Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de araña para indexación de contenidos en redes ocultas y detección de palabras clave

Autor: Javier Gil Maestro

Tutor: José Javier Martínez Herráiz

2015

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de araña para indexación de contenidos en redes ocultas y detección de palabras clave

Autor: Javier Gil Maestro

Tutor: José Javier Martínez Herráiz

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

CALIFICACIÓN: ……..

FECHA: ……

Agradecimientos

A mis padres por la vida que he tenido.

A los Primarios por la que he llevado.

A mi abuelo, por ser un ejemplo a seguir.

Al grupo de ka0labs, porque no sabría la mitad de lo que sé si no os hubiera conocido.

A mis compañeros del SoC, por el apoyo durante el desarrollo del proyecto.

Resumen

En los últimos años se ha especulado acerca del tamaño y contenido de la parte de internet conocida como “darknet”. Este trabajo trata de comprobar la viabilidad de indexar este contenido, medir el alcance obtenible, y obtener un sistema de monitorización de contenidos.

Mediante técnicas clásicas de arañado web distribuido y el uso de diversos componentes de software se ha desarrollado un sistema de arañado e indexado de contenidos web. Con una ejecución de tres meses y limitando su alcance a tres de las darknets existentes más conocidas, se ha logrado obtener resultados satisfactorios.

Abstract

During the last years, there have been speculations about the size and content of the part of the internet known as darknet. This work aims to determine the viability of indexing that content, measuring the obtainable reach, and develop a content monitoring system.

Using classic web crawling techniques and several software components, a crawling and indexing web content system has been developed. Having it running for three months and limiting its scope to the three most used darknet systems, it has been possible to get satisfactory results.

Keywords

Darknets, crawler, indexer, monitoring

Índice general

Índice de figuras

Índice de tablas

Resumen

En los últimos años ha proliferado el uso de las denominadas “darknets” para todo tipo de actividades fuera del radar, dadas sus cualidades intrínsecas de cara al anonimato y a estar fuera del alcance de los buscadores corrientes.

Dichas darknets basan su funcionamiento en el encapsulado de flujos de datos en protocolos propios de comunicación, otorgando capacidades de cifrado y comunicación anónima mediante sistemas específicos de enrutamiento de paquetes sobre IP. Mientras que es posible, por lo general, saber a ciencia cierta que un usuario localizado es partícipe de una red de este tipo, no lo es tanto determinar el origen y destino de una comunicación, menos aún su contenido.

Este proyecto no trata tanto de deanonimizar a los usuarios de estas redes, sino de localizar el máximo número posible de contenido accesible por cualquier usuario conectado, indexando el contenido textual para poder hacer búsquedas sobre el mismo, usando para ello un sistema de arañado web distribuido.

El objetivo final es la exploración automatizada de los datos recabados, de modo que la herramienta desarrollada permita definir ciertos criterios de búsqueda y ésta informe automáticamente a sus operadores del contenido encontrado en las darknets exploradas que case con dichos criterios. También puede considerarse como un indexador clásico de contenido que permite hacer búsquedas manuales. De manera esquemática, el sistema final debe:

- Ser capaz de explorar mediante técnicas de arañado web estándar el contenido de distintas darknets.
- Indexar el contenido encontrado de manera eficiente y permitiendo búsquedas de manera versátil.
- Funcionar de manera distribuida, de modo que varios nodos realicen las labores de exploración siendo coordinados por un nodo central.
- Ser escalable horizontalmente, tanto en cuestión de almacenado de información como de número de nodos de trabajo.
- Contar con una interfaz de usuario cómoda y elegante que permita realizar todas las operaciones definidas de manera simple.

Éste TFG comienza explicando el funcionamiento básico de las tres darknets que se han considerado para la exploración inicial, aun no siendo necesario este conocimiento para realizar el arañado, por ser posible abstraer los detalles internos y simplificar el acceso a las darknets como si fuese una red estándar.

Posteriormente, se detallan las características completas del proyecto a desarrollar, tanto en cuanto a especificaciones funcionales como en cuanto a componentes básicos por los que debe estar formado, sin entrar en detalles de implementación.

A continuación se describe la versión prototipo que se realizó en aras de estudiar la viabilidad de explorar redes ocultas. Se trata de una versión simplificada de la herramienta final, que implementa únicamente las funciones de exploración e indexado utilizando herramientas básicas.

A partir de los resultados satisfactorios del prototipo se sigue con la estimación en coste material y temporal del desarrollo del sistema completo.

Sigue la descripción de la versión completa del sistema, definiendo las tecnologías utilizadas para cada uno de los sistemas mencionados en la descripción de las características del proyecto, así como del funcionamiento interno del sistema y sus detalles de implementación, haciendo una breve introducción a las tecnologías empleadas. Se describe el funcionamiento de cara a un usuario del sistema.

Finalmente, se listan algunas posibilidades de mejora del sistema que se han dejado fuera del presente proyecto por la limitación temporal del mismo.

Darknets

DESCRIPCIÓN GENERAL

Denominamos *darknet* a aquellas redes o subredes de Internet que, utilizando protocolos de encapsulado propios y esquemas complejos de comunicación entre pares, proveen a sus usuarios de anonimato y privacidad ante cualquier observador de la red.

Generalmente estas redes carecen de índices centrales a partir de los cuales extraer un listado de sitios, por lo que el acceso a los distintos sitios se consigue vía:

- Relaciones de confianza entre miembros que comparten enlaces
- Listados de sitios web recopilados y publicados por miembros
- Enlaces de un sitio web a otro

Por suerte para nosotros, es común que los usuarios confíen la privacidad del contenido que comparten a este hecho, sin preocuparse de protegerlo mediante otro tipo de controles de acceso. Por tanto, si logramos un enlace de partida, es fácil que encontremos información comprometida.

Para este proyecto se han considerado tres de las darknets más usadas:

- TOR
- I2P
- Freenet

A continuación, una breve descripción de cada una de ellas.

TOR

The Onion Router es una de las redes más conocidas por el público, debido especialmente a los casos de cierres de mercados ilegales en esta red en los últimos años.

TOR provee anonimato y privacidad. Para ello, mantiene una arquitectura distribuida en la que un nodo origen pasa por un número N de nodos intermedios dentro de la red hasta alcanzar un nodo de salida que finalmente conecta con el objetivo original de la conexión. Esto se hace iterativamente, expandiendo el alcance del circuito un nodo en cada iteración. Cada nodo sólo conoce a sus pares inmediatos, y cada conexión entre pares negocia un conjunto de claves de cifrado distintas, lo que da fortaleza ante un posible compromiso de alguno de los nodos del túnel.

Gráficamente, en el sitio web de TOR podemos ver los siguientes esquemas que clarifican el proceso de conexión: (https://www.torproject.org/about/overview.html.en)

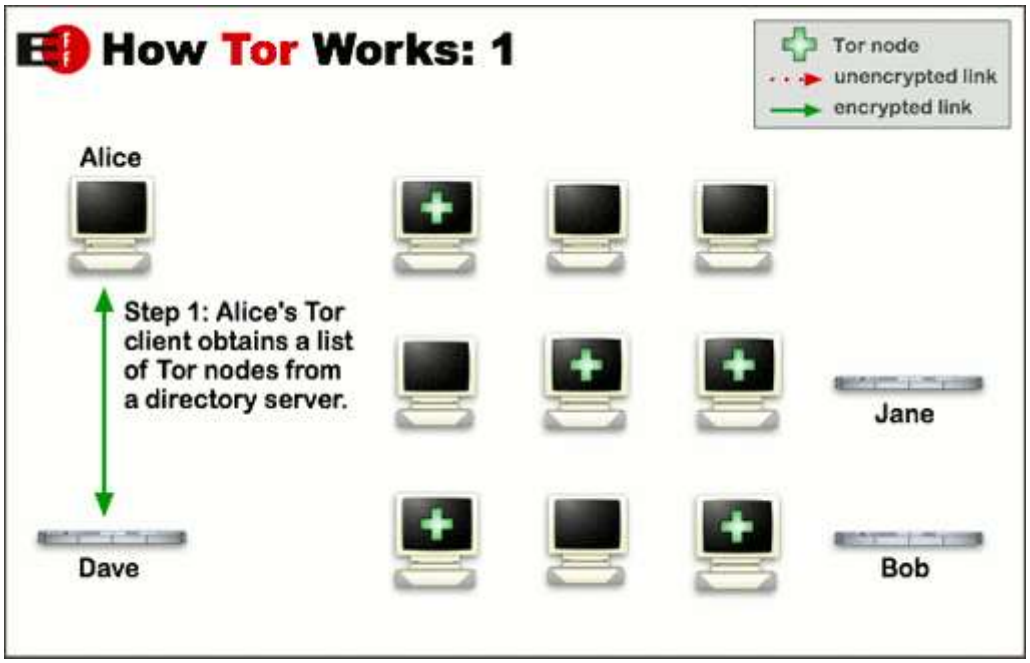


ILUSTRACIÓN 1 - CONEXIÓN EN TOR, PASO 1

Paso 1: el usuario obtiene un listado de nodos activos de la red TOR. Escoge uno de ellos como nodo de entrada a la red.

