Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Implementación de un Buscador Web Auto alojado

Implementing a self-hosted search engine

Daniel Darias Sánchez

La Laguna, 9 de septiembre de 2020

D. **Juan Carlos Pérez Darias**, con N.I.F. 45.441.625-L profesor Titular de Universidad adscrito al Departamento de Física de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Implementación de un Buscador Web Auto alojado”

ha sido realizada bajo su dirección por D. **Daniel Darias Sánchez**,

con N.I.F. 54.110.629-R.

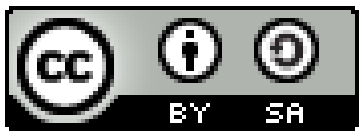
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de septiembre de 2020

Agradecimientos

Agradecerle a mi familia y a mi pareja por el apoyo incondicional durante el transcurso de mis estudios de Ingeniería Informática así como la comprensión tras cada uno de los errores cometidos durante este periodo.

A mi tutor, Juan Carlos Pérez Darias, por la fe que tuve en mi y en mi proyecto, eligiendo tutorizarme, en primer lugar tras presentarle apenas vaga idea de lo que quería conseguir y manteniendo posteriormente su decisión a pesar de mis contratiempos con las asignaturas.

Licencia



Resumen

El objetivo de este trabajo ha sido la investigación acerca de la problemática, soluciones existentes y tecnologías implicadas en el funcionamiento de los motores de búsqueda en internet, así como la creación de una herramienta prototipo capaz de funcionar como solución alternativa y auto alojada a las ya existentes.

Además, el trabajo ha sido ideado y desarrollado cumpliendo con dos condiciones adicionales establecida por el alumno. En primer lugar, el prototipo debe funcionar satisfactoriamente en computadores con gran escasez de recursos, ya que idealmente, será ejecutado en "computadoras de bolsillo" y bajo consumo como la Raspberry pi. Por otro lado, enfocando este proyecto como una oportunidad para enfrentar problemas nuevos y establecer relaciones entre conocimientos que, si bien han sido adquiridos en el marco de lo estudiado en el Grado de Ingeniería Informática, quizá no se haya profundizado en las conexiones existentes entre ellos. Además, se propone la elaboración del prototipo en su conjunto, intentando evitar el uso de librerías de alto nivel capaces de lidiar con grandes porciones de esta labor. Así, se prefiere el desarrollo de soluciones propias potencialmente subóptimas en pro del posible aumento de conocimiento por parte del alumno que conlleva el deber enfrentar la variedad de dificultades encontradas individualmente.

*El resultado del trabajo es el de una herramienta funcional y capaz escrita en **C++**, con una arquitectura modular, asíncrona y fuertemente centrada en el aprovechamiento de las capacidades multihilo de los procesadores actuales, dependiente únicamente de las librerías **libcurl y leveldb** y que reutiliza satisfactoriamente otras herramientas creadas por el alumno durante el transcurso de sus estudios en este grado. Este ha sido licenciado, cumpliéndose las restricciones de las dependencias, como software libre bajo una licencia GPL-v3.*

Palabras clave: Araña Web, Buscador Web, Multihilo, Paralelismo, Asíncrono

Abstract

The goal of this project has been the investigation towards the problematic, solutions and implied technologies on the inner mechanics of search engines, as well as the creation of a protoype tool capable of working as an alternative and self hosted solution from the existent ones.

Besides, the work has been created and developed under two additional conditions established by the student. First, the prototype must work succesfully on low-end computers, since its main goal is to be executed on "pocket pcs" like the Raspberry Pi. Secondly, I wanted to gain more knowledge about some areas that were studied during the career but I felt I had poorly conected relations between them. Additionally, I wanted do the whole elaboration of the prototype, avoiding the use of high level libraries. I prefered to come up with my own solutions, which may be suboptimal, with the goal of the increase of my knowledge occured by solving the different problems and difficulties by myself.

The result is a functional and capable tool, written in C++, with a modular architecture, asyncronous and strongly focused on the exploitation of the multihtreaded capacities of the current CPUs, that depends only on the libcurl & leveldb libraries, and that succesfully recycle other tools created by the student during his career. The project has been licensed as libre software under a GPL-v3 license.

Keywords: Crawler, SearchEngine, Multithreaded, Paralelism, Asyncronous

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

1.1 Motivación

En el campo de los buscadores web de propósito general, existe un abanico relativamente pequeño de posibilidades entre las que elegir, más aún si los clasificamos en función de su tecnología y no del cupo de mercado. Por ejemplo, si tratásemos de enumerar algunos de los existentes, probablemente comenzaríamos por *Google*, *Bing* y *Yahoo* como líderes indiscutibles por cantidad de usuarios en poblaciones occidentales. Como contraparte oriental hablaríamos de *Yandex* para Rusia o *Baidu* en China. En un segundo escalón podrían aparecer alternativas menores que tratan de competir mediante algún valor añadido de tipo ético como es el caso de *Duckduckgo*, asegurando el respeto por la privacidad, o *Ecosia*, con reclamos ecologistas. No obstante, es importante señalar que ambos utilizan de forma parcial o total infraestructura de los anteriormente mencionados[i] [ii], lo cual les otorga cierta similitud con un tercer grupo, compuesto por los meta buscadores e intermediarios con los buscadores tradicionales y el usuario. Este último grupo comprende servicios que utilizan directa y exclusivamente los servicios de los grandes buscadores pero situándose en medio para por ejemplo buscar en todos ellos a la vez o tratar de *anonimizar* las búsquedas. Cómo se ve, en mayor o menor medida y a riesgo de realizar afirmaciones sin conocer los entresijos internos de cada uno de los servicios, todos se basan en el uso de servidores privados a los que se les envía una consulta y se devuelven los resultados. Esto presenta, más allá de aquellos inherentes a la arquitectura cliente - servidor, y sin tratar esta de ser una lista extensiva, los siguientes inconvenientes de tipo ético:

• Censura o sesgos intencionados en los resultados de búsqueda. Pudiendo incluir este caso sesgos “menores” como cambiar el posicionamiento de resultados de servicios propios o afines para favorecer su uso (véase el caso el de la multa europea a Google [iii]), el bloqueo de resultados en base a reclamaciones de derechos de autor basadas en legislaciones de otros países[iv], aplicaciones laxas del derecho al olvido [v], o la omisión intencionada de resultados sobre sucesos históricos (por ejemplo, *Baidu* y la plaza de Tiananmen [vi]).

• Rastreo y violación de la privacidad de los usuarios. Dada la afluencia masiva de peticiones y el uso de *fingerprinting*, estas entidades manejan ingentes cantidades de datos sobre la población que sus servicios, quedando en manos de los usuarios la

posibilidad de confiar ciegamente en el buen uso de sus datos dada la naturaleza privada y centralizada de las tecnologías empleadas. Escándalos como el reciente caso de Cambridge Analytica [vii], si bien no está relacionado estrictamente con buscadores, no hacen sino recordarnos que el comercio con la información personal de millones de usuarios es una práctica a la orden del día y un modelo de negocio real y lucrativo.

• Cambios arbitrarios en los acuerdos de licencia. La mera existencia de estos acuerdos ya podría levantar suspicacias, ya que dado el servicio que ofrecen, podría compararse con aceptar un acuerdo de licencia por utilizar una guía telefónica, un mapa de una ciudad o una enciclopedia. Sin embargo, la dependencia existente en la actualidad de estas tecnologías para la sociedad ligada a la complejidad de estos documentos legales resulta en que a efectos prácticos estés obligado a aceptarlas ciegamente, so pena de condenarte a un ostracismo tecnológico.

Ignorando el debate técnico sobre si es posible conseguir la misma calidad de servicio sin incurrir en estas técnicas, se perfila como alternativa obvia el uso de sistemas en los que el propio usuario tenga, si no total, un mayor control sobre la infraestructura del buscador y los datos recopilados. A este respecto, bajo me juicio, la opción ideal es aquella que cumpla dos requisitos fundamentales.

- Poder ser alojada por el propio usuario.
- Ser software, como mínimo, de código abierto, preferiblemente libre.

Es fundamental que se cumplan ambas condiciones, pues si el software fuese de código abierto pero los servidores estuviesen alojados por una entidad externa, no existiría garantía de que los datos no fuesen recopilados y utilizados para su comercialización. Más aún, sería verdaderamente difícil asegurar que el software ejecutado fuese exactamente el que dice ser. Si por el contrario, las tecnologías se ejecutasen en máquinas propiedad del usuario, pero no se pudiese estudiar su código, no existen garantías de que no estén ocurriendo problemas como los sesgos comentados con anterioridad.

No obstante, el volumen de datos con los que debe trabajar este tipo de tecnologías es de una magnitud tal que se vuelve imposible para un usuario mantener su propio sistema, tanto por cantidad de almacenamiento requerida como por el procesamiento que se debe hacer. Una solución de compromiso pasa por el uso de un sistema de cómputo distribuido, donde cada usuario del buscador participe como nodo de una red mayor, almacenando y tratando un subconjunto de la información que el sistema global es capaz de manejar.

Es en este contexto en el que surge la idea de desarrollar como Trabajo de Fin de Grado un prototipo de buscador web capaz de cumplir en la medida de lo posible los requisitos establecidos con anterioridad, limitándonos al primer caso de sistema no

distribuido debido a la necesidad de acotar el trabajo, pero con vistas a una posible expansión en esta dirección.

1.2 Objetivos

El objetivo del trabajo será pues, en líneas generales, familiarizarme con las tecnologías necesarias para la creación de una herramienta capaz de servir como buscador web no especializado y realizar una implementación funcional basándome en lo aprendido. Se hace especial hincapié en la comprensión de cada uno de los problemas que intervienen en la tarea bajo la premisa de que tratar de resolverlos de forma autónoma recurriendo al menor número de librerías externas posibles afianzará las relaciones entre conceptos aprendidos durante los estudios que he cursado y me preparará para un futuro en el que potencialmente deba lidiar con problemas complejos de forma autónoma. Según lo dispuesto en el anteproyecto, el prototipo habrá de cumplir, al menos, las siguientes características:

- Indexar portales de internet.
- Extraer información relevante de dichos portales.
- Clasificar adecuadamente en base a la información extraída.
- Establecer relaciones entre los datos clasificados de diferentes portales.
- Guardar localmente los datos necesarios para la recuperación de la información de una forma óptima.
- Permitir la búsqueda en el lenguaje más natural posible entre esos datos.

1.3 Fases de desarrollo

Pormenorizando los puntos necesarios para lograr la ejecución exitosa del proyecto, podemos hablar de los siguientes tareas, cada una de las cuales surge ante la necesidad de responder un conjunto de preguntas asociado:

- Definición y acotamiento inicial. ¿Qué entendemos por un buscador de internet? ¿Cuántas funcionalidades potencialmente achacables a un buscador de páginas web vamos a querer cubrir en este proyecto? ¿Qué prioridad le asignamos a cada una de ellas?
- Familiarización con soluciones existentes. ¿Cómo están conformados los buscadores que existen en la actualidad? ¿Qué técnicas existen para resolver las tareas implicadas? ¿Qué herramientas encontramos que puedan cubrir estas necesidades?
- Adaptación a los requisitos. ¿Cómo debemos abordar la tarea para que cumpla

la condición de poder ser alojado en una máquina del usuario? ¿Qué tecnologías emplearemos?

- Creación de un prototipo de forma incremental. ¿Cómo afrontaremos los problemas surgidos? ¿A qué aspectos habremos de renunciar en pro de conseguir ciertos hitos? ¿De qué forma iteramos y en qué grado para añadir funcionalidades?
- Documentación del software generado.

1.4 Estructura de la memoria

Basándonos en las líneas expuestas en el apartado previo, el documento estará estructurado en siete capítulos, de extensión variable en función de la relevancia y granularidad requerida para cada explicación.

Subseguido a este capítulo introductorio al que da conclusión este apartado, encontramos un conjunto de definiciones más estrictas al término *buscador* que se ha empleado hasta el momento de forma laxa, así como un listado de desempeños deseados en mayor detalle.

A continuación, se da paso a la recopilación del estado del arte y la pormenorización de las necesidades específicas del problema a abordar. Se establecen aquí definiciones de componentes, comportamientos y requerimiento a fin de tener una visión certera y de mayor calado de la tarea.

Seguidamente, se realiza el primer diseño de componentes ajustado a todos los requisitos que ya habrán sido estipulados. Se trata de un paso de vital importancia, so pena de comenzar un desarrollo abocado al fracaso por culpa de un planteamiento erróneo. Detallaremos aquí información sobre la arquitectura utilizada.

Consecuentemente, el siguiente capítulo es dedicado al desarrollo de la herramienta en sí. Se trata naturalmente del segmento de mayor extensión y representa el fin último de este trabajo.

Aproximamos el final de esta memoria con un capítulo en el que mostraremos pruebas del comportamiento y rendimiento del prototipo para finalizar con una sección que disertará sobre las líneas de desarrollo futuro, reflexiones acerca de lo aprendido en la consecución del proyecto y valoración de los hitos conseguidos.

Capítulo 2
Definiciones y alcance

2.1 ¿Qué es un buscador web?

Atendiendo a la definición dada por la wikipedia [viii], un buscador web, o de forma más correcta y como será denominado de ahora en adelante, motor de búsqueda web, es

"*un sistema informático que busca archivos almacenados en servidores web gracias a su araña web. [...] Las búsquedas se hacen con palabras clave o con árboles jerárquicos por temas; el resultado de la búsqueda «Página de resultados del buscador» es un listado de direcciones web en los que se mencionan temas relacionados con las palabras clave buscadas.*"

La importancia aquí reside en la palabra archivo, pues según esta definición, un motor de búsqueda es capaz de manejar diferentes tipos de datos, como por ejemplo, -tal y como ocurre en nuestra experiencia cotidiana- contenido multimedia como imágenes o vídeos.

Además, de forma general, las herramientas de uso masivo mencionadas en el capítulo introductorio ofrecen funcionalidades accesorias como la predicción de términos de búsqueda o la devolución de segmentos cortos de los resultados a fin de facilitar el cribado por parte del usuario. Es fundamental también disociar el concepto de *página web de un buscador* del motor de búsqueda en sí. Puesto que este TFG persigue la creación de lo segundo, la obtención de resultados desde un portal web accesible desde el el navegador es tan solo deseable, pero no prioritario.

2.2 Alcance

Para nuestra labor, establecemos los siguientes requerimientos indispensables:

- La herramienta debe ser capaz de aportar resultados relevantes para la búsqueda del usuario. Independientemente de cómo se logre esta labor, el aspecto definitorio del propio término *motor de búsqueda* debe ser cumplido.
- La herramienta trabajará sobre texto plano en documentos html. El posible manejo de contenido multimedia es totalmente secundario.

