UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de una plataforma de búsqueda e indexado de contenido para el protocolo Bittorrent

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Iván José Martín García

Tutor: Jose Salvador Oliver Gil

Curso 2018-2019

Resumen

En este proyecto se aborda el análisis e implementación de una plataforma de automatización de extracción de datos para el protocolo Bittorrent. Haciendo uso de la DHT (Tabla de Hashes Distribuida) y realizando un ataque a esta red, se consigue crear un sistema totalmente autónomo y que con la ayuda de una interfaz web permite la exploración de todo su contenido de una forma sencilla para el usuario. De esta forma, este proyecto intenta afrontar uno de los problemas más grandes del protocolo Bittorrent, la descubribilidad de contenido.

Palabras clave: scraping, Bittorrent, DHT

Abstract

This project addresses the analysis and implementation of an autonomous data extraction platform for the Bittorrent protocol. A totally autonomous system is created by performing an attack to the DHT (Distributed Hash Table) network and with the help of a web interface all the content collected is exposed to the user in a convenient manner. In this way, this project tries to face one of the biggest problems of the Bittorrent protocol, the discoverability of content.

Key words: scraping, Bittorrent, DHT

Índice general

Índice de figuras

CAPÍTULO 1

Introducción

Desde los inicios de Internet, la necesidad de compartir o intercambiar ficheros ha sido una prioridad para sus usuarios, a este proceso se le denomina **transferencia de archivos**. En los años 70 y 80, la única manera de hacerlo era mediante medios físicos como podían ser los disquetes . En la actualidad se estima que entorno a un 6 % [3] del tráfico diario de internet está ocupado por la transferencia de archivos entre usuarios. Para alcanzar estos números se ha tenido que mejorar y optimizar el cómo realizamos esta tarea y no fue hasta el año 2000, que gracias a las mejoras en las capacidades de almacenamiento y velocidades de red que empezaron a aparecer las primeras versiones de las herramientas que se usan en la actualidad.

La mayoría de mejoras en ámbito de la transferencia de archivos han venido de la mano de nuevos protocolos y durante los años se han asentado los que han probado ser capaces de satisfacer las necesidades de los usuarios. Uno de estos es Bittorrent, que en algunos momentos llegó a acaparar hasta más de un 3 % del porcentaje comentado anteriormente. Este protocolo se basa en una arquitectura P2P (abreviatura de "peer-to-peer" en inglés) lo que supone que no se depende de un nodo central a la hora de realizar la transferencia y solamente es necesario encontrar uno o más pares o "peers" en la red con ese mismo archivo.

Bittorrent no ha sufrido cambios drásticos desde su creación en 2001 y algunos de los problemas que sufre a nivel de diseño como pueden ser la latencia o el uso de recursos han sido solucionados de una forma externa al protocolo. El que más importunio ha causado a los usuarios siempre ha sido la *descubrilidad*, ya que el protocolo no ofrece ninguna manera directa de explorar el contenido de su red. La forma de afrontar este problema a lo largo de los años ha sido crear buscadores alojados por terceros que dependen de que los usuarios publiquen los enlaces al contenido de forma manual, pero esta solución nunca ha sido óptima puesto que se necesita que alguien publique este enlace por lo que gran parte del contenido de la red queda invisibilizado.

Con el tiempo, Bittorrent se ha ido respaldando en otras tecnologías para mejorar las carencias anteriormente descritas y una de estas han sido las Tablas Hash Distribuidas (o DHT por sus siglas en inglés). Las DHT ha abierto un gran abanico de posibilidades y una de estas es, mediante diversas técnicas, realizar un indexado de la red. Es por esto que se ha decidido aprovechar esta funcionalidad y crear un buscador que, usando la DHT, permita a todos los usuarios de Bittorrent buscar y descubrir nuevo contenido de forma sencilla en la red de Bittorrent.

En la siguiente sección se describirá brevemente los conceptos básicos que componen el protocolo y arquitectura Bittorrent, así como las redes DHT. Muchos de estos conceptos serán utilizados posteriormente en el análisis y en la implementación para desarrollar algunas partes del proyecto.

1.1 Objetivos

El proyecto tiene como objetivo el desarrollo de una aplicación para el descubrimiento automatizado de contenido en la red Bittorrent. Para ello, se diseñarán dos componentes principales: un servicio que recopile nuevo contenido en la red y gestione su persistencia y una interfaz web que permita la búsqueda del contenido disponible.

1.1.1. Objetivos particulares

- **Buscador de contenido autónomo**: el pilar fundamental del proyecto es un buscador autónomo que vaya añadiendo el contenido a una base de datos.
- **Interfaz web para explorar contenido**: una interfaz web que permita explorar el contenido encontrado por el buscador.
- **Interfaz web simple**: la interfaz web ha de ser simple.
- **Metadatos [2] en la interfaz web**: la interfaz web deberá de mostrar un mínimo de metadatos del contenido encontrado, como por ejemplo el nombre del archivo o el tamaño de los archivos.
- **El buscador de contenido debe tener una arquitectura simple**: el componente del buscador puede llegar a ser bastante complejo, es por esto que debemos hacer especial hincapié en simplificar su arquitectura.
- **Buscador totalmente descentralizado**: el buscador no puede depender de ningún servicio que sea centralizado para garantizar que el sistema pueda funcionar con tan solo un simple acceso a internet.

- **El buscador de contenido debe ser rápido**: cada usuario que descarga el proyecto tiene que empezar a construir su base de datos desde cero, es por esto que debemos intentar que la base de datos se pueble de datos lo más rápido posible.
- **Información del torrent a descargar**: en la web se mostrará información básica como el nombre del torrent y tamaño de la descarga.
- **Simplicidad en su uso**: el proyecto debe de venir preconfigurado y el usuario deberá de realizar el menor número de pasos posibles para poder ejecutarlo.

CAPÍTULO 2

Conceptos teóricos

El objetivo de este capitulo es realizar una explicación de los conceptos teóricos que se van a tratar a lo largo de todo el trabajo, para así facilitar la comprensión y explicación de las decisiones tomadas durante el desarrollo del proyecto.

2.1 Redes P2P

Una red P2P es una red de ordenadores en la que cada ordenador que participa en la red está considerado un nodo y no existe ningún componente central que regule el comportamiento entre estos. Los nodos de la red trabajan en conjunto distribuyéndose la carga.

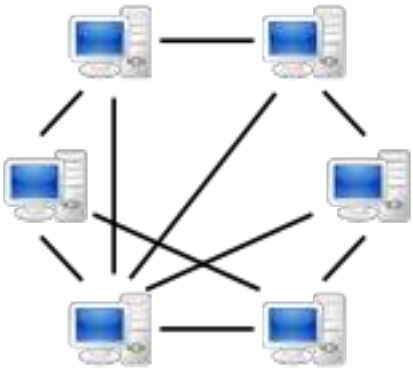


Figura 2.1: Ilustración de un conjunto de nodos formando una red P2P

Los usos de las redes P2P a día de hoy son bastante extensos y van desde la transferencia de archivos hasta su uso en telefonía para hacer más eficiente la transmisión de datos en las llamadas. Durante los últimos años este tipo de redes están auge debido a que no necesitan ningún punto central que las regule, esto permite eliminar intermediarios y hace que la censura a lo que los usuarios puedan compartir sea virtualmente nula.