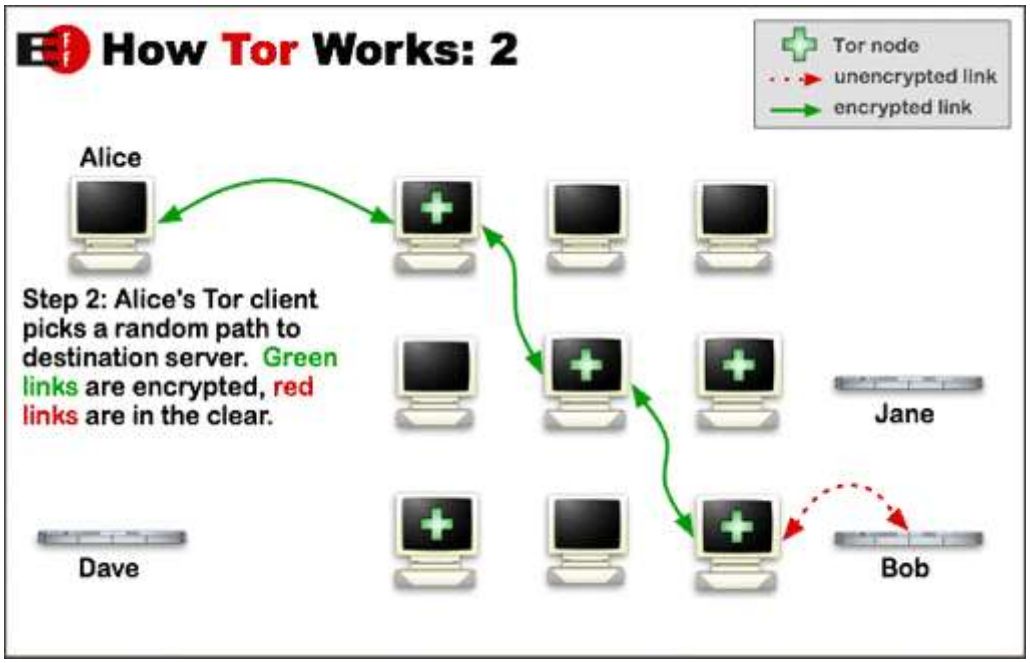


ILUSTRACIÓN 2 - CONEXIÓN EN TOR, PASO 2

Paso 2: el usuario, a partir del nodo de entrada, forma una ruta aleatoria, pasando por N nodos, hasta el nodo de salida. El nodo de salida será el que finalmente establezca la conexión. Las rutas son modificadas cada 10 minutos.

TOR soporta cualquier tipo de tráfico, siendo el único requisito que la aplicación que quiera acceder sea capaz de hacerlo a través de un proxy. Por ello, es usado para todo tipo de actividades, desde navegación web a salas de chat, pasando por compartición de archivos, por poner algunos ejemplos.

Además de funcionar como anonimizador de tráfico de paso, soporta la publicación de contenido de manera anónima, mediante los denominados *hidden services* o *servicios ocultos*. Estos permiten, mediante una URL bajo el TLD propio *.onion*, la localización y acceso en la red al contenido publicado, sin revelar la ubicación del mismo al cliente, ni la identidad del cliente al alojador.

Las URLs son de la forma [2-7a-z]{16}, derivadas de la clave pública del servicio oculto, pudiendo haber un máximo de 32^16 = 1208925819614629174706176 direcciones posibles, lo que descarta una exploración por fuerza bruta del espacio de direccionamiento.

El proceso para publicar un servicio oculto es el siguiente:

1- El servicio forma un par de claves pública / privada.
2- El servicio escoge varios *nodos de introducción* del directorio, y les comunica su clave pública.
3- El servicio forma un *descriptor de servicio* a partir de su clave pública y las direcciones de los *nodos de introducción* que ha escogido.
4- El servicio firma su *descriptor de servicio* con su clave privada, y publica el resultado en una tabla hash distribuida. La clave para buscar en dicha tabla es la URL descrita anteriormente, derivada de la clave pública del servicio.

Un cliente que quiere acceder a un servicio oculto del cual conoce su URL sigue los siguientes pasos:

1- Consulta a la tabla hash distribuida usando la URL. Si hay resultado, obtiene el *descriptor de servicio*.
2- A partir del *descriptor de servicio* obtiene la lista de *nodos de introducción* y la clave pública del *servicio oculto*.
3- Escoge un *nodo de paso* y establece un túnel con el.
4- Forma un *mensaje de introducción*, que contiene un *secreto* de uso único y la dirección del *nodo de paso*. Lo cifra con la clave pública del *servicio oculto* y pide a uno de los *nodos de introducción* del servicio que se lo envíe.
5- El servicio recibe el *mensaje de introducción*, lo descifra, y obtiene la dirección del *nodo de paso*. Conecta con el mismo, enviando el *secreto*.

6- El *nodo de paso* notifica al cliente de la recepción del secreto, y la conexión queda establecida. El *nodo de paso* actúa como nodo intermedio en las comunicaciones entre el servicio y el cliente.

La tabla hash distribuida está repartida entre gran cantidad de nodos de la red y se actualiza periódicamente.

I2P

Trata de ser una red de comunicación anónima y privada basada en túneles unidireccionales cifrados en varias capas entre nodos de la red. Cada nodo escoge el número mínimo de saltos que acepta dar hasta el otro par de la comunicación, pudiendo elegir el balance entre anonimato / seguridad que más le convenga.

Para establecer un canal de comunicación bidireccional ambos extremos establecen dos túneles, uno de envío y otro de recepción, de longitud arbitraria. En algún punto del establecimiento de la conexión, el canal de envío de uno de ellos se conecta al de recepción del otro, y viceversa.

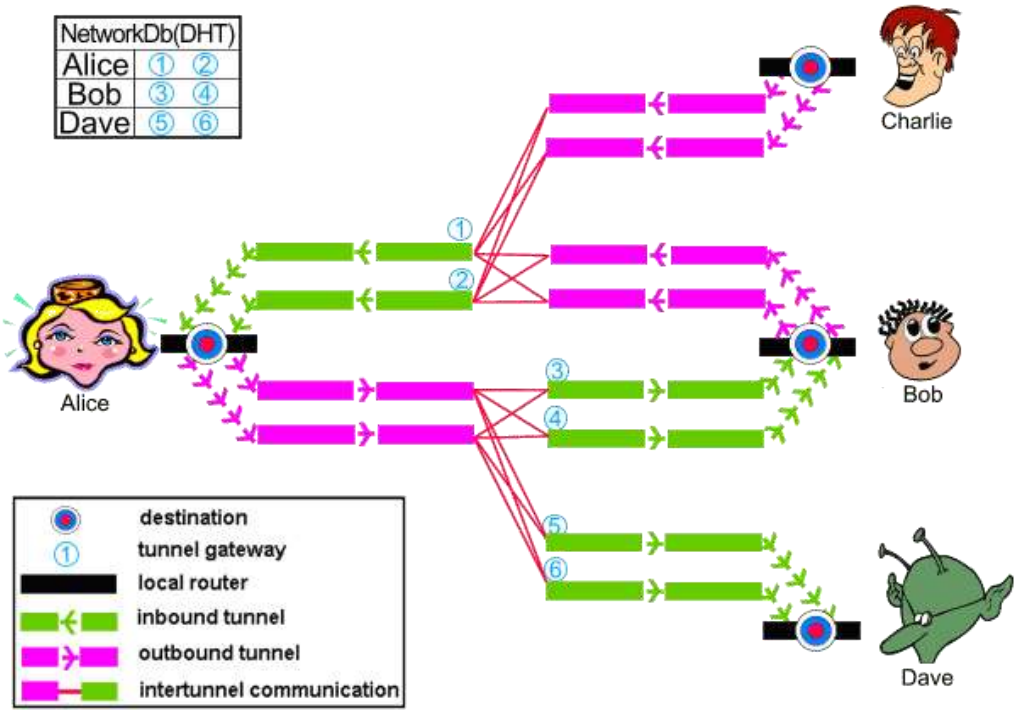


ILUSTRACIÓN 3 - CONEXIONES EN I2P

Al igual que TOR, posee una tabla de direccionamiento distribuida, con un conjunto de nodos que la gestionan. Cualquier nodo con suficiente ancho de banda puede entrar a formar parte de dicho conjunto.

La comunicación se puede representar mediante la siguiente pila de protocolos:

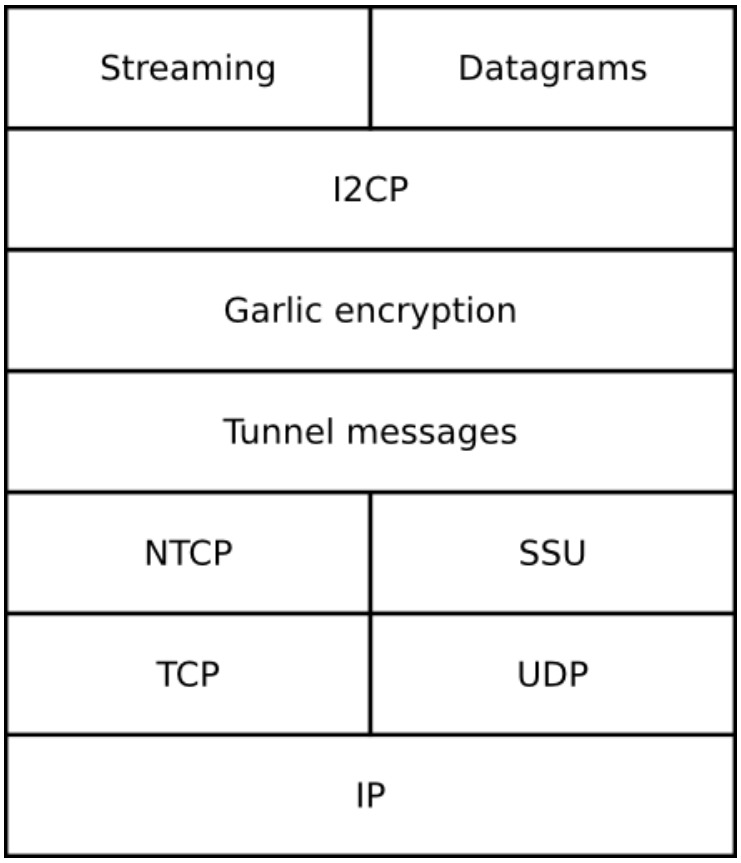


ILUSTRACIÓN 4 - PILA DE PROTOCOLOS EN I2P

Partiendo de la base, los protocolos propios de I2P tienen las siguientes funcionalidades:

- NTCP / SSU: conceptualmente similar a TCP / UDP, se utiliza en comunicaciones punto a punto, sin dar anonimato, pero sí cifrado.
- Tunnel messages: instrucciones cifradas para el establecimiento de túneles junto al contenido del mensaje del siguiente nivel. Puede, a su vez, contener otro mensaje, cifrado con otra clave, que será reenviado al siguiente nodo del túnel, hasta que no queden más mensajes encapsulados.
- Garlic encryption: protocolo de comunicación extremo a extremo. Al estar construido sobre los anteriores, en este nivel contamos con anonimato y pivacidad.
- I2CP: capa de cliente. Permite el acceso a la red de cualquier tipo de aplicación, ya sea mediante streaming de paquetes o datagramas individuales.

