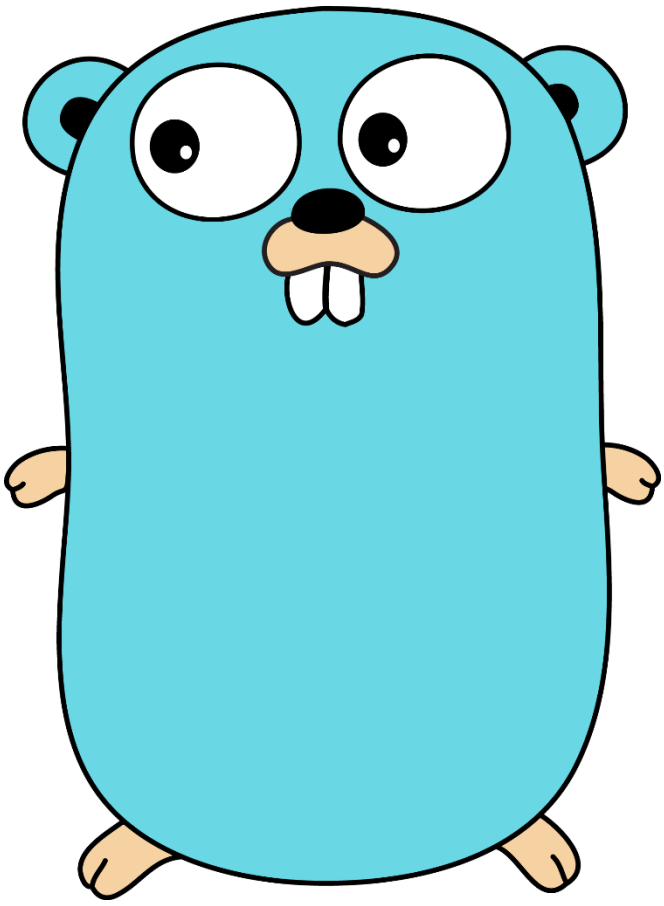
Proyecto: Sistemas Distribuidos - RPC

goChar – Reconocimiento de caracteres distribuido



David Barranco Alfonso
Jesús Bocanegra Linares
26-4-2016

Contenido

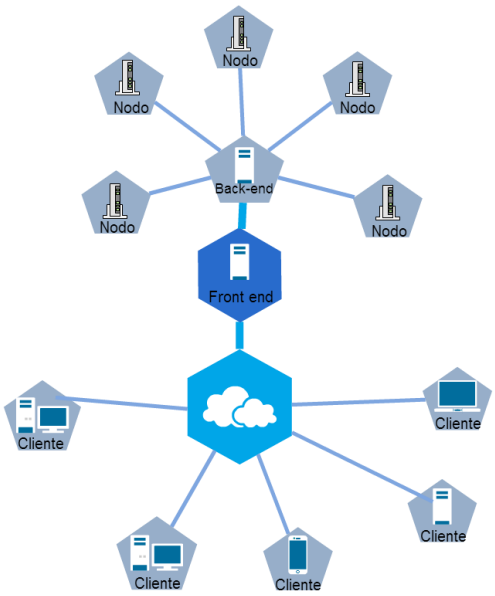
Introducción

Este es un proyecto que se ha realizado para la asignatura de **sistemas distribuidos de 4º de GITT.** En esta asignatura se propone realizar un proyecto basado en las **Remote Procedure Calling (RPC), o llamadas a procedimientos remotos.**

Una llamada a procedimiento remoto consiste en un protocolo que permite a un software o programa ejecutar código en otra máquina remota sin tener que preocuparse por la comunicación. Las RPC son muy utilizadas dentro de la comunicación cliente-servidor, siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando este de vuelta el resultado de dicha operación al cliente.