- La herramienta debe permitir la persistencia de datos. Poca utilidad tendría este sistema si dependiese exclusivamente de la cantidad de memoria RAM y además no pudiese ser apagada nunca. Esta persistencia preferiblemente se hará de una forma que aproveche el espacio lo mejor posible.
- La herramienta debe aprovechar los recursos de la máquina en la mayor medida posible. Hasta ahora hemos incidido en el punto de la capacidad de ser alojada y ejecutada en máquinas de los propios usuarios, sin embargo, no hemos tratado las implicaciones en profundidad. Este problema podríamos abordarlo de distintas formas. Por una lado, quizá podríamos tener un sistema con un uso extremadamente bajo de recursos que funcionase en un ordenador personal, tratando de coexistir con el resto de procesos lanzados por el usuario. Se trataría de un sistema válido pero potencialmente poco eficaz, pues en pro de no acaparar tiempo de cómputo tendría un estrecho margen para realizar sus cálculos. La aproximación favorita es la de que la herramienta pueda funcionar de forma aceptable en micro ordenadores como la *Raspberry pi*, bajo la premisa de que se trata de un aparato lo suficientemente barato para que se considere despreciable su coste en un hipotético uso continuado y que podría formar parte de esa posible futura red distribuida. Así, se prefiere un modelo en la que el software trate de monopolizar y aprovechar de forma efectiva las capacidades del dispositivo en el que se ejecute.

De forma adicional, se favorece la adición de las siguientes funcionalidades:

- Portal web para un acceso conveniente.
- Predicción de términos relacionados.
- Corrección de palabras mal formadas.
- Manejo de ficheros no html.
- Acercamiento al uso de lenguaje natural.
- Inicio de arquitectura distribuida.

Capítulo 3
Estado del arte

En una primera fase de recopilación de información, encontré la existencia del buscador Yacy [ix], herramienta que cumple con lo dispuesto en mis requerimientos al tratarse de Software Libre licenciado como General Public License en su versión 2 y basado efectivamente en una arquitectura *peer 2 peer.* El sistema además, permite realizar consultas incluso sin formar parte activa de (contribuir a) la red, es decir, no es necesario almacenar parte del índice de webs ni servirlas a otros usuarios para poder realizar una búsqueda.

Ignorando esta arquitectura descentralizada, pues como se ha dicho, si bien es deseable, se encuentra fuera del ámbito que pretende abarcar este trabajo, encontramos una herramienta que ofrece resultados francamente pobres pero que sirve como referencia con el que comparar el prototipo a desarrollar.

A partir de su documentación y otras fuentes mencionadas en la bibliografía se extrae el conjunto mínimo de componentes a desarrollar, los cuales son descritos a continuación.

3.1 Araña Web

También conocido como ***crawler***, se trata de una pieza cuyo cometido es, a partir de una semilla inicial de portales web, visitar el mayor número de páginas posibles y envíar su contenido al ***indexador*** para que puedan ser archivadas. Su forma de "desplazarse" por el espacio de contenidos es a través de los enlaces encontrados en un portal determinado. De esta forma, suponiendo que nuestro crawler se encuentre en un documento html "A" que contenga enlaces a "B" y "C", el componente paraseará y encontrará dichas direcciones y las almacenará en una estructura de datos, como por ejemplo una cola, cuyo primer elemento será el siguiente portal a visitar. Eventualmente, la cabeza de la cola será "B" o "C" y estas serán visitadas.

Este esquema sencillo precisa no obstante de una serie de restricciones y políticas que lo complementen a fin de hacerlo eficente y/o respetuoso con los estándares y elementos de la web.

3.1.1 Criterios de selección

Continuando con el supuesto de uso de una cola explicado previamente, es fácil percatarse de que aquellos enlaces que sean detectados por el ***parser*** en primer lugar serán los primeros en visitarse. Esto no tiene por qué ser una buena política puesto que, por ejemplo, en un momento dado podría interesarnos visitar muchos portales en "poca profundidad" en vez de rastrear cada una de las subpáginas de un dominio concreto exhaustivamente. Distinguimos en función de la prioridad de explorar en profundidad o en anchura entre rastreadores *verticales* y *horizontales.*

De igual forma, únicamente el uso de una cola no basta para garantizar un desempeño adecuado; bastan dos páginas que se enlacen mutuamente para que la araña quede atrapada en un bucle. Se debe agregar por tanto una memoria al sistema que le permita desechar un enlace a una página ya visitada. Esto sin embargo, introduce nuevos problemas en la ecuación. ¿Cómo determinamos que efectivamente dos enlaces apuntan al mismo sitio? Existen diversos casos en los que dos URL pueden ser sintácticamente diferentes pero cuyo contenido apunta a un mismo resultado: *www.dominio.com*, *www.DoMiNio.com*, *www.dominio.com/* y *www.example.com/index.html* son todos hipervínculos válidos con idéntico destino.

Es por tanto que debemos encontrar una forma de identificar unívocamente los portales que visitamos y conseguir discriminar los enlaces idénticos. Como respuesta a este problema encontramos la **normalización de url**.

3.1.2 Normalización de URL

Esta técnica consiste en la modificación del texto original de una dirección en base a determinados criterios que de como resultado en la mayor cantidad de casos posibles un string determinado, el de una url "simplificada" y sin redundancia. Para los casos anteriores, podríamos por ejemplo determinar las siguientes reglas:

• Reescribir todos los caracteres alfabéticos como minúsculas.

• Eliminar el caracter '/' si este aparece el último.

• Eliminar "/index.hml" siempre, pues probablemente una url con solo el nombre de domino redirigirá a él.

Estas reglas nos permiten no solo detectar estos casos sino también algunos más exóticos como por ejemplo www.dominio.com//index.html//, resultando todos los enlaces mostrados siempre en www.dominio.com. Sin embargo, la cantidad de combinaciones posibles y casos particulares juega también en contra y quizá exista un dominio para el que la tercera regla no deba aplicarse. Se distinguen así tres subgrupos de reglas:

- **Preservan la semántica siempre:**
  - Cambios de mayúsculas y minúsculas
  - Borrado del puerto si lo hubiera

◦ Decodificación de caracteres especiales

- **Preservan la semántica frecuentemente:**
  - ◦ Borrado de caracteres ‘.’ y “..”
- **Cambian la semántica:**
  - ◦ Borrado de “/index.html”
  - ◦ Reemplazo del nombre de dominio por la dirección IP al que apunta.
  - ◦ Borrado de caracteres ‘/’ duplicados

A este respecto debemos añadir también la problemática del uso de URLs dinámicas cuyo contenido podría deja de existir una vez indexado.

Por último, se debe prestar atención a aspectos como en qué lugar de la jerarquía html aparecen los enlaces y con qué propiedades. Podríamos encontrarnos ante una zona no renderizada por el navegador que contenga un gran número de enlaces para tratar de interferir en la labor de la araña.

3.1.3 Políticas de revisita

Si bien en el apartado anterior hemos incidido mucho en la necesidad de no re introducir en la cola portales ya visitados, esto ha sido para evitar que el crawler “de vueltas” y “pierda el tiempo”, pero existen otras razones por las que si es conveniente visitar un mismo sitio dos veces.

La gran mayoría del contenido web no permanece estático en el tiempo, por lo que es fundamental que el índice del buscador permanezca lo más actualizado posible. Esto entra en conflicto directo con la cantidad de páginas que estén indexadas. A mayor revisita de portales menor tiempo para la exploración de nuevo contenido a indexar, el cual en el caso de descubrirse pasaría a su vez a ser contenido sensible de actualización. Así, si no se limita la cantidad de contenido nuevo, es evidente que las revisitas se harán con cada vez menor frecuencia. Por otro lado, y si bien es legítimo suponer como dinámico el contenido de la vasta mayoría de internet, no todas las páginas lo hacen con las misma frecuencia o tienen contenidos con la misma relevancia. Surgen así dos estrategias:

• Revisita uniforme: Trata a todos los portales por igual tratando de lograr que el “valor de actualidad” del conjunto sea el máximo posible. Media máxima y varianza mínima.

• Revisita proporcional: Trata de averiguar la importancia y/o frecuencia de actualización de un portal y adaptarse a ella.

3.1.4 Educación

Dado que la araña es capaz de rastrear en profundidad un determinado dominio, es ético respetar ciertas directrices que rijan cómo ha de comportarse allí. Por un lado, y en lo concerniente al rastreo vertical y las políticas de revisita, se ha de tener en cuenta las limitaciones del servidor. Realizar numerosas solicitudes en muy poco tiempo puede resultar en la saturación del mismo, algo evidentemente poco decoroso. Por otra parte, debe tenerse presente y seguir las directrices en caso de existir del fichero **robots.txt**, el cual especifica determinados directorios que no deben de ser rastreados. Por último, la araña debe identificarse a si misma a fin de que los administradores de un determinado dominio puedan detectar comportamientos inadecuados o en definitiva, saber que están siendo escaneados.

3.2 Índice

Su función a es la de almacenar aquella información *útil* contenida en las páginas visitadas por el crawler en una estructura de datos que facilite y acelere la búsqueda. Se distinguen dos tareas para llevar a cabo este propósito, cada un asignada a un componente.

3.2.1 Parser

En primer lugar, se debe extraer la información relevante de los documentos suministrados por el crawler. La forma en que esta tarea se ha de realizar difiere en función del tipo de documento tratado. Suponiendo que el crawler nos sirve exclusivamente de documentos html, se deben afrontar los siguientes retos:

- **Lenguaje y tokenización**: Conseguir extraer palabras individuales del documento varía en dificultad en función del lenguaje elegido. El uso de caracteres *unicode* será descartado para el prototipo en favor de únicamente símbolos ASCII en idioma inglés.
- **Detección de etiquetas**: No todos los posibles tokens del documento tienen la misma relevancia (o lo son en absoluto). El parser debe ser capaz de detectar qué elementos de la jerarquía contienen información útil y cuáles despreciar. Los textos de botones o elementos de formularios probablemente no debieran ser registrados. Este proceso se solapa en parte con la detección de hipervínculos en la propia araña. Aparecen así dos arquitecturas, una en la que cada elemento hace uso de su propio parser diferenciado y otra en la que la se realiza una comunicación bidireccional **araña** – **indexador** en la que la primera alimenta al segundo con documentos y este a su vez le devuelve enlaces.

3.2.2 Almacenamiento

Una vez tenemos la información *tokenizada*, procedemos a almacenarla en una estructura de datos optimizada para la búsqueda. Las opciones barajadas son:

- **Árbol de sufijos (trie)** [x]: Se trata de una estructura en árbol que permite la

búsqueda tanto de la cadena exacta como de subcadenas en un tiempo de O(n). Presenta el problema de ser conceptualmente más complejo que el índice invertido. Presenta la ventaja de poder ser usado para la corrección automática de sintaxis.

- **Array de sufijos** [xi]: Similar a la estructura anterior pero utiliza menor cantidad de memoria y soporta compresión.
- **Índice invertido:** Similar a una tabla hash donde las claves son palabras (tokens) y los valores aquellos documentos donde estas aparecen. Puede ser "decorado" añadiendo más información como el lugar exacto donde aparecen, el número de veces, etc. Por su sencillez a la hora de la implementación y su arquitectura naturalmente separable (varias tablas que contengan subconjuntos del total de tokens encontrados) resulta un candidato ideal. Los resultados se encuentran en O(n).

3.3 Buscador

Dado que nos hemos preocupado por tener los datos estructurados de una forma que facilite la búsqueda, este proceso podría limitarse simplemente a la consulta de los términos solicitados. No obstante, esto nos devuelve un conjunto de documentos no ordenado, delegando en el usuario la tarea de clasificarlo en función de su relevancia. Por esta razón, se deben emplear técnicas adicionales que establezcan una jerarquía de relevancia que ajuste los potenciales resultados, en la medida de lo posible, a lo que el usuario desea encontrar.

Algunas de las técnicas valoradas son:

- **Uso de matrices documento / término**: Matriz de probabilidad que relaciona la cantidad de veces que aparece un término en un documento con el número de tokens existente. Se utiliza como forma de establecer cuál es la importancia de dicho término en el documento. Si la estructura de almacenamiento elegida es la de índice invertido, esta información puede ser introducida directamente en ella. Es importante remarcar que resulta muy sensible a las “trampas” para *crawlers* mencionadas en el aparatado 888.
- **Uso de un N-GRAMA** [xii]: Aproximación de la regla de la cadena utilizada en inferencia bayesiana que permite predecir el siguiente elemento en una secuencia en base a los n elementos anteriores. Puede ser usado tanto para expandir el radio de búsqueda (si los resultados ofrecidos para una consulta son pocos o se estiman de baja relevancia, podemos tratar de deducir qué más cosas habría buscado el usuario si hubiese seguido escribiendo, cada elemento de la secuencia sería una palabra) como para auto completado en la interfaz de búsqueda de la herramienta y corrección de sintaxis (usando en este caso caracteres como elementos de la secuencia).
- **Algoritmo de ordenación de resultados** (PageRank y similares): Tratan de establecer un orden de importancia de cada portal en función de cuántos

enlaces desde otos portales apuntan a este, ponderando cada uno por la importancia de la web que lo contiene.

- **Otros sistemas**: Ej: redes neuronales, asociaciones de tokens a las etiquetas de la cabecera html para encontrar relaciones, etc.

3.4 Interfaz de acceso

Si bien cualquier sistema que permita realizar consultas y muestre los resultados de vuelta cumpliría los requisitos necesarios, idealmente se construiría un portal web para esta tarea, al tratarse de la forma más común de realizar búsquedas en internet. Esto pues, añade la complejidad de tener que conectar los componentes mencionados en los aparatados anteriores con un servidor, añadiendo al proyecto las complejidades inherentes a estas tecnologías.

Capítulo 4
Diseño

4.1 Arquitectura del programa

Tras una puesta en común con el tutor de los aspectos mencionados en los capítulos anteriores, se llegó al siguiente consenso sobre la arquitectura a utilizar.

- Se emplea una estructura de tipo **índice invertido** como forma de almacenar la información.
- Se emplea el algoritmo de **pagerank** como forma de asignar pesos a los resultados.
- Se emplean **dos parsers** diferenciados para el *crawler* y el *índice* respectivamente.
- Se emplea un **Ngrama** para proveer búsqueda predictiva.

El programa está diseñado bajo un enfoque modular donde cada uno de los elementos implicados está contenido en su propia clase y trata de ser lo más agnóstico posible del resto de componentes. Además, las clases principales emplean un mecanismo de **resolutores multihilo** descrito en profundidad en el apartado 21 que permite que expongan una interfaz simple -existiendo además una única instancia de cada una de estas clases- al mismo tiempo que internamente pueden realizar sus cálculos de forma paralela y mantener una comunicación asíncrona entre ellas. Este mecanismo permite también modificar el uso de recursos de cada una de estas clases dinámicamente.

Además, existe un mecanismo de generación de logs aprovechado por varias clases para registrar información útil a la hora de debugger y hacer testing.