Una de las primeras grandes redes P2P fue Napster [4], creada por Shawn Fanning [5] en 1999. Este servicio tenía como fin la transferencia de música entre usuarios. Este fue el inicio del uso de las redes P2P en la transferencia de archivos tal y como las conocemos hoy en día y supuso un enfoque revolucionario.

2.2 Bittorrent

El protocolo fue diseñado en 2001 por el programador Bram Cohen y su objetivo era ofrecer una forma de transferir archivos entre usuarios sin la necesidad de costosos servidores, además de intentar ahorrar gran ancho de banda. Con estos requisitos, la arquitectura más clara en la que basar el protocolo era P2P.

A lo largo de los años se han ido confeccionando nuevas especificaciones de este. No fue hasta 2013 que apareció la versión 1.0, aunque desde sus inicios hasta esta última versión no hay cambios sustanciales en el proyecto. En la actualidad se considera uno de los protocolos por excelencia a la hora de compartir archivos de gran tamaño a través de la red.

2.2.1. Fundamentos

Las descargas a través de la red de Bittorrent consisten de la transferencia de archivos entre nodos que estén tratando de descargar el mismo archivo o que ya lo hayan descargado. A la hora de realizar la transferencia de un archivo no se hace de forma completa ya que sería ineficiente, puesto que una vez se comenzara la transferencia del contenido entre dos nodos, no podría ser interrumpida. Es por esto que los archivos se dividen en partes más pequeñas. Estas partes se conocen como piezas o *pieces* en inglés y los tamaños de estas pueden variar en función del archivo en cuestión, los tamaños más habituales son 64kB, 128kB, 512kB, 1MB, 2MB o 4MB.

En el momento de la transferencia del contenido podemos observar dos actores: *seeders* y *leechers*.

- *Leechers*: nodo que acaba de empezar a descargar un archivo o está en proceso de ello. Depende de que otros participantes de la red les envien **piezas** de los archivos
- *Seeders*: nodo que tiene todas las piezas de un archivo. Estos nodos son los más valiosos ya que se necesita al menos uno de ellos en la red por cada archivo para que la descarga se pueda finalizar.

Dado que los nodos transfieren piezas entre sí, debe de existir algún mecanismo para comprobar que la información que se está recibiendo no ha sido modificada por el nodo que la envía. Es por esto que al comenzar con la descarga de un archivo, se cuenta primero con un listado de *hashes* que hacen referencia a las distintas piezas, de esta forma cuando finaliza la descarga de cada archivo se puede comprobar su integridad.

2.2.2. Trackers

Como se acaba de explicar, existen dos tipos de actores en la red y para que estos comiencen a compartir piezas los unos con los otros, primero deberán de saber de la existencia del otro en la red. Originariamente esto se hacía con *trackers*. Los *trackers* son un tipo de servidores especiales que coordinan la comunicación entre nodos en la red.

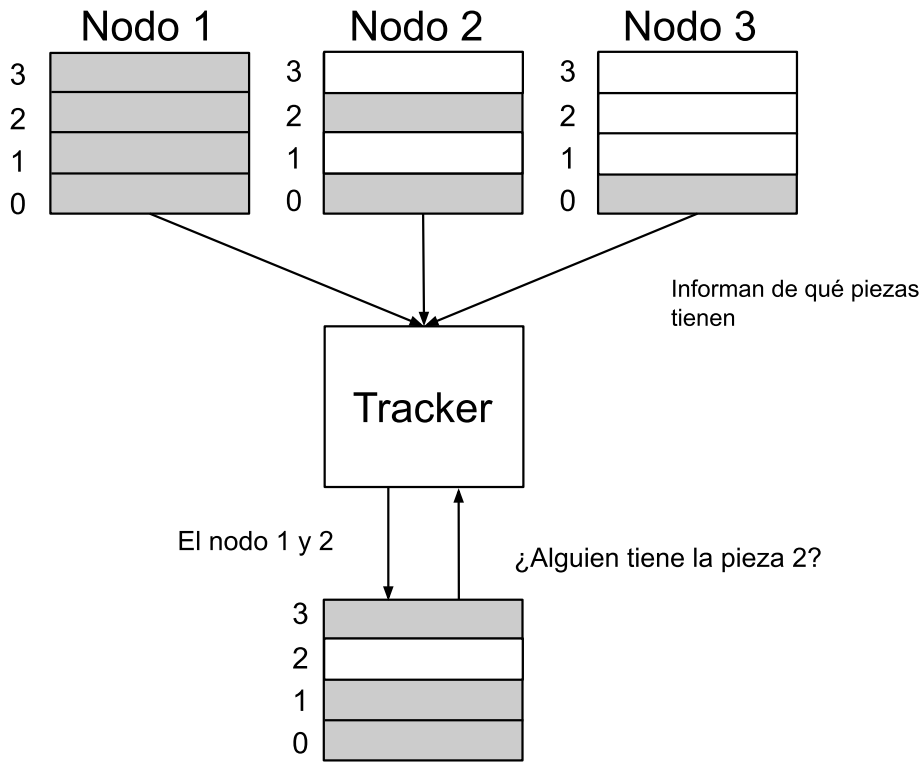


Figura 2.2: Ilustración del funcionamiento de un tracker en la red de Bittorrent

Tal y cómo se puede observar en la figura 2.2, esta arquitectura no es para nada idónea ya que añade una componente que centraliza todo el sistema, si el *tracker* no está disponible no hay forma de que los nodos puedan comunicarse o encontrar a otros en la red.

El tracker más grande de la historia de BitTorrent fue Demonii con un total de 56 millones [6] de nodos conectados.

2.2.3. DHT

Dado que los *trackers* nunca han sido una solución óptima, una alternativa era necesaria para eliminar la centralización y que permitiese a los nodos comunicarse sin ningún tipo de intermediario. Una de las formas de solucionar esto es haciendo uso de una tabla de hash distribuida. Estas son un tipo de sistema distribuido que tiene un mecanismo de búsqueda similar al de las *tablas hash*, es decir, una clave está asociada a un valor. La diferencia más notable con las tablas hash tradicionales es que en este caso los valores están distribuidos por los nodos de la red.

Para guardar un dato en la red primero es necesario producir un *hash*, en el caso de la DHT usada por Bittorrent, la función para calcular el *hash* utilizada es la SHA-1. Una vez se ha obtenido el *hash* al dato, se manda a la red junto con la clave y este es alojado en un nodo en particular. En el contexto de Bittorrent se conoce a estas las claves como *infohashes*.