En nuestro caso hemos decidido realizar un sistema de reconocimiento de caracteres óptico distribuido. Un **sistema de reconocimiento de caracteres óptico (OCR en inglés)** es un proceso dirigido a la digitalización de textos, los cuales identifican automáticamente a partir de una imagen símbolos o caracteres que pertenecen a un determinado alfabeto, para luego almacenarlos en forma de datos. El proceso básico que se lleva a cabo es convertir el texto que aparece en una imagen en un archivo de texto que podrá ser editado y utilizado como tal por cualquier otro programa o aplicación que lo necesite. Partiendo de una imagen, el reconocimiento de estos caracteres se suele realizar comparándolos con unos patrones o plantillas que contienen todos los posibles caracteres.

Partiendo de la premisa de la asignatura de realizar un proyecto basado en las **Remote Procedure Calling** hemos decidido realizar las operaciones de reconocimiento de caracteres de forma distribuida. Es decir, la estructura básica del proyecto (sobre la que profundizaremos después) es la siguiente:



Muchos clientes desde distintas plataformas pueden acceder al servidor front-end que sirve una página web donde al usuario se le presenta una interfaz donde puede subir una imagen para reconocer un carácter.

La imagen se envía mediante un HTTP POST al servidor back-end, distribuye la carga de trabajo a uno de los nodos que tenga libre para procesar la imagen. Estos nodos ejecutan un script en **Python** programado usando la librería **TensorFlow**. Gracias a esta son capaces de descifrar el carácter que contiene la imagen usando técnicas de redes neuronales convolucionales. Una vez conseguido este carácter, la respuesta se envía de vuelta al servidor back-end. De esta forma el usuario puede ver el resultado de la operación en su navegador.

Ya que es un proyecto donde la heterogeneidad de las plataformas donde se va a trabajar es evidente (un nodo puede ser un ordenador con Windows, una Raspberry Pi con una distribución Linux o un servidor basado en Mac OSX, por ejemplo) hemos decidido utilizar el lenguaje **Go**.

Lenguaje Go. De dónde viene y hacia dónde va

Como se ha comentado antes, la heterogeneidad de las plataformas usadas en este proyecto es evidente. Debido a que los nodos encargados del procesamiento de las imágenes pueden estar ubicados en distintos tipos de máquinas con distintos tipos de sistemas operativos y distintas arquitecturas de procesador es conveniente basar toda la parte interna del proyecto en un lenguaje que pueda ser compilado para diferentes tipos de máquinas de manera relativamente fácil.

Go es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de **C**. Ha sido desarrollado por Google y sus diseñadores iniciales son:

Robert Griesemer

-V8 Javascript Engine

-Diseño de Java Hotspot

-Strongtalk

Ken Thompson

-Multics OS (60's)

-Unix

-Lenguaje B, UTF-8

Rob Pike

-Plan9

-UTF-8

Se puede decir que "nació" en 2009 y es un lenguaje al que hoy en día están migrando sus servidores muchas empresas (Twitter, Docker, Google, Tumblr…).

Algunas de sus características básicas:

- *Compilado*: No se necesita instalar ningún programa para que el programa que desarrolles funcione en el sistema operativo para el que lo compilaste.
- *Estáticamente Tipado*: Las variables son tipadas de manera estática, así que, si la variable **x** se definie como entera, será entera durante todo su alcance.
- *Concurrente*: Está inspirado en CSP (Communicating sequential processes — de Charles Antony Hoare, ganador del premio Turing en 1980).
- *Uso poco usual de POO*: Go no usa clases, no usa herencia y el uso de interfaces se realiza de manera implícita. Esto con el fin de mejorar el rendimiento al momento de diseñar el software.

Es por lo anterior que su aprendizaje es bastante sencillo, sobre todo viniendo de lenguajes de programación orientada a objetos como C++ o JAVA.

Paquetes de GO

En Go no se trabaja con librerías, se trabaja con paquetes. Estos están formados por varios archivos ".go" los cuales ayudan a organizar el código y añaden funcionalidades.

