Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Grado

DISTRIBUCIÓN DE CARGAS DE TRABAJO ENTRE MÚLTIPLES CLIENTES MEDIANTE TECNOLOGÍAS WEB

Alumno: Francisco Jesús Ruiz López

Tutor: Prof. D. Francisco Charte Ojeda
Tutor: Prof. D. Antonio Jesús Rivera Rivas
Dpto: Informática

Junio, 2019

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Francisco Charte Ojeda y Don Antonio Jesús Rivera Rivas, tutores del Trabajo de Fin de Grado titulado: **Distribución de cargas de trabajo entre múltiples clientes mediante tecnologías web**, que presenta Francisco Jesús Ruiz López, autorizan su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Junio de 2019

El alumno:

Francisco Jesús Ruiz López

Los tutores:

CHARTE OJEDA FRANCISCO - 25996290B
Firmado digitalmente por CHARTE OJEDA FRANCISCO - 25996290B
Fecha: 2019.06.21 19:14:55 +02'00'

Francisco Charte Ojeda

RIVERA RIVAS ANTONIO JESUS - 26015863B
Firmado digitalmente por RIVERA RIVAS ANTONIO JESUS - 26015863B
Fecha: 2019.06.21 18:21:32 +02'00'

Antonio Jesús Rivera Rivas

Tabla de contenido

1. Introducción

En esta sección se hablará brevemente de los fundamentos del trabajo, indicando cómo estará estructurado y los diferentes aspectos que desarrollará cada apartado de la memoria.

1.1. Introducción al proyecto

El trabajo se fundamenta en el estudio y desarrollo de un sistema que explore las posibilidades de las tecnologías *web* a la hora de dividir una carga de trabajo y repartirlo entre un grupo de usuarios, con el propósito de obtener una solución similar a la que se obtendría de su ejecución al completo.

En la actualidad vivimos rodeados de dispositivos pequeños que se integran a nuestro alrededor, pero que dadas sus características específicas como pueden ser su sistema operativo, resulta complicado generar una herramienta universal que aplique dicha técnica, que sea compatible y que genere unos datos fiables y utilizables para otros fines.

Para dar lugar a dicha solución se estudiarán tecnologías que se utilizan en la actualidad, que dan lugar a lo que conocemos cómo internet y que servirán como pilares para desarrollo de una solución.

1.2. Motivación

En la actualidad la resolución de problemas de cómputo intensivo requiere de equipos costosos que precisan de una importante inversión inicial y que necesitan que la carga de trabajo que van a procesar justifique su adquisición.

En respuesta a este problema nace la idea de poder dividir un problema en tareas más simples, con un conjunto de datos reducido y que de alguna manera seamos capaces de ejecutarlo en un grupo de usuarios, utilizando para ello sus equipos en periodos de inactividad, de forma que colaboren para la obtención del

mismo resultado que obtendríamos de una tarea de cómputo realizada en una única máquina.

Como consecuencia de crear dicho sistema lograríamos una reducción teórica de los costes en inversión de equipos y del tiempo de ejecución, de manera que, a lo largo de este trabajo estudiaremos los pasos para crear dicho sistema y ver si realmente es tal su propósito.

1.3. Objetivos

- Estudio detallado de las especificaciones del protocolo HTTP, vehículo que se empleará para comunicar a los clientes con el servidor. Un repaso a nivel teórico de en que se basa el protocolo y su evolución.
- Estudio del lenguaje de marcado HTML y las hojas de estilos CSS, así como del lenguaje ECMAScript. Estos serán los elementos esenciales con los que funcionarán los clientes.
- Selección de una tecnología de servidor *web*, incluyendo lenguaje (PHP, Node.js, Ruby, etc.) y base de datos, para el desarrollo de la parte servidor de la solución. Se evaluarán las posibles alternativas y se seleccionará una para el desarrollo del proyecto.
- Diseño e implementación de un protocolo, al nivel de la capa de aplicación de la arquitectura de red, que facilite la comunicación con los clientes, distribución del trabajo y recolección de resultados.
- Implementación de la solución que, empleando el protocolo previo, distribuya una carga de trabajo concreta entre un conjunto amplio de clientes, incluyendo la recopilación de resultados y el análisis de estos.
- Redacción de los manuales de instalación y uso de la solución desarrollada, así como de la memoria correspondiente del TFG.

1.4. Metodología

- Revisión bibliográfica relativa al desarrollo de aplicaciones basadas en tecnologías *web*.
- Estudio de especificaciones de estándares, incluyendo HTTP, HTML, CSS, ECMAScript, etc.
- Análisis de los requerimientos de la solución a desarrollar, incluyendo entre sus requisitos la necesidad de que el software sea multi-plataforma y multi-dispositivo.
- Evaluación de alternativas y selección justificada de una tecnología de desarrollo en el servidor.
- Diseño e implementación del protocolo de capa de aplicación que facilitará la comunicación.
- Diseño e implementación de la parte cliente y servidor, incluyendo interfaz de usuario.
- Evaluación del sistema mediante la distribución de una carga de trabajo real entre el mayor número de dispositivos posible.
- Realización de pruebas.
- Redacción de documentación.

1.5. Estructura de la memoria

El documento redactado a continuación se divide en las siguientes secciones:

1. Introducción

Breve presentación de las motivaciones que han llevado a la realización de este trabajo. Recoge información acerca del documento, de los objetivos que debe cumplir este trabajo y de una descripción sencilla de la idea que lo motiva.

2. Información preliminar

Repaso a los contenidos que se han estudiado a lo largo de los años de formación en el grado, ya que van a ser necesarios en las siguientes fases de la creación de la aplicación.

3. Análisis

Aplicando las técnicas estudiadas en ingeniería del *software*, se realizará un estudio de los requisitos del sistema, roles que interactuarán con él, hasta llegar a una base firme en la que comenzar el diseño de la aplicación.

4. Planificación

Todo proyecto requiere de una temporización de las tareas que se van a realizar, así como de una estimación de los costes que tendrá que acometer el profesional que va a desarrollarlo.

5. Diseño

Partiendo del análisis antes realizado se representarán las diferentes operaciones que deberá contener nuestra aplicación, incluyendo las interacciones de los usuarios y los protocolos necesarios.

6. Implementación

Análisis de las herramientas, librerías e implementación del proyecto desde una perspectiva del programador, así como la creación de un prototipo que recoja los requisitos descritos en la fase de diseño.

7. Pruebas

Tras realizar una implementación del trabajo, se creará una prueba que aproveche el sistema que antes hemos creado para distribuir una carga de trabajo y comprobar que efectivamente nuestra solución es válida para el problema planteado.

8. Conclusiones

Valoración de los aspectos del trabajo realizado, así como el planteamiento de futuras mejoras o elementos que deberían replantearse en base al trabajo realizado. También se incluye una conclusión del autor tras la finalización.

9. Anexos

Apartado donde se incluyen manuales e información referente a bibliografía que amplía los documentos que componen este trabajo, que sirven para justificar y ampliar el contenido.

1.6. Introducción al problema

A continuación, paso a describir en detalle el problema que se nos plantea en este trabajo, evaluando posibles opciones de resolución.

La primera impresión que tenemos al analizar el problema es que se trata de un problema de **computación distribuida**, de forma que un sistema central, en este caso nuestra aplicación, distribuye una carga de trabajo a unos clientes, que de nuestro lado serían los usuarios *web*.

La computación distribuida es un proceso que utiliza una red de dispositivos que trabajan en paralelo con el propósito de ofrecer un servicio que reparte sus capacidades entre los dispositivos presentes.

Tradicionalmente se han creado aplicaciones que, utilizado técnicas y procesos de bajo nivel implementados mediante lenguajes de programación tales como C++ (1) o Java (2), interaccionaban usando la arquitectura de la máquina obteniendo un alto rendimiento, pero requiriendo para ello de la generación de códigos compilados que creados específicamente para dichas máquinas. La agrupación de estas máquinas ejecutando estas aplicaciones daban lugar arquitectura distribuida.

Los sistemas distribuidos (3) utilizan las capacidades de una red de computadores interconectados y que sincronizan su trabajo mediante un protocolo de

comunicación. Este procedimiento tiene tres principales ventajas: **concurrencia**, procesamiento **independiente** y **aislamiento** del sistema ante un fallo.

Los equipos que conforman el sistema distribuido pueden diferir en características físicas, pudiendo trabajar diferentes generaciones de computadores, permitiendo a los administradores de la instalación poder sustituir de forma paulatina las computadoras sin tener que realizar un gran desembolso de una sola vez, con la contra de tener que crear compilaciones específicas para cada arquitectura.

A la hora de manejar un fallo, un sistema distribuido permite que la caída fatal de uno de los equipos de la red no signifique la inutilización del sistema y la consecuente interrupción del trabajo. La parte correspondiente del trabajo que dicho equipo estaba ejecutando es replicable, y en consecuencia recuperable, y dado los datos están fraccionados, la pérdida es mínima.

1.6.1. Ejemplos de la computación distribuida

En este apartado explicaremos diferentes tipos de arquitecturas distribuidas que existen en la actualidad, y que se han diseñado desde el punto de vista un sistema distribuido.

1.6.1.1 Arquitectura orientada a servicios

Este tipo de arquitectura distribuida (4) se basa en la utilización de diferentes servicios interconectados mediante un sistema de comunicación. Estos provienen de diferentes fabricantes o tecnologías, de forma que cada uno se plantea como un elemento funcional independiente, al cual podemos acceder de manera remota con el fin de obtener su funcionalidad. Estos servicios representan una actividad muy concreta que solventa una necesidad del sistema que la contiene.

Los fundamentos de esta arquitectura se basan en que dichos servicios cumplan las siguientes características:

- Representan una actividad de la que se espera una salida en función de una entrada.
- Autonomía.
- Quien los utiliza no requieren conocer cuál es su funcionamiento (caja negra).
- Pueden hacer uso de otros servicios existentes.

En la actualidad, se ha dado un salto más, llegando a lo que se llama arquitectura de microservicios (5). Dichos microservicios son enlazados mediante una red, frente a una representación previa de todo dentro de la misma máquina. El único inconveniente es que se aumenta la dependencia de la red de conexión, y ante la falla de uno de estos enlaces, el conjunto global resultaría paralizado, lo cual nos lleva a creación de sistemas de réplicas y el consecuente aumento en los costos en infraestructuras. Estas réplicas son máquinas espejo que continúan la ejecución del servicio en caso de que las maquinas principales fallen.

1.6.1.2 Arquitectura cliente – cliente

Esta arquitectura (6) se basa en la distribución del sistema entre lo que se llaman *Peers* (Pares), clientes con los mismos privilegios sobre la red que comparten, también conocida como red *peer-to-peer* (P2P).

Los participantes ofrecen recursos de su máquina a la red, como podría ser almacenamiento, capacidad de procesamiento o ancho de banda, de forma que estos recursos están disponibles para el resto de los participantes, sin necesidad de un servidor central que los centralice. Del mismo modo, todo *peer* es a su vez consumidor de recursos, de forma que rompe con la clásica arquitectura cliente - servidor.

Conocemos ejemplos reales de éxito de este tipo de plataformas como puede ser BitTorrent (7) que se lleva utilizando desde 2001 para compartir archivos entre sus usuarios.

1.6.1.3 Arquitectura de un MMOG

Tal y como sus siglas indican, los juegos multijugador-masivos *online* (8) también están estructurados siguiendo una arquitectura distribuida, dada la gran carga que tienen que soportar en todo momento, lo que hace impensable que detrás exista un sistema monolítico que lo gestione todo.

Para ello dividen el mundo virtual en el que se mueve el jugador en zonas, que son gestionadas por diferentes servidores, pero que hacen transparente la transición de una a otra para el jugador. La necesidad de sincronización entre los diferentes elementos de la red, tanto jugadores como servidores hacen que aparezca el llamado *lag*, retraso entre las acciones del cliente y su correspondiente efecto en el mundo virtual, ocasionando que la interacción no se produzca con fluidez.

Dado que muchas de las plataformas de juego existentes tienen que dar soporte a escala mundial, en la mayoría de las ocasiones se opta por crear zonas geográficas específicas, de manera que la carga esté adecuada a una zona determinada, para así tener una mejor gestión de los recursos y ofrecer a los usuarios una mejor experiencia de juego. Se despliegan los servidores en regiones geográficas cercanas a los núcleos de usuarios más grandes, de forma que la comunicación con ellos es lo más directa posible y el tiempo de latencia se reduce considerablemente.

En la siguiente referencia (9) existe más información acerca de cómo los sistemas de juego actuales son capaces de dar servicio a 12 millones de jugadores al mes.

2. Información preliminar

En este apartado se repasarán conceptos estudiados a lo largo del grado y que serán de utilidad para sentar las bases para poder comenzar a estructurar y diseñar el trabajo a realizar.

2.1. Estudio del protocolo HTTP

2.1.1. Definición

El protocolo de transferencia de hipertexto (HTTP) (10) creado en 1996 por el *World Wide Web Consortium* (W3C) (11) y la *Internet Engineering Task Force* (IETF) (12), define la sintaxis que utilizan los elementos software de la arquitectura web para su comunicación, utilizando conexiones de red independientes para su funcionamiento. Su versión más actual es la 2.0 (13).

HTTP está orientado a conexión, de manera que cliente y servidor intercambian información mediante una conexión TCP (14), utilizando para ello un puerto de red que el administrador de la máquina servidora habilita para dicha acción. Comúnmente para las aplicaciones web se utiliza el 80.

2.1.2. Mensajes

Utilizando el texto plano, se crean mensajes que contienen la acción requerida, parámetros a usar. Además, se incorporan metadatos en la cabecera del mensaje que contienen información como a la autoría del mensaje o datos para la localización y la trazabilidad. El contenido del usuario se encuentra en el cuerpo y contiene los datos que el remitente quiere hacer llegar al otro interlocutor.

A continuación, se muestra un ejemplo de una petición HTTP de un cliente *web* solicitando el acceso al contenido de una página.

```
GET /index.html HTTP/1.1
 Host: www.example.com
 Referer: www.google.com
 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101
Firefox/45.0
 Connection: keep-alive
 [Línea en blanco]
```

2.1.3. Métodos del protocolo

Aquí se enumeran los principales métodos HTTP que se utilizan a la hora de navegar por la *web*:

- `HEAD`: utilizado para recuperar los datos de cabecera sin requerir el cuerpo del mensaje.
- `GET`: pide una representación de un recurso de forma no segura, ya que envía los parámetros a través de la URI (Identificador uniforme de recursos). Ejemplo:

  ```
  GET /images/logo.png HTTP/1.1
  ```

  Tras la ejecución de esta petición se recibe un recurso llamado *logo.png*.

- `POST`: envío de una solicitud similar a `GET`, solo que no utiliza la URI para enviar los parámetros, sino que los incluye dentro del cuerpo del mensaje.
- `PUT`: realiza una carga de un recurso, generalmente un archivo, con objetivo almacenarlo en el servidor.
- `DELETE`: borra un recurso del servidor.
- `TRACE`: solicita un mensaje cuyo contenido son metadatos con información relevante para la depuración y el diagnostico.
- `OPTIONS`: devuelve una lista con los métodos HTTP que soporta el servidor.