Para localizar un destinatario de una conexión, el cliente consulta la tabla hash distribuida. El proceso de establecimiento es:

1- El cliente forma su conjunto de túneles de entrada y salida. Para ello, consulta al directorio y obtiene varios descriptores de routers de la red. Con ellos, forma túneles de longitud arbitraria.
2- El cliente pregunta al directorio acerca de los túneles de entrada del otro peer.
3- El cliente envía a través de uno de sus túneles de salida un mensaje de enlazado a uno de los túneles de entrada del otro peer conseguidos en el paso anterior.
4- Si la conexión debe ser bidireccional, el otro peer realiza un proceso similar. Para acelerar el proceso, el cliente puede incorporar al mensaje de conexión la información sobre sus túneles de entrada.

El alcance de este proyecto (arañado web) hace que el foco de atención sea lo que denominan *eepsites*. Son nodos de la red que publican túneles de entrada asociados a un servidor web en su host. Utilizaremos el cliente de I2P como punto de entrada a la red, del mismo modo que utilizamos el de TOR.

FREENET

Freenet se distingue de las darknets descritas anteriormente por su enfoque al denominado *web of trust*: el usuario tiene control total para decidir con qué nodos se comunica directamente en cada momento. Mediante relaciones de confianza transitivas, cualquier nodo de Freenet deberá, en teoría, ser alcanzable.

También se distingue en el enfoque en cuanto al acceso a contenido. En lugar de funcionar como capa en una pila de red, se plantea como sistema de acceso a contenido distribuido, dando identificadores únicos a cada archivo publicado, pudiendo almacenar en la red distintas versiones.

Funciona como un sistema de contenido distribuido, descentralizado, autobalanceado y con recuperación de errores. Los fragmentos de archivos se almacenan en varios nodos, cifrados, sin tener el usuario control sobre cuales se almacenan en su propia máquina. El almacenamiento es gestionado por la red como conjunto, basándose en la popularidad del contenido (la frecuencia con la que los usuarios acceder a él) para decidir si se guarda o no. Un recurso que no ha sido accedido por nadie en un tiempo prolongado puede llegar a ser eliminado de la red, si el publicador original también lo ha retirado.

Las URLs en Freenet pueden ser de varios tipos:

- CHK (Content Hash Key): contienen el hash criptográfico del archivo, la clave de descifrado, y la configuración del motor criptográfico. Identifican unívocamente un archivo publicado en Freenet. Tienen la forma:
  "CHK@hash_del_archivo,clave_simétrica,configuración_criptográfica"
- SSK (Signed Subspace Keys): sirven para emular el comportamiento de una web estándar, en la que el autor publica, sin posibilidad de suplantación, versiones periódicas de su contenido. Contienen el hash de la clave pública del autor, la clave de descifrado del archivo, la configuración del motor criptográfico, el nombre escogido por el autor del documento, y la versión que se quiere obtener. Los datos identificados por una clave de este tipo incorporan la clave pública del autor y los datos cifrados con la clave simétrica de cifrado y firmados con la clave privada del autor.
  Tienen la forma
  "SSK@hash_de_la_clave_pública,clave_simetrica,configuración_criptográfica/nombre_de_archivo-versión"
- USK (Updataeble Subspace Keys): su funcionamiento de cara a la red es idéntico al de las SSK. Sirven para facilitar la obtención de la última versión publicada de un documento tras una SSK.
  Tienen la forma:
  "USK@hash_de_la_clave_pública,clave_simetrica,configuración_criptográfica/nombre_de_archivo/número/"
  Donde el número indicado determina la manera en la que el cliente de Freenet realiza la búsqueda.
- KSK (Keyword Signed Keys): contienen únicamente el nombre del archivo escogido por el autor, sin información criptográfica o de autoría. Son falsificables, en el sentido de que varias personas podrían incorporar a la red archivos con el mismo nombre, generando colisiones.
  Tienen la forma " KSK@nombre_de_archivo.txt"

Dado que el contenido está cifrado, y el descifrado se realiza una vez obtenido el archivo completo, los nodos de tránsito en la red desconocen el contenido de los flujos de datos. Esto, además, impide que los nodos conozcan el contenido que están almacenando, salvo que obtengan una URL completa al recurso.

CONCLUSIONES

Abstrayéndonos de los detalles de funcionamiento de estas darknets, para cumplir el objetivo de crear una araña que las recorra, podemos simplificar al siguiente esquema:

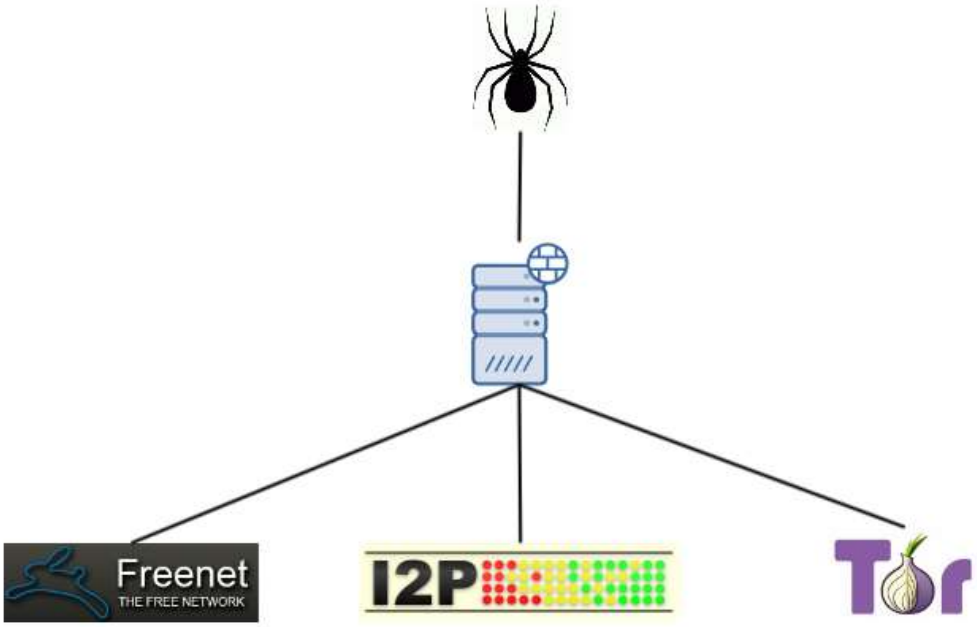


ILUSTRACIÓN 5 - ARAÑADO SIMPLIFICADO DE DARKNETS

Es decir, las arañas utilizan un proxy para acceder a las darknets como si fueran cualquier otro tipo de red. A partir de ese punto, el sistema desarrollado puede entenderse como un sistema de arañado e indexación clásico, con su alcance limitado a las darknets descritas.

Por limitaciones de conectividad del entorno en el que se ha desarrollado el proyecto, únicamente se ha implementado el acceso a TOR, dejando I2P y Freenet como implementación teórica.

Arañas web

Una araña web es un programa destinado a la exploración de sitios web, partiendo de un número de sitios conocidos, y expandiendo su conocimiento de la red a partir de los enlaces contenidos en los mismos, de manera iterativa.

Para ello emulará el comportamiento de un cliente web estándar, como puede ser un navegador web. Idealmente, tratará de ser lo más similar posible, de cara a evitar que los objetivos del arañado lo detecten como tal y devuelvan resultados falseados.

Las arañas normales se ciñen a ciertas pautas "éticas" a la hora de recopilar enlaces:

- Obedecer los meta-tags contenidos en el código HTML
- Obedecer el contenido del archivo robots.txt

Ambas medidas tratan de evitar la indexación de elementos de la web a discreción del dueño. Puesto que partimos de que queremos detectar cualquier tipo de actividad que inherentemente quiere ser ocultada, la araña que describe este proyecto no seguirá ninguna de las dos directivas.

Otro punto a tener en cuenta es el abuso de ancho de banda que una araña puede suponer a un servidor web. Es deseable repartir el trabajo de arañado entre varios objetivos en un periodo de tiempo corto, para evitar saturarlos.

El arañado en darknets no difiere especialmente del arañado en sitios estándar, habiendo únicamente que tener en cuenta los siguientes puntos:

- Punto de acceso a la darknet
  Se usará un proxy central que haga de puente a las distintas darknets.
- Tiempo de acceso
  Las conexiones en estas redes son lentas, por lo que es deseable paralelizar los accesos. Se verá más adelante cómo se ha implementado.
- Persistencia, movimiento de recursos
  Los nodos son, en muchas ocasiones, volátiles, estando online u offline según la hora, o llegando a desaparecer por periodos de tiempo indefinidos. Que un nodo deje de estar disponible no significa que no vaya a volver a estarlo, por lo que no se debe descartar la información recopilada en ningún caso.
- Duplicación de direcciones
  En muchas ocasiones se encuentran servicios alcanzables de distintas maneras.
- Probabilidad de acceder involuntariamente a contenido ilegal
  Dado el enfoque de este proyecto, es importante considerar las posibles implicaciones de un acceso descontrolado a todos los sitios localizados.

La labor de la araña es descargar contenido a partir de un conjunto de enlaces conocido, ya sea HTML u otro tipo de documento con contenido de texto. Posteriormente, debe tratar el contenido descargado, extrayendo dos conjuntos de datos:

- Contenido de texto
- Lista de enlaces a otros recursos

El contenido de texto será usado para crear el índice y el motor de búsqueda. La lista de enlaces retroalimenta a la araña, ampliando su alcance.

Descripción del proyecto

OBJETIVO

Se pretende crear una araña web distribuida capaz de explorar de manera eficaz el espacio web, enfocándose en entornos *darknet*. El sistema contará con un elemento central que coordinará un grupo de trabajo formado por varias arañas, gestionando las labores de arañado por parte de las mismas, y recibiendo sus resultados textuales para su indexación.

ESPECIFICACIÓN FUNCIONAL

El sistema, de manera global, deberá:

- Gestionar el trabajo de un conjunto de arañas, evitando en la medida de lo posible que su trabajo se solape.
- Las arañas recibirán tareas de arañado. Una tarea de arañado consiste en, para cada enlace objetivo:
    - Descargar el contenido detrás del enlace.
    - Analizar el tipo de contenido descargado.
    - Extraer el texto del mismo.
    - Extraer los enlaces contenidos en el texto.