Gracias a la gran comunidad de desarrolladores que hay detrás de Go puedes encontrar un sinfín de paquetes dedicados a tareas muy específicas. Por ejemplo, el paquete **resize** se encarga de ofrecer funciones para el redimensionamiento de imágenes, apoyándose en los paquetes de procesamiento de imágenes que ofrece Go.

Dado que existen muchos desarrolladores detrás de Go creando nuevos paquetes para ofrecer nuevas funcionalidades, Go ha de tener una gran integración con **GitHub** para poder integrar paquetes sin tener que descargar el repositorio del desarrollador.

Si quisiéramos añadir a nuestro software las funciones de redimensionamiento de imágenes, deberíamos hacer:

```
       $Go get github.com/nfnt/resize            //añadimos el paquete para trabajar
con el

       //Y en el código…

       Import (

              "github.com/nfnt/resize"

            )
```

Para la realización del proyecto hemos estudiado varios paquetes con los que implementar el sistema de conexión entre el servidor back-end (que a su vez hace de maestro) y los esclavos:

- Paquete net/rpc: Es el paquete básico que trae golang para trabajar con las rpc's. No implementa comunicación bidireccional entre 2 máquinas, con lo que imposibilitaba la idea que teníamos de que el maestro y esclavo fueran servidor y cliente a la vez. Fue descartado.
- Paquete goRPC, creado por valyala: Paquete basado en **net/rpc.** Añade muchas funcionalidades no incluidas en el paquete por defecto de rpc y simplifica bastante la creación de servidor y cliente. Entre algunas de estas funcionalidades que hizo decantarnos por este.
    - Minimiza el número de llamadas al sistema
    - Minimiza la carga de CPU
    - Minimiza el número de conexiones TCP
    - Soporta cargas de > 40.000 peticiones por segundo

  La primera versión del proyecto se creó con este paquete. Cuando comenzamos las pruebas, descubrimos que no soporta comunicaciones bidireccionales. Por lo que tuvimos que descartar el proyecto y buscar otro paquete que si implementara esto.

- Paquete rpc2, creado por cenk: Al igual que **goRPC** está basado en el paquete net/rpc. El principal objetivo de este paquete es ser **simple** e implementar **comunicaciones bidireccionales**. Fue el paquete con el que terminamos realizando el proyecto. Simplifica incluso más la creación de un cliente y un servidor para conexiones RPC, algo que nos venía bastante bien viendo el tiempo que nos quedaba hasta la entrega del proyecto.

Otros paquetes, quizás con menos importancia que los anteriores debido al propósito del proyecto fueron los siguientes:

- **Container/list:** Para trabajar con listas enlazadas
- **Image:** Para trabajar con imágenes
- **Net/http:** Conexión cliente-servidor y creación servidor http
- **Image/color:** Para la conversión de imágenes a escala de grises
- **Math:** Operaciones sobre los píxeles para la conversión de estas a escala de grises
- **Log:** Impresión de logs en la consola del servidor/cliente
- **Net:** Aportaba funciones de red
- **Strconv:** Conversión de cadenas

Funcionamiento del paqute RPC2

Dado que este proyecto se ha realizado haciendo uso de este paquete es conveniente comentar como se realiza la comunicación cliente-servidor a nivel de código.

Tanto el cliente y el servidor deben servir distintos métodos para proporcionar una comunicación bidireccional entre ellos. Para exportar un método debemos:

- Programar la función que vayamos a exportar siguiendo este formato:

  ```
  func RecibeImagen(client *rpc2.Client, args *Args_RecibeImagen, reply
  *Reply_RecibeImagen) error
  ```

  Debe recibir un objeto de tipo cliente, unos argumentos de entrada y de salida y un tipo error que devuelve (aparte de los argumentos de salida).

- Crear un objeto de **tipo Cliente**

  ```
  clt = rpc2.NewClient(conn)
  ```

  Aquí creamos el objeto cliente, usando las funciones que nos proporciona el paquete, debemos pasarle por parámetros un objeto de **tipo Conn** (o tipo conexión):

  ```
  conn, _ = net.Dial("tcp", "150.214.182.97:12345")
  ```

  El objeto tipo conexión lo obtenemos utilizando el paquete **net**, la función que devuelve este objeto es **Dial**. Esta función se encarga de crear una conexión TCP(en nuestro caso) a la dirección que le pasemos por parámetros, simple.