A continuación se muestra una serie de diagramas UML de secuencia que ejemplifica las interacciones entre los componentes principales

4.1.1 Fase de obtención de contenido

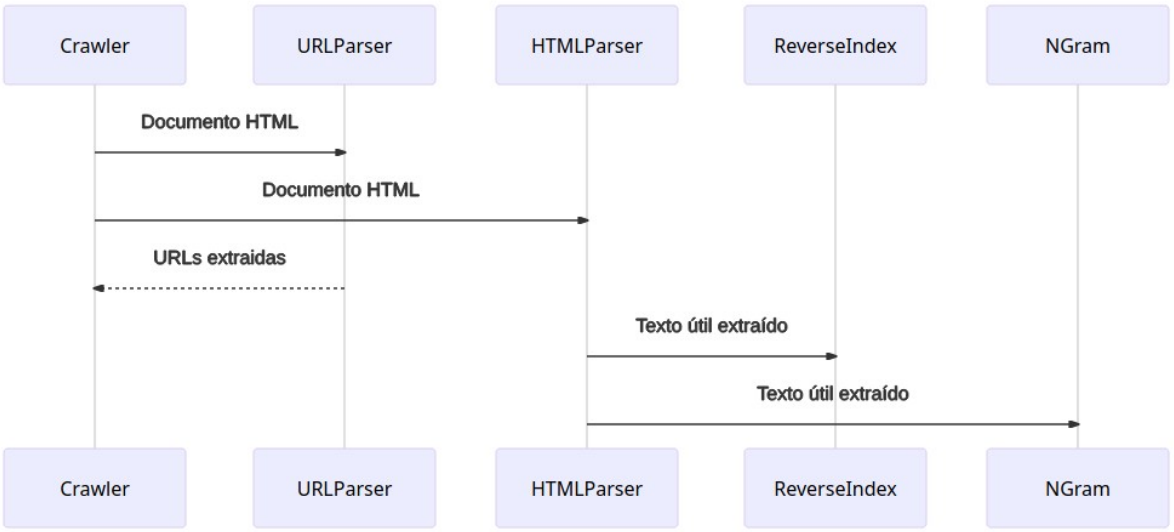


Figura 4.1: Diagrama de secuencia. Fase de obtención de contenido.

Todas estas clases se benefician del mecanismo multihilo mencionado

4.1.2 Fase de búsqueda

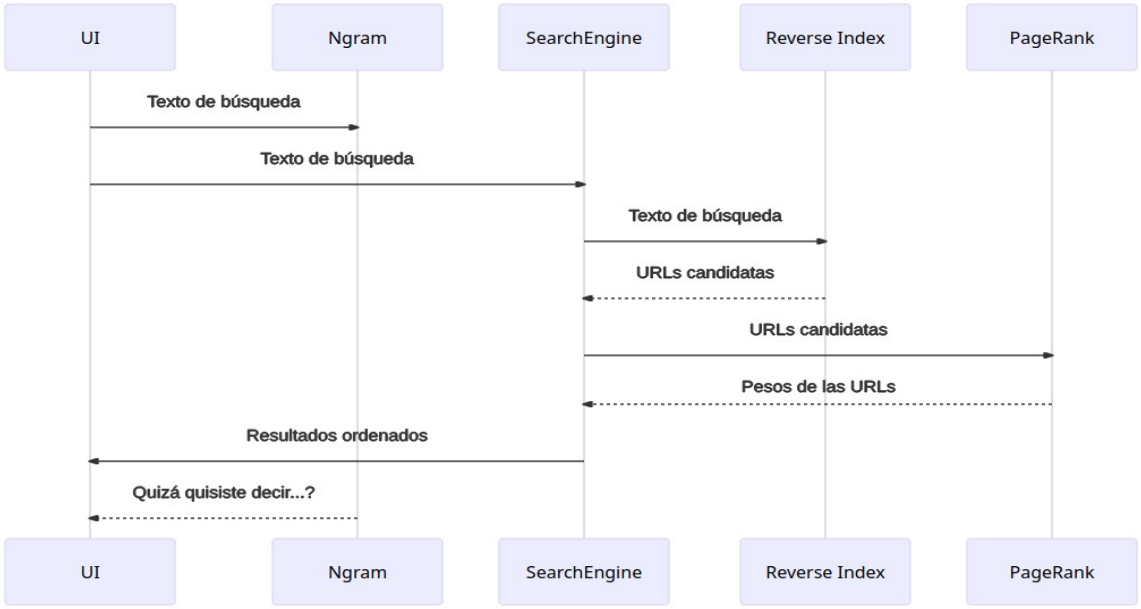


Figura 4.2: Diagrama de secuencia. Fase de búsqueda de contenido

Todas las clases se benefician del mecanismo multihilo mencionado.

4.2 Herramientas y tecnologías

En cuanto a tecnologías se refiere, se emplean las nombradas a continuación:

- Lenguaje **C++** en su versión 11 o superior. Esta decisión está justificada por varios factores.
    - Se trata del lenguaje de programación con el que encuentro mayor familiaridad, habiéndolo empleado en numerosos proyectos tanto académicos como personales y abarcando estos un número de horas sensiblemente mayor al de otros lenguajes. Se considera que este hecho puede paliar en parte el posible aumento de complejidad al elegirlo sobre otras tecnologías.
    - Las necesidades de alto rendimiento lo hacen un candidato ideal para el trabajo asíncrono y multihilo, más aún, en su versión 11 o posterior, donde se añaden funcionalidades que facilitan este tipo de programación además de otras mejoras que serían utilizadas como el soporte de *funciones lambda* , el uso de *punteros inteligentes* que facilitan la gestión de memoria o el soporte para “raw” strings.
    - Dado que el uso de frameworks que realicen gran parte de alguna funcionalidad (como es el caso por ejemplo de ***scrappy***, un crawler escrito en python) era evitado en la medida de lo posible, dejaba de existir la necesidad de utilizar los lenguajes ligados dichas herramientas.
- Librería **libcurl**[xiii] (licencia MIT compatible con el proyecto, *bindings* con C++ nativos al estar escrita en C) como medio para descargar los ficheros html de los portales web.
- Framework **cutelyst**[xiv](licencia LGPL 2.1 compatible con el proyecto, *bindings* con C++ nativos al estar escrita en C++) como medio para servir el portal web de búsqueda.
- Librería **leveldb**[xv] (licencia BSD 3-Clause "New" or "Revised", compatible con el proyecto) como método de almacenamiento. Se trata de una implementación de tabla hash que permite el uso de claves y valores de un tamaño arbitrario. Esto es útil debido a la forma en que funciona el índice invertido, donde los “valores” crecen a medida que se añaden nuevas URLs. Además soporta el acceso multihilo y comprime los datos automáticamente.
- Librería **jsoncpp**[xvi] (licencia MIT compatible con el proyecto, *bindings* con C++ nativos al estar escrita en C++) desarrollada por el propio alumno **con anterioridad a este TFG** como medio de serialización de clases.
- Librería **lib_async_cpp**[xvii] (licencia GPL 3 compatible con el proyecto, *bindings* con C++ nativos al estar escrita en C++) desarrollada por el propio alumno **en el marco de este TFG** como marco de trabajo para el desarrollo de programas asíncronos multihilo en C++.

Capítulo 5
Desarrollo

5.1 Librería async_cpp

Se trata de una librería *single hpp* la cual da soporte al proyecto aportando funcionalidades para la ejecución de algoritmos en diferentes hilos **desacoplados del principal** así como permite la coordinación de estos de forma asíncrona mediante un sistema de señales basado vagamente en el empleado por Qt.. Al contrario que esta última, la librería creada emplea únicamente código C++ en su versión 11 sin necesidad de recurrir a un preprocesado. El sistema emplea asiduamente estas tres características:

- Funciones lambda.
- Combinación de std::mutex + std::unique_lock + std::condition_variable. Mecanismo que permite dormir y despertar hilos con facilidad.
- std::bind como forma de encapsular llamadas a funciones junto a sus parámetros en funciones anónimas que no reciben ni devuelven argumentos.

Provee las siguientes clases.

5.1.1 asyncManager

Se trata de la piedra angular de la librería, una clase singleton encargada de crear y destruir los hilos en función de lo solicitado. Las operaciones internas de control son atómicas para evitar condiciones de carrera.

```
class asyncManager {
private:
  std::atomic<bool> end_loop;
  long long n_threads;
  std::mutex m;
  std::condition_variable cv;
```

Figura 5.1: Clase manejador asíncrono

Sus únicos atributos relevantes son un contador de hilos *n_threads* y un bool *end_loop* que sirve para determinar si se deben terminar las ejecuciones de los hilos.

Permite la ejecución de funciones lambda que le sean enviadas bajo tres políticas distintas:

- **Funciones "estándar":** Sirven para ejecutar una función en un nuevo hilo generado expresamente para la misma y eliminar este al finalizar la ejecución. Recibe una lambda "function", genera un nuevo hilo y le asigna a este la ejecución de una segunda lambda "ff" que encapsula a "function" y gestiona información adicional. En este caso, únicamente reducir el contador de hilos existentes cuando la ejecución de "function" termine. La función "function" no recibe ni devuelve ningún parámetro. Este esquema básico es el empleado por los dos casos posteriores.

```
void add_function (std::function<void (void)> function) {
  auto ff = [&, function]() {
    function();
    --n_threads;
  };
  ++n_threads;
  std::thread t (ff);

  t.detach();
}
```

Figura 5.2: Funciones asíncronas estándar

- **Funciones persistentes:** Sirven para ejecutar funciones de forma cíclica en un nuevo hilo, valorando a cada nueva repetición si se debe volver a ejecutar la función o no en base al valor devuelto por esta y el de *end_loop.* Una vez se sale del ciclo, se elimina el hilo. Similar al caso anterior, esta vez la lambda enviada "function" sigue sin recibir parámetros pero debe devolver un bool. Este valor informará de si la función debe ser llamada nuevamente (retorno de true) o no (retorno de false) una vez finalice su ejecución. Para ello, en este caso la lambda encapsuladora "persistent" contiene un bucle que tiene en cuenta tanto el valor devuelto por "function" como si se requiere el cierre de todos los hilos (is_alive() == false). La existencia de esta función se justifica en la necesidad de realizar bucles potencialmente largos que no bloqueen el hilo indefinidamente.

```
void add_persistent_function (std::function<bool (void)> function) {
  auto persistent = [&, function](void) {
    while (is_alive() && function ());
    --n_threads;
  };
  ++n_threads;
  std::thread t (persistent);
  t.detach();
}
```

Figura 5.3: Funciones asíncronas persistentes

- **Funciones persistentes "adormecidas":** En este caso, permite ejecución de una función en su propio hilo generado expresamente para la misma con la particularidad de que una vez finaliza, si esta devuelve *false*, no se elimina el *thread.* En su lugar, se duerme el hilo y se espera a que este sea despertado desde otro thread. Esto se produce mediante la invocación de la lambda "signal" devuelta por el propio método *add_persistent_function.* Esta tiene la capacidad de despertar el hilo tanto para reiniciar la ejecución de "function" como para eliminarlo por completo, en base al valor del parámetro bool con el que se invoque.

```
// Returns a function to be called to wake up the thread
std::function<void (bool)> add_persistent_sleepy_function (
                           std::function<bool (void)> function) {

  std::mutex* l_m = new std::mutex();
  std::condition_variable* l_cv = new std::condition_variable();
  std::unique_lock<std::mutex>* l_lk =
                                    new std::unique_lock<std::mutex> (*l_m);

  bool* wakeup_condition = new bool (false);
  bool* alive_condition = new bool (true);

  auto persistent = [&, function, l_m, l_cv, l_lk, alive_condition,
                                                   wakeup_condition] (void) {
    while (is_alive() && *alive_condition) {
      if (!function ()) {
        *wakeup_condition = false;
        l_cv->wait(*l_lk, [&]{return (*wakeup_condition);});
      }
    }
    --n_threads;

    delete l_m;
    delete l_cv;
    delete l_lk;
    delete wakeup_condition;
    delete alive_condition;
  };

  auto signal = [l_cv, wakeup_condition, alive_condition](bool keep_alive) {
    *wakeup_condition = true;
    *alive_condition = keep_alive;
    l_cv->notify_one();
  };

  ++n_threads;
  std::thread t (persistent);
  t.detach();

  return signal;
}
```

Figura 5.4: Funciones asíncronas persistentes adormecidas

Nótese que para este caso se deben generar multitud de objetos de forma dinámica y que posteriormente estos son destruidos adecuadamente por la lambda encapsuladora "persistent" cuando se establece que el ciclo de ejecución de "function" ha finalizado. No hay pues fugas de memoria en este proceso.

Adicionalmente, encontramos las siguientes funciones de control:

- **start:** Duerme el hilo que ha llamado a esta misma función. Para entender el por qué, debemos primero ser conscientes de que esta librería está pensada para ser invocada desde un main mínimo que simplemente provoca la ejecución de múltiples hilos, dejando a estos libertad para interactuar entre sí. Según este esquema, si no durmiésemos el hilo principal, el main seguiría ejecutándose hasta eventualmente finalizar, terminando así el proceso y cerrándose el programa con todos sus hilos.

```
void start() {
  std::unique_lock<std::mutex> lk(m);
  cv.wait(lk, [&]{return !is_alive();});
}
```

Figura 5.5: Inicio del manejador asíncrono

- **end_safe**: En el caso de que no lo esté ya, establece la variable *end_loop* a false a fin de que los procesos recurrentes descritos en *add_persistent_function* y *add_persistent_sleepy_function* puedan terminar. Una vez han terminado despierta el hilo que llamó a start a fin de finalizar su ejecución. **No** existen garantías de que ninguno de los hilos generados en las funciones *add_function, add_persistent_function* y *add_persistent_sleepy_function* terminen en algún momento, puesto que estos pueden ejecutar la llamada a un bucle infinito.

```
bool end_safe () {
  if (!end_loop.load()) {
    end_loop.store(true);
    #ifdef VERBOSE
    std::cerr << "Waiting for threads to finish " << n_threads << std::endl;
    #endif
    while (n_threads > 1) {
      std::this_thread::sleep_for(std::chrono::nanoseconds(3));
    }
    #ifdef VERBOSE
    std::cerr << "All threads finished" << std::endl;
    #endif
    cv.notify_one();
    return true;
  }
  return true;
}
```

Figura 5.6: Cierre seguro del manejador asíncrono

- **end_unsafe:** Medida contra el inconveniente descrito con anterioridad. En lugar de esperar al final de las ejecuciones de los hilos generados, despierta inmediatamente al hilo que llamó a la función start.

```
void end_unsafe () {
  if (!end_loop.load()) {
    end_loop.store(true);
    cv.notify_one();
  }
}
```

Figura 5.7: Cierre inseguro del manejador asíncrono

5.1.2 asyncObject

Clase virtual extremadamente simple cuyo único propósito es servir como base de la que heredar objetos que empleen el sistema de señales propuesto en la librería.

```
class asyncObject {
public:
  virtual ~asyncObject();
  void send_signal (std::function<void (void)> slot) {
    asyncManager::get_instance().add_function(slot);
  }
};
```

Figura 5.8: Clase objeto asíncrono

Como puede apreciarse, contiene un método **send_signal** que simplemente automatiza el envío del objeto lambda recibido al método de adición de función no persistente en el manejador asíncrono. La existencia de este método obedece a una cuestión estilística que busca simplificar la sintaxis de comunicación entre objetos.

