 3. Protocolos

Enunciado

Se pide crear una aplicación formada por cuatro módulos:

- Client
- Server
- Worker
- External Service

En el siguiente diagrama se muestra cómo se comunican estos módulos entre sí.



Cada uno de estos módulos tendrá la siguiente funcionalidad:

- **Client**
  - Código ejecutado en el contexto de un navegador web. Aquellos alumnos que no tengan conocimientos de desarrollo frontend, puede simular este cliente con una aplicación node.js.
  - Podrá solicitar la realización de una tarea invocando una API REST proporcionada por el servidor.
  - Se conectará por WebSocket para recibir mensajes con el progreso de la tarea que acaba de lanzar (que mostrará en la consola).
  - Al crear una tarea se envía un texto.

    - Si el cliente se implementa para ser ejecutado en el navegador web, estará alojado en la carpeta “public” del server.
- **Server**
    - Servidor implementado con Node.js y Express
    - Implementa una API REST con la que los clientes pueden solicitar tareas
    - Se asumirá que la realización de las tareas lleva un tiempo considerable y no se pueden llevar a cabo en el contexto de una petición REST.
    - Cada vez que se solicita una tarea, el servidor retorna inmediatamente al cliente un id de la tarea y la tarea se prepara para ejecutarse en segundo plano.
    - El cliente puede consultar por el porcentaje de completitud de la tarea con la API REST usando el id creado con anterioridad.
    - El formato de la API y los mensajes puede ser el siguiente:
        - URL /tasks/
        - Body: { text: 'content' }
        - Response: { id: 1, text: 'content', progress: 0, completed: false }
    - El cliente puede conectarse a un WebSocket para recibir un evento cada vez que la tarea actualiza su porcentaje de completitud.
    - El formato de actualización del progreso de la tarea puede ser:
        - { id: 0, completed: false, progress: 10 }
    - Cuando la tarea finaliza, el mensaje puede tener el formato:
        - { id: 0, completed: true, progress: 100, result: ‘CONTENT’ }
    - Se asume que las tareas son computacionalmente intensivas y se ejecutan en el servicio Worker.
    - El Servidor notifica al Worker mediante una cola de mensajes AMQP con RabbitMQ. Ese mensaje llevará el texto enviado por el cliente.
        - La queue por la que se envían estos mensajes se llamará “newTasks” y se asumirá ya creada en RabbitMQ
        - Los mensajes estarán codificados en JSON con el formato:
            - { id: 0, text: “content” }
    - Para simplificar el desarrollo, se asume que el servidor sólo puede gestionar una única tarea a la vez. Si un cliente intenta crear una nueva tarea mientras la anterior está en proceso, se le devolverá un error.
- **Worker**
    - Worker implementado en Java y SpringBoot.
    - Atiende peticiones de tarea mediante una cola AMQP
    - Para iniciar la tarea, se obtiene el texto del mensaje AMQP y se hace una llamada a un servicio externo que ofrece una API gRPC.
    - Este servicio externo recibirá el texto original y devolverá un nuevo texto procesado.
    - El worker simula la ejecución de una tarea costosa progresando un uno por cierto cada segundo (Bloqueando el hilo con Thread.sleep(1000)).
    - Para la notificación del porcentaje de completitud al Servidor se usará otra cola AMQP.
        - La queue por la que se envían estos mensajes se llamará “tasksProgress” y se asumirá ya creada en RabbitMQ

        - Los mensajes estarán codificados en JSON con el mismo formato que recibirá el cliente vía WebSockets, es decir:
            - { id: 0, completed: false, progress: 10 }
    - El resultado de la tarea se envía desde el Worker al servidor mediante un mensaje especial de “completado” con el texto procesado por el servicio externo.
        - Se usará la misma cola que para el envío del progreso (“tasksProgress”)
        - El formato será el mismo que recibirá el cliente por WebSockets:
            - { id: 0, completed: true, progress: 100, result: ‘CONTENT’ }
- **External Service**
    - Ofrece una API gRPC usada por el Worker al realizar su tarea.
    - Recibe un texto y lo devuelve en mayúsculas.
    - Se podrá implementar en Java con SpringBoot o en Node.
    - El formato de la API gRPC será:
        - Request: { string text }
        - Response: { string result }
        - Service: toUpperCase(Reques) returns Response

La práctica se estructurará de la siguiente forma:

- Una carpeta con las siguientes subcarpetas: server, worker y external service.
- Si el cliente se implementa como una aplicación node, también habrá una carpeta client
- Habrá un script node (install.js) en la raíz que permitirá la instalación de las dependencias después de descomprimir el código fuente de la práctica. Este script se ejecutará como: ‘node install.js’.
- A continuación se muestra un install.js siguiendo las convenciones vistas en los ejemplos:

```
const { spawnSync } = require('child_process');

function exec(serviceName, command, cwd){

   console.log(`Installing dependencies for [${serviceName}]`);

   console.log(`Folder: ${cwd} Command: ${command}`);

   spawnSync(command, [], { cwd, shell: true, stdio: 'inherit' });
}

exec('externalservice', 'npm install', './externalservice');
exec('server','npm install', './server');
exec('worker','mvn install', './worker');
```

- Habrá un script node (exec.js) en la raíz que permitirá la ejecución de los diferentes módulos. Este script se ejecutará como: ‘node exec.js’.
- A continuación se muestra un ejemplo de exec.js considerando que el client se ejecuta en el navegador web y siguiendo las convenciones vistas en los ejemplos:

```
const { spawn } = require('child_process');

function exec(serviceName, command, cwd){

   console.log(`Stated service [${serviceName}]`);

   let cmd = spawn(command, [], { cwd, shell: true });

   cmd.stdout.on('data', function(data){
       process.stdout.write(`[${serviceName}] ${data}`);
   });

   cmd.stderr.on('data', function(data){
       process.stderr.write(`[${serviceName}] ${data}`);
   });
}

exec('externalservice', 'node src/server.js', './externalservice');
exec('server','node src/server.js', './server');
exec('worker','mvn spring-boot:run', './worker');
```

Formato de entrega

La práctica se entregará teniendo en cuenta los siguientes aspectos:

- La práctica se entregará como un fichero .zip. Dentro del fichero .zip deberá haber 4 carpetas: Cliente, Servidor, Worker, Servicio Externo. El nombre del fichero .zip será el correo URJC del alumno (sin @alumnos.urjc.es).
- En la raíz del fichero .zip debe existir un fichero README.md que explique cómo ejecutar los cuatro módulos de la aplicación partiendo del código fuente del .zip.
- Los proyectos se pueden crear con cualquier editor o IDE (para el proyecto Node se recomienda VSCode).
- La práctica se entregará por el aula virtual con la fecha indicada.
- No se deberán incluir en el .zip las carpetas node_modules ni target ya que contienen dependencias o código compilado que se puede regenerar partiendo de los fuentes.

Las prácticas se podrán realizar de forma individual o por parejas. En caso de que la práctica se haga por parejas:

- Sólo será entregada por uno de los alumnos
- El nombre del fichero .zip contendrá el correo de ambos alumnos separado por guión. Por ejemplo `p.perezf2019-z.gonzalez2019.zip`