- `CONNECT`: nos permite saber si la navegación se realiza a través de un *proxy* (15), sirve también para comprobar que SSL (16) está activo.
- `PATCH`: similar a `PUT`, se utiliza para actualizar un archivo en el servidor.

A modo de mención, otros métodos existentes menos utilizados: `SEARCH`, `COPY`, `LOCK`, `UNLOCK`, `MOVE`, `MKCOL`, `PROPFIND`, `PROPPATCH`, `MERGE`, `UPDATE`, `LABEL`.

Cuando un servidor responde a una de estas solicitudes, genera un código de respuesta que resume si la operación es exitosa o no, en función de la resolución de la petición que ha enviado el cliente.

Esta tabla muestra un resumen de los códigos más comunes que se utilizan en los servidores *web*:

Código	Significado
1xx	Información y eventos en la comunicación
2xx	Eventos en el procesamiento de las peticiones
3xx	Redirecciones
4xx	Errores en las solicitudes del cliente
5xx	Errores las respuestas del servidor

1 Códigos de respuesta

Cada uno de estos códigos engloba mensajes relacionados (17), permitiendo la definición de otros por parte de los administradores del sistema en caso de que los necesiten para tareas de depuración y monitorización.

2.1.4. Utilidad en el proyecto

Como vemos el protocolo HTTP es la base para la distribución de contenido en una página *web*, desde un servidor hasta un cliente que utiliza un navegador *web*. En nuestro proyecto lo utilizaremos para la comunicación y distribución de los archivos necesarios, así como medio de transporte para las peticiones que los usuarios realicen, y de su respuesta por parte de la aplicación. Diseñaremos un protocolo que cubra las necesidades funcionales de ambos para que el proyecto funcione correctamente.

2.2. Lenguaje de marcado y hojas de estilo

En la actualidad, la gran mayoría de las páginas *web* basan la estructura de su contenido en HTML (18) (*Hyper Text Markup Language*, Lenguaje de Marcado de Hipertexto),y su versión mas actual la 5 (19). Las *webs* están estructuradas en etiquetas que organizan el contenido de la web para su correcta representación en el navegador.

Estas etiquetas definen el esqueleto del documento de forma que el navegador web interprete de manera correcta el contenido y lo muestre al usuario que lo ha solicitado.

2.2.1. Estructura de un documento HTML

`<html>`

Esta etiqueta sirve para abrir y cerrar el documento HTML, de manera que indicamos al navegador *web* que lo que hay entre ellas debe interpretarse como código en HTML.

`<head>`

Esta es la sección de cabecera, que no contiene contenido visible de cara a su representación de lado del cliente, pero que contiene metadatos, estilos y librerías que deben cargarse para el funcionamiento de la *web*. Se especifican también características de compatibilidad que requiere el navegador del cliente que accede al documento para garantizar una interpretación correcta.

`<body>`

Contiene todo el contenido visible por el usuario. Recogerá la información que se va a representar en el cliente, lo que podríamos considerar como la información que este quería visualizar cuando realizó la petición del documento HTML.

Todas estas etiquetas requieren de su correspondiente etiqueta de cierre, que indica al navegador el fin de una sección, siendo igual a la etiqueta de apertura, pero precedida de **‘ / ’**.

El aspecto mínimo de un documento en HTML será el siguiente:

```
<html>
     <head>
          // Cabeceras del documento
     </head>
     <body>
          // Contenido a mostrar al usuario
     </body>
</html>
```

Partiendo de este esqueleto, un navegador *web* puede transformarlo en una representación con la estructura definida que muestra el contenido que el usuario ha solicitado.

Es necesario cumplir con las especificaciones que el W3C recoge en una serie de documentos (20), de forma que se asegure el buen funcionamiento, tanto de la parte del servidor generando un contenido compatible, como de los diferentes navegadores *web* que existen, para que generen en todos los casos una representación similar para el usuario.

2.2.2. Estructura de una hoja de estilos CSS

La hoja de estilos basa su funcionamiento en establecer una serie de modificaciones a aplicar en el contenido definido dentro del documento HTML, utilizando para ello las etiquetas, clases e identificadores que se utilizan en la definición de la estructura.

Dichas modificaciones son los estilos, que dan forma y color al contenido, de manera que el usuario no visualice un documento en texto plano, favoreciendo la experiencia y transmitiendo aún más información.

A continuación, mostraré un ejemplo de cada uno de los identificadores que se utilizan en el navegador para aplicar los estilos:

```
P {       //Ejemplo de selector HTML
     color: red;
}

.parrafo {          //Ejemplo de clase
     background: blue;
}

#boton_aceptar {         //Ejemplo de identificador
    border-radius:1px;
}
```

En su última revisión, CSS3 (21) incorpora novedades a nivel de gestión del contenido multimedia en detrimento del uso de Flash (22), favoreciendo junto a HTML5 la mejora de compatibilidad de los documentos HTML y mejorando la información que los usuarios reciben mediante la *web*.

Al igual que en el caso de HTML, existen una serie de recomendaciones por parte de la W3C (23) en las que se recogen las directrices a seguir para conseguir una máxima compatibilidad.

2.2.3. Utilidad en el proyecto

Estas dos herramientas serán útiles para la creación de las vistas. Para ello definiremos las necesarias para asegurar que se cubren las necesidades de interacción de los diferentes usuarios de nuestra aplicación, asegurando una compatibilidad con la mayor cantidad de dispositivos.

2.3. Ejemplos reales del problema planteado

Al igual que nos vamos a plantear el uso de varios lenguajes o tecnologías, también debemos investigar que exista alguna aplicación que cubra las necesidades

a las que se enfrenta este trabajo, ya que volver a hacer algo que existe, y puede que incluso más eficiente que lo que vamos a crear, sea innecesario.

Existen algunas interpretaciones para este problema que mostraremos a continuación:

2.3.1.1 BoomerangJS

Como explican en su pagina *web* (24) se trata de una librería para *Javascript* inspirada en BOINC (25), un proyecto de la Universidad de Berkeley que aprovechar el tiempo de inactividad de tu PC para colaborar en proyectos científicos que requieren de altas necesidades de potencia de procesamiento.

El proyecto tiene desarrollado un cliente que recibe código por parte de un servidor central y este le devuelve el resultado de su ejecución. Aunque prometedor, el proyecto lleva cuatro años sin recibir actualizaciones.

Sin duda es una de las representaciones más aproximadas a lo que necesitamos, pero dado que no está al día, no podemos contar con que las librerías o las medidas de seguridad que se han ido solucionando a lo largo de estos cuatro años lo hagan una solución fiable.

2.3.1.2 deThread

Se trata de una librería (26) similar a la anterior, pero que está pensada para que el servidor trabaje de un modo *headless* (27), es decir, solo gestionando la distribución de trabajo y manejando posibles errores que remitan los clientes.

Incorpora elementos interesantes como un control de los usuarios y una implementación basada en WebWorkers (28), que mejora la ejecución en paralelo del lado del cliente. El proyecto lleva tres años sin recibir actualizaciones.

Es una implementación más compleja que emplea Sockets (29) para la comunicación con los clientes, lo que supone un protocolo más complejo, pero más

seguro, más complicado de programar. De la misma forma que el anterior el no estar actualizado supone una solución poco fiable.

2.3.2. Capacidad de cómputo de Javascript

Después de hablar de ejemplos existentes que tratan de solucionar este problema, me parece interesante introducir este apartado para evaluar las capacidades de Javascript a la hora de orientarlo al procesamiento, ya que puede no ser la respuesta a las necesidades del problema.

La información que encontramos acerca de cómo se ejecuta Javascript en nuestro navegador se basa en la existencia de un intérprete (30) que se encarga de realizar una compilación del código, que en realidad es una precarga de los métodos y variables definidas dentro del documento y de unas tareas de comprobación de la sintaxis de cara a evitar errores en tiempo de ejecución.

Por lo general, cada navegador posee su propia implementación del motor del intérprete, siendo la más eficiente hoy en día la V8 de *Chrome* (31).

Si investigamos un poco por la *web* vemos (32) que el rendimiento de Javascript es muy bueno (33), usando su entorno nativo fuera del navegador mediante Node.js, llegando a compararse al rendimiento obtenido en lenguajes como C++ o Java.

Entonces, porqué la tendencia es la de seguir utilizando aplicaciones de procesamiento a nivel de la arquitectura de la máquina, en vez de utilizar las capacidades que nos brinda el navegador *web*, que teóricamente, debería tener un rendimiento similar.

La realidad (34) es bastante diferente. La optimización de un código en Javascript es compleja, ya que las funcionalidades no están implementadas de igual manera en todos los intérpretes, con lo que la eficiencia de una función dentro de un navegador puede diferir mucho de la interpretación que realiza otro, llegando a existir diferentes niveles de soporte del estándar.

Otro de los problemas que nos encontramos es que las acciones sobre el DOM (35) son lentas. Cualquier interacción con la parte gráfica desde nuestro código en Javascript va a afectar de manera directa al rendimiento, pudiendo empobrecer la experiencia del usuario.

Como conclusión, podemos decir que Javascript es eficiente, pero la implementación actual por parte de los diferentes intérpretes y su incorporación dentro de los navegadores no es la opción más optima para su uso en computación.

2.3.3. Web Workers

Sería un error no hablar de los Web Workers (28) cuando estamos evaluando la capacidad de cómputo de Javascript en un navegador.

Estos serían una interpretación de los hilos de ejecución en el navegador, implementados dentro del motor de Javascript y a los que un desarrollador puede tener acceso para implementar operaciones que se deben completar en segundo plano durante la visualización de la página.

Podría pensarse como una ventaja a la hora del procesamiento, ya que dividir el código en diferentes tareas nos facilitaría la independencia de la interfaz de usuario de las tareas de procesamiento. El problema (36) es la dificultad de comunicarse con estos procesos desde el proceso principal.

Otro de los problemas es que los *Web Workers* no tienen acceso al sistema de archivos del navegador. Por ejemplo, si queremos adjuntar un fichero con datos para su uso por parte de los hilos de ejecución, tendríamos que programar desde el hilo principal el paso de los datos a los *Web Workers* mediante paso de mensajes, lo cual aumenta la complejidad de la aplicación.

Son útiles en trabajos en segundo plano, como por ejemplo la comprobación de la sintaxis en un editor *online*, gestión de una base de datos local creada dentro del navegador o para precargar recursos que van a ser utilizados después.

3. Análisis

Utilizando las herramientas que nos proporciona la ingeniería del software empezaremos a definir nuestra aplicación, de cara al desarrollo y el mantenimiento de esta.

3.1. Requisitos

En esta tabla enunciaremos los requerimientos que nuestra aplicación necesita cubrir basándonos en los objetivos que se describen anteriormente en este trabajo:

Requisito	Descripción
R1	Solicitar una carga de trabajo
R2	Envío de los resultados tras el procesamiento
R3	Informe de la ejecución
R4	Control de las peticiones de los usuarios
R5	Cálculo de estadísticas en base a los resultados

2 Requerimientos de la aplicación

A partir de ellos comenzaremos a analizar los elementos necesarios para cubrir las necesidades que nos plantean.

3.2. Roles

A continuación, se describirán los diferentes roles que estarán presentes en la aplicación:

- **Usuario anónimo**: este será el cliente que acceda a la aplicación para hacer su aportación en forma de procesamiento. Deberá tener una experiencia buena a su paso por la aplicación teniendo constancia en todo momento de lo que está haciendo.

- **Usuario administrador**: encargado de configurar y mantener la aplicación en funcionamiento, requerirá de herramientas que le faciliten la monitorización y el seguimiento del estado en el que se encuentra la ejecución, necesitando que se le transmita la mayor cantidad de información de manera resumida.

Estos serán los usuarios que utilicen nuestra aplicación y es necesaria su definición desde el comienzo del análisis ya que nos basamos en sus necesidades a las que nuestra aplicación ha de responder.

3.3. Descripción de los requisitos

En esta sección evaluaremos cuáles son los requisitos funcionales y no funcionales de nuestra aplicación basándonos en los requerimientos que antes hemos descrito.

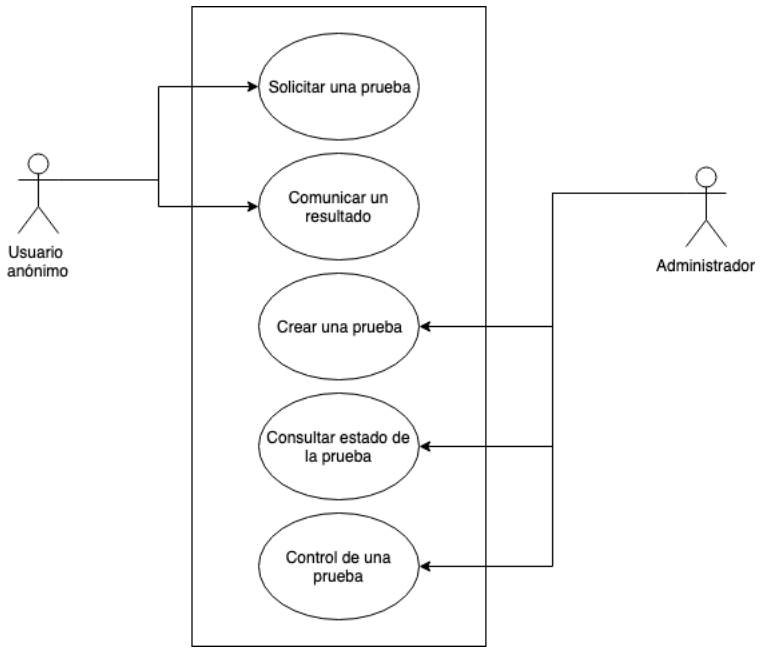


Ilustración 1 Diagrama de casos de uso

Como vemos las operaciones descritas representan las necesidades que estos usuarios plantean al hacer uso de la aplicación.

3.3.1. Requisitos funcionales

A continuación, se describen los requisitos funcionales de los que partirá nuestra aplicación. Estos deben detallar la funcionalidad que responde a las necesidades de los usuarios.

3.3.1.1 Solicitud de trabajo

Un usuario solicita una carga de trabajo a nuestra aplicación para su evaluación, siendo esta la funcionalidad básica a la que tiene que responder nuestro trabajo. Consistiría en responder una petición por parte del usuario anónimo, proporcionándole los recursos que tengamos programados para que él colabore.

3.3.1.2 Envío de los resultados

Nuestra aplicación ha de ser capaz de recoger los datos del trabajo realizado por todos los usuarios que colaboren en la ejecución. Dicha información resultante será el objetivo de la creación de esta aplicación y el resultado que se espera al finalizarla.

3.3.1.3 Informe de la prueba

Nuestra aplicación debe generar una serie de datos del usuario que colabore, sin que estos puedan llevar a su identificación. En caso de ser necesarios, se podrían emplear a la hora de generar algún tipo de estadísticas que resumieran los resultados de la ejecución.