La idea es que un objeto heredado de asyncObject tendrá parámetros objeto-lambda públicos que actuaran como *señales* y métodos tradicionales que actuarán como *slots*. En determinado momento estos serán *conectados,* permitiendo a partir de ese instante invocar al objeto *señal* como si de una función tradicional se tratase, pero provocando esto la ejecución del método *slot* en un nuevo hilo. Dado que la función send_signal solo acepta lambdas homogéneas que no reciben ni devuelven parámetros (lo primero por la imposibilidad del lenguaje de compilar lambdas parametrizadas, lo segundo además por falta de significado -una señal es enviada y la ejecución en el hilo que la envió continua-), la "conexión" requiere un *binding* de parámetros a fin de generar ese objeto carente de parámetros / devolución.

Queda pues definida la forma de usar estos objetos como se aprecia continuación:

```
class Sender : public asyncObject {
public:
  // SIGNAL
  std::function <void (int)> signal_1;
};

class Receiver : public asyncObject {
public:
  // SLOT
  void receive (int data) {
    std::cout << "I've received some data! " << data << "\n";
  }
};

int main (void) {
  Sender sender;
  Receiver receiver;

  // CONNECT, actually assigns the lambda a body
  sender.signal_1 = [&](int i) {
    // creates a binding with the slot and the parameters
    // encapsulating them as a compatible anonymous function
    sender.send_signal(std::bind(&Receiver::receive, &receiver, i));
  };

  sender.signal_1(23);   // I've received some data! 23
  asyncManager::get_instance().start();
}
```

Figura 5.9: Ejemplo de uso de objetos asíncronos

5.1.3 AsyncDispatcher

Este última clase aprovecha la infraestructura anterior para proveer un sistema que permita crear objectos capaces consumir elementos a través de una cola de entrada y realizar una operación sobre los mismos de forma paralela mediante un conjunto de hilos cuyo número puede variar dinámicamente según las necesidades de carga. La adición de nuevos elementos a la cola de estos **objetos resolutores** multihilo ocurre de forma asíncrona por parte de otros objetos externos.

En el diagrama 21 se aprecia un ejemplo de uso de este método. En él, una clase resolutor posee 3 hilos de ejecución encargados de resolver los problemas que periódicamente se les vaya asignando. Una clase externa provee al resolutor de conjuntos de problemas de formas asíncrona y sin conocer la cantidad de problemas resueltos o el tamaño de la cola. El sistema permite que el programador pueda aprovechar la información de la clase resolutor para añadir o eliminar hilos según se necesite.

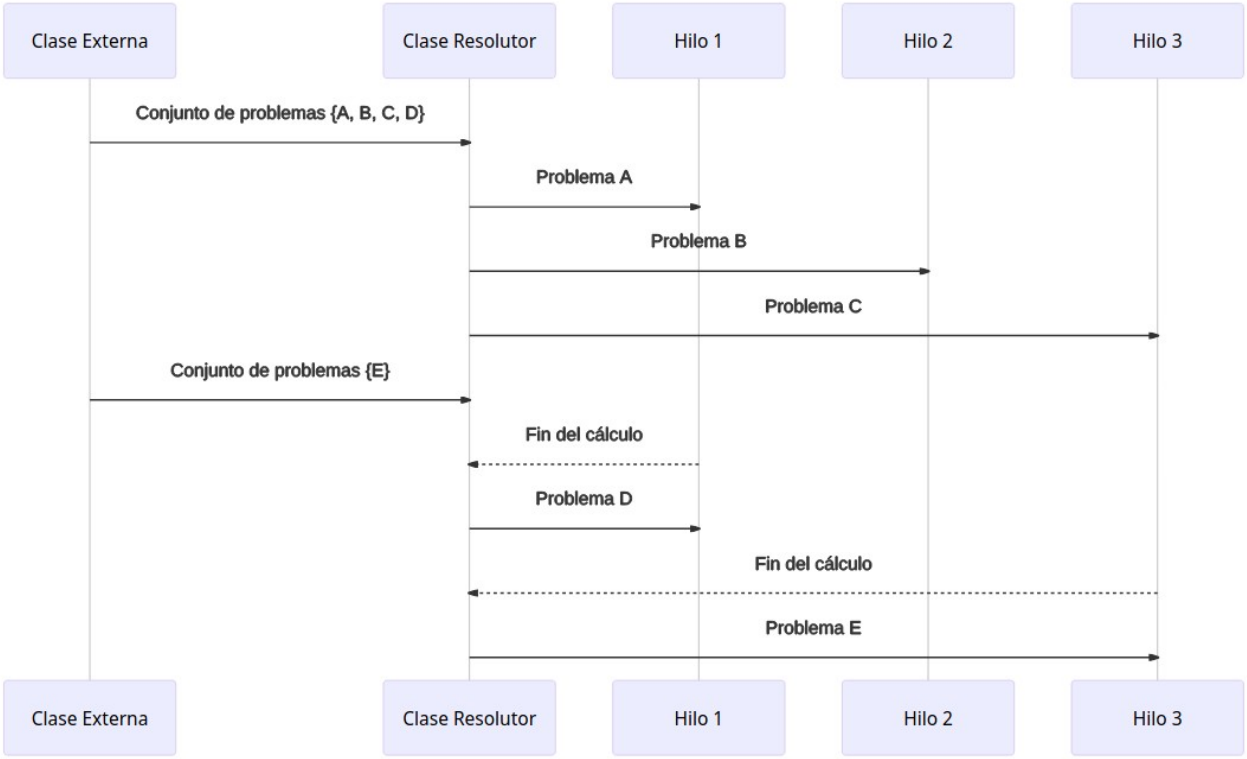


Figura 5.10: Diagrama de secuencia. Ejemplo de AsyncDispatcher

Se trata de una clase abstracta parametrizada de la cual el usuario debe heredar y cuyo template ha de ser el tipo de objeto que se quiere consumir. La operación que se realiza sobre los objetos consumidos debe ser definida reimplementando el método *operation.* Provee además otras dos funciones puramente virtuales *on_queue_empty* y *on_queue_full* que se llaman cuando la cola se encuentra vacía o llena respectivamente.

Una vez instanciado un objeto heredado de esta clase, el manejo ocurre a través

de las funciones *set_initial_queue* para inicializar los primeros valores de la cola, *set_buffer_size* para establecer el tamaño máximo de la cola, *resume_dispatching / pause_dispatching* para interrumpir / restablecer el envío de objetos por parte de la cola a los hilos y *start_dispatching,* la cual recibe el número de hilos que se quieren dedicar e inicializa el proceso.

```
template <class T>
class asyncDispatcher : public asyncObject {
private:
  std::mutex _dispatcher_mtx;
  std::queue <T> queue_t;
  std::list <std::function <void (bool)>> wake_ups;
  bool dispatcher_enabled;

  unsigned buffer_size;

  int working_threads;
```

Figura 5.11: Clase resolutor asíncrono

```
void start_dispatching (unsigned n_threads) {
  for (unsigned i = 0; i < n_threads; i++) {
    wake_ups.push_back (asyncManager::get_instance().add_persistent_sleepy_function(
      std::bind (&asyncDispatcher::try_dispatch, this)
    ));
  }
}

inline void pause_dispatching () {
  dispatcher_enabled = false;
}

inline void resume_dispatching () {
  dispatcher_enabled = true;
  wake_up_threads();
}

inline bool dispatching () {
  return dispatcher_enabled;
}

virtual void operation (T) = 0;
```

Figura 5.12: Funciones de manejo del resolutor asíncrono

Se distingue claramente que el sistema internamente funciona aprovechando las funciones *persistentes adormecidas* que fueron introducidas en el apartado 16, enviándose como lambda a ejecutar el método privado *try_dispatch*. Este se trata simplemente de un bucle que cíclicamente comprueba que la ejecución no se haya pausado y si aún quedan objetos que tratar en la cola a fin de llamar función *operation* reimplementada. En caso de no quedar objetos informa de que la cola está vacía y se detiene el hilo.

Para alimentar la colas de objetos se puede llamar en cualquier momento desde cualquier hilo al método *push_to_queue.* Este comprueba que la cola no esté llena, añade el objeto y despierta el número de hilos mínimo necesario para ejecutar la tarea. Devuelve un booleano que indica que si el objeto se encolo correctamente o la cola estaba al máximo de capacidad.

El programa entero está concebido en torno al uso de estos objetos *AsyncDispatcher.* Cada uno de los actores indicados en los diagramas 14 y 14 está planteado como un objeto heredado de esta clase.

```
bool push_to_queue (T element) {

  if (queue_t.size() >= buffer_size) {
    on_queue_full();
    return false;
  }

  queue_t.push (element);

  wake_up_threads();
  return true;
}
```

Figura 5.13: Método de adición a la cola del resolutor asíncrono

5.2 Crawler

Como se ha visto, este componente es el encargado de descargar el html de los portales a visitar y alimentar a *url_parser* y *html_parser* con dichos datos. Está constituido por una única clase heredada de *AsyncDispatcher* cuya cola recibe objetos std::string que representan urls por visitar. El contenido html descargado es enviado directamente a las colas de ambos objetos en forma de puntero inteligente compartido evitando así copias innecesarias. Su cola es alimentada con los datos enviados de vuelta por el URLParser una vez se ha comprobado que no hay URLs ya exploradas.

El componente es tremendamente simple y únicamente se destaca el uso de la librería libcurl para la descarga de contenido en el método reimplementado *operation* y el hecho de que genera una entrada en el reverse index vacía en cuanto detecta una nueva url.

Debido a su naturaleza expansiva, es una pieza cuya cola se prevee alcance su máxima capacidad con prontitud, pues en la visita de un único portal web pueden encontrarse multitud de URLs nuevas.

```
void Crawler::operation(std::string url) {
  CURL *curl;

  curl = curl_easy_init();

  std::string buff;

  visited_urls.insert(url);
  curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
  curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
  curl_easy_setopt(curl, CURLOPT_WRITEDATA, &buff);
  curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
  curl_easy_perform(curl);              // return a error code
  curl_easy_cleanup(curl);              // free memory

  std::vector<std::string> tags{"some", "example", "tags"};
  auto readBuffer = std::make_shared<std::string>(buff);

  index->insert_document(url, tags);

  bool b = true;

  b &= url_parser->push_to_queue({readBuffer, url});
  b &= html_parser->push_to_queue({readBuffer, url});

  if (!b) {
    std::cout << "Crawler stopped" << std::endl;
    pause_dispatching();
  }

  counter++;
}
```

Figura 5.14: Operación asíncrona realizada por el crawler

En el momento de su construcción, este objeto recibe un listado con las urls iniciales denominado “semilla”, el cual es leído del fichero de *seed.json.* En caso de no existir, se genera uno nuevo automáticamente.

5.3 URL Parser

Clase heredada de asyncDispatcher que consume objetos *ContainerURLParser* que han sido enviados por el crawler y sobre los cuales se realizan las siguientes operaciones:

```
struct ContainerURLParser {
  std::shared_ptr<std::string> content;
  std::string url;
};
```

Figura 5.15: Estructura de datos recibida por el URLParser

5.3.1 Parseo y extracción de URLs

Parseo simple e ingenuo en el que se buscan todos los hipervínculos en el contenido html mediante el uso de una expresión con regular que casa con las cadenas de texto incluidas dentro de un atributo href=”*” .

5.3.2 Normalización y comprobación de validez de URLs

Se comprueba que el hipervínculo utilice los protocolos http o https y no finalice en una extensión de archivo binario (por ejemplo pdf o png). Si la url es válida, esta es tratada por el objeto URL Normalizer y el resultado es enviado de vuelta al Crawler. Dado que su cola no se espera esté llena en ningún momento, mantiene comunicación con el Logger para informar de eventos de vaciado y completitud.

5.4 URL Normalizer

Esta objeto **no** hereda de asyncManager y únicamente expone un método de normalizado de URLs que recibe strings y devuelve el resultado de realizarle las siguientes operaciones en el orden mencionado.

- Conversión de minúsculas
- Sustracción de parámetros dinámicos (parámetros get)
- Sustracción de anclas (#)
- Adición del carácter ‘/’ al final

Todos estos cambios preservan la semántica según lo estipulado en 8

5.5 XML Parser

Empleada por el HTML parser para realizar sus funciones, esta librería desarrollada sin acoplamiento a otros elementos del proyecto permite obtener datos de un string que contenga información estructurada como html en base a una serie de condiciones de búsqueda estipuladas por el programador. Explicaremos en primer lugar la interfaz que expone y a continuación su funcionamiento interno.

5.5.1 Interfaz y concepto

La idea de esta librería es recuperar información **de texto** contenida entre ciertas etiquetas a definir por el usuario, por ejemplo:

<**etiqueta parámetro_no_recuperado**=**2**> *Solo este texto se recupera* </**etiqueta**>

No permite pues obtener datos de las etiquetas en sí o de sus parámetros. Su uso gira en torno al uso de condiciones que estipulan qué etiquetas encierran información relevante y cuáles, por el contrario, no deben ser exploradas. Existen dos tipos de condiciones -de inclusión y de exclusión- consistentes en ambos casos en dos strings y un vector de strings.

```
void add_inclusion_condition (std::string tag, std::string key,
                              std::vector<std::string> values);

void add_exclusion_condition (std::string tag, std::string key,
                              std::vector<std::string> values);
```

Figura 5.16: Condiciones de inclusión y exclusión del XML Parser

Como se ve, la información asociada a las condiciones consiste en el nombre de una etiqueta, seguido de una “key” -potencial parámetro de la etiqueta- y finalmente el conjunto de valores de la etiqueta a los que nos referimos.

Las condiciones de inclusión establecen aquellas combinaciones de etiqueta-parámetro-valores que encierran contenido útil, mientras que en el caso de exclusión las combinaciones a ignorar. En caso de conflicto, se favorece la exclusión. Además, si una combinación etiqueta-parámetro-valores detectada como de inclusión encierra a su vez otra etiqueta, por defecto la etiqueta anidada hereda la condición de inclusión.

Las condiciones permiten el uso del comodín “*” a modo de cierre de Kleene.