- Crear un manejador del cliente:

  ```
  clt.Handle("RecibeImagen", RecibeImagen)
  ```

  En el cual le pasamos por parámetros los métodos que queremos exportar

- Lanzar el cliente:

  ```
  go clt.Run()
  ```

  La función **Run** se encarga de poner a funcionar el cliente. A partir de este punto ya estaríamos exportando el método.

  Se le incluye la sentencia **go** delante para que el propio sistema cree un hilo paralelo a la ejecución del programa donde el cliente esté esperando para ejecutar ese método cuando le llegue la petición a ese método.

Si ahora quisiéramos acceder a un método del cliente o del servidor, deberíamos hacer lo siguiente:

- Crear un nuevo objeto de tipo cliente apuntando a la dirección de la máquina que exporte los métodos
- Llamar a la función **Call**

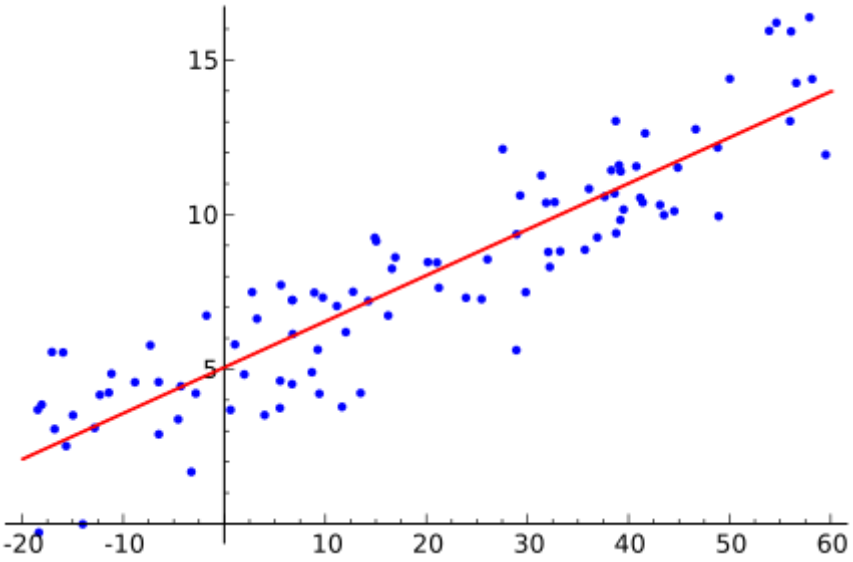
```
err = clt.Call("CierraConexiones", identificador_nodo,
&identif_desconex)
```

Llamamos a la función Call donde le pasamos por parámetros el nombre del método al que queremos acceder, el argumento de entrada y la dirección de una variable que se rellenará al ejecutar ese método. Tan simple como eso.

Reconocimiento de caracteres en Python

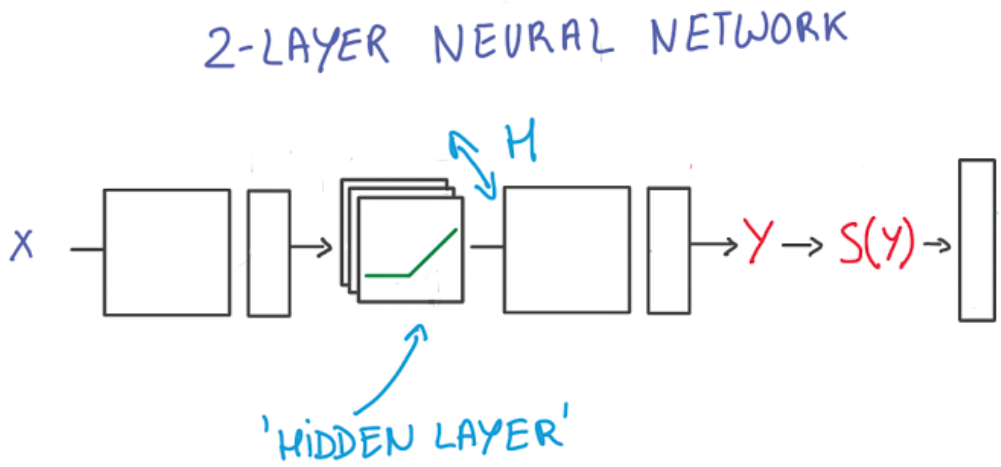
Cada nodo ejecuta un script en Python para el reconocimiento de los caracteres. El programa toma una imagen PNG de 28x28 píxeles en escala de grises. Su única salida es el carácter que ha reconocido.