  Una vez extraídos texto y enlaces de cada uno de los enlaces dados por el coordinador, deberá devolver los resultados a este y esperar nuevas tareas.
- Cada araña debe estar unívocamente identificada en el sistema. El gestor debe solicitar autenticación a las arañas antes de cualquier otra interacción posterior.
- El nodo coordinador, al recibir resultados de las arañas, deberá:
    - Indexar el contenido de texto utilizando algún sistema de búsqueda.
    - Añadir los enlaces a la cola de trabajo, de manera que los accesos de las arañas no se centren durante mucho tiempo en los mismos hosts objetivo para evitar su sobrecarga.
- El nodo coordinador debe almacenar estadísticas de cada araña:
    - Tareas ejecutadas
    - Tareas fallidas
    - Enlaces extraídos
    - Tiempos de actividad
- Se debe almacenar un histórico de progreso de la exploración de la red.
- El nodo coordinador debe contemplar la posibilidad de que una tarea asignada a una araña no reciba respuesta. La tarea asignada no debe perderse, sino que debe ser reencolada con su prioridad original.

- El índice de contenido textual debe poder ser consultado mediante una interfaz web, en forma de un buscador.
- Debe haber un panel de control web en el que se puedan definir grupos de consulta. Un grupo de consulta tiene un nombre, un conjunto de búsquedas a realizar periódicamente, y un conjunto de direcciones de correo a las que enviar notificaciones.
    - Los resultados devueltos por las consultas realizadas periódicamente serán almacenados y mostrados en el panel. Podrán clasificarse en tres grupos:
        - *Resultado desconocido*: estado por defecto, no se conoce el contenido del resultado.
        - *Resultado vigilado*: funciona a modo de *bookmark*, permite tener un listado de resultados que se quiere monitorizar.
        - *Resultado ignorado*: resultados que no son relevantes o son falsos positivos.
    - Se debe poder establecer filtros para categorización automática de resultados. Un filtro debe poder comprobar los campos
        - Título
        - Texto
        - Enlace

      La condición de la comprobación podrá ser:
        - Contiene
        - No contiene
        - Regex matchea
        - Regex no matchea

      Un filtro que se cumple debe poder clasificar automáticamente el resultado al que se aplica en cualquiera de las tres categorías.

      Cada filtro va asociado a una consulta de un grupo de consulta. Un grupo de consulta puede tener cero o más filtros.
    - El panel debe permitir la modificación de la lista de grupos de consulta, las consultas y las direcciones de notificación de cada grupo, y los filtros de cada consulta, así como categorizar manualmente resultados individuales.

COMPONENTES

Sigue una descripción genérica de los componentes contemplados en el diseño inicial del proyecto, sin entrar en detalles de implementación o tecnologías utilizadas.

NODO COORDINADOR

Contendrá los siguientes elementos:

- La interfaz de comunicación con las arañas, en forma de una API REST a las que accederán para autenticarse, recibir tareas y reportar resultados.
- Conexión con la base de datos de tareas de arañado. Accederá a esta para:
    - Extraer enlaces para enviar como tarea a las arañas.
    - Eliminar los enlaces de la cola una vez devueltos resultados por las arañas.
- Conexión con la base de datos de datos de arañado. Ver más adelante. Accederá a la misma para calcular la puntuación de los enlaces devueltos por las arañas.

BASE DE DATOS DE TAREAS DE ARAÑADO

Contiene el conjunto de enlaces a visitar por las arañas, con su puntuación asignada para hacer este acceso de manera ordenada.

BASE DE DATOS DE ARAÑADO

Contiene el histórico de acceso a sitios web, con las fechas de primer y último acceso, así como el número de veces accedidos, de cada dominio, cada ruta dentro de un dominio, y cada conjunto de parámetros GET dentro de cada ruta.

Esta información será utilizada para calcular la puntuación de los enlaces recibidos, dando puntuaciones más altas (menos prioritarias) a las URLs que más se hayan accedido o se hayan accedido más recientemente.

BASE DE DATOS DE ARAÑAS

Contiene datos de autenticación para cada araña identificada en el sistema y sus estadísticas de actividad.

ÍNDICE DE TEXTO

Base de datos donde se indexará el contenido textual de los sitios arañados.

MONITOR DE TAREAS EXPIRADAS

Comprobará que las tareas asignadas a las arañas no permanezcan más tiempo del debido en estado “asignado”, donde el tiempo máximo se estima mediante el número de URLs objetivo que contiene la tarea y el tiempo por URL que se considere aceptable.

Una tarea puede expirar en caso de que sea asignada a una araña y dicha araña no devuelva nunca los resultados, por cualquier tipo de fallo del sistema.

En el caso en el que una tarea expire, será reencolada para su posterior asignación a otra araña.

ARAÑA

Programa que será ejecutado en varias ubicaciones y realizará los accesos a las *darknets*.

Su ciclo de trabajo es:

1- Identificación con el gestor
2- Solicitud de tarea
3- Ejecución de tarea asignadas
    a. Descarga del contenido para cada URL contenida en la tarea
    b. Extracción de texto del contenido
    c. Extracción de enlaces del contenido
4- Devolución de resultados al gestor
5- Repetir desde **2**

Cada máquina en la que se ejecute una araña, por tanto, debe tener acceso a cada *darknet* contemplada, ya sea mediante instalación local del *Proxy de acceso* o mediante proxys externos.

PROXY DE ACCESO

Punto de acceso centralizado a las *darknets* alcanzadas por el proyecto. Su misión es reenviar las solicitudes de las arañas al punto de entrada a la *darknet* correspondiente.

PANEL DE CONTROL Y DE BÚSQUEDA

Se trata de la interfaz de usuario para los operadores de la herramienta. Mediante un sitio web, permite gestionar los grupos de consulta descritos en el apartado *Descripción funcional*, así como realizar búsquedas de cara a navegación manual o prueba de las consultas definidas en los grupos de consulta.

También permite la visualización de todos los datos históricos y estadísticos.

LÍNEA DE DESARROLLO

Se propone seguir las siguientes etapas:

1- Desarrollo de una versión prototipo
   Elaborar una versión inicial, renunciando a parte de las especificaciones funcionales y simplificando el conjunto de componentes, con los siguientes objetivos:
   - Evaluar la viabilidad del objetivo final del proyecto, es decir, la elaboración de un índice competente del contenido de varias *darknets*
   - Estimar los recursos hardware necesarios a largo plazo
   - Estimar el tiempo real de desarrollo del proyecto

2- Elaboración del diseño final, teniendo en cuenta los resultados obtenidos y los problemas encontrados a partir de la ejecución de la versión prototipo durante un periodo limitado de tiempo.

3- Desarrollo de la versión completa del proyecto.

Prototipo

LIMITACIONES

Con la versión preliminar se pretende únicamente evaluar la viabilidad del arañado y los recursos necesarios para almacenar los resultados, por lo que se dejan sin implementar las siguientes funcionalidades:

- Arañado distribuido: toda la tareas de arañado, indexado y búsqueda serán realizadas en la misma máquina. Por tanto, quedan sin implementar:
  - Lógica de reparto de tareas entre arañas
  - API REST de acceso para las arañas
- Panel de control: no es relevante en ninguno de los aspectos a evaluar.
- Indexado en motor de búsqueda: se quiere evaluar la capacidad de almacenamiento al alza, por lo que todo el contenido textual obtenido será almacenado en una base de datos. Para realizar búsquedas, se utilizarán las funciones simples de búsqueda de texto dadas por la misma.
- Estadísticas de actividad de las arañas: puesto que se trata de un sistema unificado, carece de sentido.
- Autenticación de arañas: ídem al punto anterior.
- Monitor de tareas expiradas: las tareas serán extraídas y ejecutadas linealmente por el mismo programa, por lo que no tiene sentido plantear fallos de sistemas externos.
- Únicamente se cuenta con acceso a la red TOR.

DISEÑO

Se implementó un único script que, usando la librería de Python Scrapy (**ver más adelante**), hiciese un arañado constante, almacenando los resultados en una base de datos MongoDB con índice FTS. Usando esta base de datos, se implementó un panel web de búsqueda para poder explorar los resultados obtenidos.

Scrapy, por defecto, guarda sus datos de arañado en memoria (la cola de objetivos y el histórico de sitios explorados para evitar repeticiones). Para lograr persistencia de esta información, se hizo uso de la librería **scrapy_redis**, que externaliza el almacenado a una base de datos en Redis. De este modo, durante el desarrollo del prototipo, se puede reiniciar la araña sin perder los datos anteriores. No se requirió ninguna configuración específica del servidor Redis.

Aunque Scrapy gestiona correctamente la evasión de enlaces repetidos, no implementa ningún tipo de control de acceso excesivo al mismo conjunto de hosts, ni

hace nada para evitar *sinks* (sucesiones de enlaces en el mismo sitio con modificaciones en la ruta o en los parámetros de manera infinita o demasiado abundante, sin ganar relevancia). Para corregir este defecto, se creó una segunda colección en MongoDB, conteniendo los datos de acceso a los diferentes sitios. Un documento de esta colección tiene el siguiente aspecto:

ILUSTRACIÓN 6 - DOCUMENTO MONGODB PROTOTIPO, ACCESO A SITIOS

Es decir, para cada dominio explorado, se guarda el número de veces que se ha accedido al mismo, junto a las rutas que se han explorado. Para cada ruta, se guarda a su vez el número de veces que se ha descargado, con la lista de parámetros HTTP GET utilizados. Cada conjunto de parámetros se normaliza a la concatenación de los nombres de los mismos ordenados alfabéticamente y separados por comas. Se guarda la cantidad de veces que se ha accedido a la ruta a la que pertenecen con dichos parámetros, junto a una lista de los valores que han tomado en cada acceso, normalizados de igual manera.

De este modo, se implementa un filtro de links en Scrapy. Se definen los siguientes máximos:

- max_domain_access: número máximo de veces que se puede acceder a un dominio antes de descartar todos los nuevos enlaces que apunten al mismo, sea cual sea la ruta.
- max_path_access: número máximo de veces que se puede acceder a una ruta dentro de un dominio antes de descartar todos los enlaces a la misma, sean cuales sean los parámetros.

- max_param_access: número máximo de veces que se puede acceder a una ruta dentro de un dominio utilizando los mismos parámetros, tomen el valor que tomen, antes de empezar a descartar enlaces.