3.3.1.4 Control de las pruebas creadas

Un administrador debe poder tener información del estado actual de la ejecución de la prueba creada en la aplicación, permitiéndole un seguimiento del progreso y facilitándole herramientas que le permitan gestionarla.

3.3.1.5 Almacenamiento de los resultados

Nuestra aplicación debe de ser capaz de guardar toda la información referente a la ejecución por parte de los usuarios, de forma que, en caso de fallo, todo el trabajo realizado sea recuperable. Deben establecerse mecanismos que aseguren que esta información se almacena de alguna forma dentro de la aplicación para que no se pierda.

3.3.1.6 Cálculo de estadísticas en base a los resultados

Debemos ser capaces de obtener una puntuación de las ejecuciones, para saber lo buena o mala que ha sido la aportación de un usuario, de forma que podamos comparar entre todas las aportaciones cuales son mejores o cuales son peores en base a este criterio objetivo.

3.3.2. Requisitos no funcionales

Este apartado recoge los requisitos no funcionales, que no afectan de manera directa a los usuarios, pero deben incorporarse ya que mejoran la experiencia con nuestra aplicación.

3.3.2.1 Compatibilidad de dispositivos

Nuestra aplicación tiene que ser compatible con la mayor cantidad de dispositivos de forma que sea accesible por la mayor cantidad de público. La manera de asegurar que llegamos al mayor número de usuarios es asegurando que dispondrán de la mejor experiencia posible, desde donde sea que accedan, utilizando las herramientas adecuadas para ello.

3.3.2.2 Aplicación segura

Nuestra aplicación debe ser segura y estar protegida contra errores que pudieran hacer que se interrumpa su ejecución. Esto asegurará la fiabilidad a los administradores y a nosotros como desarrolladores nos evitará problemas futuros de mantenimiento.

3.4. Narrativa de los principales casos de uso

En este apartado se detallan los principales casos de uso de nuestra aplicación. Describiendo de manera detallada en que consiste la acción podremos establecer un escenario al que después daremos respuesta en el diseño.

RF- 1	**Solicitud de un trabajo**	
Descripción	Un usuario accede a la aplicación para colaborar en la ejecución.	
Precondición	Ninguna	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario accede a la aplicación
	2	Acepta un acuerdo de confidencialidad
	3	Recibe el trabajo a realizar
	4	Genera unos datos como resultado
Postcondición	Ninguna	
Excepciones	**Paso**	**Acción**
	1	Si no acepta el acuerdo, no recibe trabajo
Frecuencia esperada	Muchas veces por minuto	
Importancia	Vital	
Urgencia	Inmediato	

RF- 2	**Comunicar un resultado**	
Descripción	El usuario tras finalizar su trabajo enviará de vuelta el resultado.	
Precondición	El usuario debe haber completado una ejecución	
Secuencia Normal	**Paso**	**Acción**
	1	El usuario empaqueta los resultados
	2	Solicita a la aplicación el envío de los datos
	3	Envía los datos
	4	Recibe la confirmación de la recepción
Postcondición	Ninguna	
Excepciones	**Paso**	**Acción**
	1.1	No se ha realizado la ejecución
	4.1	Error en la recepción
Frecuencia esperada	Muchas veces por minuto	
Importancia	Vital	
Urgencia	Inmediato	

RF- 3	**Crear una prueba**	
Descripción	Un administrador accederá a la aplicación y configurará el estado inicial y los posibles elementos que sean necesarios para el funcionamiento.	
Precondición	El administrador debe estar autenticado en la aplicación	
Secuencia Normal	**Paso**	**Acción**
	1	El administrador accede al panel de control
	2	Accede al apartado de configuración
	3	Modifica las diferentes opciones disponibles
	4	Confirma los cambios
Postcondición	Ninguna	
Frecuencia esperada	Pocas veces al mes	
Importancia	Alta	
Urgencia	Media	

RF- 4	**Consultar el estado de una prueba**	
Descripción	Un administrador accederá a la aplicación y comprobará el estado de la prueba.	
Precondición	El administrador debe estar autenticado en la aplicación	
Secuencia Normal	**Paso**	**Acción**
	1	El administrador accede al panel de control
	2	En la vista principal observa los estadísticos de las pruebas
Postcondición	Ninguna	
Frecuencia esperada	Pocas veces al mes	
Importancia	Alta	
Urgencia	Media	

RF- 5	**Control de una prueba**	
Descripción	Un administrador accederá a la aplicación para gestionar el estado de una prueba.	
Precondición	El administrador debe estar autenticado en la plataforma	
Secuencia Normal	**Paso**	**Acción**
	1	El administrador accede al panel de control
	2	Accede al apartado de configuración
	3	Cambia el estado de la prueba
	4	Confirma los cambios
Postcondición	Ninguna	
Frecuencia esperada	Pocas veces al mes	
Importancia	Alta	
Urgencia	Media	

4. Planificación

En este apartado se analizará el tiempo, costes y posibles recursos necesarios para el desarrollo de la aplicación. Debemos tener esto en cuenta ya que a la hora de crear un presupuesto para el cliente que nos pudiera encargar el desarrollo, debemos de compensar los costes del proyecto para así calcular el beneficio que obtendremos.

4.1. Metodología de temporización

Durante la realización de este proyecto se seguirá una metodología clásica, que hace uso del paradigma tradicional. Cada una de las fases del proyecto será dependiente de la anterior. El porqué de esta metodología es que se sigue un proceso secuencial que da mayor control sobre las fases.

Ilustración 2 Metodología tradicional

4.2. Estimación de tiempos

La estimación de tiempos nos servirá para calcular el tiempo teórico para la realización de todas las tareas que recogen la realización de este trabajo.

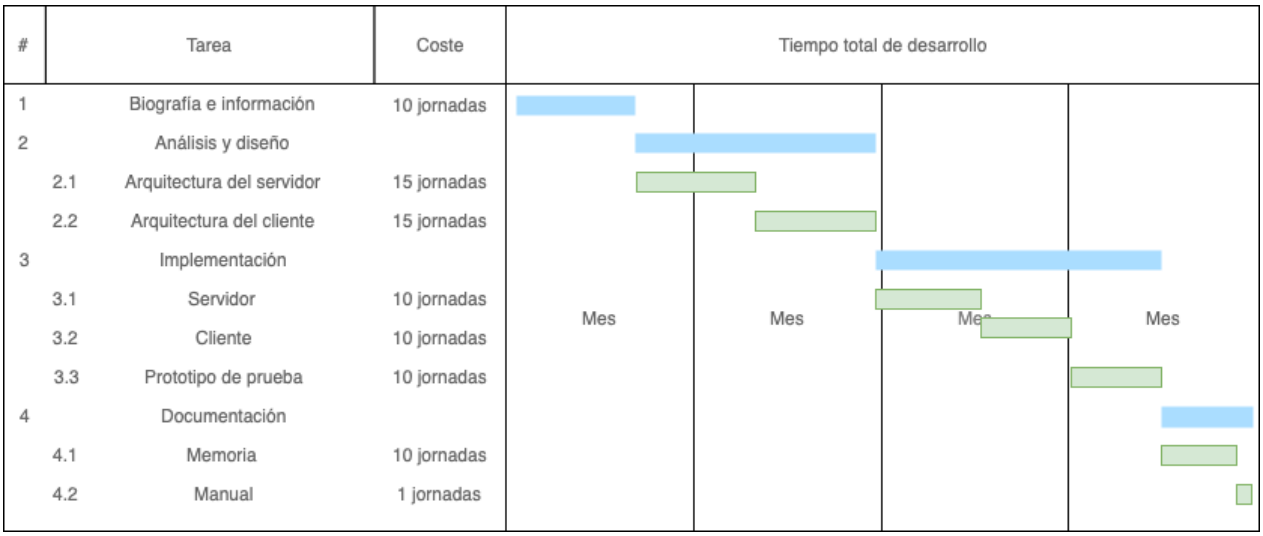


Ilustración 3 Diagrama de Gantt

Como vemos el tiempo total es de 74 jornadas, lo que equivaldrían aproximadamente a cuatro meses de trabajo. Las estimaciones de tiempo se han calculado en base a que es necesario un estudio y documentación previa de las tecnologías. La planificación está planteada con tiempo suficiente, aunque al tratarse de una estimación siempre puede verse sujeta a modificación.

4.3. Estimación de costes

Analizaremos los diferentes elementos que forman parte del desarrollo y su consecuente gasto.

4.3.1. Costes de equipos

Para el desarrollo de este trabajo se han requerido los siguientes equipos:

- Ordenador portátil MacBook Pro - 2018 (37), coste estimado ~1300€.

- Servidor VPS 2GB RAM, 1CORE 3,62€ / mes (38).

Suponiendo que los equipos de desarrollo tienen un tiempo de amortización máximo de 6 años (39), 1300€ / 6 años * 12 meses / año = 18,06€ mes.

El servidor puede contratarse de forma mensual con lo que no se evalúa su coste de amortización, sino que se trata como un servicio.

4.3.2. Costes de personal

Para la realización de este trabajo, se precisa de personal cualificado que basándose en este documento sea capaz de completar la tarea que se le encomiende. El coste mensual de un ingeniero informático es de 17.544,24€ anuales (40), 1.253,16€ mensuales.

4.3.3. Costes en infraestructuras y servicios

Durante la realización de este trabajo serán necesarios una serie de recursos asociados a cubrir las necesidades del personal que participa en el desarrollo.

- La conexión a internet tendrá un coste de 32,30€ / mes (41).
- La conexión y consumo eléctrico tendrá un coste de 91,67€ / mes (42).
- El consumo de agua y saneamiento tendrá un coste de 18,55€ mes (43).
- La oficina tendría un coste de 150€ / mes (44).

4.3.4. Coste total del desarrollo del proyecto

La siguiente tabla recoge todos los costes antes descritos y nos proporciona un total resultante de la realización de este trabajo.

Concepto	Coste parcial	Coste mensual
Costes de equipos	3,62€	21,68€
	18,06€	
Costes de personal	1.253,16€	1.253,16€
Costes en infraestructuras y servicios	32,30€	292,52€
	91,67€	
	18,55€	
	150€	
	Total mensual	1567,36€

Suponiendo 4 meses de trabajo el coste del proyecto sería: **6269,44€.**

Como anotación, he de añadir que toda la información acerca de dónde se han obtenido los precios de los servicios y bienes de consumo que se utilizan se encuentra en el enlace que aparece al final de cada uno de los elementos.

5. Diseño

Partiendo de las tecnologías antes descritas, pasaré a describir un protocolo de comunicación entre cliente y servidor. Se basará en peticiones HTTP, que tendrán como objetivo solventar las necesidades de los usuarios.

5.1. Diagrama de paquetes

El diagrama mostrado a continuación muestra las diferentes partes en las que se debe dividir la aplicación.

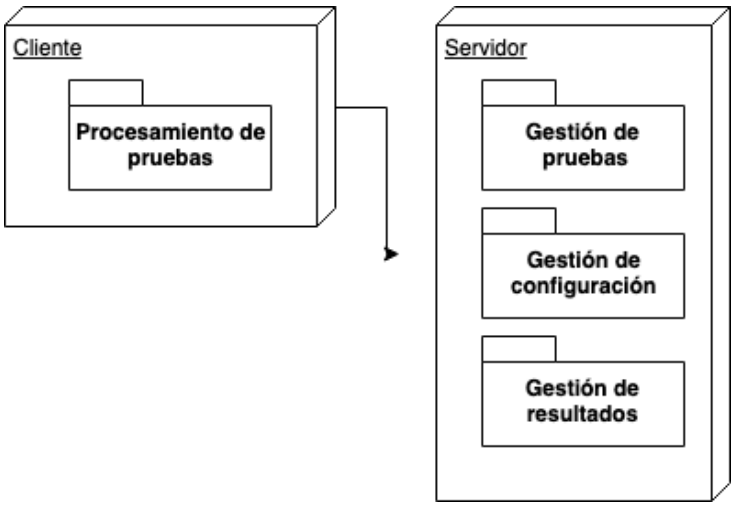


Ilustración 4 Diagrama de paquetes

Como observamos disponemos de una parte del cliente y otra del servidor. De parte del cliente englobamos la realización del trabajo que el servidor envía para su ejecución, incluiría funciones para la ejecución y comunicación de los resultados obtenidos. En el lado del servidor, tenemos todo lo relacionado con la gestión de las respuestas a los clientes, así como el soporte a la gestión del administrador. La gestión de pruebas manejaría el envío de los trabajos a ejecutar, la gestión de resultados la recepción el trabajo que realicen los usuarios, así como de almacenarlos en el sistema. La gestión de configuración recoge tanto el control como la parte de las estadísticas.

5.2. Diagramas de secuencia

Los diagramas de secuencia nos van a ser útiles para describir la interacción entre los distintos elementos que toman parte en el funcionamiento de la aplicación. Con ellos representaremos la interacción que realizan los usuarios.

5.2.1. Solicitud de una prueba

Los clientes accederán al portal y deberán aceptar una serie de clausulas que nos permitan a nosotros saber que el usuario, en este caso un cliente que hace uso de la aplicación es consciente de que vamos a utilizar su capacidad de cómputo para resolver el problema que le enviemos, y que no vamos a hacer uso de sus datos privados. Hemos separado lo que sería la parte de la aplicación y la base de datos ya que son elementos diferentes que pueden estar alojados en diferentes servidores, con lo que también representamos la interacción que tendrían dichos elementos.

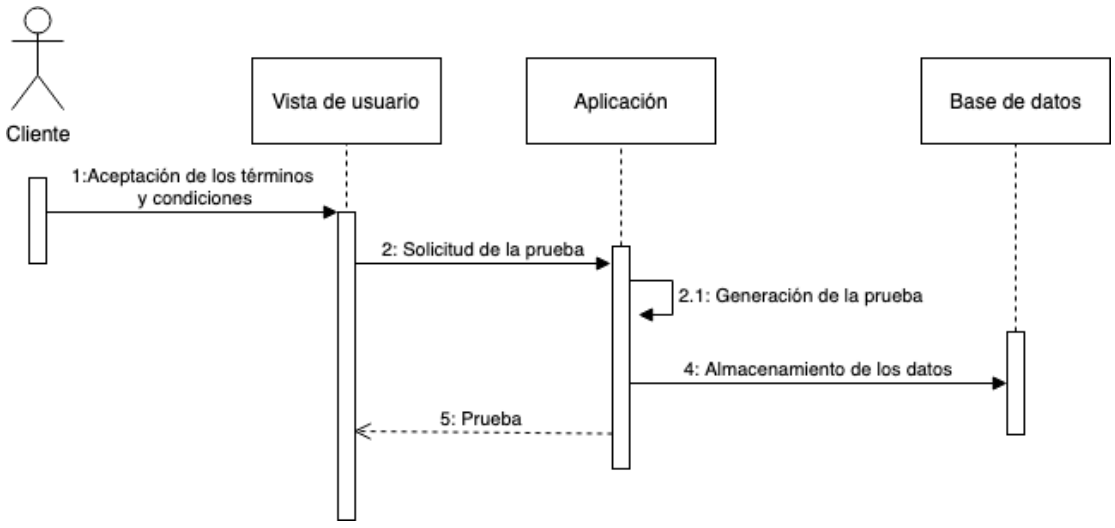


Ilustración 5 Diagrama solicitud de prueba

5.2.2. Subida de los resultados

La aplicación, una vez ha completado la ejecución del lado del cliente, recibe los resultados, tanto datos como las estadísticas. Del lado del servidor, en este caso la aplicación *web*, almacenamos los ficheros que pueda remitir el cliente, y del lado de la base de datos la información extraída del sistema del cliente, así como de elementos que evalúan lo buena que ha sido su ejecución.