Para conseguir reconocer los caracteres, se basa en un conjunto de redes neuronales y convolucionales. Tras pasar por estas capas, se somete a la imagen a una regresión logística para determinar de qué carácter se trata.



1. Regresión logística

Los coeficientes de las matrices utilizadas en las capas anteriores, así como en la regresión, se obtuvieron mediante un proceso de entrenamiento usando un juego de ejemplos. El entrenamiento lo realiza otro script. Las matrices obtenidas se serializan y se almacenan en un archivo que el programa encargado del reconocimiento puede cargar.



Estructura del proyecto

Servidor Front-end

Se trata simplemente de un servidor Apache que sirve la página del proyecto y sus recursos asociados.

Servidor Back-end – Maestro

El programa de Back-end es al mismo tiempo un servidor HTTP y de RPC. El protocolo HTTP le permite interaccionar con el usuario, esto es, aceptar imágenes y exportar los resultados. La parte de RPC le permite comunicarse con los nodos esclavos.

HTTP

Se trata de un servidor muy simple que implementa dos handlers para responder a dos direcciones:

- subir: Admite un método HTTP POST con la imagen a procesar. Responde con el id de trabajo que se ha asignado al usuario.
- estado: Recibe como parámetro en la petición el id de trabajo por el cual se pregunta. Devuelve el resultado de la ejecución del programa de reconocimiento en uno de los nodos.

Añade a las respuestas las cabeceras CORS (Cross-Origin Resource Sharing) para permitir su llamada mediante AJAX desde otro sitio web, es decir, el front-end:

- **Access-Control-Allow-Credentials:** `true`
- **Access-Control-Allow-Headers:**
  `Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token`
- **Access-Control-Allow-Methods:**
  `POST, GET, OPTIONS, PUT, DELETE`
- **Access-Control-Allow-Origin:** *

RPC

Los métodos que exporta para que el cliente pueda utilizarlos son:

- `func AceptaConexiones(client *rpc2.Client, args *Args_Conexiones, reply *Reply_Conexiones) error`

  Este método es el encargado de establecer la comunicación con el cliente. Cuando un esclavo se quiere conectar con el maestro utiliza este método, donde en el primer parámetro de la función le envía su **tipo Cliente** con el que el maestro podrá establecer una comunicación con el cliente, despreocupándose este de la dirección IP del esclavo. También se le envía un tipo boolean (ya que hay que enviar algo obligatoriamente). En la respuesta de este método se le asigna al esclavo el id con el que haremos referencia a él después.

  La función principal de este método es guardar los datos de cada esclavo en una lista enlazada donde se guarda la información de cada esclavo. Como en golang no hay clases, la estructura de un tipo **Nodo (esclavo)** es la siguiente:

```
type Nodo struct {
	idTrabajo int
	idNodo    int
	cliente   *rpc2.Client
	resultado byte
}
```

  Como se ve, se guarda un id de trabajo utilizado para comprobar el estado del trabajo de un nodo, un id de Nodo para saber que nodo envía información, su tipo Cliente (con el que se establece comunicación) y un byte para almacenar el carácter que resulta después de la conversión en ASCII.