Cada vez que se accede a una URL, se actualiza el documento de la base de datos, sumando uno al campo "count" del dominio, de la ruta accedida, y del conjunto de parámetros. Cuando Scrapy extrae los enlaces del contenido descargado, se consulta esta información, y si la cuenta de alguno de los tres valores limitados es igual al máximo, se descartan. Así, se impide que la araña pase demasiado tiempo en el mismo sitio, y se fuerza a que haga una exploración del resto de sitios descubiertos, lo cual es el objetivo del prototipo.

Para conseguir el acceso de la araña a las darknets se implementa un *middleware* de Scrapy. Un *middleware* es una clase que permite interceptar varios puntos del flujo de datos dentro del framework de Scrapy, modificando los datos si es necesario.

El *middleware* que cumple esta función es trivial: intercepta las peticiones HTTP cuando pasan del motor de arañado al motor de descarga, extrae el dominio de la URL objetivo de la petición, y según a qué darknet pertenezca, asigna un valor u otro al campo "proxy" de la petición. El motor de descarga de Scrapy, una vez hecho esto, utiliza el proxy indicado para la conexión.

Para exportar los datos descargados a MongoDB se utiliza un *pipeline* de Scrapy. Los *pipelines* son procesadores de elementos obtenidos del arañado que se invocan secuencialmente. El procesado que se aplica es almacenar los campos *contenido*, *url*, *primera aparición* y *último acceso* en la base de datos.

En la colección de resultados se establece un índice de texto en el campo *contenido.* En este momento no se busca tener un indexador de texto potente, sino un simple almacén de datos sobre el que ejecutar búsquedas sencillas, por lo que la función de índice de texto de MongoDB cumple suficientemente bien.

El panel de búsqueda cuenta únicamente con una entrada de texto en el que introducir una consulta y una tabla para mostrar los resultados, en forma de enlace y extractos del contenido relevantes a la búsqueda. Se utiliza bootstrap como framework de diseño web y un pequeño servidor web en Python utilizando la librería Flask, encargado de recibir las consultas y devolver los resultados mediante una API REST.

RESULTADOS

La URL inicial del arañado fue http://deepweblinks.org/ conteniendo 200 enlaces a sitios web dentro de la red TOR.

Tras un periodo de mes y medio con la araña prototipo en ejecución, los resultados fueron:

- 10 000 dominios distintos localizados
- 300 000 URLs distintas descargadas
- 10 GB de espacio en disco duro ocupados por la colección de resultados, contando el almacenado del documento y el índice de texto
- El total de la memoria RAM de la máquina en uso (4 GB)
- CPU con picos del 100% de uso por parte del proceso de MongoDB (2 vCores a 2.6 GHz) causados por la labor de actualización del índice de texto.
- Las búsquedas al índice de texto pueden llevar hasta un minuto con la base de datos utilizada.

De lo que se obtienen las siguientes conclusiones:

- La exploración de *darknets* mediante técnicas de arañado web es viable y da resultados a partir de una entrada relativamente pequeña.
- La estimación de espacio en disco necesaria es 0.22 GB por cada araña por cada día de exploración, o bien 35KB por cada URL indexada, utilizando MongoDB como índice de texto.
    - MongoDB no es viable como índice de texto, es necesario buscar alternativas y extrapolar la estimación.
- Es necesaria más de una máquina para mantener el sistema, en correcto funcionamiento. Se observó que, incluso con una sola araña, el sistema acabó volviéndose inestable por sobrecarga.

Versión final

ARQUITECTURA

El diseño final tiene la siguiente arquitectura en el lado del nodo coordinador / indexador:

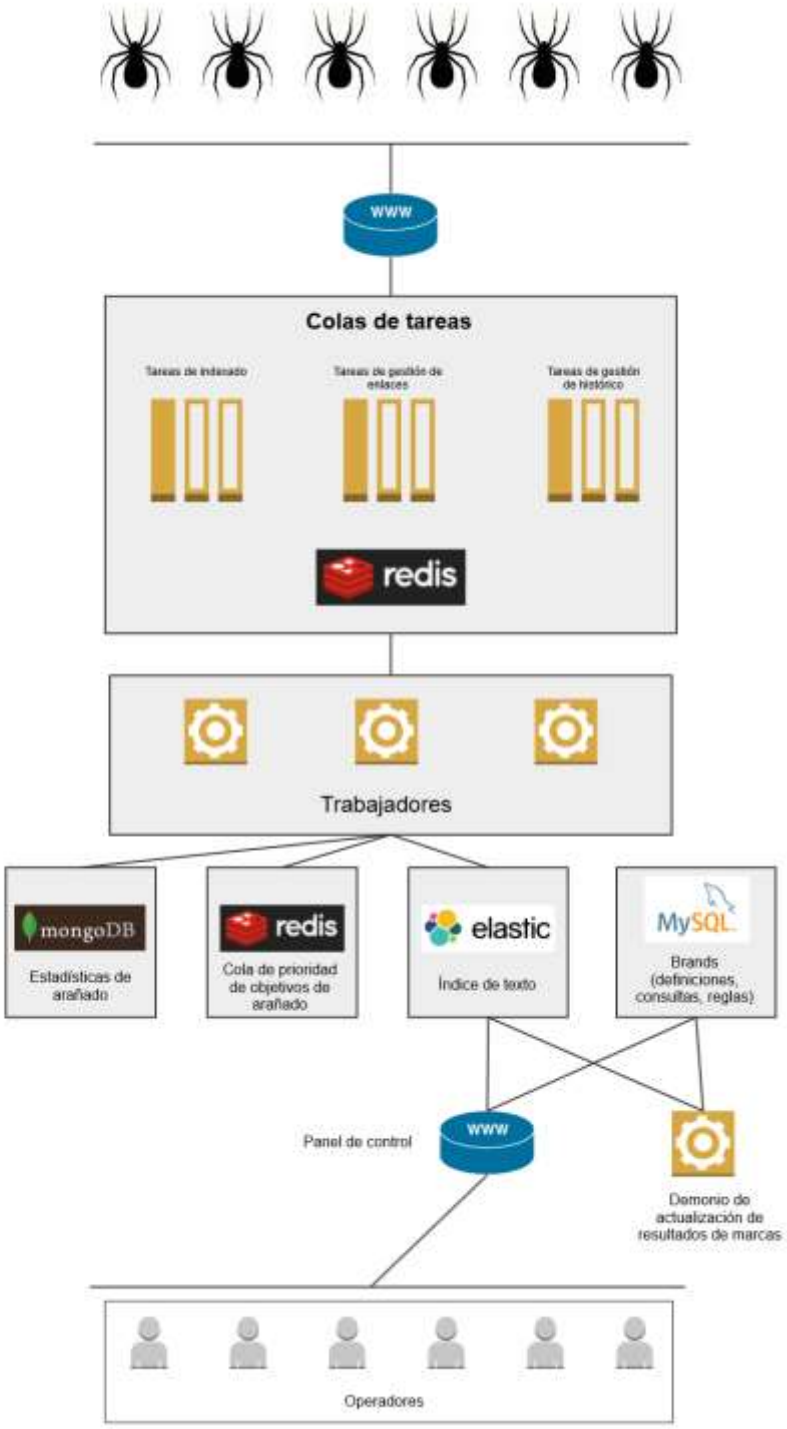


ILUSTRACIÓN 7 - ARQUITECTURA DEL SISTEMA

El punto de entrada al nodo coordinador es una api REST, encargado de comunicarse con las arañas para la asignación de tareas y recuperación de resultados. El protocolo de este intercambio se verá más adelante.

Para optimizar el tiempo que las arañas pierden en la comunicación con el coordinador, cualquier tarea que este último deba realizar que pueda ser extensa en tiempo, como puede ser añadir resultados al índice de texto, calcular las prioridades de los enlaces obtenidos y añadirlos a la cola, o actualizar las estadísticas de arañado de una URL cuyo escaneo ha sido completado, se encola para ser ejecutada por otro proceso. Para ello se utiliza Redis como cola de tareas, teniendo una cola distinta para cada una de las actividades mencionadas.

Se encolan tareas en las siguientes ocasiones:

- Al recibir texto para indexar, se crea una tarea de indexado.
- Al recibir enlaces extraídos, se crea una tarea de cálculo de prioridades para los mismos.
- Al recibir resultados de una URL, se crea una tarea de actualización de sus estadísticas de arañado.

Estas colas son procesadas por procesos cuya única finalidad es ejecutar las tareas encoladas, en un bucle infinito, esperando a tareas nuevas si la cola se vacía.

Estos trabajadores interaccionan con las distintas bases de datos que soportan el sistema:

- Índice de texto
- Base de datos de estadísticas de arañado
- Cola de objetivos

Hasta aquí se ha expuesto el sistema de exploración del proyecto. Para la explotación de la información recolectada, se dispone de un panel de control web que implementa las funcionalidades enumeradas anteriormente.

El panel interacciona con el índice de texto para obtener resultados en el panel de búsqueda, y con una base de datos relacional en la que se almacenan los datos de brands configuradas.

Para la actualización de los resultados expuestos en el panel a partir de las consultas de los brands, un demonio se encarga de ejecutarlas periódicamente, incorporando los resultados nuevos a la base de datos relacional, obteniendo dichos resultados del índice de texto.

DEFINICIÓN DE TECNOLOGÍAS EN COMPONENTES

BASE DE DATOS DE TAREAS

Se requiere una base de datos con soporte para colas de prioridad que sea capaz de trabajar a gran velocidad tanto para añadir como para extraer elementos.

Se ha optado por Redis por los siguientes motivos:

- Trabaja en memoria, permitiendo persistencia periódica en disco
- Soporte nativo para colas
- Buena integración con Python
- Utilizado en otros de los componentes del sistema

TRABAJADORES

Se trata de un programa independiente que extrae tareas de la base de datos descrita previamente para ser ejecutadas. En el momento de escribir este documento, el sistema es estable utilizando un proceso dedicado a cada una de las tres colas definidas en el apartado anterior.

Se utiliza la librería *rq*. Sus características son:

- Desarrollada en Python, por lo que su integración con el proyecto es automática.
- Permite empaquetar la referencia a una función definida por el usuario junto a sus parámetros en una tarea y encolar la misma para su posterior procesado.
- Incluye el programa *rqworker*, que al ser ejecutado, extrae y ejecuta tareas de la cola en Redis pasada como argumento. La integración, por tanto, con la base de datos de tareas viene de serie.