5.5.2 Sintaxis

```
- add_inclusion_condition("head",
                          "value",
                          {"2", "3"});

  <head value = "2">    CAPTURED
  <head value = "3">    CAPTURED
  <head value = "1">    IGNORED
  <head other = "a">    IGNORED
  <head>                IGNORED
  <other>               IGNORED

- add_inclusion_condition("head",
                          "value",
                          {"*"});

  <head value = "2">    CAPTURED
  <head value = "3">    CAPTURED
  <head value = "1">    CAPTURED
  <head other = "a">    IGNORED
  <head>                IGNORED
  <other>               IGNORED
```

```
- add_inclusion_condition("head",
                          "*",
                          {});

  <head value = "2">    CAPTURED
  <head value = "3">    CAPTURED
  <head value = "1">    CAPTURED
  <head other = "a">    CAPTURED
  <head>                CAPTURED
  <other>               IGNORED

- add_inclusion_condition("*",
                          "",
                          {});

  <head value = "2">    CAPTURED
  <head value = "3">    CAPTURED
  <head value = "1">    CAPTURED
  <head other = "a">    CAPTURED
  <head>                CAPTURED
  <other>               CAPTURED
```

Figura 5.17: Sintaxis de las condiciones de inclusión y exclusión del XML Parser

5.5.3 Funcionamiento

El parser está programado como un analizador predictivo descendente recursivo cuyas fases de análisis léxico y sintáctico están unidas. Trabaja observando la cadena de entrada caracter a caracter y decidiendo para cada uno si debe ser añadido al string de caracteres útiles de salida o descartado. En ciertos casos se permite el *lookahead.* La gramática del lenguaje es una versión simplificada de la de que describe los documentos XML pero que no realiza comprobaciones innecesarias para el propósito del programa, como que el nombre de una etiqueta de apertura y cierre coincidan o los nombres de las etiquetas sean válidos. Queda definida como se muestra:

- *G = (Σ, V, S, P)*
- *Σ: {'<', '>', '!', '=', ' " ', blank = \s, word = [^<>!="\s] }*
- *V: {SCOPE, TAG, COMMENT, OPEN_TAG, CLOSE_TAG, KEY_VALUE*
- *S: SCOPE*
- *P:*
  - *SCOPE → (word | blank | '<' TAG)* '<' '/' CLOSE_TAG*
  - *TAG → '!' COMMENT | |word+ OPEN_TAG)*
  - *COMMENT → (word | blank)+ '-' '-' '>'*
  - *CLOSE_TAG → word+ '>'*
  - *OPEN_TAG → (blank+ KEY_VALUE)* ('>' SCOPE | '/' '>')*
  - *KEY_VALUE → word+ blank* '=' blank* word+ blank**

Para cada uno de los símbolos no terminales existe una función asociada en el parser, realizándose las comprobaciones sobre las condiciones de inclusión / exclusión en las funciones para **KEY_VALUE** y **OPEN_TAG**. Además, debido a la flexibilidad con la que algunos navegadores parsean html – y la inconsistencia con la que los programadores trabajan – se incluye un conjunto de nombres de etiquetas que si bien formalmente habrían de cumplir las mismas reglas sintácticas que el resto, es común encontrar de la forma. Esta conjunto, por defecto vacío, es completado por el HTML parser.

<**esta_etiqueta_no_cierra_nunca**> *resto de texto [...][Fin del string]*

El contenido útil recuperado por esta librería se encuentra en forma de una única cadena de caracteres, por lo que se requiere una fase adicional de separación de palabras o tokens que es realizada en el HTML parser.

5.6 HTML Parser

Esta clase heredad de AsyncDispatcher hace uso del parser XML descrito anteriormente para obtener la información de texto relevante de los documentos que recibe desde el Crawler. Recibe objetos "*ContainerHtmlParser*" que, al igual que en el caso del URLParser, contienen un puntero inteligente compartido a un string con el contenido de un documento html y otro string con la url de la que proviene.

En el momento de su construcción, emplea la librería *json_cpp* y su aproximación a los objetos serializables en C++ para cargar un fichero de configuración "*htmlparser_config*" que contiene las condiciones de inclusión y exclusión así como el de nombres de etiquetas que no requieren cierre. En caso de no existir este fichero, la clase D*efaultFileGenerator* se encarga de crearlo.

Figura 5.18: Fichero htmlparser por defecto

```
class XMLInfo : public json::Serializable {
public:
  std::vector<XMLCondition> inclusion_conditions;
  std::vector<XMLCondition> exclusion_conditions;
  std::vector<std::string> void_elements;

  SERIAL_START
    "inclusions",    inclusion_conditions,
    "exclusions",    exclusion_conditions,
    "void_elements", void_elements
  SERIAL_END
};
```

Figura 5.19: Uso de clases serializables en el html parser

Una vez inicializado, la operación que se realiza sobre cada uno de los elementos de la cola consiste en la llamada a la función de parseo del XML Parser, la descomposición del resultado en tokens separados por caracteres en blancos (pues como se vio previamente el resultado del parseo es una única cadena de texto) y el envío de un vector con dichos tokens a las colas del Ngrama y el Índice Inverso.

Tal y como sucede en el URLParser, en caso de llenarse o quedar vacía la cola se informa al Logger y se realiza la comunicación oportuna con el Crawler a través del sistema de señales y slots.

5.7 Ngram

Este componente, heredado una vez más de asyncDispatcher, es el encargado de proveer búsqueda predictiva al usuario. Para ello, recibe en su cola strings enviados por el HTML Parser los cuáles contienen la información útil de un documento html, esto es, aún sin estar separada en tokens. Posteriormente, en el momento de realizar una búsqueda, atiende la petición de autocompletado por parte del objeto UI.

5.7.1 Explicación del N-Grama

Atendiendo a la definición de wikipedia[xviii], "Un modelo de n-grama es un tipo de modelo probabilístico que permite hacer una predicción estadística del próximo elemento de cierta secuencia de elementos sucedida hasta el momento. Un modelo de n-grama puede ser definido por una cadena de Markov de orden n-1. " Esto es, nos permite predecir cuál será el siguiente elemento en una secuencia basándonos en los n-1 elementos anteriores.

Para ello, este método se vale de la regla de la cadena[xix], la cual nos permite reformular la probabilidad de existencia de una secuencia completa ordenada de elementos como una serie de multiplicaciones de probabilidades condicionadas de subsecuencias de menor tamaño (regla del producto).

$$P(A1, A2, ..., An) = P(A1 | A2, ..., An)\ P(A2 | A3, ..., An)\ P(An-1 | An)\ P(An)$$

Figura 5.20: Regla de la cadena

En el caso de un N-Grama, restringimos el cálculo de dicha probabilidad a únicamente los N-1 elementos anteriores, por lo que las secuencias serán siempre de tamaño **N** fijo y no un **n** arbitrario como se deduce de la fórmula anterior. Por ejemplo, para N=2:

$$P(el\ caballo\ es\ verde) \approx P(caballo | el) * P(es | caballo) * P(verde | es)$$

Figura 5.21: Aproximación por un 2-grama

El mecanismo de predicción se basa en las siguientes fases.

- Obtención de todas las subsecuencias de tamaño N de un conjunto de secuencias inicial denominado *corpus.*
- Precalcular las probabilidades de cada una de esas subsecuencias y almacenarlas en una estructura de datos acorde.
- Una vez recibida una secuencia sobre la cual se desea realizar una predicción, seleccionar la última subsecuencia de tamaño N-1 y comprobar en nuestra estructura de datos anterior cuál es el token que más probablemente suceda a dichos elementos.

Ejemplificaremos a continuación estas fases suponiendo el caso de un un 2-Grama (Bigrama) y el uso de una estructura en forma de árbol.

Texto inicial

El caballo es azul. Fíjate en el caballo verde. Mira el caballo verde.

Corpus

Se han normalizado todos los caracteres como minúsculas separado las oraciones en función de la puntuación como secuencias aisladas. Además se han añadido los símbolos ^ y $ que denotan inicio y fin de oración respectivamente.

^ *el caballo es azul $* ^ *fíjate en el caballo verde $* ^ *mira el caballo verde $*

Obtención de subsecuencias de tamaño N

De cada secuencia u oración se obtienen las subsecuencias de tamaño 2.

^ el caballo es azul $	**^ fíjate en el caballo verde $**	**^ mira el caballo verde $**
^ el	^ fíjate	^ mira
el caballo	fíjate en	mira el
caballo es	en el	el caballo
es azul	el caballo	caballo verde
azul $	caballo verde	verde $
	verde $	

Árbol de ocurrencias

A continuación se genera una estructura en forma de árbol donde la raíz representa el token cadena vacía o ausencia de token y cuyos hijos son el primer elemento de cada subsecuencia de tamaño 2. A su vez, el segundo nivel de anidación se utiliza para el segundo elemento de la subsecuencia hasta llegar al nivel de anidación N, que se corresponde con las hojas.

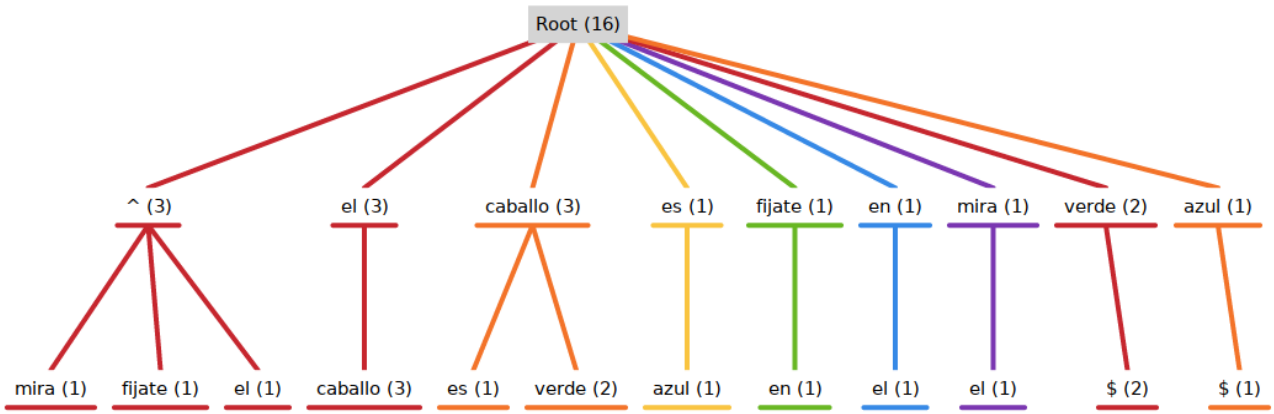


Figura 5.22: Árbol de ocurrencias de un bi-grama

Si nos fijamos, para cada nodo hemos añadido el número de ocurrencias de de dicho token en una subsecuencia dada, el cual siempre será la suma de los números de ocurrencia de todos sus nodos hijos. Por ejemplo, en nuestro conjunto de subsecuencias encentramos “caballo verde” dos veces y “caballo es” una sola vez, por tanto, de la raíz cuelga un nodo caballo con valor 3 (pues existen tres ocurrencias totales de esta palabra en el corpus) seguido de dos hojas, la primera para verde con un valor de 2 y la segunda para es. Es fácil deducir pues que, **para este caso de N=2,** el nodo raíz contiene como valor el número total de palabras encontradas en el corpus – 1 (pues se considera que todas las secuencias terminan en $ y este símbolo no puede colgar del árbol, pues su inclusión implicaría una subsecuencia de tamaño N=1)

Árbol de probabilidades

Una vez se ha montado el árbol de ocurrencias, este se decora añadiendo la probabilidad asociada a cada elemento de una secuencia. Esto es, para un nodo cualquiera, la probabilidad su ocurrencia como hijo de su nodo padre teniendo en cuenta además la probabilidad de existencia del propio padre. Es aquí donde se calculan las probabilidades condicionadas de cada ngrama. Por ejemplo:

$$P(verde \mid caballo) = C(caballo, verde) / C(caballo)$$

Donde C es el número de ocurrencia de la secuencia. Dado que dicha información ya la tenemos en el árbol, el resultado es

$$P(verde \mid caballo) = 2 / 3 = 0{,}66$$

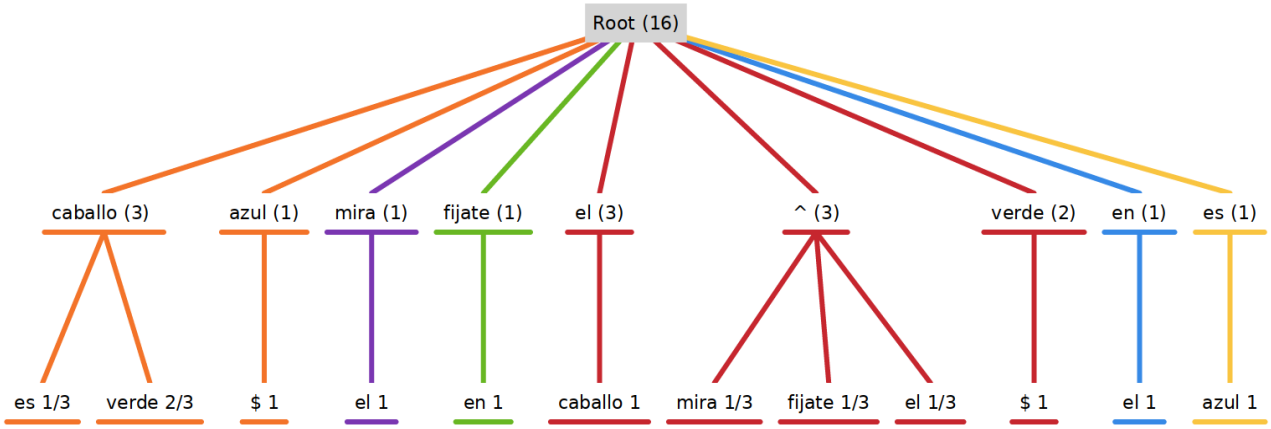


Figura 5.23: Árbol de probabilidades de un bigrama

Se puede apreciar como, por ejemplo, tras el token “el” hay un 100% de probabilidad de encontrar el token “caballo” o que tras “verde” debe finalizar la frase.

Predicción de siguiente elemento

Existiendo dicho árbol, el proceso para predecir nuevos elementos en una secuencia consiste en la búsqueda de la subsecuencia de N-1 tokens en el árbol y la selección del nodo hijo de mayor probabilidad, usándose un umbral como forma de averiguar si merece la pena añadirlo a la oración actual o por el contrario su ocurrencia es altamente improbable. Por ejemplo:

^ Adoro el caballo [...]

$$P(\text{caballo} \mid \text{el}) = 1 \rightarrow P(? \mid \text{caballo}) > \text{umbral} \rightarrow P(\text{verde} \mid \text{caballo}) = 2/3$$

^ Mi caballo favorito es **verde** [...]

$$P(\text{verde} \mid \text{caballo}) = 2/3 \rightarrow P(? \mid \text{verde}) > \text{umbral} \rightarrow P(\$ \mid \text{verde}) = 1$$

^ Mi caballo favorito es **verde $**

En la práctica el cálculo de probabilidades se realiza en forma de logaritmo para evitar *underflow* y facilitar los cálculos, tal y como se refleja en las fuentes consultadas[xx].