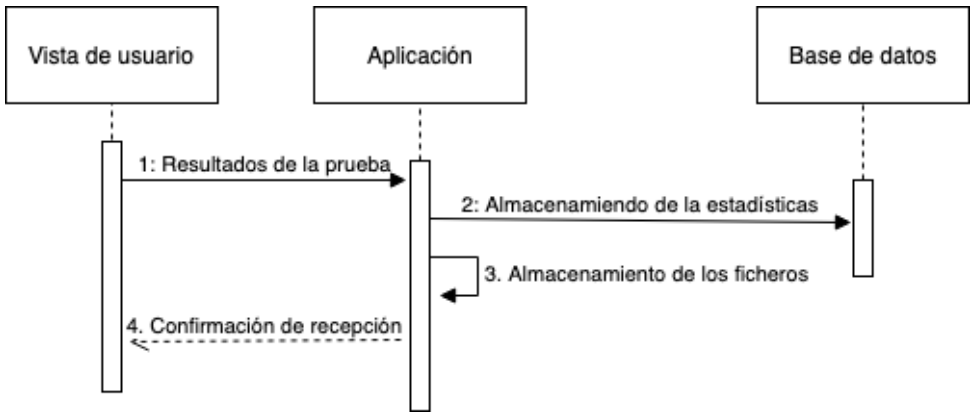


Ilustración 6 Diagrama subida de resultados

Observamos que en todo momento se intenta que el cliente sea consciente de qué se está haciendo y se le confirma que las acciones que realiza han tenido el efecto deseado, y que efectivamente su trabajo ha sido almacenado.

5.2.3. Acceso del administrador

El papel del administrador de nuestra aplicación será el de mantener el control del estado de esta, así como de ser quien monitorizará el progreso e identificará posibles errores en el proceso.

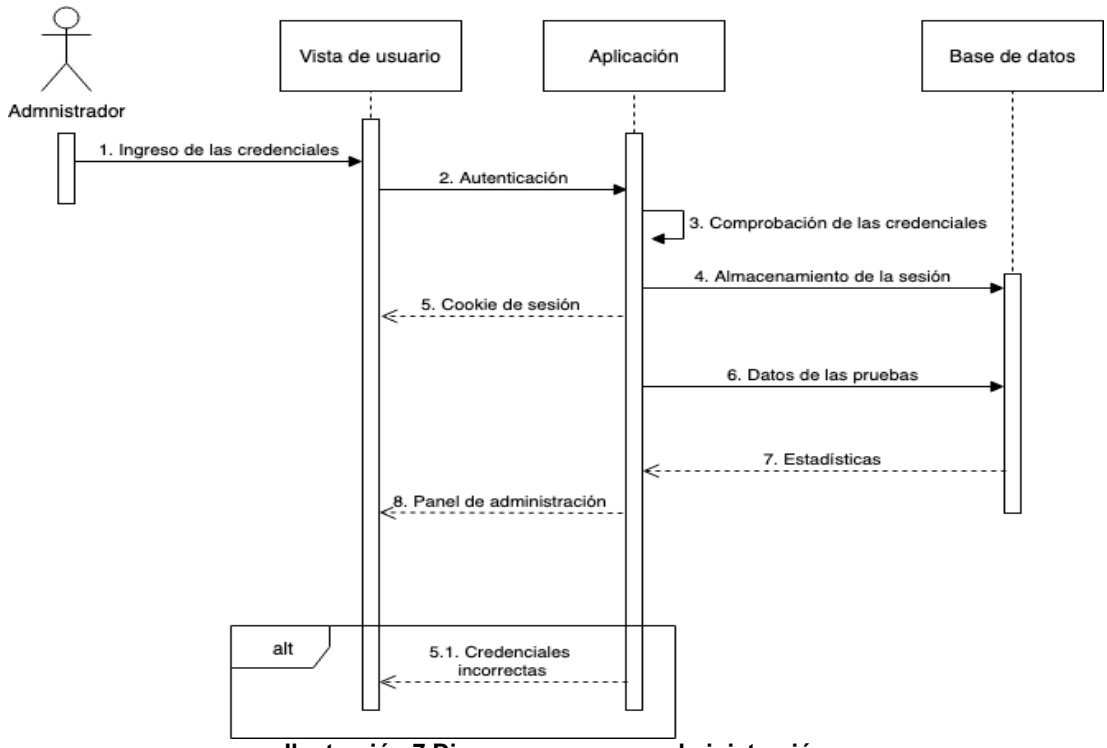


Ilustración 7 Diagrama acceso a administración

En este diagrama de secuencia se muestran los pasos que se seguirán cuándo este acceda al panel de control para consultar el estado del proyecto. El acceso a las demás funcionalidades como pueden ser la configuración del proyecto se realizarán siguiendo el mismo diagrama.

5.3. Diagrama de comunicación

Tras estudiar cuáles serían las interacciones básicas de los usuarios con nuestra aplicación, pasaremos a describir las diferentes clases y operaciones que va a implementar nuestro servidor. Ya que al tratarse de un sistema *web*, no seguiremos el sistema tradicional de diseñarlo mediante objetos y clases, ya que la mayoría de la aplicación se va a basar en la respuesta a peticiones HTTP de los usuarios.

A continuación, se incluye un diagrama con la interacción de las diferentes partes de la aplicación, a nivel funcional, que más adelante contendrán los métodos necesarios.

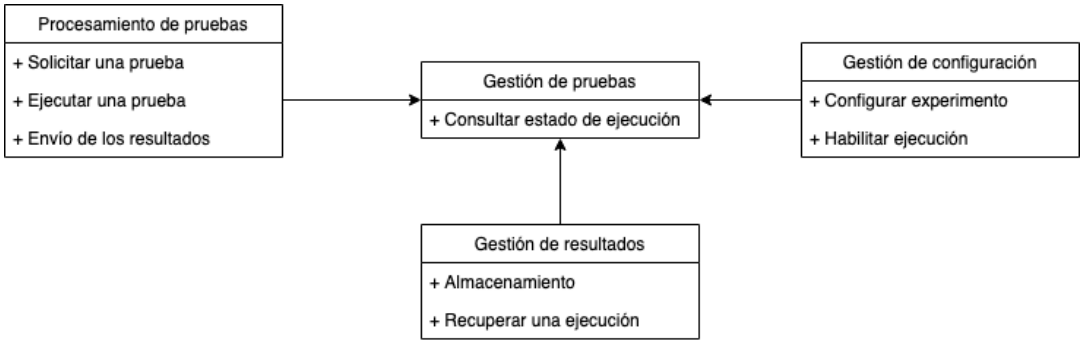


Ilustración 8 Diagrama de comunicación

Los métodos que se definan del lado del cliente serán aquellos que correspondan a la ejecución de la prueba, más adelante en la descripción de la aplicación de prueba, plantearemos los métodos que se ejecutarán de esta parte.

5.4. Patrón de diseño

Como proceso de la ingeniería de software, cuando creamos una aplicación web debemos seguir un patrón de diseño que defina el proceso que los datos seguirán para mostrarse al usuario.

Para ello usaremos un patrón Modelo - Vista - Controlador (45), que separaría los datos, la lógica que los maneja y la agrupación para su interacción con el usuario. Las ventajas que nos aporta son que, al establecer la independencia entre las diferentes partes, se separan los conceptos y se simplifica el proceso de desarrollo, siendo necesario solo saber cómo interaccionan entre ellas.

La parte del modelo vendrá representada por nuestro sistema de gestión de base de datos, que será el encargado de las operaciones de creación, búsqueda y modificación sobre estos. El controlador lo implementaremos en nuestra aplicación, que actuará de mediador respondiendo a las peticiones que los clientes envíen y

solicitando los datos al modelo. En cuando a las vistas, usaremos un gestor de plantillas, que definirá los esqueletos que nuestro controlador rellenará con los datos procedentes del modelo.

5.5. Diseño de la base de datos

Este diagrama muestra los esquemas que nuestra base de datos ha de almacenar para dar soporte a nuestra aplicación.

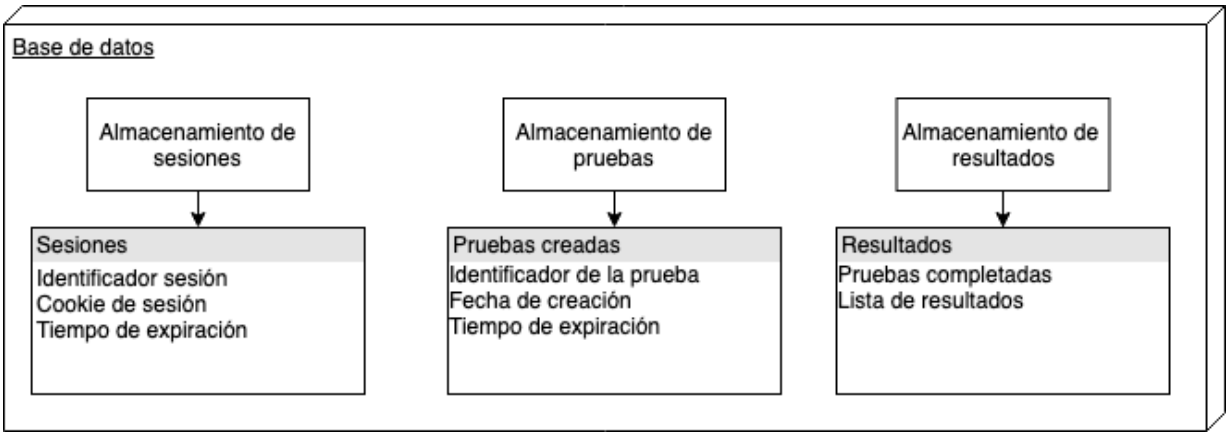


Ilustración 9 Base de datos

Partiendo de este esquema tendremos que ajustar, en base a la tecnología que escojamos para implementar la base de datos, las posibles tablas u otra alternativa defina el esquema antes descrito.

5.6. Diseño de las vistas

Esta sección contiene los prototipos de las vistas que compondrán la experiencia de los usuarios de la aplicación.

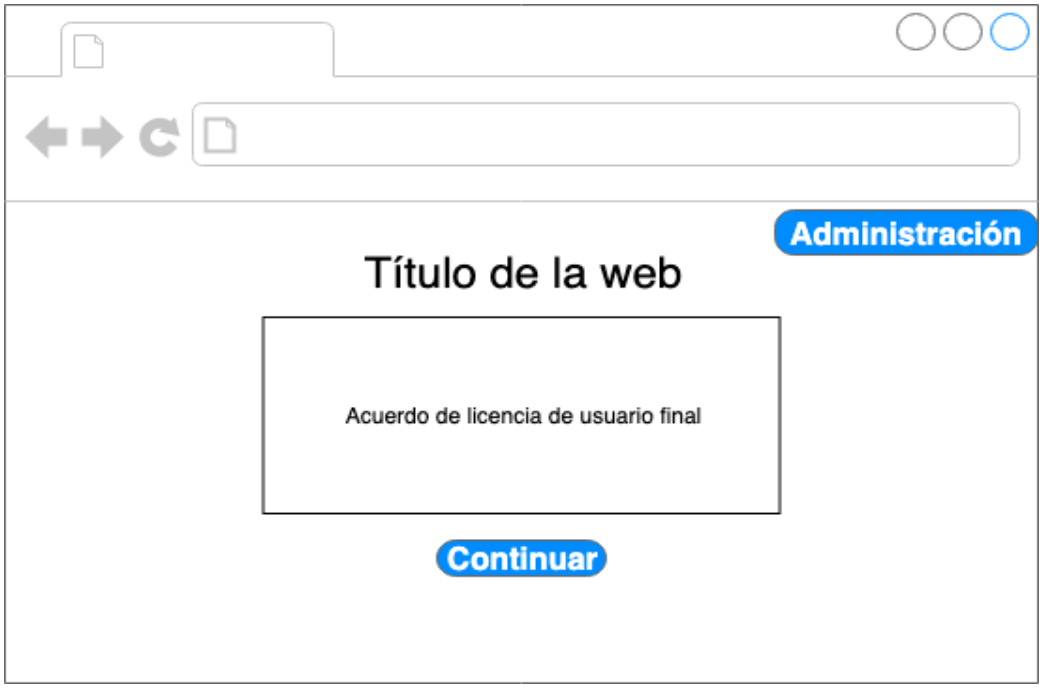


Ilustración 10 Diseño vista principal del usuario

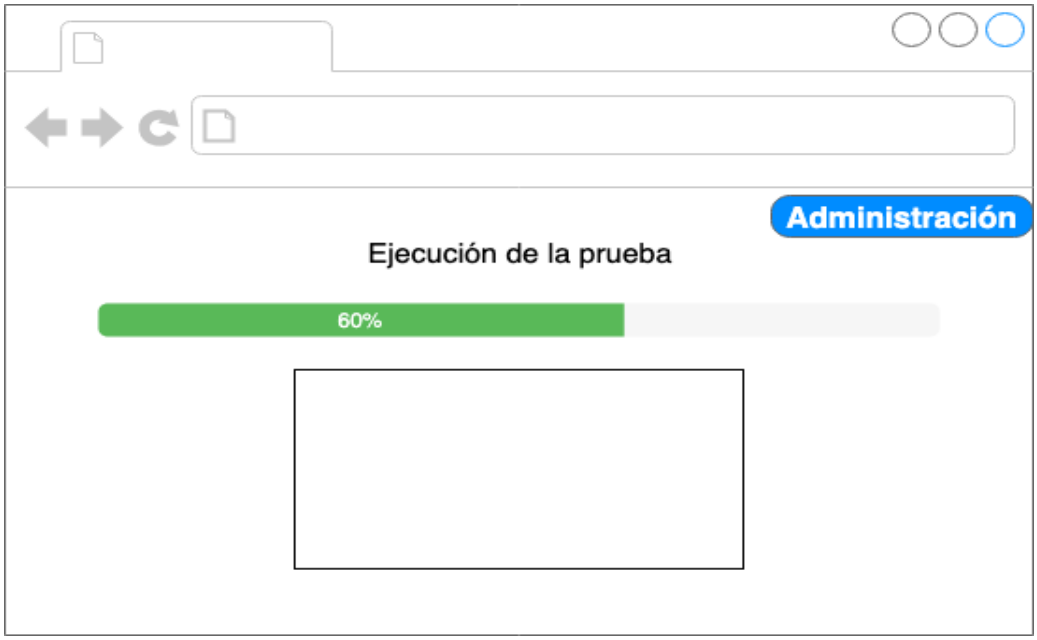


Ilustración 11 Diseño vista de ejecución

Estas dos primeras vistas servirían para cubrir las necesidades del usuario anónimo de nuestra aplicación, también llamado cliente, proporcionándole la respuesta ante su necesidad de participar en la ejecución del proyecto. Hay que destacar la inclusión de elementos que hacen mas amena su participación informándole del progreso o mostrándole mensajes con información de interés.