Las funciones a encolar por *rq* se localizan en el mismo archivo. Una tarea ejemplo puede tener, conceptualmente, la forma:

{ “function”: “some.package.function”, “args”: [“arg_1”, “arg_2”] }

rq importa el fuente de Python referenciado por el campo *function*, buscando la función definida, por lo que es necesario establecer el entorno correctamente (la variable PYTHONPATH). Al importar el archivo de manera estándar, se pueden establecer una única vez los elementos necesarios para el funcionamiento de las tareas, como pueden ser las conexiones a las bases de datos, en lugar de hacerlo en cada ejecución de la función que realiza la tarea.

BASE DE DATOS DE ARAÑADO

Contiene documentos JSON similares a los descritos para la gestión de los datos de arañado en la versión prototipo, añadiendo las fechas de último acceso, ya que son utilizadas en el cálculo de prioridad:

```
{
    "domain":"domain.onion",
    "last_access":"1/2/2015",
    "count":42,
    "paths":[
        {
            "path":"/test_path_1",
            "last_access":"1/2/2015",
            "count":19,
            "params":[
                {
                    "paramlist":"x,y",
                    "last_access":"1/2/2015",
                    "count":13,
                    "values":[
                        "1,2",
                        "a,b"
                    ]
                }
            ]
        }
    ]
}
```

ILUSTRACIÓN 8 - DOCUMENTO MONGODB, ACCESO A SITIOS

Se ha optado por utilizar una base de datos NoSQL con esquema no fijo como es MongoDB. Las ventajas que supone frente a una base de datos relacional para este caso de uso son:

- Esquema no fijo. Permite flexibilidad a la hora de añadir o eliminar campos de información si en el futuro se modifica el algoritmo que usa estos datos o se necesita almacenar más información.
- Es escalable: de ser necesario, añadir nodos para distribuir el almacenamiento y la carga de trabajo es sencillo.
- No es necesario que cumpla con ACID: podemos aceptar cierto nivel de error siendo el resultado una mínima repetición de tareas de arañado.

Se ha creado un índice en el campo *domain* de cada documento, puesto que es el único utilizado para buscar documentos.

El acceso a esta base de datos se hace de manera idéntica a como lo hacía la versión prototipo. Cuando se completa el análisis del contenido de una URL, se actualiza la información para el dominio en cuestión, incrementando los valores *count* del mismo,

del subdocumento asociado a la ruta accedida, y del subdocumento asociado a los parámetros GET utilizados.

El servidor MongoDB se ha configurado con sus parámetros por defecto, salvo por el motor de almacenado. Se ha optado por utilizar WiredTiger, ya que aporta las siguientes funcionalidades:

- Compresión de datos: la información se almacena comprimida en disco por defecto. En el momento de escribir este documento, la base de datos cuenta con más de 30000 documentos, de los cuales se utiliza una pequeña parte en un periodo de tiempo corto. Dichos documentos llegan a ocupar varios megabytes en dominios que han sido explorados en profundidad, conteniendo por lo general cadenas o subcadenas de texto repetidas que son susceptibles de reducción al pasar por un algoritmo de compresión. Por tanto, el ahorro de almacenamiento compensa el escaso retardo que la operación de compresión / descompresión genera.
- Bloqueo de base de datos a nivel de documento: con el motor de almacenado clásico utilizado por MongoDB, las operaciones de escritura sobre un documento bloquean la colección entera que lo contiene. Esto genera fácilmente problemas de rendimiento de manera innecesaria, ya que los documentos almacenados nunca hacen referencia unos a otros. Con WiredTiger se introduce bloqueo a nivel de documento, con lo que el rendimiento aumenta y podemos tener varios trabajadores actualizando información de la colección de estadísticas de arañado sin tener que esperar unos a otros. Esto cobra aún mayor importancia en los documentos que contienen los datos de arañado de dominios muy recorridos por la araña, que al ocupar varios megas pueden tener cierto retardo al actualizarse.

COLA DE OBJETIVOS DE ARAÑADO

Es una cola de pares (prioridad, url) ordenada según la prioridad, donde url ha de ser un valor único en la cola.

Se utiliza, de nuevo, Redis, dado que cuenta con un tipo nativo que cumple bien con las necesidades: los sets ordenados.

Este tipo de datos permite mantener un *set* de datos, donde cada dato se asocia a una prioridad, y se ordena según la prioridad dada. Si se añade un dato que ya existe en el *set*, se actualiza su prioridad a la nueva.

La cola será accedida desde su elemento de menor prioridad, extrayendo elementos según sean asignados a arañas, y añadiendo elementos según se extraigan enlaces de sitios webs arañados.

ÍNDICE DE TEXTO

El objetivo es poder hacer búsquedas de texto sobre el contenido de los sitios web arañados. Para ello se evaluaron varios motores de búsqueda (Solr, ElasticSearch, Sphinx), considerando el más apropiado ElasticSearch, por los siguientes motivos:

- Fácil de instalar: basta con descomprimir el paquete y ejecutar los binarios para tener una instancia de ElasticSearch corriendo y lista para aceptar consultas.
- Escalable: es sencillo añadir nodos a la arquitectura y distribuir la carga entre ellos. ElasticSearch lo hace de manera automática.
- Fácil de configurar para casos de uso sencillos: soporta datos sin esquema fijo. Es posible empezar a trabajar sin tan siquiera hacer configuraciones en los índices, obteniendo resultados de búsqueda aceptables. Para el caso que nos ocupa, más adelante se describe la configuración utilizada.
- Api REST: compatible, por tanto, con cualquier lenguaje de programación, y utilizable (con cuidado, debido a la carencia de seguridad de la versión gratuita) directamente desde el navegador vía JavaScript.
- Consultas muy versátiles: es posible hacer consultas formando combinaciones de cualquier número de campos, utilizando tipos de búsqueda distintos en cada uno de ellos, y pudiendo configurar el peso de cada elemento para ordenar de manera óptima los resultados obtenidos.
- Muy rápido: con los suficientes recursos hardware, los resultados son en tiempo real.

Se ha definido un índice llamado *crawl*. Dado que los índices de ElasticSearch son de tamaño limitado, es deseable poder cambiar de índice a la hora de añadir contenido de manera transparente a los usuarios. Para ello, se hace uso de la funcionalidad de *alias*. Ésta permite crear referencias a índices con distintos nombres, pudiendo un alias referenciar varios índices simultáneamente. La implementación es:

1. Crear un índice llamado *crawl_1*
2. Crear un alias llamado *crawl_index* apuntando a *crawl_1*. Será el nombre que los usuarios de la base de datos conozcan a la hora de añadir contenido.
3. Crear un alias llamado *crawl_search* apuntando a *crawl_1*. Será el nombre que los usuarios de la base de datos conozcan a la hora de hacer búsquedas.
4. En el momento en el que *crawl_1* se llena, crear un nuevo índice llamado *crawl_2*
5. Actualizar el alias *crawl_index* para que apunte a *crawl_2*. Esto permite seguir añadiendo contenido de manera transparente e ininterrumpida.
6. Actualizar el alias *crawl_search* para que apunte a *crawl_1* y *crawl_2*. De este modo, las búsquedas abarcan todo el contenido disponible.
7. En el momento en el que *crawl_2* se llene, repetir desde el paso 4 incrementando el contador.

ElasticSearch permite definir analizadores para utilizar en los campos y en las búsquedas. Un analizador está compuesto por tres etapas, donde cada etapa puede estar formada por varios elementos encadenados[1]:

ILUSTRACIÓN 9 - ETAPAS DE PROCESADO EN ELASTICSEARCH

- Filtro de caracteres: acepta una cadena de entrada y permite la transformación de caracteres en otras secuencias.
- Tokenizador: a partir de la cadena de texto, extrae las palabras o tokens individuales que serán utilizadas para construir el índice.
- Filtro de tokens: a partir de cada token recibido del paso anterior, permite eliminar, modificar o añadir tokens extra que serán utilizados de la misma manera que el original.

A la hora de definir el analizador hay que tener en cuenta que:

- El texto a indexar puede estar en cualquier idioma.
- El texto a indexar puede estar en cualquier conjunto de caracteres.
- No se quiere implementar función de autocompletado en la búsqueda.

Para el primer punto, se evita utilizar cualquier filtro que tenga en cuenta cuestiones semánticas, como pueden ser los encargados de sustituir palabras en plural por su singular. Esto es debido a que para poder aprovechar estas funcionalidades, es necesario detectar previamente en qué idioma está el texto original, función que no se ha implementado. De querer hacerse, sería necesario crear un índice por idioma y aplicar un analizador distinto a cada uno utilizando los filtros semánticos pertinentes.

Para el segundo punto, se utilizan los filtros de la familia icu_*. Permiten normalizar el texto desde cualquier formato y juego de caracteres, incluso a la hora de tokenizar.

El tercero implica que se omite la definición de analizadores destinados a búsqueda incremental, mediante *shingles* (secuencias de tokens) o *n-grams* (subsecuencias de caracteres extraídas de cada token)

También mencionar que ElasticSearch permite utilizar analizadores distintos en el indexado y en la búsqueda. Esto es útil si se ha implementado la funcionalidad descrita en el punto anterior, ya que operaciones como crear subsecuencias son útiles al actualizar el índice, pero no al analizar una cadena de texto que forma una consulta. En nuestro caso, se utiliza el mismo analizador para las dos operaciones.

[1] https://www.found.no/foundation/text-analysis-part-1/

```
"web_analyzer_search": {
    "type": "custom",
    "filter": [
        "icu_normalizer",
        "stop_filter",
        "icu_folding"
    ],
    "tokenizer": "icu_tokenizer"
},
"web_analyzer_index": {
    "type": "custom",
    "filter": [
        "icu_normalizer",
        "stop_filter",
        "icu_folding"
    ],
    "tokenizer": "icu_tokenizer"
}
```

ILUSTRACIÓN 10 - CONFIGURACIÓN DE ANALIZADORES EN ELASTICSEARCH

La configuración de los documentos indexados cuenta con los siguientes puntos:

- Campo *all* desactivado. Es un campo que se forma automáticamente concatenando todos los campos del documento. Puesto que no se ha considerado interesante buscar de esa manera, se desactiva esta función, ahorrando espacio.
- El identificador del documento es el campo *url_hash*. Esto sirve para localizar un documento de manera unívoca dentro del índice. Dado que debe haber un único documento por cada URL, se ha creído provechoso que el campo identificador sea un valor derivado de la misma. Esto permite buscar documentos de manera ágil al hacer una actualización del contenido tras una URL.
- El campo *source* (que contiene una copia del documento original indexado) contiene todos los campos del documento.