5.7.2 Implementación del N-Grama

La funcionalidad del objeto n-grama, como ya hemos visto en su explicación teórica, se divide en dos fases, estructura que también respeta su implementación. Dado que se trata de un objeto heredado de asyncDispatcher, las funcionalidades obtenidas de esta clase padre son empleadas para la primera fase, ya que se considera que las labores de cálculo de probabilidades y montaje del árbol son más frecuentes y costosas que la búsqueda ocasional de términos nuevos en una secuencia introducida por el usuario. Cabe destacar que la implementación no utiliza símbolos de inicio / fin de secuencia y se basa puramente en el valor de las probabilidades para decidir cuándo terminar de añadir nuevos tokens a predecidos. Este diseño se basa en la consideración de que es inusual que un usuario de un buscador escriba frases correctas y completas, en su lugar, suele aparecer un conjunto de términos asociados en una estructura gramatical parcialmente correcta o correcta en subagrupaciones.

En el momento de construcción, el N-Grama recibe el valor de la N a utilizar, el cual está especificado en el fichero *config.json* y que por defecto es 3. En caso de no existir dicho fichero, este es generado automáticamente.

Fase de análisis de secuencias

Inmediatamente después de extraer un string de la cola, se llama al método *loadCorpus* enviándole dicho objeto. Este consiste en la llamada secuencial a los métodos de parseo, montaje del árbol y decoración del árbol.

- Parseo: Como ya hemos nombrado, el string recibido contiene el total de tokens de utilidad según han sido leídos y sin separar. Ese primer método se encarga de separarlos en secuencias (oraciones) en base a símbolos delimitadores como punto, salto de línea o coma. Estas secuencias son empujadas a un *corpus* temporal. El proceso tiene además como consecuencia la adición de nuevos términos encontrados a un *vocabulario.*
- Cálculo de ocurrencias. Para cada línea existente en el corpus, se extraen todas las subsecuencias continuas de tamaño N. A continuación, cada una de las mismas es insertada en el árbol. Esta inserción se lleva a cabo utilizando un proceso **iterativo** que, comenzando desde el nodo raíz y el primer token de la subsecuencia, incrementa el contador de ocurrencias para el nodo actual y comprueba la existencia de un nodo hijo con el valor del token. En el caso de no existir, lo crea y existiese o no previamente, se incrementa su contador en 1. A continuación se repite el proceso para el siguiente token de la subsecuencia estando situados en el nodo que representa el token anterior.

```
struct NGramContainer {
  unsigned n_ocurrences;
  double prob;
  std::map <std::string, NGramContainer*>  content;
  ~NGramContainer() {
    for (auto& element : content)
      delete element.second;
  }
};
```

Figura 5.24: Estructura utilizada para cada nodo del árbol

- Cálculo de probabilidades. Consiste en un mecanismo en este caso **recursivo** que evalúa todos los nodos y les establece su valor de probabilidad según lo establecido en 32.

Una vez finalizado este ciclo, el corpus temporal es desechado y la estructura queda lista para la fase predictiva.

Fase de predicción

Esta capacidad es proveída a través del operador función, el cual recibe un vector de tokens con la información introducida por el usuario y que devuelve una estructura *prediction* que incluye el siguiente token más probable y su probabilidad. El método comienza con el aislamiento de la última subcadena de N-1 elementos y se procede a una búsqueda **recursiva** a través del árbol. En caso de no encontrarse un candidato adecuado se devuelve un token vacío con una probabilidad de 0.

5.8 Reverse Index

Objeto heredado de asyncDispatcher, se encarga del almacenamiento de información sobre los portales visitados por el crawler estructurada de forma que facilite su recuperación en base a términos buscados. Emplea las librería *leveldb* para aportar persistencia a dicho almacenamiento.

5.8.1 Índice Inverso

La estructura de datos utilizada consiste en una tabla hash inversa de documentos html / tokens. Este esquema se basa en el uso de cada uno de los tokens encontrado en los documentos html descargados como una clave en la tabla hash y cuyo valor será el conjunto de todos los documentos donde dicho token fue hallado. Además, a cada uno de los documentos se le asocia el número de ocurrencias del token en el mismo y la cantidad total de tokens que contiene. Esta adición se realiza bajo la premisa de que para dos documentos de igual número de palabras, aquel que contenga más veces un token determinado tendrá más probabilidad de contener información útil acerca del mismo. Organizada la información de esta forma, la búsqueda consiste simplemente en el acceso a las claves y la devolución de todas las webs que la contienen, ordenándose estas en base a la proporción resultante de dividir el número de apariciones del token entre los tokens totales de la web.

Posibles documentos HTML

Web1 (sz = 5)	Web1 (sz = 3)
Caballos. Los caballos son animales [...]	[...] paseando vi caballos [..]

Índice Inverso

caballos	Web1 (n = 2, sz = 5)	Web2 (n = 1, sz = 3)
los	Web1 (n = 1, sz = 5)	
son	Web1 (n = 1, sz = 5)	
animales	Web1 (n = 1, sz = 5)	
paseando	Web2 (n = 1, sz = 3)	
vi	Web2 (n = 1, sz = 3)	

Para una búsqueda del término “caballos”, los resultados ordenados serían *Web1 (n= 2/ sz = 5) > Web2 (n = 1/ c = 3).*

Es fácil ver que existe cierta duplicidad en cuanto a los valores de la tabla hash, pues para cada Web se incluye además siempre su tamaño y cantidad de veces que contiene el token. Este problema empeoraría además si se quisiesen añadir metadatos a cada entrada. Por otra parte, términos comunes provocarían valores enormes en la tabla, pues aparecerían en la mayoría de webs consultadas. Por ello, el

mecanismo de almacenamiento real utilizado difiere, empleándose una tabla propia para cada token (**tablas token**) que contendrá pares url y número de ocurrencias como clave / valor. Por otra parte, una tabla única adicional (**tabla documents)** contendrá pares url y metadatos de la web para completar la información.

Tabla Documents		**Tabla "caballos"**		**Tabla "los"**	
Web1	(sz = 5. metadatos)	Web1	2	Web1	1
Web2	(Sz = 3, metadatos)	Web2	1		

5.8.2 Implementación

Este objeto aprovecha los mecanismos de asincronía y paralelismo de su clase padre únicamente para las funcionalidades de actualización del índice inverso, quedando relegada a la llamada de un método tradicional la fase de búsqueda. El objeto consume a través de su cola instancias de "*ContainerReverseIndex*", una estructura enviada por el HTML Parser que encapsula la url de un documento y el vector de de tokens asociados.

Para sus labores, este objeto utiliza varias clases y estructuras de datos auxiliares, a saber:

WebDocument

Representa un documento HTML en el cual ha sido encontrado un token determinado, es decir, el valor en la tabla "Documents". Contiene el número de veces que se halló el token y un vector de tags hallados en el html. Dado que *leveldb* trabaja únicamente con claves y valores en forma de string, la estructura posee la capacidad de serializarse en este formato a través de los métodos *serialize_in* y *serialize_out.* La cadena de caracteres contenida en el string en binario de la siguiente forma:

Tamaño	u_long	sizeof(V)
Contenido	Número de tokens	Contenido del vector de tags

Donde el V es el string que serializa a su vez el vector de tags

Tamaño	u_long	u_long	tags[0].size()	u_long	[...]
Contenido	tags.size()	tags[0].size()	tags[0]	tags[1].size()	[...]

IndexEntry

Representa la cantidad de veces que un tokens fue hallado en un documento, es decir, el valor en una tabla token. Contiene únicamente un número *unsigned long* y el igual que en el caso anterior, provee la capacidad de serializarse. En este caso, como

el tamaño es estático, su funcionamiento es trivial.

DocumentStorage

Encapsula las labores de almacenamiento y recuperación de información de la tabla “Documents” proveyendo una interfaz simplificada y gestionando el acceso concurrente mediante el uso de un mutex. En el momento de su construcción realiza las operaciones necesarias para la apertura y adquisición de la tabla y la mantiene abierta hasta el cierre del programa.

IndexStorage

De forma análoga al objeto anterior, encapsula Las operaciones relevantes a las “tablas token”. Sin embargo, dado que estas se crean dinámicamente y su relevancia no es continua, estas no son mantenidas y se cierran tras la inclusión de entradas.

IndexResult

Se trata de la estructura utilizada para representar resultados cuando se hace una consulta. Contiene un string con la url del portal web y una variable double con el peso del tokens (calculado como ya se ha visto como cantidad de ocurrencias entre cantidad total de tokens). Un vector de estos objetos es devuelto tras una consulta.

5.9 PageRank

Este objeto, heredado de asyncDispatcher, provee la capacidad de asignar pesos a cada página web según una estimación de su importancia calculada en base al algoritmo *pagerank*[xxi]. Si bien el índice inverso ya aporta ciertos indicios de relevancia a través de la cuenta de tokens y su normalización, este método resulta ingenuo y es fácilmente manipulable a través de, por ejemplo, el uso de páginas que contengan numerosas veces el mismo término sin contexto o información alguna. Es por tanto que se utiliza este sistema, como forma adicional de baremar los potenciales resultados que en primera instancia han sido proveídos por el índice inverso.

5.9.1 Explicación del algoritmo

El sistema pagerank trata averiguar la relevancia de un portal web no en base a su contenido sino al número de enlaces que llevan a ella. La intuición es la de que una página web que es señalada por una gran cantidad de portales externos deberá tener mayor importancia que aquella que apenas es enlazada. Además, tiene en cuanta la importancia propia de las webs que enlazan, de forma que una web poco enlazada podría considerarse muy relevante si sus enlaces provienen de páginas a su vez muy relevantes.

El mecanismo, cuya demostración matemática puede ser consultada en la bibliografía, consiste en la generación de una matriz A que represente el grafo dirigido existente en los enlaces de las webs. Así, cada fila y columna representan webs y cada posición a_{ij} podrá contener un valor 0 ó 1, y representará la existencia o no de un enlace que nos lleve desde la web i hasta el portal j.

	Web1	Web2	Web3	Web4	Web5	Web6
Web1	0	**1**	0	0	0	**1**
Web2	0	0	**1**	0	0	0
Web3	**1**	0	0	0	0	0
Web4	**1**	0	0	0	0	0
Web5	**1**	0	**1**	0	0	0
Web6	**1**	0	0	0	**1**	0

De este ejemplo puede deducirse que la Web1 es muy relevante, pues casi todas enlazan hacia ella. Al mismo tiempo, la Web6, a la que casi nadie enlaza es una de las únicas dos a las que lleva la Web1, por lo que debe tenerse en consideración también.

Una vez creada esta matriz, se normalizan cada uno de los valores de la filas diviéndolos entre la suma total de la fila, dando como resultado M'. Para evitar la existencia de ceros en ninguna posición, se realiza un ajuste en base a la siguiente fórmula, donde c = 8.75:

$$\mathbf{M}'' = c\mathbf{M}' + (1-c)\begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}(1,\ldots,1)\,,$$

Una vez construida M", se procede a calcular su polinomio característico y se extraen sus *autovalores* y *autovectores*. Aquel autovalor mayor en módulo tendrá asociado un autovector cuyas componentes serán todas del mismo signo. El valor absoluto de cada una de esas componentes se corresponden con el peso o importancia de cada web del mismo índice.

5.9.2 Implementación

Debido a la complejidad espacial de este problema, la matriz de enlaces no es construida o actualizada a cada enlace nuevo que se añade. En su lugar, el objeto PageRank contiene un vector de *vectores dispersos,* en cuya interpretación matricial cada fila (índice del primer vector) se corresponde a una web y su contenido (segundo vector) contiene los índices de aquellas webs con las que enlaza. Además, una tabla hash registra pares url / índice asignados para poder hacer las traducciones pertinentes. Esta información nos permite construir la matriz de enlaces (o submatrices contenidas en esta) cuando sea requerido.

El proceso para construirla y calcular los pesos, disparado a través de la llamada al método *get_rank,* consta de los siguientes pasos:

- Una instancia de la clase auxiliar *Matrix* es utilziada para "descomprimir" la información contenida en el vector disperso. Esta es la matriz A.
- Los valores de las filas son sustituidos por la normalización de los mismos.
- Se aplica el ajuste para ceros descrito con anterioridad.
- La clase auxiliar Gauss contiene en su operador función el algoritmo de resolución de sistemas de ecuaciones de Gauss. Este es utilizado para triangular superior e interiormente la matriz A, quedando elementos tan solo en la diagonal principal.
- Se calcula el polinomio característico como $|A - I\lambda| = 0$. Dado que todos los elementos fuera de la diagonal principal son nulos, se puede calcular el determinante mediante el método de los menores adjuntos con facilidad. El resultado es una serie de multiplicaciones de binomios con la forma

$$(a_{00} - \lambda) * (a_{11} - \lambda) * ...(a_{nn} - \lambda).$$

- Nótese que todos los binomios son positivos ($+1 \times (A_{ii} - \lambda)$). Los autovalores son entonces cada uno de los elementos de la diagonal principal.
- Se normaliza el valor de cada autovalor dividiéndolo entre la suma de todos los autovalores. Aquel cuyo valor absoluto sea mayor es escogido.
- Sabiendo que cada componente del autovector asociado a este autovalor y que la matriz es triangular en ambos sentidos, se calculan solo las componentes cuyo índice coincide con las direcciones web candidatas proveídas por el índice inverso. Un vector con estas componentes ordenadas es devuelto para su uso como peso en la ordenación de resultados.

Si bien la implementación es funcional, cabe destacar aquí ciertos aspectos de importancia.

- La creación y de la matriz de enlaces, si bien es retrasada hasta que su creación sea solicitada, sigue requiriendo ingentes cantidades de espacio. Además, se trata de un problema cuyas dimensiones van a crecer indefinidamente a medida que se añadan nuevas webs.
- El proceso de normalización de las filas provoca números razonablemente pequeños para portales con una cantidad generosa de enlaces. La dimensión de estos números se ve además fuertemente decrementada tras pasar por el resolutor de sistemas de gauss.
- El cálculo no utiliza ninguna capacidad de paralelismo. En general es costoso y comprende varias copias de vectores de grandes dimensiones.

Respecto a la obtención de información para poder montar la matriz en sí, la cola de entrada requerida por su herencia recibe desde el URL Parser una estructura consistente en un string con la dirección de una página arbitraria y un vector de strings con todas los enlaces encontrados en la misma. Con esta información se nutren el vector disperso y la tabla hash.

5.10 SearchEngine

Clase simple estándar cuyo propósito es el de realizar las consultas pertinentes a los objetos ReverseIndex y PageRank para obtener una lista ordenada de resultados. Su único método, *search*, devuelve un vector de estructuras *Result* que simbolizan una dirección web, su peso según el algoritmo PageRank y su proporción de existencias del token. El vector está ordenado en base a los resultados de page rank.

```
struct Result {
  std::string url;
  double token_weigth;
  double rank;
};
```

Figura 5.25: Estructura Result

5.11 UI

Clase heredada de **asyncObject** (en lugar de asyncDispatcher como se ha visto frecuentemente hasta el momento) encargado de mostrar una interfaz CLI simple en la que se distingan parámetros de los componentes principales y se puedan realizar búsquedas y recuperar los resultados.