  Cada vez que llegue una nueva petición de conexión de un esclavo se añade un elemento más a la lista enlazada.

- `func CierraConexiones(client *rpc2.Client, args *Args_Conexiones, reply *Reply_Conexiones) error`

  Este método es el inverso al anterior. Se recibe un id de nodo previamente asignado y se responde un boolean, a modo de debug.
  Este método se encarga de buscar el nodo en la lista enlazada y quitarlo de la misma para que no se le vuelvan a asignar trabajos.
  Después de la llamada a este método en el esclavo, se cierra el programa.

- `func RecibeRespuesta(client *rpc2.Client, args *Args_RecibeRespuesta, reply *Reply_RecibeRespuesta) error`

Gracias a este método recibimos por parámetros una estructura en la que se incluye un entero con el id del nodo para poder encontrarlo en la lista y el byte donde está el carácter donde está el carácter.
Una vez recibido esto se actualiza la información del objeto nodo donde se almacena el byte.

Cuando el cliente pregunte desde la web cual es el estado de su trabajo las funciones del servidor HTTP buscarán en la lista y comprobarán si se ha actualizado ese byte. Si es así, se enviará el resultado al cliente

Esclavo

El esclavo es el encargado de conectarse al Maestro para proporcionar el servicio de reconocimiento de caracteres. Este está completamente programado en Go. Las funciones que realiza este son:

- Conectarse (y registrarse) con el maestro
- Esperar a que se le asigne un trabajo
- Recibir la imagen y convertirla a formato .png
- Llamar al script de Python que realiza la conversión de caracteres pasándole por parámetros la imagen que acaba de exportar
- Leer la salida del programa para recoger el byte que devuelve el script de Python con el código ASCII del carácter
- Enviar el carácter al servidor
- Queda a la espera de que le envíen otro trabajo

El método que exporta para que el servidor pueda utilizar es:

- `func RecibeImagen(client *rpc2.Client, args *Args_RecibeImagen, reply *Reply_RecibeImagen) error {`

Este método recibe un objeto **tipo imagen** y devuelve un boolean, para comprobar que la llamada al método se ha realizado correctamente. Dentro de este método se realizan las llamadas al script de Python y se llama al método que exporta el servidor **RecibeRespuesta** devolviéndole ahí el carácter que este necesita.

Ejecución del proyecto

En esta sección se muestra una ejecución de ejemplo para ver el funcionamiento del sistema. Lo primero que tendremos que hacer será iniciar el back-end/maestro y conectar a él uno o varios nodos.

En la ejecución del esclavo podremos ver logs para seguir la ejecución del programa:

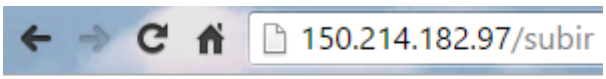
```
PS C:\Work\github - goRPC\GoChar> go run .\esclavo_rpc2.go
2016/04/28 11:02:56 Esclavo de goRpc iniciado.
2016/04/28 11:02:56 Conexion a servidor RPC....
2016/04/28 11:02:56 Conectado correctamente a servidorRPC.
2016/04/28 11:02:56 Registrado en servidorRPC correctamente.
```

Y de la misma forma en el servidor back-end:

```
PS C:\Work\github - goRPC\GoChar> go run .\http.go
2016/04/28 11:02:56 Conectado cliente con id:  0
2016/04/28 11:04:11 Conectado cliente con id:  1
```

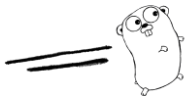
Entonces podremos acceder al servidor front-end donde dispondremos de la interfaz para poder subir las imágenes al servidor back-end:

Una vez se suba una imagen al servidor backend este la redimensionará y la convertirá a una escala de grises. Una vez hecho esto la enviará a uno de los esclavos que estén libres para que este inicie el proceso de la conversión. Algo a lo que es importante hacer referencia es que ya que estamos trabajando con id de nodos para asignar los diferentes trabajos conseguimos que los esclavos hagan un **callback** para devolver el resultado de la conversión. Así conseguimos que el maestro no quede bloqueado esperando recibir el resultado de la conversión. En los logs del servidor podremos ver a que esclavo se le ha asignado el trabajo:



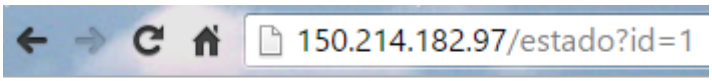
1

De forma invisible al usuario (aunque en esta versión sin contenido si podemos verlo) el servidor back-end nos devolverá el id del nodo al que se le ha asignado el trabajo. Gracias a esto, el cliente preguntará cada cierto tiempo el estado de ese nodo, esperando recibir la respuesta. Si no se hiciera esto la petición HTTP podría cancelarse al llegar al timeout, perdiendo así ver el resultado.



Cuando el esclavo termine su trabajo, este enviará la respuesta al servidor back-end, haciendo que este actualice el campo del objeto nodo (esclavo) destinado a almacenar la respuesta.

En la próxima pregunta del cliente sobre el estado del trabajo este le devolverá el carácter que ha obtenido en la digitalización de la imagen. Después el esclavo volverá a quedar libre y a la espera de un nuevo trabajo.



G

Con esto quedaría terminada la ejecución normal que un usuario haría del sistema.

Herramientas utilizadas

Para el desarrollo de este proyecto, las herramientas que hemos utilizado han sido:

- **LiteIde**: Pequeño IDE para programar en Go. Es de código abierto y funciona en varias plataformas. Incluye un debugger de código, auto identado, intellisense y mucho más. Ha resultado muy útil en el desarrollo del código de Go.

- **Github**: Para control de versiones del código. Creamos un repositorio (este) en el que fuimos subiendo las diferentes versiones que íbamos consiguiendo. Se puede ver en el repositorio que hay 2 ramas, en la rama secundaria se desarrolló el cambio de paquete de RPC's de Go.

- **Dropbox**: Para compartición de archivos.

- **Waffle.io:** Unido a github, se utiliza para llevar una lista de tareas (siguiendo la metodología **SCRUM**) de los issues que se van abriendo en GitHub. Como nosotros partimos de un repositorio vacío añadimos varios issues con las distintas tareas que teníamos que realizar. Nuestro waffle.io -> link

- **Comunicación:** Whatsapp, Skype… Herramientas básicas de comunicación.

Conclusiones y trabajo futuro

Trabajo futuro

Es cierto que el reconocimiento de un carácter en una imagen hoy en día no es muy útil, pero este proyecto nos sirve como base para trabajar con algo más difícil aún. La idea original del proyecto era enviar al servidor back-end una página de un libro escrito en castellano antiguo, la cual se distribuyera entre distintos nodos para al final conseguir un texto completo.

Las dificultades que suponía el realizar ese proyecto no estaban relacionadas con las RPC's, si no con el tratamiento de la imagen en distintas partes para distribuir a los nodos, contando con que el tamaño de los espacios no es uniforme en una página y la tipografía de una letra puede cambiar a lo largo de un texto.

Por lo tanto, se podría trabajar con esta plataforma para en un futuro realizar un reconocimiento óptico de páginas de libros distribuido.

Conclusiones

Este trabajo ha sido uno de los más interesantes que hemos realizado a lo largo de nuestra carrera en la universidad. Hemos conseguido algo que al principio dudábamos si podríamos, el aprender un lenguaje nuevo desde cero para aplicarlo a un proyecto. Es cierto que no ha sido fácil, pero el resultado es más que satisfactorio.

También hemos podido profundizar un poco más en el estudio de las RPC, cosa que estamos seguros que utilizaremos en otros proyectos, debido a la facilidad que ofrece a la hora de conectar diferentes máquinas y la rapidez que aporta esto al desarrollo de un proyecto.

Sin más, decir que estamos muy contentos y orgullosos del resultado.

David Barranco Alfonso

Jesús Bocanegra Linares