Ilustración 12 Diseño vista administración

Esta vista contendría la vista del usuario administrador, que contendría todas las herramientas, así como la información acerca de la progresión del proyecto.

5.7. Protocolo de interacción

Este apartado muestra el protocolo que hará que interactúen las diferentes partes de nuestra aplicación.

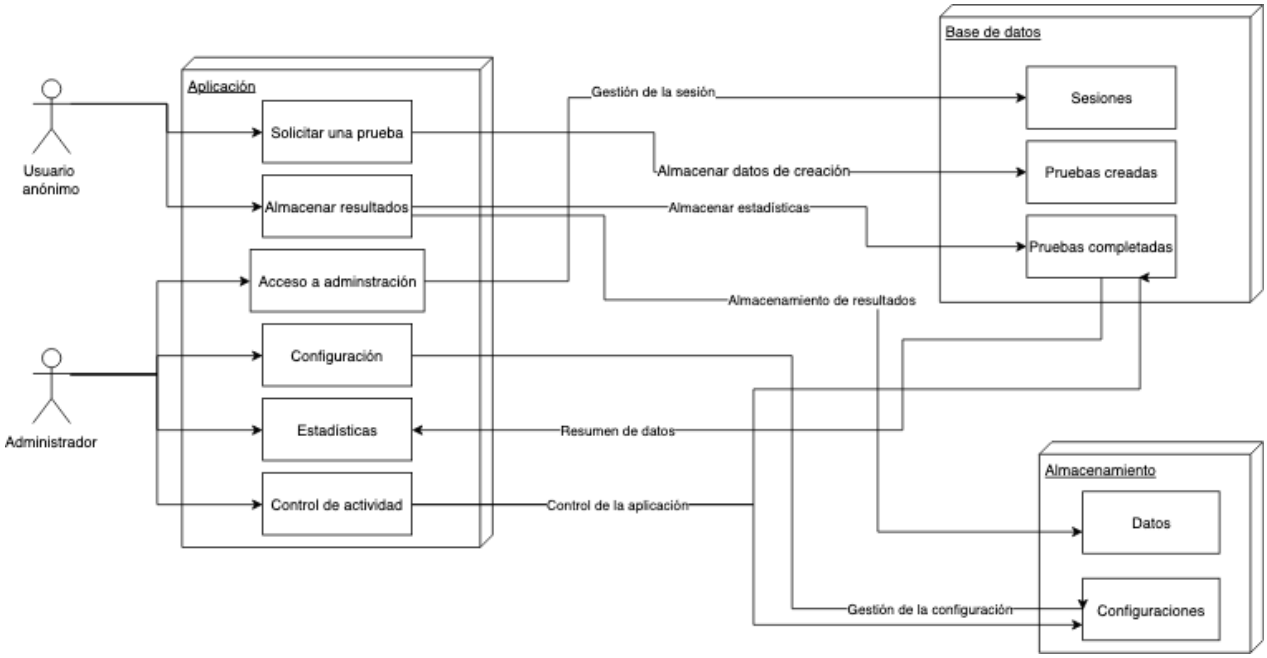


Ilustración 13 Protocolo de interacción

Este diagrama recoge el diseño que partiendo del diagrama de casos de uso y extendido mediante el proceso de análisis y diseño, ha dado lugar a lo que sería la composición de la aplicación. Cabe destacar la división que existe en los datos, existiendo la base de datos y un almacenamiento en el sistema de ficheros del propio servidor, que aparece separado. Esto es debido a que uno de los requisitos era que el trabajo fuese recuperable en caso de un fallo del sistema, con lo que si la aplicación o la base de datos no estuvieran funcionales, en última instancia un administrador podría acceder al sistema que aloja la aplicación y recuperar los ficheros resultantes de la ejecución de los clientes y no dar el proyecto por perdido.

6. Desarrollo

6.1. Selección del lenguaje de programación, tecnología del servidor y del motor de la base de datos.

En este apartado estudiaremos las diferentes tecnologías disponibles y nos decantaremos por una para la utilización dentro del proyecto. Comencemos por la evaluación de los lenguajes de programación, ya que en base a ellos vamos a definir el resto de los elementos.

6.1.1. Lenguajes de programación

6.1.1.1 Comparativa de los diferentes lenguajes

Realizaremos una comparativa entre las diferentes opciones disponibles en el mercado y, en base a las conclusiones y mis propias preferencias, seleccionaremos uno para su uso en el desarrollo del proyecto.

Los criterios que tendremos en cuenta serán los siguientes:

- Extensibilidad: debe de disponer de librerías que faciliten la ampliación de lo que sería nuestra estructura base. Nos ahorraría tiempo de desarrollo además de facilitarnos seguir los estándares del lenguaje. Otorgaremos la puntuación en función de lo numerosas que sean y del soporte que tengan por parte de la comunidad, también siendo una parte importante los criterios de seguridad que se apliquen a la hora de distribuirlas al público.
- Soporte: la disponibilidad de documentación tanto del lenguaje como de sus herramientas es básica, de cara a la resolución del problema. La puntuación vendrá dada por lo buena que sea dicha documentación, así como la existencia de foros de consulta para los usuarios, que serían la alternativa en caso de que la documentación no fuese completa.
- Rendimiento: el lenguaje debe tener un buen rendimiento que aproveche los recursos disponibles. La puntuación vendrá otorgada en función del esfuerzo

que supondría implementar un sistema similar al que queremos, evaluando la complejidad de la programación para obtener un buen rendimiento del lado del servidor.

- Complejidad: un lenguaje debe ser sencillo, tanto para aprenderlo como para comprender código de terceros y saber si no conviene su incorporación al proyecto. La puntuación vendrá en base a evaluar la curva de aprendizaje del desarrollador y su posterior productividad al momento de tener que usarlo.

Lenguaje	Extensibilidad	Soporte	Rendimiento	Complejidad
PHP	5/5	5/5	5/5	4/5
Python	3/5	5/5	5/5	4/5
Ruby On Rails	4/5	4/5	4/5	5/5
ASP. NET	3/5	4/5	3/5	5/5
Node.js	4/5	4/5	5/5	5/5

3 Lenguajes de programación web

A continuación, se hacen unos comentarios puntuales de cada uno de los lenguajes que han llevado a otorgarles esa puntuación:

- PHP: es un lenguaje muy **fiable**, que se lleva utilizando desde el origen de la *web* para la creación de aplicaciones. No es lo más sencillo sobre todo con los últimos cambios en las versiones más recientes que tratan de asemejarlo a lenguajes más populares. Es fiable, pero tarda mucho en recibir actualizaciones. Es el más respetuoso con el desarrollo tradicional siguiendo unos patrones diseño muy estructurados.
- Python: eficiente pero **complejo**. Su uso está extendido en el ámbito científico (46), dada la cantidad de librerías que existen para ello, pero para el uso dentro de la web es quizá demasiado complejo. Su rendimiento es muy bueno, pero para conseguir la misma aplicación que en otros lenguajes aquí presentados haría falta utilizar más horas de trabajo.
- Ruby On Rails: es el *Framework* de **Ruby** para el desarrollo de aplicaciones *web*, existen alternativas, pero este es el estándar. La estructura de la aplicación no es sencilla, es decir, crear una estructura de objetos es

compleja, aunque una vez hecho es sencilla su utilización, también es altamente extensible y eficiente.

- ASP.NET: Es un *Framework* de desarrollo en **.Net** creado por **Microsoft**. Se dice que es un buen lenguaje a la hora de crear **aplicaciones empresariales** ya que es fácil crear interacciones con el usuario. Como contra su última versión estable data del año 2012, con lo que cualquiera del resto de lenguajes de esta lista está más al día en cuanto a seguridad y compatibilidad con las tecnologías de la *web*.
- Node.js: similar a Ruby On Rails, solo que este tiene más variedad en cuanto a *Frameworks* de desarrollo para aplicaciones *web*. No es sencillo ya que **Javascript** es complejo y su aprendizaje puede ser costoso, muy **eficiente** en cuanto a su ejecución como aplicación a nivel de usuario, su núcleo está implementado en C++. Altamente **extensible** y personalizable a las necesidades del programador.

6.1.1.2 Node.js

Node.js (47) es un lenguaje de programación que se basa en JavaScript, siendo la sintaxis la misma solo que se usa a nivel de aplicación de usuario y no desde un navegador.

A pesar de que la sintaxis de JavaScript no es la más sencilla y entendible, la potencia de este y la cantidad de documentación, librerías y una comunidad activa lo hacen viable para utilizarlo dentro de este proyecto.

En comparación con PHP, JavaScript permite un funcionamiento asíncrono, es decir, cada petición de cada cliente se hace de forma independientemente del estado del servidor, y se pueden utilizar varias hebras de procesamiento para responder múltiples peticiones.

En resumen, este trabajo va a tener desarrollada su mayor parte en JavaScript, si además la llevamos a la parte del servidor, va a ser mucho mas sencillo tanto a nivel de manejo de objetos, funciones, etc. Además de que me aseguro una compatibilidad

100%, pudiendo usar las librerías tanto en cliente como en el servidor, y de que a la hora de aprender el lenguaje, podré aplicarlo a todo el proyecto.

6.1.2. Tecnología del servidor

En este apartado realizaré una breve información de cómo alojaré el código de la aplicación.

Si bien podríamos pensar que para alojar una aplicación necesitaríamos de una herramienta externa que nos diera el soporte necesario para alojar y mantener activa nuestra aplicación, como se lleva haciendo durante años con servicios como Apache (48) o NGINX (49). Sin embargo, los lenguajes más actuales como es el caso de Node.js o Ruby On Rails (50), incorporan internamente de una librería encargada de montar un pequeño servidor y de proporcionar los servicios básicos para distribuir la aplicación que estos implementan, evitando la necesidad concreta de depender de otra herramienta a parte.

Node.js contiene la librería HTTP (51), de manera que cuando programemos nuestra aplicación, uno de los apartados necesarios será el de configurar mediante esta librería nuestro servidor *web*.

6.1.3. Bases de datos

Aquí compararé algunos motores de bases de datos que están disponibles para incorporar al proyecto. Si bien existen diferentes tecnologías, se podrían definir dos grandes patrones de diseño, SQL y NOSQL.

6.1.3.1 SQL

Las bases de datos que siguen el patrón SQL (52) ofrecen una forma de almacenar los datos de manera estructurada, existiendo entre ellos una fuerte relación de dependencia.

Las características principales de una base de datos SQL son las siguientes:

1. **Lenguaje de definición de datos**: incorpora un lenguaje de definición de datos (LDD) que facilita la creación de las tablas y la definición las relaciones entre ellas.
2. **Lenguaje interactivo de manipulación de datos**: existe un lenguaje (LMD) que permite la consulta, modificación y eliminación de los datos alojados en el sistema.
3. **Integridad**: se integran mecanismos que aseguran la integridad de la base de datos cuando se altera el estado de esta, y que evitan la aparición de inconsistencias.
4. **Definición de vistas**: se pueden crear vistas, que son resúmenes de datos con las características en comunes que el creador defina en su creación.
5. **Control de transacciones**: se permite la definición de operaciones previas y posteriores a la ejecución de consultas sobre los datos.
6. **SQL incorporado y dinámico**: la mayoría de los lenguajes conocidos poseen integrados mecanismos para interaccionar con cualquier motor de base de datos del tipo SQL.
7. **Autorización**: un sistema de permisos permite definir quién tiene poder sobre las operaciones que acceden a los conjuntos de datos y la autoría de los mismos.

Las bases de datos SQL también cumplen con los principios ACID (53) (*Atomicity, Consistency, Isolation and Durability*: Atomicidad, Consistencia, Aislamiento y Durabilidad):

- **Atomicidad**: las operaciones que se realizan sobre la base de datos siguen una serie de pasos, de modo que o se realizan todos o la transacción falla y no se ve alterada la base de datos.

- **Consistencia**: se mantiene la integridad de la base de datos, de modo que los datos para una transacción están en su versión más reciente.
- **Aislamiento**: las operaciones son independientes, de manera que diferentes operaciones sobre los mismos datos no produzcan errores, es decir, se realizan de forma sincronizada y en estricto orden de llegada para asegurar la consistencia.
- **Durabilidad**: los cambios son registrados en la base de datos de forma física, ante una caída del sistema los datos permanecen en la última versión confirmada.

Las más conocidas son MYSQL (54), Oracle (55) o Access (56).

6.1.3.2 NOSQL

En este apartado englobaríamos todo aquello que no sigue un modelo estructurado, como por ejemplo MongoDB (57) o Redis (58).

La peculiaridad de estas bases de datos es que guardan los datos en estructuras diferentes a las tablas, y de que no garantizan los principios ACID a costa de tener un mejor rendimiento. La elección de un motor de base de datos no relacional se traduce en ofrecer alta disponibilidad, extensibilidad y replicación.

La alta disponibilidad es gracias a su alto rendimiento, la extensibilidad es gracias a que se puede modificar fácilmente la estructura de la base de datos y de los elementos que la componen, la replicación se consigue creando réplicas de base de datos original que actúa en modo espejo.

MongoDB es útil a la hora de almacenar datos con una estructura similar a la que tendría un documento, mientras que Redis sigue un modelo de almacenamiento de objetos, utilizando para ello estructuras de datos que se incluyen en los lenguajes de programación.

Para este proyecto utilizaré Redis, un motor no relacional que ofrece un rendimiento enorme. Suponiendo que nuestra aplicación tiene que gestionar miles de peticiones

por segundo, Redis ofrecería un rendimiento de hasta 10000 clientes de forma simultanea.

Normalmente Redis se utiliza en servidores de cacheo o almacenamiento de sesiones, de manera que el tiempo para servir una vista ya procesada de una web sea mínimo o el tiempo en comprobar la sesión activa de un usuario sea inmediata. Aún así lo que nos interesa a nosotros es su alto rendimiento a la hora de resolver las operaciones.

6.1.4. Alojamiento e integración continua

Nuestra aplicación necesita de un servicio que contenga todo el software que vamos a utilizar y dé soporte a los servicios que hemos definido. Del mismo modo necesitamos de una herramienta de control de versiones que nos permita modificar nuestra aplicación, manteniendo un histórico de cambios de manera que podamos revertirlos en caso de detectar un problema en una nueva versión del software.