Tras su construcción, añade al manejador dos funciones asíncronas persistentes16: *menu_loop y wait_for_user_input.* La primera se encarga de mostrar periódicamente el tamaño de las colas de todos los objetos heredados de asyncDispatcher así como la cantidad de documentos registrados en el índice inverso. Tras esto duerme el hilo durante 20 mili segundos antes de repetir la operación. La segunda gestiona la entrada de usuario, pudiendo introducir 'q' o '0'. La primera de las opciones provoca el **cierre seguro16** del programa mientras la segunda pausa temporalmente la impresión de *menu_loop* y permite introducir la búsqueda.

El proceso de búsqueda consta de los siguientes pasos:

- Tokenización de la cadena introducida, separando por espacios.
- Llamada al objeto SearchEngine para la devolución de resultados.
- Llamada al objeto Ngram para la obtención de cadena más probable.
- Impresión ordenada de resultados.

```
→  WebSearch git:(master) ✗ ./WebSearch
[0] Realizar búsqueda
[q] Salir

Crawler        URL Parser     HTML Parser    Reverse Index  NGram          n_records
 4404           0              3              80             0              4
```

Figura 26: UI mostrando la cantidad de elementos en las colas

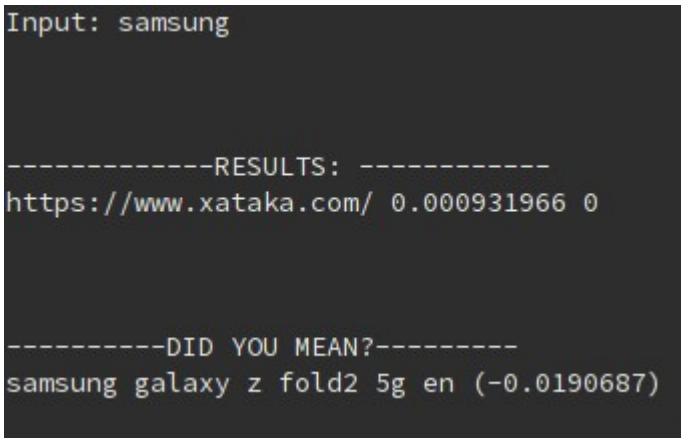


Figura 26: UI mostrando los resultados a una búsqueda

5.12 Logger

De nuevo, estamos ante un objeto heredado de asyncObject, el cual, como su nombre indica, se encarga de crear logs con información general recogida periódicamente así como eventualidades que puedan ocurrirle al programa.

Para realizar esta acción, emplea un mecanismo que consta de una cola de objetos *LogEntry* que contendrán información que debe ser escrita en los logs. Estas estructuras están conformadas únicamente por dos strings, uno reservado para un *mensaje* y el otro para un *timestamp.* En el momento de su construcción se genera automáticamente el timestamp y se le asigna el texto del mensaje que ha sido enviado como parámetro.

```
struct LogEntry {
  std::string message;
  std::string time_stamp;


  LogEntry (const std::string& m) {
    std::chrono::system_clock::time_point today =
    std::chrono::system_clock::now();

    time_t tt;
    tt = std::chrono::system_clock::to_time_t ( today );

    time_stamp = std::string (ctime(&tt));

    message = m;
  }
};
```

Figura 27: Estructura LogEntry

La función asíncrona persistente *log* se encarga de recorrer la cola y concatenar su información a un fichero ubicado en logs/ que ha sido generado en el momento de construcción de la clase Logger. El nombre de este fichero es a su vez el timestamp de su creación. Es en la construcción además cuando se realizan las conexiones que permiten a los diferentes objetos relevantes del programa comunicar situaciones de interés. Esto se consigue a través de la asignación de un cuerpo de función efectivo a los objeto función lambda que son contenidos en las clases que quieren realizar comunicaciones hacia el logger. Estas conexiones consisten simplemente en la creación y adición a la cola de objetos LogEntry.

```
// CONNECTS
html_parser->to_logger_full = [&] {
  push_to_queue (LogEntry ("HTMLParser is full"));
};

html_parser->to_logger_empty = [&] {
  push_to_queue (LogEntry ("HTMLParser is empty"));
};

ngram->to_logger_full = [&] {
  push_to_queue (LogEntry ("Ngram is full"));
};
```

Figura 29: Conexiones que permiten la comunicación con el logger

```
void HTMLParser::on_queue_empty(){
  if (have_been_full) {
    try_to_resume_crawler ();
    to_logger_empty();
    have_been_full = false;
  }
}

void HTMLParser::on_queue_full() {
  if (!have_been_full) {
    have_been_full = true;
    to_logger_full ();
  }
}
```

Figura 28: Disparo de las señales de llenado y vaciado de cola para el HTML Parser

Adicionalmente a estos eventos que podríamos considerar sucesos anómalos, se registra regularmente el tamaño de todas las colas y la cantidad de documentos registrados en el Índice Inverso.

```
0               0               0               0               0
105             0               0               0               2
192             0               0               0               6               URLParser is full Wed Nov 14 10:28:42
2018

465             0               0               0               13
1052            0               0               0               25              URLParser is full Wed Nov 14 10:28:44
2018

1271            0               0               0               32
1510            0               0               0               36              URLParser is full Wed Nov 14 10:28:46
```

Figura 5.26: Ejemplo de fichero de log

5.13 WebSearch

Clase estándar cuyo único propósito es inicializar todos los objetos y mandar la señal de comienzo al asyncManager. Emplea la librería *json_cpp* para inicializar correctamente los objetos en base a la información contenida en el fichero *config.json* -el cual describe el tamaño del Ngrama y el número de hilos asignado a cada asyncDispatcher- y añade la semilla inicial obtenida de *seed.json.* En el caso de que alguno de estos ficheros no se encuentre, llama a una instancia de la clase auxiliar *default_file_generator* que se encarga de crearlo según las plantillas que contiene en forma de cadenas literales “raw”.

La función del main mínimo proveído es únicamente llamar al constructor de este objeto.

Capítulo 6
Pruebas y resultados

6.1 Problemas en el proceso de desarrollo

Tal y como se espera de un trabajo de estas características, el surgimiento de numerosos problemas tanto debido a decisiones de diseño como a imprevistos o estimaciones temporales retrasaron el desarrollo y / o obligaron a modificar ciertos aspectos. A continuación se enumeran los más importantes, desglosando los motivos que lo ocasionaron y posibles soluciones.

6.1.1 Falta de persistencia

Si bien el objetivo de mayor importancia era la familiarización con todas las piezas implicadas y el desarrollo de un prototipo mínimo funcional, se contemplaba como característica altamente deseable el hecho de que la información obtenida fuera almacenada y recuperada en cada nueva sesión. Actualmente, esto solo ocurre con el índice inverso, pero no para el Ngrama, el PageRank ni la cola de URLs a explorar por el Crawler.

6.1.2 Diseño potencialmente inadecuado de la persistencia

Para el caso del índice inverso, el esquema que se sigue actualmente implica el uso de numerosas tablas de dos columnas cada una de las cuales tiene un tamaño que puede ser acotado. Además, existe un cierto nivel de relación entre las tablas y las entradas no se repiten. Esto nos lleva a la cuestión de por qué no se está utilizando un sistema de bases de datos relacional en lugar de varias tablas hash. La razón es que originalmente, el índice inverso se pensó para contener menos información (ahora mismo almacena metadatos sobre las etiquetas y potencialmente podrían añadírsele más con facilidad) y para comportarse estrictamente como el modelo indica. Esto es, como una única tabla cuyas entradas son cada una de longitudes heterogéneas. Sin embargo, tras añadir esa posibilidad de metadatos, la duplicidad fue tal que se prefirió cambiar el diseño ajustándose a lo que ya se tenía.

6.1.3 Rendimiento del algoritmo de PageRank

Se trata de un problema intrínseco al mecanismo. Se requiere una matriz de un

tamaño abrumador y cálculos acorde a su tamaño. Aventuro que debe existir una forma de atacar el problema que ofrezca un mejor comportamiento, pero el tiempo asignado a esta pieza así como mi nivel de conocimiento matemático es limitado y no se exploró más su optimización. Si se compila el programa utilizando PageRank para las valoraciones el tiempo de búsqueda se incrementa hasta niveles inaceptables.

6.1.4 Falta de interfaz web acorde

A pesar de que esta sí fue desarrollada utilizando el framework Cutelyst tal y como se indicaba en la planificación, la comunicación entre dicha pieza y todo el resto de componentes implicaba el uso de librerías dinámicas, algo que si bien ya había experimentado anteriormente en otros proyectos, aumentó la complejidad de la tarea y provocaba un comportamiento errático y difícil de depurar.

6.1.5 Gramática del lenguaje del XML Parser

Siguiendo un razonamiento que resultó desacertado, esta fue diseñada en pro de conseguir únicamente información útil entre etiquetas intentando evitar en la medida de lo posible llevar control sobre las mismas con la esperanza de que los documentos estuviesen bien formateados siempre. Aparentemente, el mundo de la web es muy distinto y está plagado de portales web cuyos ficheros contienen erratas que de alguna forma los navegadores se ingenian para solventar. Si bien se ha intentado activamente evitarlo, aún aparecen ocasionalmente caracteres extraños debido a etiquetas cerradas de formas exóticas.

6.2 Pruebas de rendimiento

Aprovechando la información registrada por el Logger, mostramos a continuación la evolución del programa según distintas configuraciones y dispositivos.

La metodología utilizada ha sido la linealización de cero del motor sin ninguna tabla del índice inverso creada y la observación de su comportamiento hasta alcanzar los 50 portales visitados. Para cada uno de los tests se adjunta una captura del fichero de configuración utilizado y un gráfico que muestra el comportamiento de las colas exceptuando la del Crawler (pues esta siempre es creciente).

Los dispositivos utilizados se muestran a continuación

- **Portátil**. IntelCore I7 4750HQ. 7,7GB Ram DDR3 2666Mhz. 240GB SSD
- **RaspberryPi 4B**

6.2.1 Ordenador portátil. Test 1

En este primer test, se asigna el mayor número de hilos al crawler para potenciar la descarga de múltiples documentos en paralelo. Sin embargo, esto provoca que el índice inverso no sea capaz de lidiar con la información, probablemente por el sobrecoste de tener que escribir en el disco las tablas para cada uno de los términos.

```
{
  "n_threads" : {
    "crawler"      : 8,
    "urlparser"    : 1,
    "htmlparser"   : 1,
    "reverseindex" : 1,
    "ngram"        : 1
  },
  "queues_max_sizes" : {
    "urlparser"    : 100000,
    "htmlparser"   : 200000,
    "crawler"      : 100000,
    "reverseindex" : 100000,
    "ngram"        : 100000
  },
  "ngram" : 3
}
```

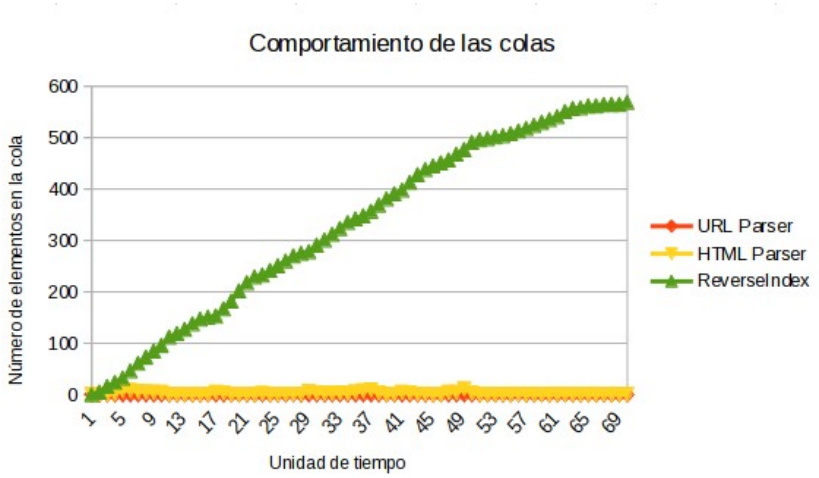


Figura 6.1: Comportamiento de las colas en el Test 1

Se aprecia como para el resto de componentes se mantienen estables, no necesitando, en apariencia, más hilos dedicados a los mismos.

6.2.2 Ordenador portátil. Test 2

```
{
  "n_threads" : {
    "crawler"      : 5,
    "urlparser"    : 1,
    "htmlparser"   : 1,
    "reverseindex" : 3,
    "ngram"        : 1
  },
  "queues_max_sizes" : {
    "urlparser"    : 100000,
    "htmlparser"   : 200000,
    "crawler"      : 100000,
    "reverseindex" : 100000,
    "ngram"        : 100000
  },
  "ngram" : 3
}
```

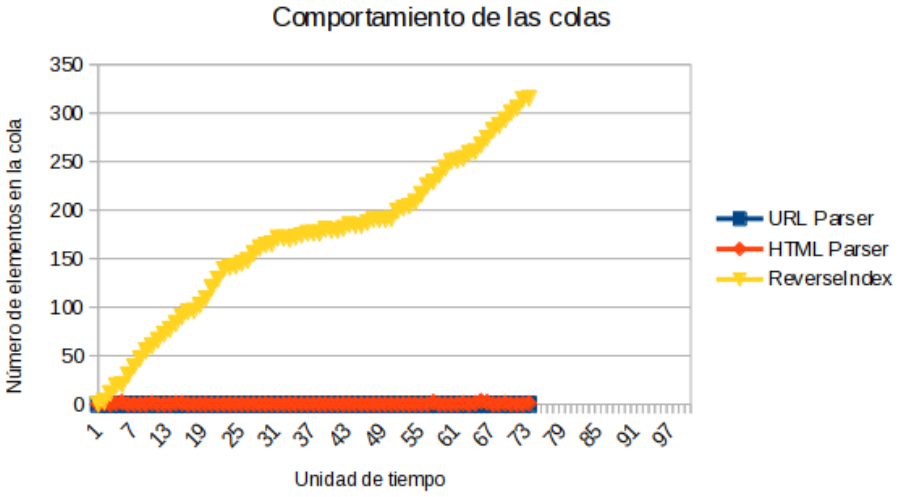
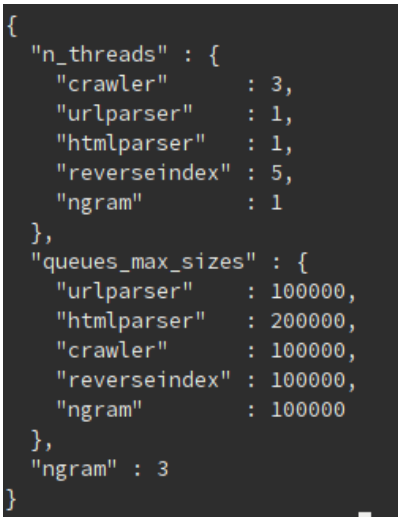


Figura 6.2: Comportamiento de las colas en el Test 2

Se aprecia en este caso una diferencia significativa en la evolución de la cola del Índice Inverso en comparación con el test anterior, decrementándose esta en un 50% al tiempo que el periodo de obtención de los 50 resultados tan solo se incrementa en un 5%. Es importante notar que si bien se reducido su pendiente, sigue tratándose de una cola ascendente sin mínimos locales.