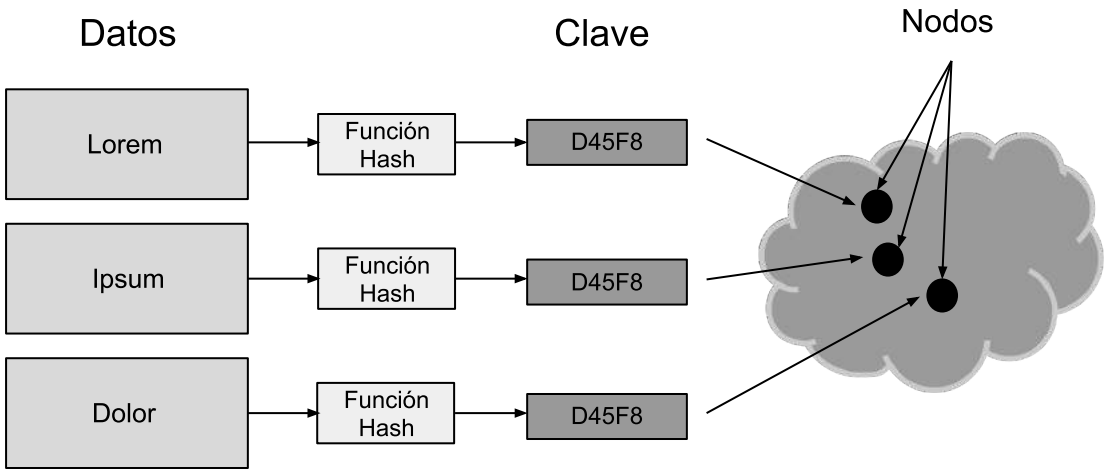


Figura 2.3: Ilustración de una DHT

Espacio de claves

La acción de guardar el contenido en la red no se realiza de forma arbitraria y es que hay unas reglas para realizar esto. En el momento en el que un nodo se une a la red, debe de auto-asignarse un identificador. Para hacer esto se utiliza la misma función de *hashing* comentada anteriormente, por lo que tanto los archivos como los nodos de la red podemos identificarlos con *hashes*. Gracias a esto, es posible considerar que los archivos y los nodos se encuentran en el mismo espacio de claves y por lo tanto, se determinar la **cercanía** de un nodo a un archivo gracias al *hash* de cada uno.

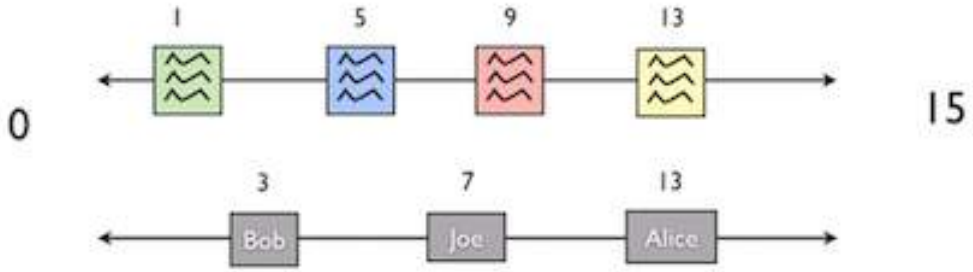


Figura 2.4: Ilustración simplificada del espacio de claves archivos/nodos

Esta cercanía no es física, es tan solo una forma lógica de ordenar la DHT. La formula básica que se utilizaría para calcular la distancia entre dos puntos sería:

$$\boxed{d_{euclidean} = (a, b) = |a - b|} \tag{2.1}$$

El problema con esta formula es que puede haber casos en los que la distancia entre dos nodos y un archivo sea la misma. Como por ejemplo en la figura 2.4 el caso del nodo 3 y 7 con el archivo 5. Dado que $5 - 3 = 2$ y $7 - 5 = 2$. Es por esto, que para evitar este tipo de situaciones es necesario calcular la distancia de otra forma. La solución que aplica la DHT es usar una función *XOR*:

$$\boxed{d_{xor} = (a, b) = bitwise\_xor(a, b)} \tag{2.2}$$

Gracias a la función *xor* se puede calcular la distancia entre nodos y además eliminamos el caso de que dos nodos estén igual de cerca de un archivo. Si comprobamos el caso de antes podemos ver como ahora el nodo más cercano al archivo sería el 7.

$$\begin{array}{r} 5 \\ \wedge 3 \\ \hline 6 \end{array} = \begin{array}{r} 0101 \\ \wedge 0011 \\ \hline 0110 \end{array} \qquad \begin{array}{r} 7 \\ \wedge 5 \\ \hline 2 \end{array} = \begin{array}{r} 0111 \\ \wedge 0101 \\ \hline 0010 \end{array}$$

Enrutado

Dado que la red tiene un gran tamaño, es imposible que un nodo conozca a todos los demás participantes. Es por esto, que existen unas normas para que los nodos puedan comunicarse con otros sin haberlos descubierto antes.

Los nodos de la DHT tan solo tienen que conocer una parte de la red para poder comunicarse con todo el resto, a esta parte de la red se les suele conocer como *vecinos*.

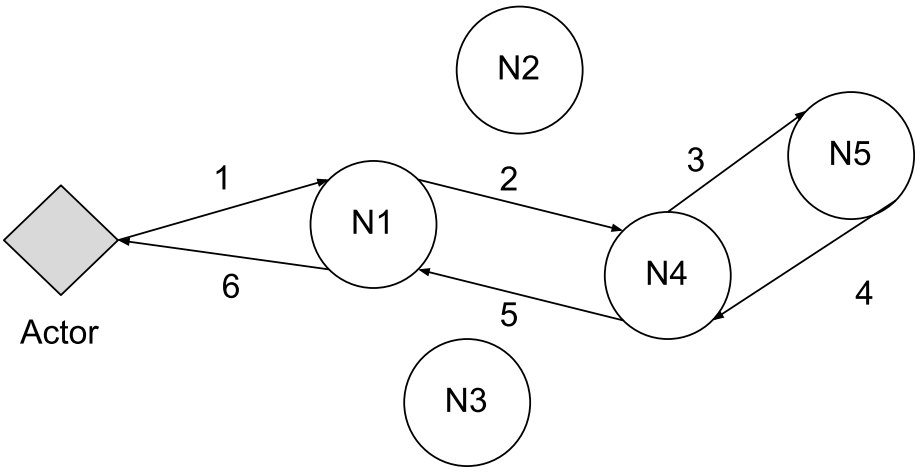


Figura 2.5: Ilustración de un ejemplo de enrutado

Es por esto que los nodos siempre tratarán de comunicarse con sus vecinos para obtener información de la red en el caso de que necesiten algo.

Toda la información relacionada con la topología de la red se almacena en memoria por cada nodo y a esta estructura se le denomina **tabla de enrutado**. La implementación de la estructura de estos datos varia en función al cliente que se utilice para conectarse a la red.

La figura 2.5 demuestra el *dialogo* seguido por los nodos cuando un actor externo hace uso de la DHT e intenta recuperar un archivo que se encuentra en N5. Este ejemplo en concreto sería si el actor realizase una llamada anónima, porque en su defecto el nodo N5 podría contestar de forma directa al actor. En detalle lo sucedido sería:

1. **Actor**: GET *X*, *N1*.
2. **N1**: *X* no está gestionado por mi, *N4* es más cercano.
3. **N4**: *X* no está gestionado por mi, *N5* es más cercano.
4. **N5**: OK, aquí está *X*.
5. **N4**: OK, devolviendo *X* a *N3*.
6. **N1**: OK, aquí está *X*.

Llamadas

En la DHT de Bittorrent existen cuatro llamadas de tipo búsqueda con las que operar:

- ***ping***: sirve para comprobar si un nodo está activo y funcional.
- ***find_node***: permite descubrir más información sobre un nodo sabiendo su ID.
- ***get_peers***: permite obtener más nodos que estén tratando de descargar un archivo dado su ID.
- ***announce_peer***: se anuncia a la red que se ha comenzado la descarga de un archivo con un ID determinado.