Cada documento indexado tiene los siguientes campos:

- content: contenido textual del sitio arañado. Es de tipo *string* y se le aplica el analizador definido previamente.
- title: título extraído del contenido. De tipo *string*, mismo analizador que *content*.
- first_found: timestamp del momento en el que se arañó la URL por primera vez.
- last_check: timestamp del momento en el que se arañó la URL por última vez.
- available: valor binario que representa si se pudo acceder a la URL la última vez que se intentó o no.

- url: dirección web arañada.
- url_hash: hash del campo *url*.

BASE DE DATOS DE BRANDS

Almacena los parámetros de configuración para la monitorización de los resultados del arañado, y los resultados de dicha monitorización.

Se trata de datos con un esquema fijo, y que no se prevé vaya a cambiar a menudo. Es importante su consistencia, ya que pueden ser modificados por varias personas simultáneamente. Se ha optado por utilizar un sistema de base de datos tradicional SQL, en concreto, MySQL.

El esquema es el siguiente:

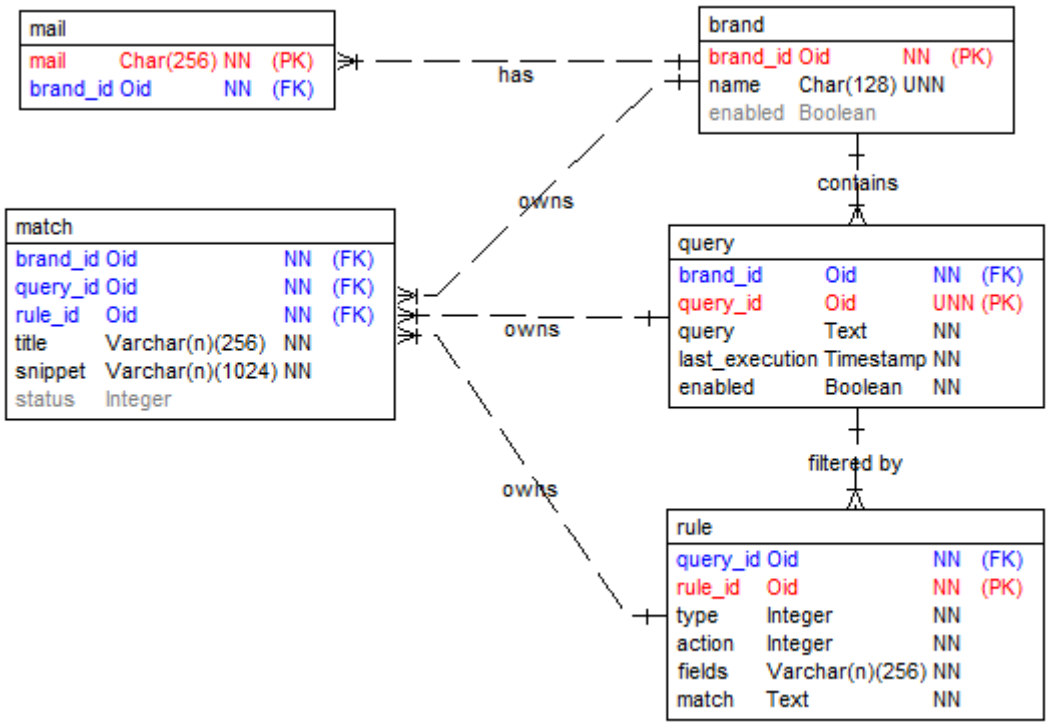


ILUSTRACIÓN 11 - ESTRUCTURA SQL DE LA BASE DE DATOS DE BRANDS

Para cada brand puede haber cero o más direcciones de correo a las que enviar notificaciones. Así mismo, un brand contiene cero o más consultas que se realizarán periódicamente a la base de datos de indexado. Cada consulta puede estar filtrada por cero o más filtros, que modifica el modo en el que se almacenan las resultados encontrados por las consultas.

Un brand con el campo *enabled* con valor *false* no ejecuta ninguna de sus consultas asociadas. Se pueden desactivar consultas individualmente.

Los filtros, tal y como se describió en la especificación del proyecto, hacen que los *matches* modifiquen su estado, ya sea en el primer momento en el que se localizan, o con efecto retroactivo al definir una nueva regla.

No ha sido necesaria una configuración específica del servidor MySQL. Se ha creado una base de datos para el sistema, y dos usuarios:

- query_daemon: será el usuario utilizado por el demonio de actualización de resultados. Tiene permiso de lectura en la tabla de consultas, y de escritura en la de resultados.
- web: es el usuario utilizado por el panel de control. Debe tener acceso de lectura y escritura en todas las tablas, ya que se muestra información de todas ellas, y el panel permite insertar o modificar valores.

ARAÑA DISTRIBUIDA

Este apartado expone el diseño del sistema distribuido que forma la araña. Se tratarán individualmente el nodo maestro o coordinador y la araña o esclavo, ejecutada en múltiples hosts, así como el protocolo de comunicación que implementan.

MAESTRO

Coordina los esfuerzos de las arañas individuales, interaccionando con todas las bases de datos involucradas.

El punto de acceso es una API REST, con los siguientes puntos de acceso:

POST /login
Las arañas han de autenticarse para poder interaccionar con el maestro. Este punto de acceso recibe las credenciales y devuelve una cookie firmada que representa la sesión.

GET /targets
Devuelve las N primeras URLs de la cola de prioridad.

POST /results
Subida de resultados por parte de los esclavos.

La API se ha creado en Python, utilizando la librería Flask.

Se trata de una librería de Python destinada a la creación de sistemas web. Es ligera, se basa en otras librerías especializadas para cumplir cada una de sus funciones, y muy simple de utilizar. Puesto que no se trata de un panel web complejo ni requiere gran funcionalidad, se ha considerado la opción más acertada, frente a frameworks más complejos como Django.

Está formado por dos componentes y código propio para hacer de puente:

- werkzeug: es una librería para crear servicios HTTP. Se encarga del manejo del protocolo HTTP y la interacción con WSGI.
- Jinja2: es un motor de plantillado. Permite crear plantillas programables que serán transformadas según los parámetros que se le pasen.

A partir del punto de acceso HTTP, el maestro utiliza las siguientes clases para cumplir su función:

TARGETQUEUE

Se encarga de la cola de prioridad de objetivos de arañado, interaccionando con la base de datos Redis.

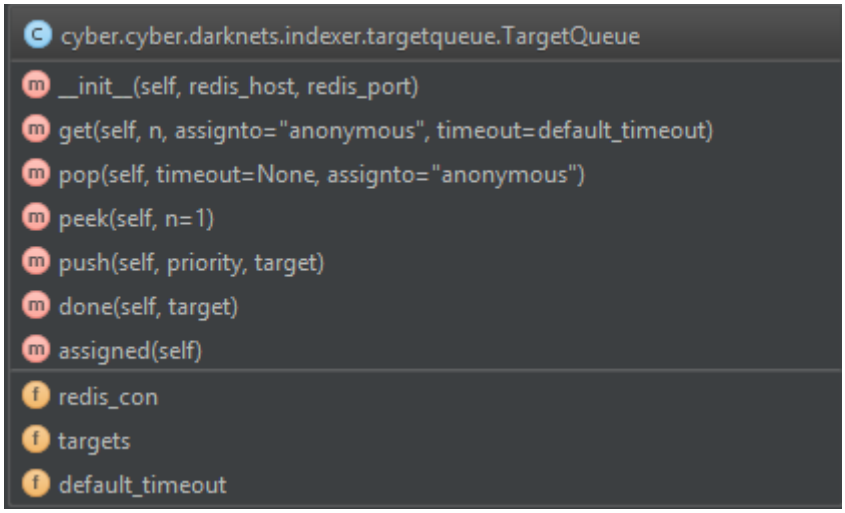


ILUSTRACIÓN 12 - UML, TARGETQUEUE

Incluye los métodos típicos de una cola, más algunos propios :

- push (prioridad, objetivo)
  Añade un objetivo a la cola con una prioridad dada.
- pop (timeout, assignto)
  Desencola el objetivo de mayor prioridad, reencolándolo si no se ha devuelto usando el método *done()* antes de los segundos dados por el parámetro *timeout*. Si no se especifica a qué araña se ha asignado, se utiliza la araña inexistente *anonymous*. Si no se especifica *timeout*, el objetivo no será reencolado en el caso de que la araña a la que se le había asignado no lo devuelva nunca.
- peek (n)
  Devuelve los *n* objetivos más prioritarios de la cola, sin desencolarlos.
- get (n, assignto, timeout)
  Devuelve los *n* objetivos más prioritarios, asignándolos a la araña denominada por el parámetro *assignto* con tiempo de caducidad *timeout*.
- assigned ()

Devuelve la lista de objetivos asignados en un momento dado.

- done (target)
  Marca un objetivo asignado como completado, evitando que sea reencolado cuando su *timeout* expire.

Delega el manejo de la cola de prioridad de Redis en un objeto de la clase *PriorityQueue* de la librería *qr* (el campo *targets* del diagrama).

El campo *redis_con* es una conexión a Redis, utilizada para la gestión de objetivos asignados.

Para detectar la expiración de un *timeout* de un objetivo, se utilizan los eventos de Redis. Cuando se añade un objetivo, se crea una clave con la forma assigned:*nombre_de_araña*:*prioridad*:*objetivo* en Redis, aplicando un tiempo de expiración de la clave. A partir de la versión 2.8.0, Redis añade soporte para notificaciones de eventos. Una de las notificaciones que se puede activar es la de expiración de clave. Con ella, y creando un monitor de eventos de expiración, podemos detectar cuándo un objetivo ha caducado. El tiempo de envío de la notificación no es ni mucho menos preciso, pero no es necesario que lo sea.

El monitor de eventos expirados recibe mensajes representando los eventos. En el campo *data* del mensaje se encuentra el nombre de la clave expirada, con la forma assigned:*nombre_de_araña*:*prioridad*:*objetivo.* A partir de esta cadena puede, por tanto, reencolar el objetivo con la misma prioridad con la que fue encolado originalmente, y saber qué araña no ha devuelto el resultado de su tarea asignada.