6.2.3 Ordenador portátil. Test 3



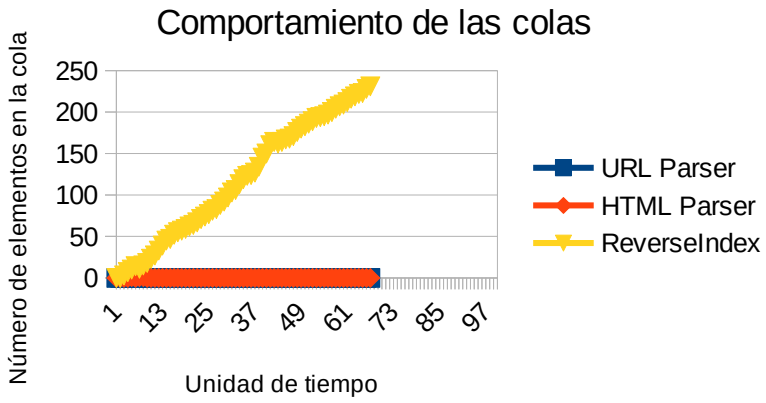
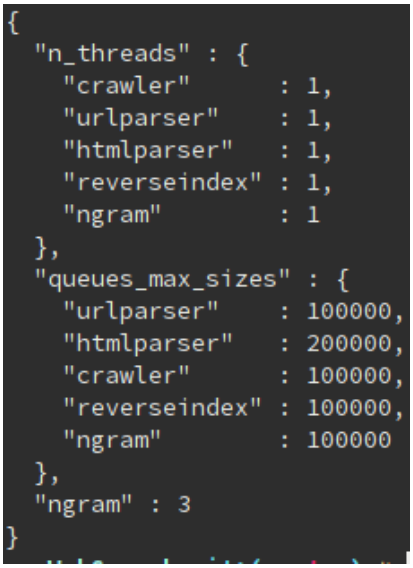


Figura 6.3: Comportamiento de las colas en Test 3

Continuando con la tendencia vista anteriormente, no se distingue un aumento del tiempo de escaneo (de hecho se reduce ligeramente hasta valores similares a los del primer test) y si se consigue disminuir de nuevo la pendiente de la función de la cola del índice inverso.

6.2.4 Ordenador portátil. Test 4



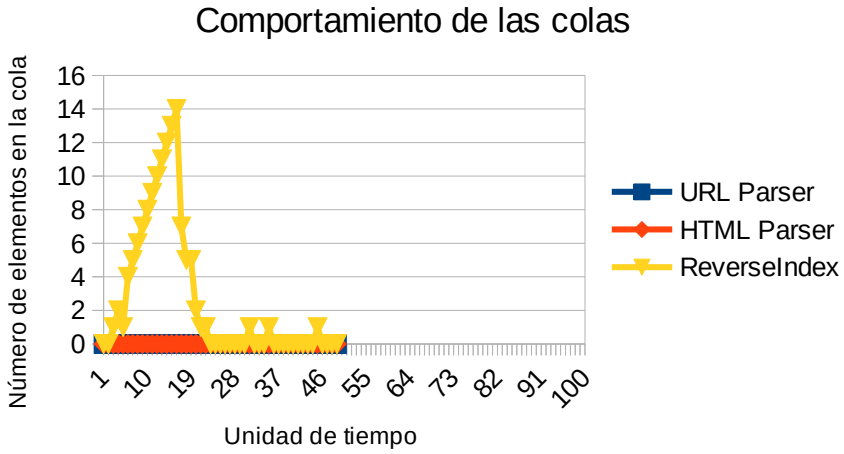


Figura 6.4: Comportamiento de las colas en el Test 4

En este último test por fin se muestra una gráfica con picos y valles. Sin embargo, el uso de hilos resulta aparentemente contradictorio, pues solo se le asigna uno a cada elemento. Tras analizar este comportamiento se detectó que el uso de mutex en las operaciones de escritura y lectura de la base de datos estaba poco refinado, provocando multitud de bloqueos innecesarios (pues el acceso a una tabla token ocasiona el bloqueo a cualquier hilo que quiera acceder a cualquier otra tabla).

Se aprecia además un decremento del tiempo total de en torno al 40%

6.2.5 RaspberryPi. Test 5

```
{
  "n_threads" : {
    "crawler"      : 1,
    "urlparser"    : 1,
    "htmlparser"   : 1,
    "reverseindex" : 1,
    "ngram"        : 1
  },
  "queues_max_sizes" : {
    "urlparser"    : 100000,
    "htmlparser"   : 200000,
    "crawler"      : 100000,
    "reverseindex" : 100000,
    "ngram"        : 100000
  },
  "ngram" : 3
}
```

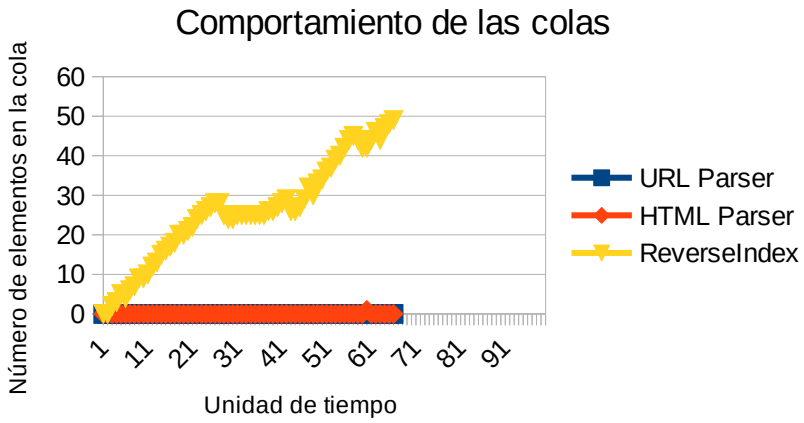


Figura 6.5: Comportamiento de las colas en el Test 5

Comenzando con la configuración usada en el mejor caso posible para el portátil, vemos que en este caso las limitaciones del dispositivo se hacen patentes y si bien el incremento del tamaño de la cola no es lineal sí vuelve a estar acentuado. Además, el tiempo total de escaneo se ha incrementado en torno a un 30%. Hay que destacar aquí que el tipo de memoria utilizada es totalmente distinta, pues se trata de una tarjeta microSD.

Dado que el test se está realizando sobre el escaneo de 50 portales web, parece acertado considerar que el crecimiento es parejo, pues se llega a un tamaño de cola de exactamente 50 elementos para el Índice Inverso.

6.2.6 RaspberryPi. Test 6

```
{
  "n_threads" : {
    "crawler"      : 1,
    "urlparser"    : 1,
    "htmlparser"   : 1,
    "reverseindex" : 3,
    "ngram"        : 1
  },
  "queues_max_sizes" : {
    "urlparser"    : 100000,
    "htmlparser"   : 200000,
    "crawler"      : 100000,
    "reverseindex" : 100000,
    "ngram"        : 100000
  },
  "ngram" : 3
}
```

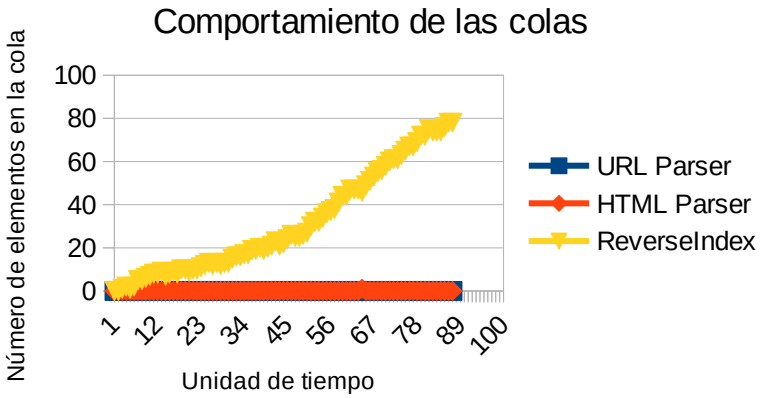


Figura 6.6: Comportamiento de las colas en el test 6

Al alterar el fichero de configuración para asignar más hilos al Índice Inverso volvemos nuevamente a una situación similar a la de los primeros tests.

Capítulo 7
Conclusiones y líneas futuras

7.1 Conclusiones

Bajo la perspectiva del alumno, se han cumplido los objetivos planteados con un alto grado de eficacia, pues el desarrollo del proyecto ha ocasionado:

- Aumento de conocimiento sobre la problemática a resolver y las técnicas existentes para abordarla. En la primera fase se realizó un análisis relativamente exhaustivo de los elementos necesarios para la creación de este tipo de herramienta y distintas formas de conseguir su funcionamiento.
- Creación de un prototipo funcional con todas las características que se propusieron. A excepción de la interfaz web, que no pudo ser **conectada** pero si fue creada, los componentes identificados tanto como estrictamente necesarios como opcionales fueron creados e integrados.
- Afianzamiento de conocimiento y aumento de interrelaciones de distintos campos de la informática. Como se ha podido ver a lo largo de esta memoria, la consecución de este proyecto ha requerido la participación de diferentes ramas del conocimiento como lo son por ejemplo el procesamiento de lenguaje natural, el álgebra lineal, el tratamiento de lenguajes formales o el manejo de las capacidades de paralelismo de un computador.
- Uso de buenas prácticas de programación y generación de librerías reutilizables.. El desarrollo del prototipo no se vio limitado a la creación de elementos ad-hoc pobremente desarrollados sino que siguió un esquema pensado desde el primer momento como modular y causando la obtención de componentes fácilmente utilizables en otros proyectos. Este es el caso, por ejemplo, de la librería de parseo de XML y mayor grado, la de manejo de eventos asíncronos paralelos, la cual he comenzado a utilizar en otros proyectos.

7.2 Líneas futuras

A lo anteriormente comentado en el apartado de problemas 41, se debe añadir las siguientes consideraciones:

- El sistema asíncrono multihilo ofrece a mi juicio una buena infraestructura sobre la que construir el proyecto, pero está terriblemente infrautilizado. Como los componentes han sido desarrollados de forma mínima, en muchos casos el tiempo de cómputo que requieren para sus labores es muy pequeño, como es el caso del URL Parser, HTML Parser y en menor medida el Ngrama. Una posible solución es la desaparición completa de URL Parser y su integración, quizá, en el Crawler. Los otros dos componentes sí tienen potencial para ser expandidos realizándose mejores comprobaciones o aumentando el tamaño de los Ngramas.
- Es probable que la forma en la que se está utilizando la librería libcurl sea tosca y desaproveche recursos, una mejora en esta sección ocasionaría descargas más rápidas de los documentos.
- Si se quisiese migrar a una arquitectura distribuida, el sistema del índice inverso tal y como está concebido podría no resultar la mejor opción, se hace pues necesaria una reevaluación de las estructuras de almacenamiento existentes para este tipo de tareas.

Capítulo 8
Summary and Conclusions

8.1 Conclusions

From the student's point of view, the goals have been achieved successfuly, since the developement of the project has resulted on:

- The increase of knowledge regards the issue to solve and the acknowledgement of the different techniques to tackle the problem. During the first phase, I made a retively exhaustive analysis of the fundamental elements towards the creation of this kind of tool and different ways to achieve it's goals.
- The creation of a functional prototype with all the characteristics that were proposed. With the exception of the web interface, which despite being drafted it couldn't be connected, the totality of the components (even the optional ones) were created and integrated.
- Strengthening of the knowledges and increase of the inner connections in different fields of computer science. As can be seen through this memory, the ainment of this project has required the participation of different branches of knowledge such as the natural language, linear algebra, treatment of formal languages or taking advantage of the paralelism methods allowed by modern machines.
- The usage of good programming practices and production of reusable libraries. The creation of the different elements of the prototype weren't developed ad-hoc, a modular architecture was planned from the very beginning, bringing the possibilities of the easy obtention of components that can be recycled in other projects. This is the case of the XML Parsing library, and even more, the Asynchronous Parallel Manger, which I've started to use in another projects.

8.2 Future Works

Besides the problems seen at 41 section, we must consider the following aditional issues:

- The async multithreaded system offers a suitable infrastructure that allows the elaboration of the project, but it's, in my opinion, terribly untaped. Since the components have been developed with the minimal functional requirements, they are blanzingly fast right now (like the URL Parser, Html Parser and in some ways the Ngram). A possible solution could be the complete desaparition of URL Parser, superseeding it in the Crawler. The other two components do have the potential to be expanded.
- It's highly possible that the way libcurl has been used was rough and a waste of resources. An improvement in this section would result on quicker downloads.
- If a different distributed architectured were desired, it would be necessary to reevaluate the storage structures and check if are sitable for that task.

Capítulo 9
Presupuesto

Recogemos en este capítulo un resumen del presupuesto del proyecto. Tratándose de un proyecto de “investigación” más que de el desarrollo de una herramienta funcional para, por ejemplo, el uso por parte de una empresa, los costes materiales físicos son casi inexistentes.

Tareas	Horas	Presupuesto
Investigación Inicial	30	10€/h
Análisis de requisitos	20	10€/h
Desarrollo del proyecto	180	20€/h
Pruebas	5	10€/h
Total	175	4150€

Tabla 9.1: Tareas, horas y costes

Otros costes	Presupuesto
Portátil Utilizado	750€
Raspberry Pi 4 B	50€
Total	800€

Tabla 9.2: Otros costes

Coste final: 4950€[xxii]

Bibliografía

i https://ecosia.zendesk.com/hc/en-us/articles/206153381-Where-do-Ecosia-s-search-results-come-from-

ii https://help.duckduckgo.com/results/sources/?redir=1

iii https://ec.europa.eu/commission/presscorner/detail/en/IP_17_1784

iv https://transparencyreport.google.com/copyright/overview

v https://transparencyreport.google.com/eu-privacy/overview

vi https://rsf.org/fr/node/16494

vii https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal

viii https://es.wikipedia.org/wiki/Motor_de_b%C3%BAsqueda

ix https://yacy.net/

x https://es.wikipedia.org/wiki/Trie

xi https://en.wikipedia.org/wiki/Suffix_array

xii https://es.wikipedia.org/wiki/N-grama

xiii https://curl.haxx.se/libcurl/

xiv https://github.com/cutelyst/cutelyst

xv https://github.com/google/leveldb

xvi https://github.com/Dariasteam/JSON-cpp

xvii https://github.com/Dariasteam/cpp_async_signals/blob/master/lib_asynchronous.hpp

xviii https://es.wikipedia.org/wiki/N-grama

xix http://www.eecs.qmul.ac.uk/~norman/BBNs/Chain_rule.htm

xx https://web.stanford.edu/~jurafsky/slp3/slides/LM_4.pdf

xxi Bol. Soc. Esp. Mat. Apl. n o 30(2004), 115–141

xxii https://universidadeuropea.es/blog/cuanto-gana-un-ingeniero-informatico