CAPÍTULO 3

Análisis y diseño conceptual

En este capítulo se profundizará en el análisis de los objetivos expuestos para proponer un diseño que pueda satisfacerlos.

Dado que este proyecto tiene dos componentes separados, debemos diferenciar y tratar a cada uno de estos de forma independiente, ya que los objetivos de cada uno son diferentes y por lo tanto se necesitarán arquitecturas distintas.

En primer lugar está el componente de *búsqueda* de contenido. Este componente se encargará de descubrir nuevo contenido y tendrá la necesidad de persistir y hacer comprobaciones en una base de datos. En segundo lugar se encuentra la interfaz web que se encargará de servir los datos encontrados por el buscador consultando de la base de datos .

Ya que ambas componentes necesitan acceder a la base de datos, tiene sentido que toda esa lógica sea compartida entre los dos componentes, para aumentar la reusabilidad del código.

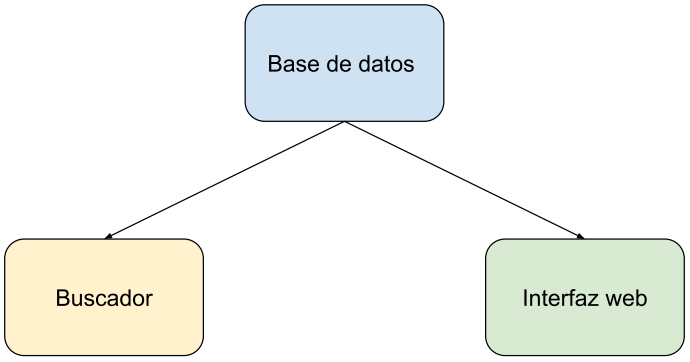


Figura 3.1: Organización del código

3.1 Buscador de contenido

El primer reto que plantea la creación de un buscador de contenido para el protocolo Bittorrent y la DHT es la exploración del contenido en la red, ya que ninguna de las dos da soporte en la actualidad para descubrir ficheros disponibles. Por ello, es necesario implementar un mecanismo de rastreo activo que sea capaz de interceptar paquetes e indexar el contenido existente.

La comunidad es consciente de este problema y por ello, en las mejoras y propuestas que se hacen para mejorar Bittorrent existe un plan [7] con modificaciones a la DHT que permitiría pedir información sobre los archivos que tiene un nodo. Por ahora esto no es más que una propuesta y por lo tanto no es una solución al problema que se plantea.

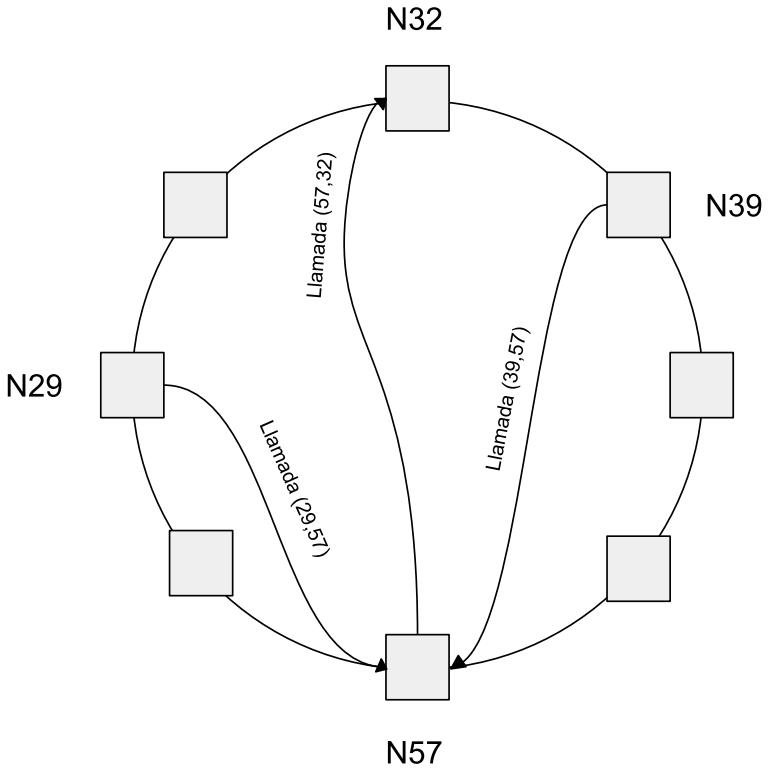
En la introducción se ha expuesto brevemente el funcionamiento de la DHT y uno de los matices de su implementación es que los nodos, por su limitado conocimiento de la red en general, siempre harán uso de nodos vecinos para hacer llegar los datos a dónde necesiten. Esto quiere decir que por el mero hecho de un nodo estar en la red recibirá datos de otros nodos.

Si se tiene en cuenta este último punto una de las opciones es que el buscador realizase el papel de un nodo normal y poco a poco fuese obteniendo información relacionada con archivos de la red. El problema de esto es que sería ineficiente puesto que solamente se obtendría información de los nodos más cercanos y de una forma muy lenta.

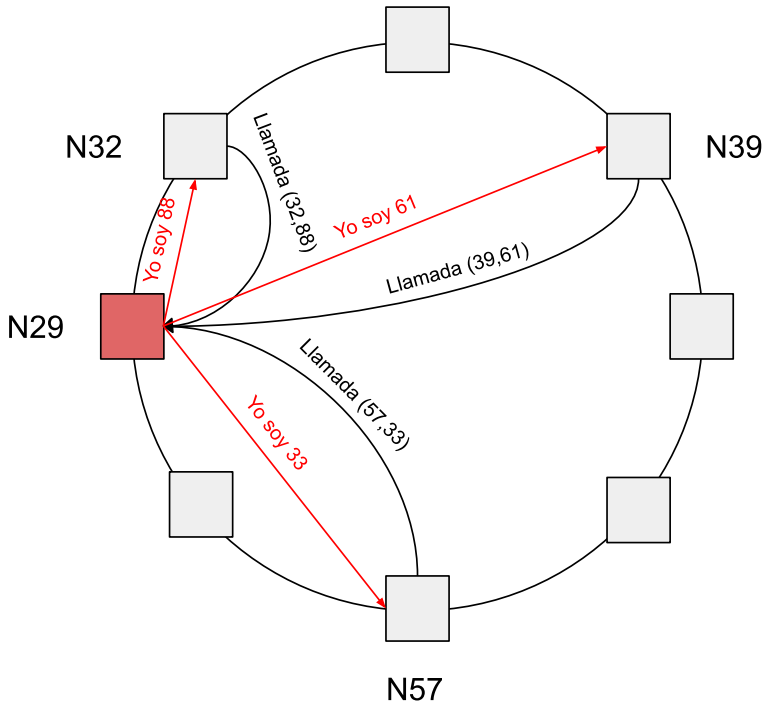
3.1.1. Ataque horizontal

En 2002, un investigador de Microsoft presentó una investigación llamada "The Sybil Attack"[8]. En ella se hablaba de cómo aunque las redes P2P consiguen resistir ataques gracias a la redundancia de datos entre nodos, era posible realizar cierto tipo de ataques haciendo uso de distintas identidades.