6.1.4.1 Docker

Docker (59) es un servicio de virtualización de aplicaciones mediante contenedores, que al contrario de soluciones más tradicionales que requerían de la instalación de una maquina completa, este utiliza servicios más sencillos alojados en la máquina base, mejorando la modularidad e independencia. Otra de las ventajas es que nos permite ejecutar múltiples instancias de los diferentes servicios y crear un balanceo de carga.

Dividiremos nuestra aplicación en dos servicios. Uno de Node.js y otro de Redis para la base de datos.

6.1.4.2 Git

Todo proyecto de desarrollo actual dispone de un sistema de control de versiones que asegure un seguimiento de los cambios en el código durante el desarrollo de la aplicación, permitiendo definir diferentes versiones.

Mi preferencia es Git (60), un sistema de control de versiones de **código libre**, alojado en una plataforma como es GitHub (61), la cual me permitirá desarrollar correctamente el proyecto.

El proyecto estará disponible al público tras su entrega en el siguiente enlace:

- https://github.com/elmendacorp/tfg2

6.2. Librerías

En esta sección explicaré y justificaré el uso de las librerías de terceros utilizadas en el desarrollo de este proyecto. Su utilización nos ahorrará trabajo y nos facilitará el tener funcionalidades eficientes.

6.2.1. Express.js

Esta es la librería base sobre la que creamos la estructura del servidor, tiene un funcionamiento similar al de una *API web* y su funcionamiento se basa en responder peticiones HTTP que llegan al puerto que tenemos configurado. Express.js servirá para establecer las respuestas a las peticiones de la aplicación, aunque será necesario añadirle ciertos paquetes para que se complete su funcionalidad.

Link: https://expressjs.com/es/

6.2.2. http-errors

Librería que extiende la funcionalidad de Express.js, facilita el manejo de los errores que se generan, de modo que cuando se requiere crear una página de error específica, esta librería nos facilita la generación de la información de la traza asociada. Cuando un usuario navega por un sitio web se generan errores, a los que el servidor en consecuencia informa al usuario o al administrador del sitio de ellos, esta librería ayuda creando un reporte sencillo similar al que aparece en los sitios *web* al recibir un error 404.

Link: https://github.com/jshttp/http-errors#readme

6.2.3. Compression

Esta librería facilita la compresión del contenido de respuesta que el servidor manda al cliente. Es una mejora de rendimiento que reduce la cantidad de datos que el servidor tiene que enviar. Mejora el rendimiento a costa de algo de capacidad de procesamiento, pero se reducen los tiempos de carga del contenido. Al ser un proyecto pequeño la ganancia es poca, pero es una buena práctica contar con una optimización como esta.

Link: https://github.com/expressjs/compression#readme

6.2.4. cookie-parser

Esta librería nos va a facilitar la tarea de manejar las cookies del cliente, sin tener que parsear nosotros a mano el contenido de la petición del cliente. Esta se encarga de procesarlas y almacenarlas en una variable de entorno para su posterior uso en el lado de la aplicación. Para realizar lo que hace la librería deberíamos acceder a la información que el cliente nos ha enviado en su petición HTTP de forma manual manipulando el mensaje, mientras que al usarla, automáticamente al llegar a nuestro controlador esa información esta disponible.

Link: https://github.com/expressjs/cookie-parser#readme

6.2.5. Redis

Ya que utilizamos el motor de Redis como base de datos necesitamos establecer una conexión con el servicio. Esta librería nos facilita la creación de una interfaz.

Podríamos crear un controlador con las funciones necesarias para hacer las consultas pertinentes a la base de datos, pero usando esta librería se obtiene una herramienta limpia y abstracta para consultar la base de datos, que además utiliza los métodos del cliente estándar del propio motor.

Link: https://github.com/NodeRedis/node_redis

6.2.6. Helmet

Helmet es una librería de seguridad de Express.js que configura por defecto las cabeceras HTTP de nuestra aplicación evitando los errores más comunes.

Express.js tiene ciertos agujeros de seguridad que no son difíciles de controlar, así como los presentes en el protocolo HTTP. Esta librería agrupa todas las soluciones y las aplica. Además, al contar con una herramienta especializada no tenemos que preocuparnos de que nuestra aplicación deje de estar actualizada en materia de seguridad. Esto nos asegura que futuras vulnerabilidades sean corregidas en esta librería, resolviéndolas con una simple actualización de la dependencia.

Link: https://helmetjs.github.io/

6.2.7. Morgan

Se trata de un gestor del *log* para Node.js que se encarga de recoger información de las conexiones y respuestas que se gestionan en nuestra aplicación. Para que el *logger* de la aplicación contenga información relevante y no tengamos que estar lanzando mensajes a la consola, este paquete se encarga de recoger, a distintos niveles configurables, la información relevante y nos la añade automáticamente, de

manera que, a la hora de ir comprobarlos, tenemos la información formateada correctamente.

Link: https://github.com/expressjs/morgan

6.2.8. Express-session

Se trata de una librería que se encarga de manejar las sesiones y gestionarlas de una manera transparente en el motor de base de datos, nos ahorra el tener que gestionar nosotros la información a mano. Funciona en conjunto con cookie-parser y se encargan de la gestión de las sesiones de usuario.

Link: https://github.com/expressjs/session

6.2.9. EJS

Se trata de un gestor de plantillas. Su utilidad es la de añadir un sistema de marcado basado en Javascript dentro de los documentos HTML de forma similar a lo que haríamos con PHP a la hora de definir una vista. De esta forma cuando queremos responder a una petición HTTP, pasamos un esqueleto de la vista y unos datos procedentes del modelo, que juntos se transforman mediante el controlador en una vista personalizada en base a esa petición.

Link: https://ejs.co/

6.3. Estructura del servidor

Tal y como se describen las tecnologías antes descritas que se van a usar del lado de la aplicación, paso a detallar la estructura del servidor y de los métodos que dispone tanto el usuario anónimo como el administrador.

En este caso he implementado un controlador que atienda las peticiones de tipo GET y POST y que son gestionadas por Express.js de manera similar al funcionamiento de una API *web*.

La estructura básica de la cual partiría cualquier experimento sería la de una respuesta a una petición de trabajo, y otra para recoger los datos resultantes, y remitirlos al servidor.

La lógica del lado del cliente se crearía en un fichero Javascript almacenado en los archivos públicos del servidor, de manera que, al recibir una petición por parte del cliente, este recibiría en respuesta los documentos que contienen el trabajo a realizar.

Nos aseguramos que en la creación de dichos documentos se utilicen librerías que funcionen en todos los navegadores *web* disponibles, para maximizar el alcance y la **compatibilidad** de la aplicación, para ello utilizaremos Bootstrap (62) para crear los estilos de las vistas, así como JQuery (63) a la hora de invocar procedimientos en segundo plano que requieran comunicación con el servidor.

6.3.1. Esqueleto de la aplicación

La configuración de una aplicación viene dada por una serie de ficheros bajo el directorio `/config`. En ellos se almacena en objetos de tipo JSON la configuración necesaria para que la aplicación funcione.

El fichero `experimento.json` contiene la información referente a la vista del usuario anónimo y a la configuración correspondiente a la temporización del experimento.

A continuación, una lista de las variables configurables dentro del fichero:

Variable	Función
Fecha de creación	Día en el que se realizó la configuración
Configurado	Activo si el experimento ha sido configurado
Activado	Muestra si el experimento está activo en ese momento
Duración	Días que durará activo el experimento.
TimeOut	Minutos hasta que el servidor considere una prueba perdida
NumPruebas	Numero de evaluaciones que durará el experimento
Título pagina	Cabecera de vista del cliente
Título experimento	Título del experimento
Descripción	Descripción del experimento
Eula	Acuerdo de confidencialidad

4 Variables de configuración

Otro fichero llamado *login.json* contiene la información referente al inicio de sesión de los administradores. Las credenciales de acceso de los usuarios normalmente se almacenan en la base de datos, pero en este caso requerimos solo a un usuario administrador, de manera que los parámetros están almacenados y cifrados dentro de este fichero. Si se altera su contenido no será posible el acceso a la parte de administración. Este procero lo logramos haciendo uso de BCrypt (64), una librería que crea un hash (65) a partir de la clave original y la encripta para poder utilizarla de forma segura.

El fichero *redis.json* se encarga de almacenar los elementos de configuración relacionados con la conexión a la base de datos de Redis, en este caso tiene especificado el nombre del host de nuestro contenedor de Docker, pero en caso de ser necesario, modificando este fichero podríamos configurar otro servidor dedicado para usarlo como base de datos, manteniendo la necesidad de usar Redis.

6.3.2. Recopilación de datos de ejecución

Como sabemos tras la ejecución, se realiza un envío de la información resultante del procesamiento al servidor. Esta información proviene de una variable de datos de tipo cadena declarada dentro del experimento llamada *datos*. Dado que Redis

permite el almacenamiento de objetos, podemos almacenar en dicha cadena con toda la información que creamos importante.

También se dispone de un método de envío de archivos *resultUpload*. Dicho método se encarga de almacenar en la carpeta configurada en la aplicación de los archivos que le mandemos.

6.4. Elementos en la ejecución en el cliente

Como antes mencionaba, el fichero con el código a ejecutar por parte del cliente debe encontrarse la carpeta de contenido público. El nombre que la aplicación usa para indicar su carga en las cabeceras es *CodigoCliente*. En caso de querer sustituir este, debe modificarse su inclusión, ya que si no el cliente no recibirá el contenido y consecuentemente la prueba será errónea.

Para modificar la inclusión en las cabeceras hay que modificar el fichero *header.ejs* bajo el directorio */views*. Dentro de este fichero se pueden incluir todo tipo de librerías que nuestro código en Javascript necesite, y que se encuentren a nivel local o que su carga se haga desde algún CDN. Existen ejemplos de ambos casos en el esqueleto.

La vista de ejecución del usuario anónimo dispone además de ciertos elementos modificables. Concretamente de una barra de progresión con el identificador *myBar*, la cual podemos modificar su porcentaje para indicar el progreso de la prueba. Para ello usaremos un método en Javascript similar al siguiente:

```
document.getElementById("myBar").style.width = 10 + '%';
```

Disponemos también de un cuadro de texto a modo de *log* en el cual podemos escribir mensajes para informar al usuario de qué estamos haciendo o de datos que le puedan resultar interesantes.

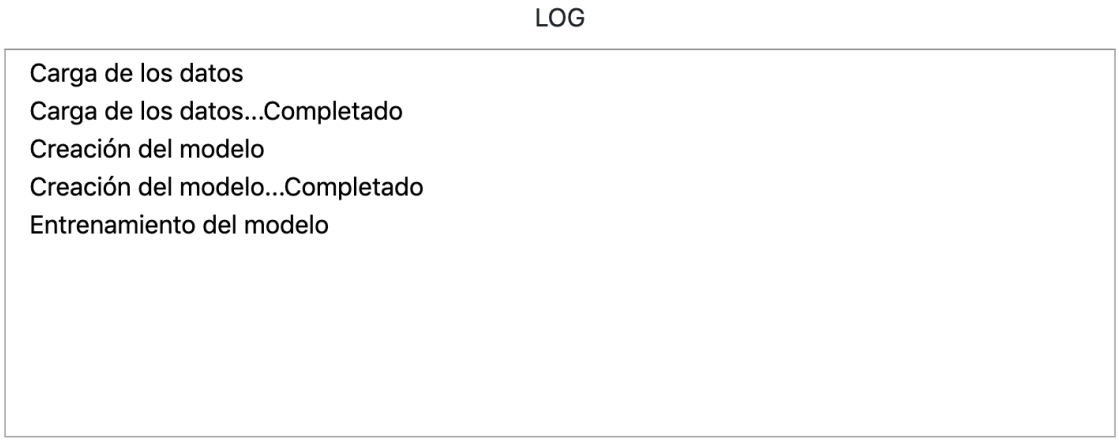


Ilustración 14 Cuadro de información

Podemos utilizar el siguiente método para imprimir mensajes en el log:

```
$('#logbox').append(**mensaje**\n').scrollBottom();
```

Todas las pruebas vienen identificadas con una cadena única que la servirá para tener control de esta y poder identificar problemas, la generamos de manera única usando la librería NanoID (66).

Ilustración 15 Identificador generado

6.5. Vista del cliente

Cualquier usuario que acceda a la web verá una vista similar a la mostrada a continuación.

Ilustración 16 Vista de la página principal

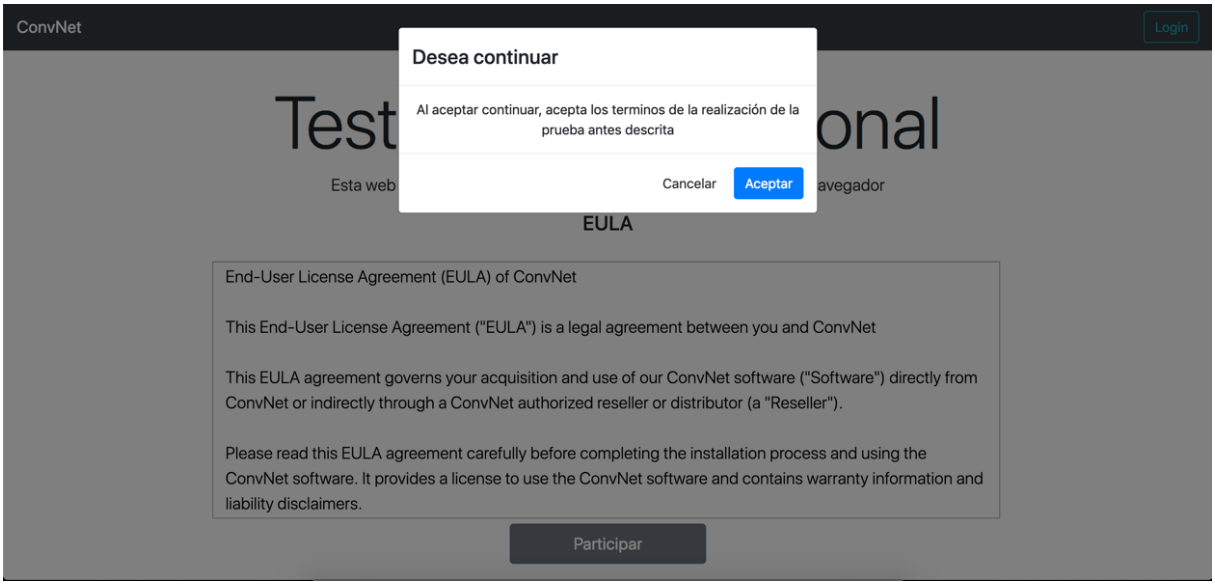


Ilustración 17 Confirmación de lectura del acuerdo de confidencialidad

Tras la aceptación del acuerdo de confidencialidad, el usuario será redirigido a la vista del experimento donde este esperará a que se ejecute en su totalidad. Se le mostrará información acerca del progreso la ejecución, utilizando los métodos antes descritos.