Uno de los componentes necesarios a la hora de unirse a la DHT de Bittorrent es un identificador y aunque existen ciertas buenas prácticas a la hora de obtenerlo, no hay ningún requisito real que impida que este identificador sea manipulado a placer para manipular las tablas de enrutado de los demás nodos de la red.



(a) Funcionamiento normal de la red DHT.



(b) Funcionamiento de la red DHT en un ataque horizontal.

Figura 3.2: Diferencia de funcionamiento de la red DHT en condiciones normales y estando bajo ataque.

En la figura 3.2 se muestra como manipulando el ID e introduciendo múltiples identidades en las tablas de enrutado de otros nodos, es posible recibir más llamadas de los demás participantes de la red ya que los nodos vecinos pensarán que el atacante es el nodo al que estaba destinada la llamada o en su defecto alguien cercano a él.

Una de las partes negativas de esta técnica es que no se garantiza de ninguna forma que las llamadas que se reciban contengan información sobre un archivo y por ello muchas de estas no aportarán ningún valor a descubrir nuevo contenido. Pero aún con este inconveniente esta técnica es más que suficiente para que el sistema sea funcional y tenga un rendimiento que satisfaga las necesidades de que el buscador sea rápido.

3.2 Modelo de datos

La arquitectura de los modelos de datos va a ser muy simple ya que para esta versión del proyecto solamente se busca almacenar los archivos encontrados por el buscador. Por lo tanto, los modelos que podemos encontrar son:

- ***Torrent***: albergará la información básica de un archivo torrent.
- ***File***: destinado a almacenar información sobre un archivo perteneciente a un torrent.

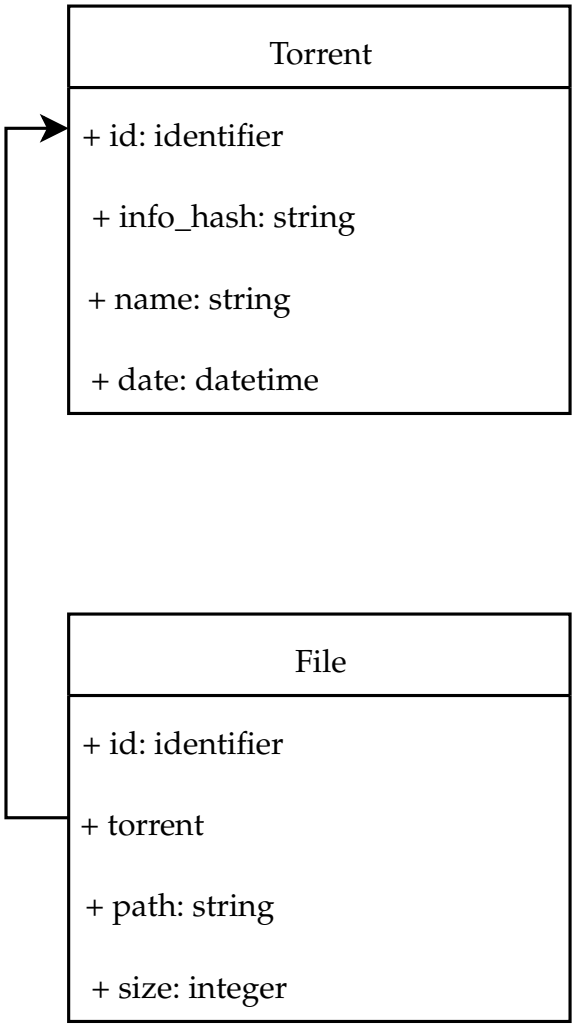


Figura 3.3: Organización del código

3.3 Análisis del marco legal y ético

La legalidad de Bittorrent siempre ha estado abierta a debate dado que en la red no existe ningún tipo de organismo central que regule el contenido que puede ser publicado en ella. Es por esto que existe controversia sobre si Bittorrent como tal es legal o no.

Uno de los principales puntos que se deben tratar en el marco legal respecto a Bittorrent es su uso para la descarga de archivos sujetos a derechos de autor o fuera de la legalidad vigente del país en el que se realiza dicha descarga. Sin embargo, este hecho sería como el culpar a un navegador por acceder a una página que sirva contenido con copyright en lugar de al usuario. Por esto, se puede decir que la tecnología en sí, **sí es legal**.

Además, en lo que respecta a este proyecto se debe de tener en cuenta que en ningún momento se guardan los archivos encontrados, sino los *metadatos* de estos. Por lo que en el hipotético caso de que algún archivo ilegal fuese encontrado por el buscador, serían solamente sus *metadatos* los que se guardarían. Otro matiz importante a recalcar es que la base de datos de este proyecto será mantenida en el sistema del usuario y nunca expuesta al exterior.

El realizar las cosas de esta forma garantiza que el proyecto se mantenga en los márgenes legales ya que en caso de proyectos similares, la justicia ha determinado [9] que estos servicios eran ilegales por almacenar contenido relacionado con los archivos, como podrían ser capturas de un archivo de vídeo.

CAPÍTULO 4

Implementación

En este capítulo se tratará sobre los aspectos de la implementación de los distintos elementos que componen el proyecto, concretamente el **buscador de contenido** y la **interfaz web**. Además se ha dividido la **base de datos** como un componente independiente por la arquitectura de la implementación.

Una de las particularidades que comparten ambas componentes es que se han implementado haciendo uso de un paradigma **asíncrono** [10]. Esta decisión se debe a que el buscador de contenido ha de mantener un gran número de conexiones abiertas de forma concurrente y por lo tanto el uso de este paradigma garantizará que se puedan estar respondiendo a algunas llamadas mientras se esperan a otras. Respecto a la interfaz web también se ha decidido el uso de este paradigma ya que aunque no aportará tanta mejora en el rendimiento garantiza la uniformidad de usar el mismo paradigma a lo largo de todo el proyecto.

4.1 Base de datos

4.1.1. Tecnología y framework

Para la implementación de la base de datos se consideró apropiado utilizar un *framework* [11] que implementase técnicas de mapeo Objeto-Relacional, también conocido como *ORM*. Gracias a esto tanto el tiempo de desarrollo como el nivel de complejidad del código relacionado con la base de datos se pueden ver reducidos notablemente.

Debido a la decisión tomada de que todo el proyecto debe adoptar un paradigma asíncrono, se debe comprobar que el *framework* elegido soporte asincronía.

Con las características técnicas descritas, se ha elaborado un listado para comparar las opciones existentes en el ecosistema de Python.

Framework	Asincronía
Peewee	
ORM	X
Tortoise ORM	X

Después de explorar las alternativas que satisfacían los requisitos expuestos, se decidió optar por *ORM* por ser la más simple de las dos opciones disponibles.

4.1.2. Implementación del modelo de datos

Dado que se ha decidido usar un *ORM* se deberán definir las clases en un archivo y registrar dichos modelos para que el *framework* realice los cambios pertinentes en la base de datos.

```
1  class Torrent(orm.Model):
2      __tablename__ = "torrent"
3      __database__ = database
4      __metadata__ = metadata
5
6      id = orm.Integer(primary_key=True)
7
8      info_hash = orm.String(max_length=40, unique=True, index=True)
9      name = orm.String(max_length=512)
10     date = orm.DateTime()
11
12     def __str__(self):
13         return self.name
```

Figura 4.1: Definición de un modelo en ORM

Para que el framework encuentre dichos modelos y haga los cambios pertinentes en la base de datos, es necesario referenciarlos de alguna forma. Esto se hace en las líneas 3 y 4 de la figura 4.1, con los elementos *database* y *metadata*.

```
1      database = databases.Database(DATABASE_URL)
2      metadata = sqlalchemy.MetaData()
```

Figura 4.2: Declaración de *database* y *metadata*

Por último, se debe inicializar la base de datos, para esto se ha definido una función de ayuda.

```
1  def init() -> None:
2      engine = sqlalchemy.create_engine(str(database.url))
3      metadata.create_all(engine)
```

Figura 4.3: Inicialización de la base de datos

4.2 Buscador de contenido

El buscador de contenido es la parte más importante del proyecto, además de la más complicada. Por ello se ha puesto gran empeño en simplificar su arquitectura.

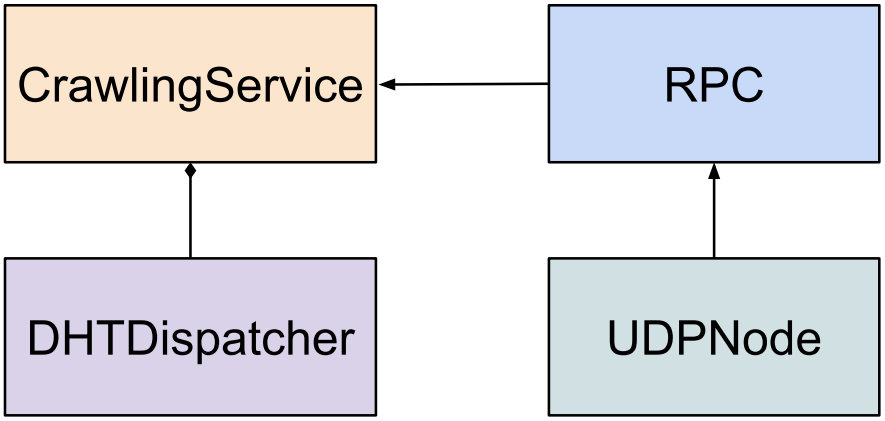


Figura 4.4: Arquitectura del buscador de contenido

A continuación se describe cada elemento de la figura 4.4 de forma individual:

- ***UDPNode***: este elemento será utilizado por el objeto ***RPC***, se trata de una implementación a nivel bajo de lo necesario para poder interactuar a nivel de red con la *DHT*.
- ***RPC***: se encargará de realizar las llamadas pertinentes a la *DHT*, contiene lógica concreta relacionada con esta tecnología.
- ***DHTDispatcher***: clase abstracta que define los métodos necesarios que debe de implementar una clase para interactuar con la *DHT*.
- ***CrawlingService***: clase principal que se encarga de la interacción con la *DHT* y en gestionar las respuestas recibidas.

4.2.1. Implementación del ataque para la obtención de contenido

Como se explicó en el apartado 3.1.1, para la obtención de contenido se va a hacer uso de un *ataque horizontal*.

Para ello, se ha implementado en la clase CrawlingService la clase abstracta *DHTDispatcher*. La clase *DHTDispatcher*, cuenta con tres métodos que son equivalentes a las diferentes tipos de respuesta que puede recibir el nodo.

```
class DHTDispatcher:
    @abstractmethod
    def on_announce_peer(
        self, tid: bytes, nid: bytes, info_hash: bytes, address: Tuple[
            str, int]
    ) -> None:
        pass

    @abstractmethod
    def on_find_node(self, nodes: List[Node]) -> None:
        pass

    @abstractmethod
    def on_get_peers(
        self, info_hash: bytes, tid: bytes, address: Tuple[str, int]
    ) -> None:
        pass
```

Figura 4.5: Código simplificado de *DHTDispatcher*

Estos métodos de respuesta permitirán reaccionar de forma distinta a las diferentes llamadas que reciba el nodo. El método importante de la figura 4.5 en relación con el ataque es *on_find_node*. Este método será llamado cada vez que se encuentre un nuevo nodo en la red.

```
    def on_find_node(self, nodes: List[Node]) -> None:
        if not self.routing_table.is_full:
            nodes = [node for node in nodes if dht_utils.is_valid_node(
                self.node, node)]
            for node in nodes:
                self.routing_table.add(node)
```

Figura 4.6: Implementación de *on_find_node*

Cada vez que esta función es llamada, se pasa por argumentos un listado de los nodos encontrados y se añaden a la tabla de enrutado. Todos esos nuevos nodos añadidos en la tabla, son después utilizados para crear nuevos vecinos realizando el ataque. Para realizar esto se utiliza el código de la figura 4.7.

```
1     def _make_neighbours(self) -> None:
2         for node in self.routing_table.nodes:
3             neighbour_nid = dht_utils.generate_neighbor_nid(self.node.
                  nid, node.nid)
4             self.rpc.find_node(nid=neighbour_nid, address=(node.address
                  , node.port))
```

Figura 4.7: Implementación de *on_find_node*

La función *generate_neighbor_nid*, mostrada en la figura 4.8, es la encargada de falsificar el identificador para hacer que el otro nodo tenga la sensación de cercanía. Para ello, se utilizan los 15 primeros bytes del identificador del nodo remoto y los 5 finales del nodo local.

```
1 def generate_neighbor_nid(local_nid: bytes, neighbour_nid: bytes) ->
      bytes:
2     """
3     Generates a fake node id adding the first 15 bytes of the local
          node and
4     the first 5 bytes of the remote node.
5
6     This makes the remote node believe we are close to it in the DHT.
7     """
8     return neighbour_nid[:15] + local_nid[:5]
```

Figura 4.8: Implementación de *generate_neighbor_nid*

Una vista general del código simplificado de la implementación de la clase *CrawlingService* se puede ver en la figura 4.9.

```python
class CrawlingService(DHTDispatcher):
    def _make_neighbours(self) -> None:
        for node in self.routing_table.nodes:
            neighbour_nid = dht_utils.generate_neighbor_nid(self.node.nid, node.nid)
            self.rpc.find_node(nid=neighbour_nid, address=(node.address, node.port))

    async def _tick_periodically(self) -> None:
        while self._running:
            await asyncio.sleep(self._tick_interval)

            if not self.routing_table.nodes:
                await self._bootstrap()

            self._make_neighbours()
            self.routing_table.nodes.clear()

    def on_announce_peer(
        self, tid: bytes, nid: bytes, info_hash: bytes, address: Tuple[str, int]
    ) -> None:
        self.rpc.respond_announce_peer(
            tid=tid,
            nid=dht_utils.generate_neighbor_nid(self.node.nid, nid),
            address=address,
        )
        self.metadata_fetcher.fetch(info_hash, address)

    def on_find_node(self, nodes: List[Node]) -> None:
        if not self.routing_table.is_full:
            nodes = [node for node in nodes if dht_utils.is_valid_node(self.node, node)]
            for node in nodes:
                self.routing_table.add(node)

    def on_get_peers(
        self, info_hash: bytes, tid: bytes, address: Tuple[str, int]
    ) -> None:
        self.rpc.respond_get_peers(
            tid=tid, info_hash=info_hash, nid=self.node.nid, address=address
        )

    def on_metadata_result(self, info_hash: bytes, metadata: bytes) -> None:
        metadata_decoded = decode(metadata)
        asyncio.ensure_future(
            db_utils.store_metadata(info_hash, metadata_decoded, logger)
        )
    def run(self) -> None:
        asyncio.ensure_future(self._tick_periodically())
```

Figura 4.9: Código simplificado de *CrawlingService*

Obtención de información

Cuando se recibe una llamada del tipo *announce_peer*, existe una posibilidad de que el id que se reciba esté relacionado a un archivo. Por esto, estas llamadas son las que se pasan a una clase llamada *MetadataWorker*. Esta clase es la encargada de intentar descargar los metadatos del archivo.