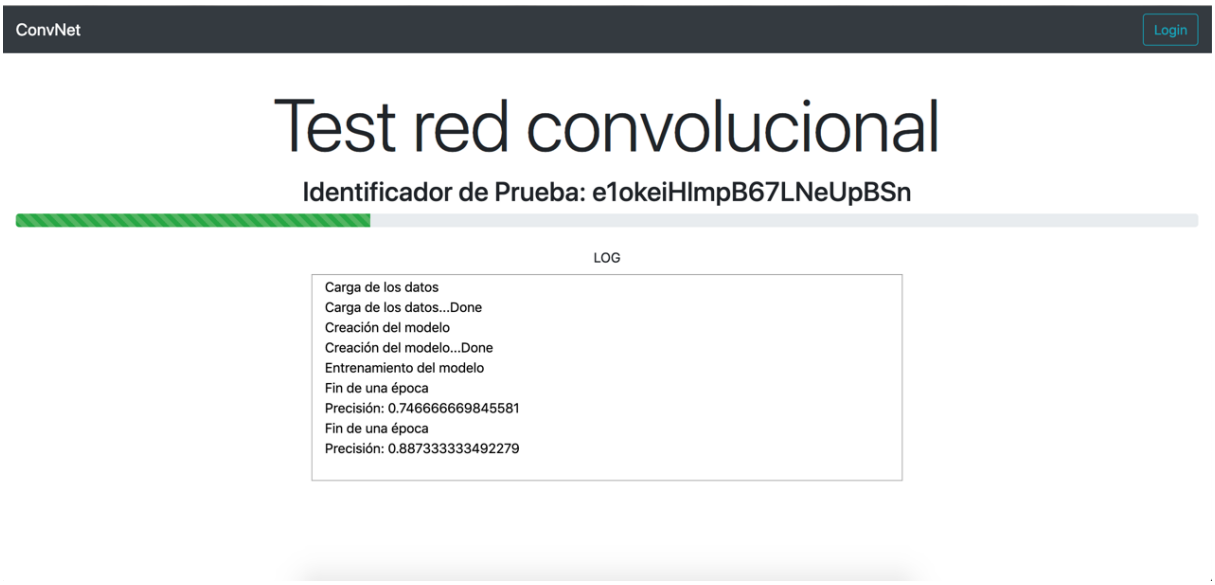


Ilustración 18 Vista de la prueba

Tras la realización se le ofrecerá repetir otra ejecución o salir del mismo.

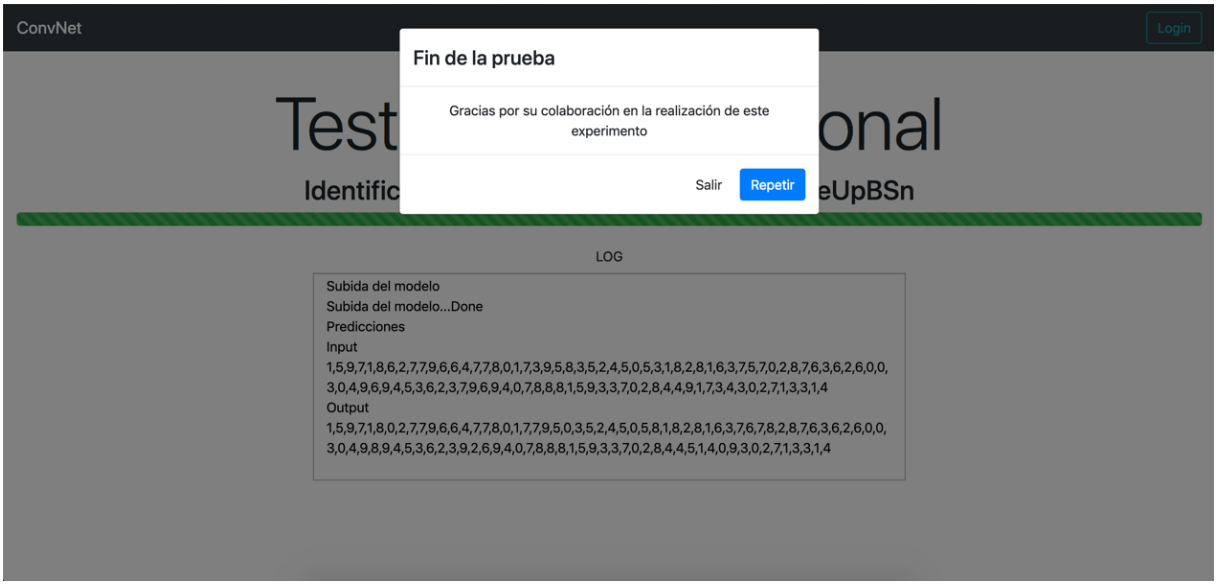


Ilustración 19 Vista final de la prueba

6.6. Vista del administrador

Ilustración 20 Vista panel de administración

El administrador tiene una vista representada en la anterior ilustración, en la que de un simple vistazo es capaz de ver el estado de progreso del proyecto. Dispone además de unos botones que le facilitan el acceso a ciertas tareas de administración que puede usar en cualquier momento.

La siguiente ilustración muestra del formulario desde el que el administrador puede configurar la vista del usuario anónimo modificando las cadenas de texto que se muestran.

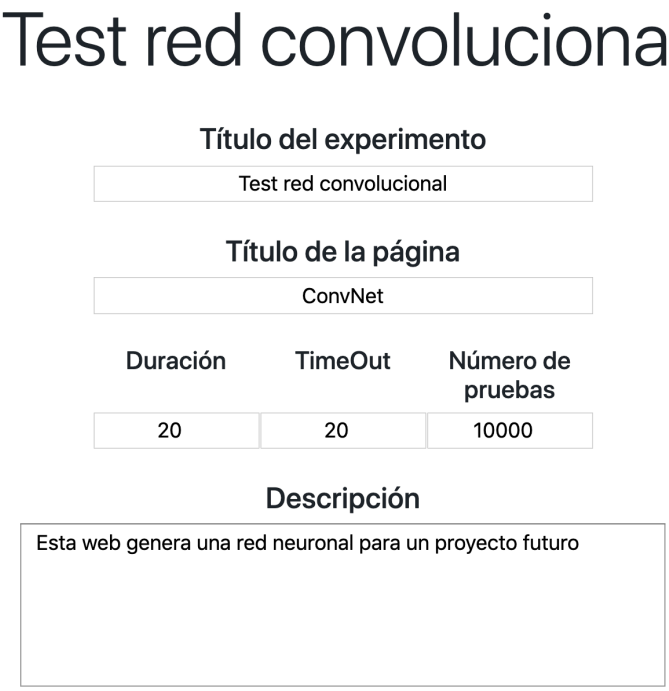


Ilustración 21 Configuración de la vista de usuario y variables de ejecución

Como se observa tanto en las vistas del usuario anónimo como del administrador se ha optado por un diseño simple que favorece la accesibilidad de la aplicación por parte de la mayor cantidad de público posible.

7. Pruebas del sistema

7.1. Resumen del problema implementado

La función esencial de este trabajo era la de estudiar el reparto de trabajo entre clientes mediante tecnologías *web*. Partiendo de esto, el sistema está planteado para entregar al cliente un trabajo a ejecutar y luego recibir por parte de este una respuesta que contenga los resultados del trabajo realizado.

Al cliente se le envía una aplicación para ejecutar el problema descrito a continuación.

Se trata del entrenamiento de una red neuronal que resuelve el problema del reconocimiento de dígitos manuscritos, mas conocido como MNIST (67). Dicho problema consta de un conjunto de 65000 imágenes, comprimidas en un solo archivo, las cuales, en la parte del cliente, definen los 65000 elementos a utilizar por la red. Este es el conjunto de datos que después utilizaremos para las tareas de entrenamiento y test.

Utilizando la librería de TensorFlow.js (68) implementaremos mediante los métodos que contiene la ejecución del lado del cliente. Para ello partiendo del conjunto de datos, se define un modelo cuya estructura la define la siguiente imagen y que aplicará una serie de transformaciones a estos datos de entrada:

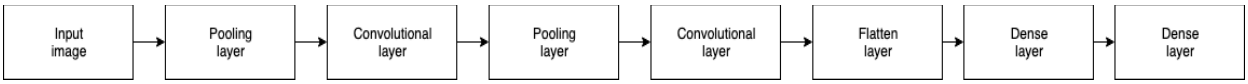


Ilustración 22 Capas de la red neuronal

Comenzando con una entrada de 28x28, o lo que sería lo mismo 784 entradas correspondientes a los pixeles que forman la imagen del dígito manuscrito, aplicamos las siguientes operaciones descritas como capas de la red neuronal:

- Una capa de *Pooling*, para reducir la cantidad de información sin perder los valores importantes, en concreto reducimos una matriz de 2x2 a un único elemento lo que comprime la información de entrada.
- La siguiente capa, de tipo convolucional, consta de 32 filtros de entrada, y la función de activación es de tipo unidad lineal rectificada (ReLU) (69).
- Volvemos a aplicar otra capa de *Pooling* con las mismas características que la anterior, así como otra vez la de tipo convolucional.
- El último paso es la aplicación de dos capas conectadas, la primera con 64 entradas y una función de activación ReLU, y la segunda con 10 entradas y la función de activación Softmax (70).

Esta última capa es la que aplicaría una categorización sobre el conjunto de datos de entrada, en este caso los 10 dígitos manuscritos [0,1,2,3,4,5,6,7,8,9].

El tamaño de la muestra que evalúa el cliente, así como la de test se configura en el fichero del experimento que hemos definido en *CodigoCliente*. También es posible configurar otros parámetros como son las épocas de entrenamiento y otros parámetros que conciernen a la red neuronal.

Una vez hecho este procesamiento, el cliente manda en respuesta la red entrenada al servidor, de manera que se pueden exportar los datos resultantes para realizar operaciones sobre las redes entrenadas bajo el mismo experimento, de modo que mediante un sistema por ejemplo de votaciones se puede tener un resultado común ante una muestra de entrada igual.

También se obtienen algunos datos de interés como datos acerca del sistema operativo de la maquina, características del navegador o el tiempo y la precisión de la prueba ejecutada.

7.2. Comparativa de los resultados

En este apartado realizaremos una comparación estadística en cuanto a rendimiento. Más concretamente veremos si es viable la división del problema en función del tiempo de ejecución y de la precisión obtenida.

Partiendo de un caso base en el que se ejecutará el problema completo en un cliente, dicho problema tiene las siguientes características:

Datos entrenamiento	50000
Datos test	15000
Número de épocas de entrenamiento	10

5 Configuración base

Partiendo de la configuración antes descrita, obtenemos los siguientes resultados:

Ejecución	Precisión	Tiempo ejecución
1	0.9956470727920532	6 minutos 58 segundos
2	0.9959294199943542	6 minutos 56 segundos
3	0.9955999851226807	6 minutos 55 segundos
4	0.9955294132232666	6 minutos 55 segundos
5	0.99541175365448	6 minutos 57 segundos

6 Resultados de partida

Tras esto podemos calcular que la precisión media es de 99,562352896% y que el tiempo medio que tardaría en ejecutarse la prueba sería de 6 minutos 56 segundos.

Para comparar con la división del trabajo plantearemos dos escenarios, uno que reduce el tamaño de la muestra y otro en el que se reducen las épocas de entrenamiento

A continuación, se muestran los resultados obtenidos en ambos casos.

Datos entrenamiento		25000
Datos test		7500
Número de épocas de entrenamiento		10
Iteración	**Precisión**	**Tiempo ejecución**
1	0.9915764927864075	3 minutos 39 segundos
2	0.9923765063285828	3 minutos 33 segundos
3	0.9922823309898376	3 minutos 33 segundos
4	0.9914823770523071	3 minutos 34 segundos
5	0.99541175365448	3 minutos 34 segundos

7 Variante A

Datos entrenamiento		50000
Datos test		15000
Número de épocas de entrenamiento		5
Iteración	**Precisión**	**Tiempo ejecución**
1	0.9848000407218933	3 minutos 33 segundos
2	0.977066695690155	3 minutos 33 segundos
3	0.9884666800498962	3 minutos 38 segundos
4	0.9901999831199646	3 minutos 34 segundos
5	0.9890000224113464	3 minutos 34 segundos

8 Variante B

La interpretación de los datos nos lleva a las siguientes conclusiones.

Como vemos al reducir el tamaño de la muestra mantenemos una precisión media similar a la original, con lo cual en el supuesto de una ejecución distribuida del problema por parte de varios usuarios nos llevará a un resultado similar al que tendríamos si lo ejecutara uno solo al completo.

Las pruebas han sido realizadas sobre una sola máquina, y considerando que el tiempo inicial de ejecución del problema al completo es de 6 minutos 56 segundos, al dividirlo en un problema más sencillo que requiere de la mitad de datos para entrenar la red, vemos que en el supuesto de ejecución en varias máquinas de este problema más simple, se obtiene un tiempo similar de ejecución, con lo que tendríamos el doble de redes entrenadas, con una precisión bastante similar a la del problema original, empleado para ello la misma cantidad de tiempo.

8. Conclusiones

8.1. Extensibilidad

El prototipo creado da lugar a diferentes mejoras que se describen a continuación.

Comenzando con la estructura de la aplicación en Node.js, la cual debería dividirse en dos partes, una dedicada a la de distribución de contenido, y la otra dedicada a la recepción de los datos resultantes.

La distribución de contenido debe concebirse como un servidor con una conexión de red rápida, cacheando el contenido para reducir al máximo el tiempo de espera de los usuarios que quieran participar. El funcionamiento sería parecido al de una CDN (71).

En cuanto a la segunda parte, la recepción de los datos, este servidor debería estar centrado en el volcado de los resultados al sistema de archivos y a la base de datos.

Sería también interesante la implementación de réplicas, usando por ejemplo Kubernetes (72). El prototipo esta preparado para su funcionamiento mediante contenedores, con lo que lo único que sería necesario desacoplar la parte del administrador creando para ello una aplicación secundaria que se encargue de gestionar y agrupar la información necesaria.

Durante el proceso de documentarme para este trabajo se me ocurrió una alternativa a la hora de diseñar la aplicación, pero lo descarté debido a la dificultad que supondría.

La alternativa era la de crear una *web* en la que el cliente mantuviera una sesión continua con el servidor, similar a la que existiría en una sesión de SSH (73). De esta forma, mediante una serie de funciones creadas en el cliente similar a un RPC (74),

un hipotético administrador podría enviar cargas de trabajo a ejecutar a los clientes activos usando dicha conexión.

8.2. Conclusiones propias

El desarrollo de este trabajo me ha permitido abordar un proyecto real desde cero, partiendo del problema inicial, utilizando tecnologías que eran nuevas para mi ya que, por ejemplo, mi conocimiento de Javascript era bastante básico. El aprender Node.js y Redis ha sido un reto, así como el manejo básico de una librería como es Tensorflow.js. Sin duda ha sido una manera de mostrar la capacidad de adaptarme a la hora de abordar un problema.