Este proceso es similar al que realizan los clientes de Bittorrent antes de realizar la descarga de un archivo.

4.2.2. Estructuras

Para la implementación de este componente se ha decidido crear una serie de estructuras que modelan partes fundamentales de la red, tales como los nodos o las tablas de enrutado.

```python
class Node:
    def __init__(self, nid: bytes, address: str, port: int):
        self.nid = nid
        self.address = address
        self.port = port

    @property
    def hex_id(self) -> str:
        return self.nid.hex()

    def __eq__(self, other: object) -> bool:
        if isinstance(other, Node):
            return (
                self.nid == other.nid
                and self.address == other.address
                and self.port == other.port
            )
        return False

    def __repr__(self) -> str:
        """
        Represents the node in an hex format using the nid.
        """
        return self.hex_id

    @classmethod
    def create_random(cls, address: str, port: int) -> Node:
        """
        Creates a random node with the desired address and port.
        """
        nid = Node.generate_random_id()
        return cls(nid, address, port)

    @staticmethod
    def generate_random_id() -> bytes:
        """
        Generates a random node id which consists of 20 bytes.
        """
        return os.urandom(20)
```

Figura 4.10: Estructura para el nodo

```python
class RoutingTable:
    def __init__(self, max_size: int):
        self.max_size = max_size
        self.nodes: List[Node] = []

    @property
    def is_full(self) -> bool:
        return len(self.nodes) > self.max_size

    def add(self, node: Node) -> bool:
        if not self.is_full:
            self.nodes.append(node)
        return not self.is_full
```

Figura 4.11: Estructura la tabla de enrutado

4.3 Interfaz web

4.3.1. Tecnología y framework

Dado que la interfaz web debe de tener funcionalidad de búsqueda, este componente debe de estar funcionando gracias a un servidor web. El abanico de *frameworks* web disponibles en el ecosistema de Python es muy abundante. Es por esto que se tuvo que realizar un análisis de las opciones que existen actualmente. El único requisito impuesto debía ser que soportase asincronía.

Framework	Asincronía
Django	
Flask	
Falcon	X
Sanic	X
Bocadillo	X

Dos de los grandes *frameworks* del ecosistema Python quedaron rápidamente descartados debido a la falta del soporte de programación asíncrona. De las opciones restantes, *Falcon* tuvo que ser descartado también debido a que está destinado a la programación de APIs [12]. De las dos opciones restantes, *Sanic* y *Bocadillo*, se decidió optar por *Bocadillo* debido a que parece tener un futuro prometedor dentro del ecosistema.

Respecto a la maquetación se decidió usar un *framework* que permitiese desarrollar la interfaz de forma rápida y sencilla, ya que sin este, el tiempo de implementación de la interfaz se habría extendido de manera notable.

En la actualidad existen multitud de *frameworks*, es por esto que se tuvo que realizar un análisis de las opciones existentes. Dado que en el apartado de la interfaz web no existe ningún requisito, la decisión tomada fue totalmente subjetiva.

Las opciones que se analizaron fueron (ordenadas por popularidad): *Bootstrap*, *Semantic-UI*, *Skeleton* y *Bulma*. Se decidió optar por *Bulma*.

4.3.2. Implementación de las vistas

Esta componente está formada por dos vistas principalmente: Página principal de búsqueda y resultados.

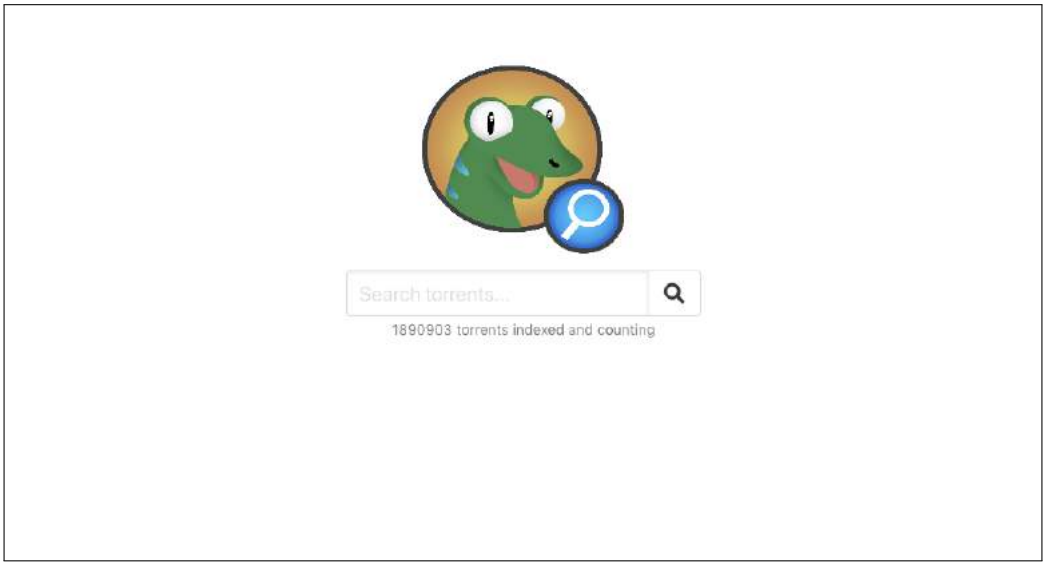


Figura 4.12: Página principal

Tal y como se puede ver en la figura 4.12, esta vista cuenta con un campo para introducir el nombre del archivo así como con el número de torrents que existen en base de datos actualmente.

```
@app.route("/")
async def index(req, res):
    count = await total_torrents_count()
    res.html = await templates.render("index.html", count=count)
```

Figura 4.13: Código de la vista principal

Gracias a haber utilizado *ORM*, el código para obtener el número de *torrents* que existen en la base de datos queda muy simplificado tal y como se muestra en la figura 4.14.

```
async def total_torrents_count() -> int:
    count = await Torrent.objects.count()
    return count
```

Figura 4.14: Código para contar los *torrents* en base de datos

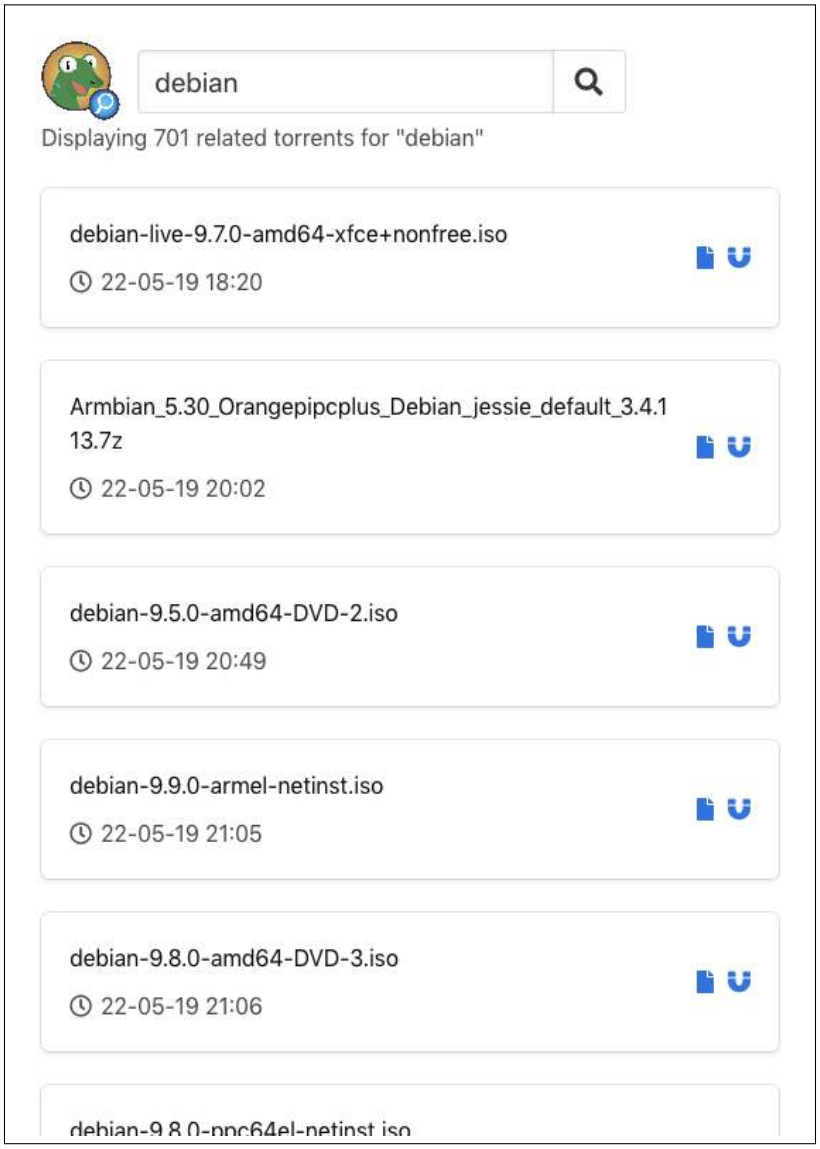


Figura 4.15: Página de resultados

La figura 4.15 representa un ejemplo de búsqueda.

Además del nombre del *torrent* y la fecha en la que se encontró el resultado, se puede interactuar con dos botones que permiten añadir el archivo para su descarga al cliente de Bittorrent que tenga instalado el usuario. El primero de ellos hace uso de una página externa para generar un archivo *.torrent*, mientras que el segundo hace uso del identificador del archivo la DHT.

```
1    <a href="magnet:?xt=urn:btih:{{ torrent.info_hash }}&dn={{ torrent.
         name }}">
2    <i class="fas fa-magnet"></i>
3    </a>
```

Figura 4.16: Código para lanzar la descarga del archivo en el cliente del usuario

CAPÍTULO 5

Conclusiones

5.1 Resultados y pruebas

Dado que el proyecto debe de ser un sistema autónomo, se dejó funcionando durante un largo periodo de tiempo para comprobar si era capaz de funcionar sin errores y sin la necesidad de intervención humana.

En el momento de escribir este trabajo, el proyecto lleva en funcionamiento 1 mes y 15 días. Durante ese tiempo ha sido capaz de descubrir un total de 1.927.317 archivos, casi 2 millones de torrents. Para poner en contexto este número, servicios que llevan en funcionamiento durante años tienen en base de datos entorno a unos 20 millones [14] de torrents, con el añadido de que muchos de estos servicios tienen ayuda de sus usuarios.

Con estos datos se puede decir que el rendimiento del proyecto es muy prometedor si se compara con las opciones privadas que existen en el mercado.

5.2 Trabajo futuro

El objetivo al inicio del proyecto era la creación de un producto mínimo viable [15], por ello, muchas funcionalidades fueron descartadas por no entrar dentro de las funcionalidades básicas. Es por esto, que para futuras iteraciones del proyecto se han considerado las siguientes mejoras:

- ***Número de pares conectados descargando un torrent***: Esta funcionalidad permitiría ordenar por popularidad de los torrents, siendo los más descargados en este momento los más populares.
- ***Panel de analíticas***: Permitirá ver el número de torrents añadidos a lo largo del tiempo con estadísticas.
- ***Sistema de autenticación***: Aunque el objetivo del proyecto es que sea usado en el sistema local de un usuario, podría existir la necesidad de que ese usuario quisiera acceder al buscador desde fuera de la red. Por ello, un sistema de autenticación garantizaría que solo ese usuario pueda acceder.

Bibliografía

[1] Wiki theory *Recuperado el 20 de julio de 2018* https://wiki.theory.org/

[2] Metadatos, definición y características *Recuperado el 24 de junio de 2019* https://www.powerdata.es/metadatos

[3] Application Usage & Threat Report. *Recuperado el 13 de junio de 2019* https://web.archive.org/web/20131031153132/http://researchcenter.paloaltonetworks.com/app-usage-risk-report-visualization/

[4] Napster history. *Recuperado el 13 de junio de 2019* https://history-computer.com/Internet/Conquering/Napster.html

[5] Shawn Fanning. *Recuperado el 14 de junio de 2019* https://sistemas.com/11659.php

[6] Top Torrent Trackers Now Handle Up to 56 Million Peers – Each. *Recuperado el 14 de junio de 2019* https://torrentfreak.com/top-torrent-trackers-now-handle-up-to-56-million-peers-each-150531/

[7] DHT Infohash Indexing *Recuperado el 16 de junio de 2019* http://www.bittorrent.org/beps/bep_0051.html

[8] The Sybil attack *Recuperado el 16 de junio de 2019* https://www.microsoft.com/en-us/research/wp-content/uploads/2002/01/IPTPS2002.pdf

[9] European court of justice rules Pirate Bay is infringing copyright *Recuperado el 19 de junio de 2019* https://www.gu.com/technology/2017/jun/15/pirate-bay-european-court-of-justice-rules-infringing-copyright-torrent-sites

[10] What Is Asynchronous Programming? *Recuperado el 21 de junio de 2019* https://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html

[11] Software frameworks *Recuperado el 21 de junio de 2019* https://www.techopedia.com/definition/14384/software-framework

[12] What is an API? In English, please. *Recuperado el 21 de junio de 2019* https://www.freecodecamp.org/news/what-is-an-api-in-english-please-b880a3214a82/

[13] Templating in Python *Recuperado el 23 de junio de 2019* https://wiki.python.org/moin/Templating

[14] Torrent sites *Recuperado el 26 de junio de 2019* https://twitgoo.com/best-torrent-sites/

[15] Producto mínimo viable *Recuperado el 26 de junio de 2019* https://www.emprenderalia.com/que-es-el-mvp-producto-viable-minimo/