Anexo A: Bibliografía

1. C++. [En línea] https://es.wikipedia.org/wiki/C%2B%2B.
2. JAVA. [En línea] https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n).
3. Computación distribuida. [En línea] https://en.wikipedia.org/wiki/Distributed_computing.
4. Arquitectura orientada a servicios. [En línea] https://en.wikipedia.org/wiki/Service-oriented_architecture.
5. AMS. [En línea] https://es.wikipedia.org/wiki/Arquitectura_de_microservicios.
6. P2P. [En línea] https://en.wikipedia.org/wiki/Peer-to-peer.
7. BitTorrent. [En línea] https://es.wikipedia.org/wiki/BitTorrent.
8. MMOG. [En línea] https://es.slideshare.net/ChrisMohritz/introduction-to-mmog-architecture.
9. MMOModel. [En línea] https://hackernoon.com/authoritative-mmo-data-models-5dc4c1aa30fa.
10. HTTP. *Protocolo de transferencia de hipertexto.* [En línea] https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto.
11. W3C. *W3C.* [En línea] https://www.w3c.es/.
12. IETF. *IETF.* [En línea] https://www.ietf.org/.
13. Hypertext Transfer Protocol Version 2 (HTTP/2). *Hypertext Transfer Protocol Version 2 (HTTP/2).* [En línea] https://tools.ietf.org/html/rfc7540.
14. Tcp. *https://es.wikipedia.org/wiki/Protocolo_de_control_de_transmisi%C3%B3n.* [En línea]
15. Proxy. [En línea] https://es.wikipedia.org/wiki/Servidor_proxy.
16. SSL. [En línea] https://es.wikipedia.org/wiki/Transport_Layer_Security.
17. HTTP status codes. [En línea] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.
18. HTML. [En línea] https://es.wikipedia.org/wiki/HTML.
19. HTML 5. [En línea] https://es.wikipedia.org/wiki/HTML5.
20. W3C html 5.1 recommendations. [En línea] https://www.w3.org/TR/2017/REC-html51-20171003/.
21. CSS 3. [En línea] https://developer.mozilla.org/es/docs/Web/CSS/CSS3.
22. Flash. [En línea] https://es.wikipedia.org/wiki/Adobe_Flash.
23. CSS recommendations. [En línea] https://www.w3.org/Style/CSS/Overview.en.html.
24. BoomerangJS. [En línea] http://www.boomerangjs.org/.
25. BOINC. [En línea] https://boinc.berkeley.edu/.
26. deThread. [En línea] https://github.com/deThread/dethread.
27. Headless. [En línea] https://en.wikipedia.org/wiki/Headless_software.
28. Web worker. [En línea] https://en.wikipedia.org/wiki/Web_worker.
29. Socket. *https://es.wikipedia.org/wiki/Socket_de_Internet.* [En línea]
30. Javascrip Engine. [En línea] https://en.wikipedia.org/wiki/JavaScript_engine.
31. V8 chrome. [En línea] https://v8.dev/.
32. JS Bench. [En línea] https://www.linkedin.com/pulse/algorithmic-performance-comparison-c-javascript-java-malyshev/.
33. JS out of the web. *https://hackernoon.com/the-status-of-javascript-outside-of-the-browser-2018-beyond-ee0b79ee059f.* [En línea]
34. JS Real performance. [En línea] https://itnext.io/the-reality-of-javascript-performance-4ec9747882d3.
35. DOM. [En línea] https://es.wikipedia.org/wiki/Document_Object_Model.
36. Web worker performance. [En línea] https://developer.mozilla.org/es/docs/Web/Guide/Performance/Usando_web_workers.

37. MacBookPro. [En línea] https://www.apple.com/es/macbook-pro/specs/.
38. OVH. [En línea] https://www.ovh.es/vps/vps-ssd.xml.
39. Amortización de equipos. [En línea] https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml.
40. Salario de un programador. [En línea] https://www.boe.es/boe/dias/2017/01/18/pdfs/BOE-A-2017-542.pdf.
41. Movistar 30Mb. [En línea] https://www.movistar.es/particulares/internet/adsl-fibra-optica/oferta-fibra-optica-30mb.
42. Consumo medio de luz. *https://enpromedio.com/promedios-de-consumo/factura-energetica-promedio.* [En línea]
43. Consumo agua. *https://enpromedio.com/promedios-de-consumo/consumo-medio-de-agua.* [En línea]
44. Oficina. [En línea] https://www.idealista.com/inmueble/85723618/.
45. MVC. *https://es.wikipedia.org/wiki/Modelo–vista–controlador.* [En línea]
46. Python scientific use. [En línea] https://www.stat.washington.edu/~hoytak/blog/whypython.html.
47. NodeJS. [En línea] https://nodejs.org/es.
48. Apache. [En línea] https://www.apache.org/.
49. Nginx. [En línea] https://www.nginx.com/.
50. Rails. [En línea] https://rubyonrails.org/.
51. Node.js HTTP. [En línea] https://nodejs.org/api/http.html.
52. SQL. [En línea] https://es.wikipedia.org/wiki/SQL.
53. ACID. [En línea] https://es.wikipedia.org/wiki/ACID.
54. MYSQL. [En línea] https://www.mysql.com/.
55. Oracle SQL. [En línea] https://docs.oracle.com/en/database/oracle/oracle-database/18/index.html.
56. Access. [En línea] https://products.office.com/es-es/access.
57. MongoDB. [En línea] https://www.mongodb.com/es.
58. Redis. [En línea] https://redis.io/.
59. Docker. [En línea] https://www.docker.com/.
60. Git. [En línea] https://git-scm.com.
61. GitHub. [En línea] https://github.com/.
62. Bootstrap. *https://getbootstrap.com/.* [En línea]
63. JQuery. *https://jquery.com/.* [En línea]
64. bcrypt. *https://github.com/kelektiv/node.bcrypt.js#readme.* [En línea]
65. Hash password. *https://www.php.net/manual/es/faq.passwords.php.* [En línea]
66. NanoID. *https://github.com/ai/nanoid#readme.* [En línea]
67. MNIST. [En línea] http://yann.lecun.com/exdb/mnist/index.html.
68. TensoFlow.js. [En línea] https://www.tensorflow.org/js.
69. Relu. *https://js.tensorflow.org/api/latest/#layers.reLU.* [En línea]
70. Softmax. *https://js.tensorflow.org/api/latest/#layers.softmax.* [En línea]
71. CDN. [En línea] https://es.wikipedia.org/wiki/Red_de_distribuci%C3%B3n_de_contenidos.
72. Kubernetes. [En línea] https://kubernetes.io/.
73. SSH. *https://es.wikipedia.org/wiki/Secure_Shell.* [En línea]
74. rpc. [En línea] https://es.wikipedia.org/wiki/Llamada_a_procedimiento_remoto.
75. Autoencoder. [En línea] https://en.wikipedia.org/wiki/Autoencoder.

76. Javascript. [En línea] https://es.wikipedia.org/wiki/JavaScript.
77. CSS. [En línea] https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada.
78. SASS. [En línea] https://es.wikipedia.org/wiki/Sass_(lenguaje_de_hojas_de_estilo).
79. Webassembly. [En línea] https://webassembly.org/.
80. SpreadSheet Webassembly. [En línea] https://webassembly.github.io/spec/core/bikeshed/index.html.
81. SandBox. [En línea] https://es.wikipedia.org/wiki/Aislamiento_de_procesos.
82. Wordpress. [En línea] https://es.wordpress.com/.
83. Laravel. [En línea] https://laravel.com/.
84. Codeigniter. [En línea] https://codeigniter.com/.
85. Symfony. [En línea] https://symfony.com/.
86. Apache . [En línea] https://en.wikipedia.org/wiki/Apache_HTTP_Server.
87. ExpressJS. [En línea] http://expressjs.com/.
88. NodeJS http. [En línea] https://nodejs.org/api/http.html#http_http.
89. Express.js. [En línea] https://en.wikipedia.org/wiki/Express.js.
90. QUIC. [En línea] https://www.chromium.org/quic.
91. Precio luz. *https://www.helpmycash.com/cat/energia/comparador-de-energia/.* [En línea]

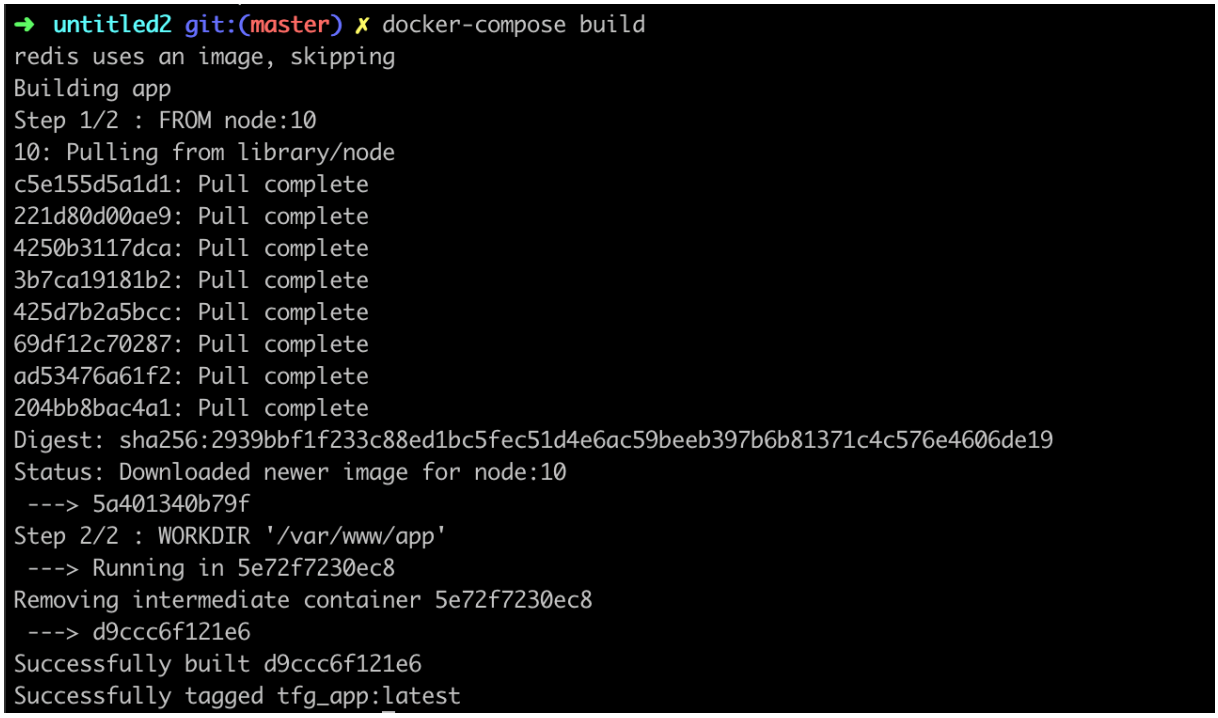
Anexo B: Instalación

En este anexo describiré como ejecutar el proyecto implementado en una máquina que vaya a funcionar de servidor.

Necesitamos una instalación de Docker, así como del paquete Docker-compose que nos permite definir grupos de máquinas de Docker y las redes que las conectan de una manera mas sencilla a la que nos supondría hacerlo de manera manual.

Estando dentro de la raíz de la carpeta que contiene todo el proyecto ejecutaríamos:

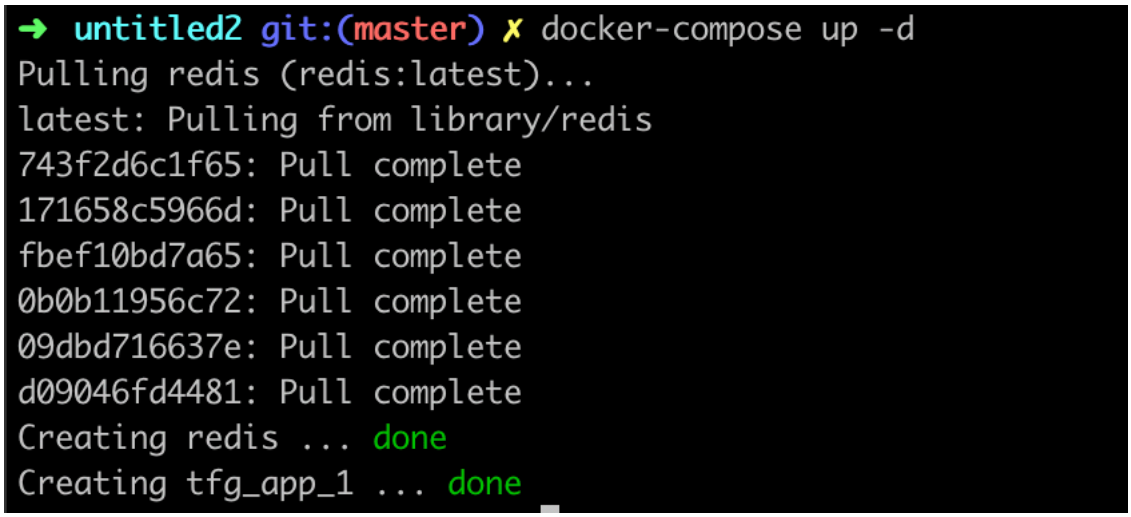
```
docker-compose build
```



Esto nos descargará las imágenes necesarias para el funcionamiento y las modificará en función de la configuración predefinida.

Después ejecutaremos:

```
docker-compose up -d
```



Esto pondrá en funcionamiento los contenedores con las imágenes, y lanzará la instancia de la aplicación que será accesible en el puerto 3000 de la dirección IP de la máquina que ejecuta la instancia de Docker.

Para detener el proyecto ejecutar:

```
docker-compose stop
```

Anexo C: Configuración de un trabajo

En este apartado explicaré como configurar a carga de trabajo para su posterior distribución partiendo del esqueleto base de la aplicación.

Partiríamos de los ficheros en la carpeta del esqueleto, seguiríamos los pasos descritos a continuación:

1. Copiar el código en Javascript a ejecutar dentro de `/public/javascript` bajo el nombre `CodigoCliente.js`
2. Añadir a las cabeceras definidas en `/views/header.ejs` las librerías procedentes de un CDN o del almacenamiento local para que nuestra aplicación flas cargue en el cliente.
3. Realizar los pasos del Anexo B.
4. Acceder con las credenciales por defecto: *admin*, *1234*, las cuales se recomienda encarecidamente cambiar dada su baja seguridad.
5. En el apartado de configuración rellenar los campos de la vista de usuario con la información del experimento, así como de la configuración de este.

Personalización avanzada

Hay ciertas funciones adicionales que requieren conocimientos avanzados para su configuración, pero que son fácilmente configurables por alguien con experiencia.

Se puede habilitar la subida de archivos al servidor modificando el fichero `/routes/index.js`. Al final de este existe una función que permite la subida de archivos mediante el método POST a una carpeta dentro del servidor. Sería necesaria su modificación de acuerdo con las necesidades del problema alojado.