STATMANAGER

Gestiona la base de datos de estadísticas de arañado, utilizada para el cálculo de prioridad de los objetivos.

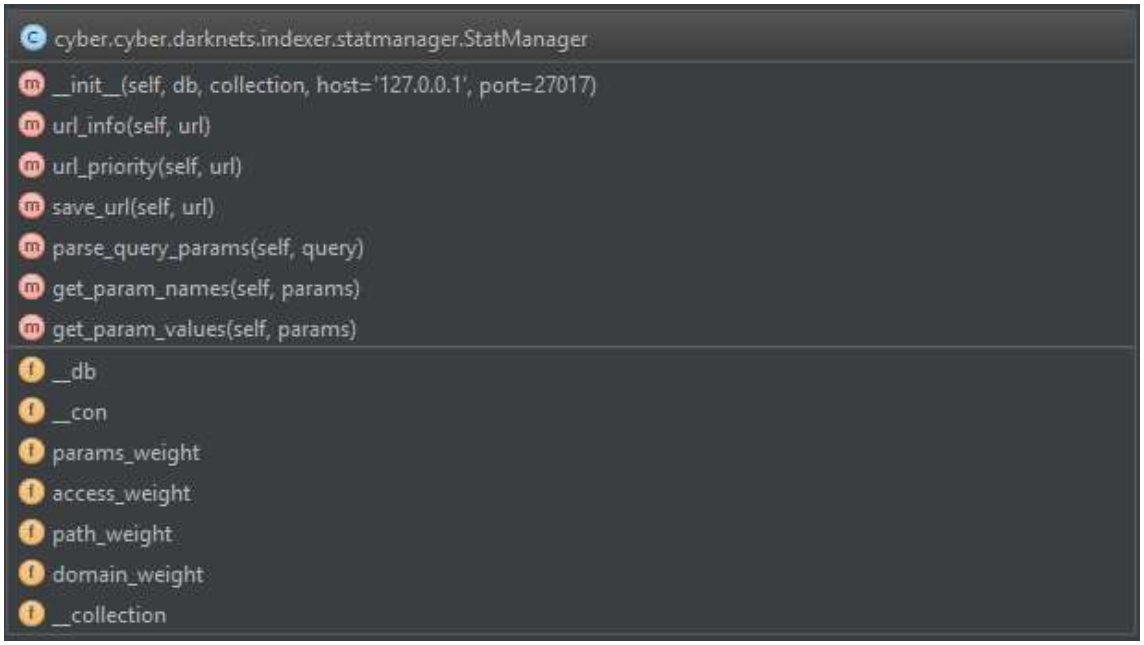


ILUSTRACIÓN 13 - UML, STATMANAGER

Su constructor recibe los parámetros de conexión a MongoDB. Sus métodos son:

- url_priority (url):
  Devuelve la prioridad para una URL. Para ello, divide la URL en:
  - dominio
  - ruta
  - parámetros

  Para cada uno de ellos, se consulta en la base de datos cuántas veces se ha accedido (X veces en total al dominio, Y veces a la ruta del dominio en cuestión, Z veces con los mismos parámetros). Cada uno de estos tres valores tiene un peso, Wx, Wy y Wz.

  La prioridad es el resultado de (X * Wx + Y * Wy + Z * Wz) / d, donde d es (timestamp_actual – timestamp_ultimo_acceso) * Wd.

- save_url (url)
  Actualiza los datos de acceso para una URL, incrementando los contadores de acceso y los timestamps.
- url_info (url)
  Dada una URL, extrae el dominio, ruta y parámetros.
- parse_query_params (query)

Dada la cadena de parámetros GET de una URL (por ejemplo, param_1=123&param3=456&param_2=789), devuelve una lista de diccionarios con la forma {“nombre”: x, “valor”: y} ordenada alfabéticamente por nombre de parámetro (para el ejemplo anterior, devolvería [{“nombre”: “param_1”, “valor”: 123}, “nombre”: “param_2”, “valor”: 789 }, “nombre”: “param_3”, “valor”: 456 }])

- Get_param_names / get_param_values (params)
  Devuelve una lista de los valores de los campos “nombre” o “valor” en la lista devuelta por *parse_query_params*.

Los campos de la clase contienen las prioridades mencionadas y los parámetros de conexión a MongoDB.

El flujo de trabajo, desde una perspectiva de alto nivel (ya que las tareas son encoladas y procesadas por otro proceso independiente), es el siguiente:

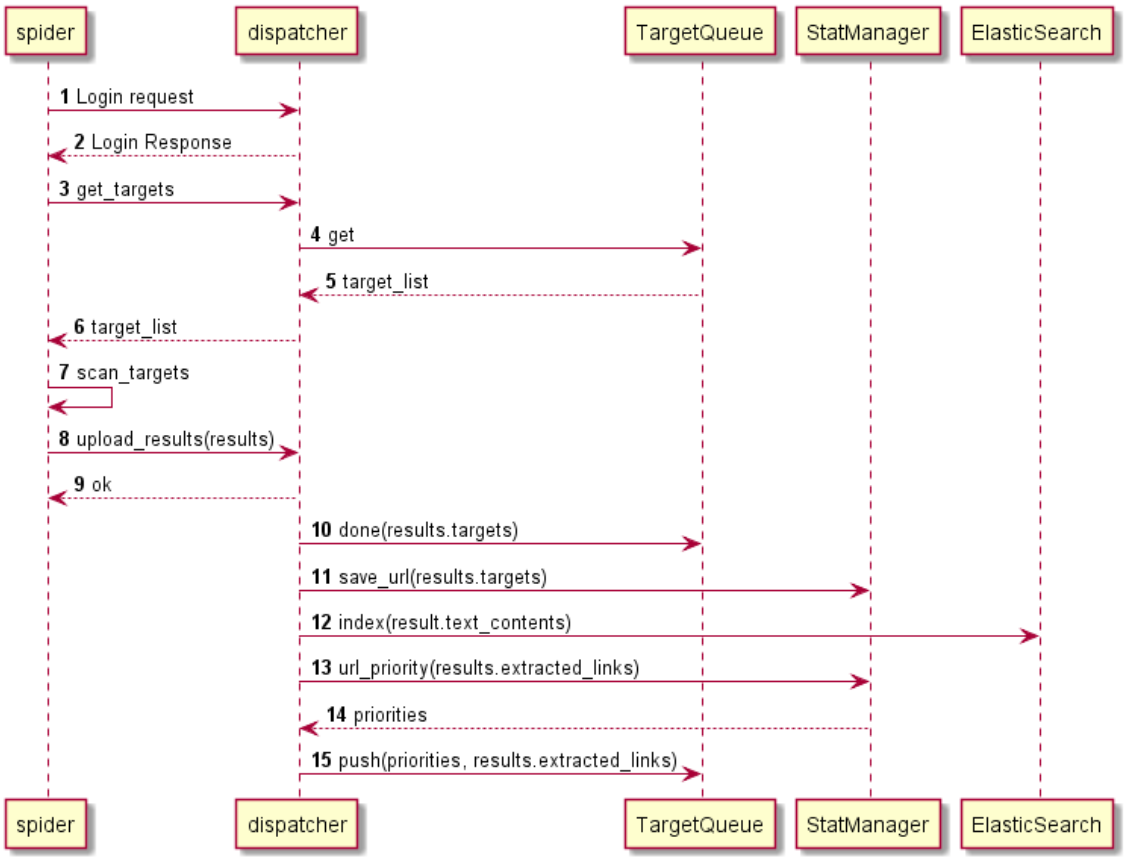


ILUSTRACIÓN 14 – UML SECUENCIA, FLUJO DE TRABAJO

El diagrama es fiel a la realidad entre los pasos 1 y 6, puesto que la petición de objetivos, seguida de la extracción de los mismos de la cola de prioridad y su devolución son tareas síncronas.

El paso 7 será descrito más adelante, en el apartado dedicado a los nodos esclavos.

A partir del paso 9, las tareas no son secuenciales. El funcionamiento es:

- Marcar como completados los objetivos de la cola utilizando el método *done* de TargetQueue (paso 10).
- Encolar una tarea para que StatManager actualice la información de los objetivos visitados pasando los mismos al método save_url (paso 11).
- Encolar una tarea para añadir el contenido de texto extraído a ElasticSearch (paso 12).
- Encolar una tarea para obtener las prioridades de los enlaces extraídos usando el método *url_priority* de StatManager, y encolarlos con dicha prioridad utilizando el método *push* de TargetQueue (pasos 13, 14 y 15, secuenciales).

Las tres tareas serán procesadas por procesos distintos en cualquier orden.

NODO ESCLAVO

Realizan el trabajo de acceso a los sitios web, seguido de la extracción de enlaces y texto del contenido descargado. Obtienen enlaces a descargar del maestro y reportan los resultados de vuelta.

El nodo esclavo requiere de un punto de acceso a las darknets. Es posible configurar proxys HTTP de acceso para cada una de ellas, usando la forma de la URL para determinar cuál usar. Para la implementación, se ha usado una instancia de Privoxy configurada para redirigir peticiones a la red TOR.

En cada araña es necesaria una de estas dos configuraciones:

- Punto de acceso externo a las darknets.
- Punto de acceso local a las darknets (instalación de los clientes, instalación del Proxy HTTP intermediario).

Al utilizar la librería Scrapy, diseñada como librería de programación asíncrona, no es posible pensar el flujo de trabajo del programa como una serie de bucles, sino como una serie de funciones que se llaman consecutivamente recibiendo la salida de la función anterior como parámetro en respuesta a eventos, como pueden ser la finalización de una tarea, una conexión, o una función. Se mantienen dos ramas paralelas de cadenas de funciones, la cadena de *callbacks* y la cadena de *errorbacks*, utilizadas para gestionar el flujo normal del programa y el flujo en condiciones de

error, respectivamente. El programa comienza añadiendo un *callback* con el que se iniciará la ejecución y lanzando el *reactor* de Twisted, encargado de, entre otras cosas, gestionar el flujo de ejecución.

Está formado por una clase principal y varias subclases:

CRAWLSLAVE

Esta es la clase principal. Su constructor recibe las credenciales de autenticación con el maestro, el nombre de la araña, y la ruta en la que se encuentra la API de comunicación HTTP en el maestro.

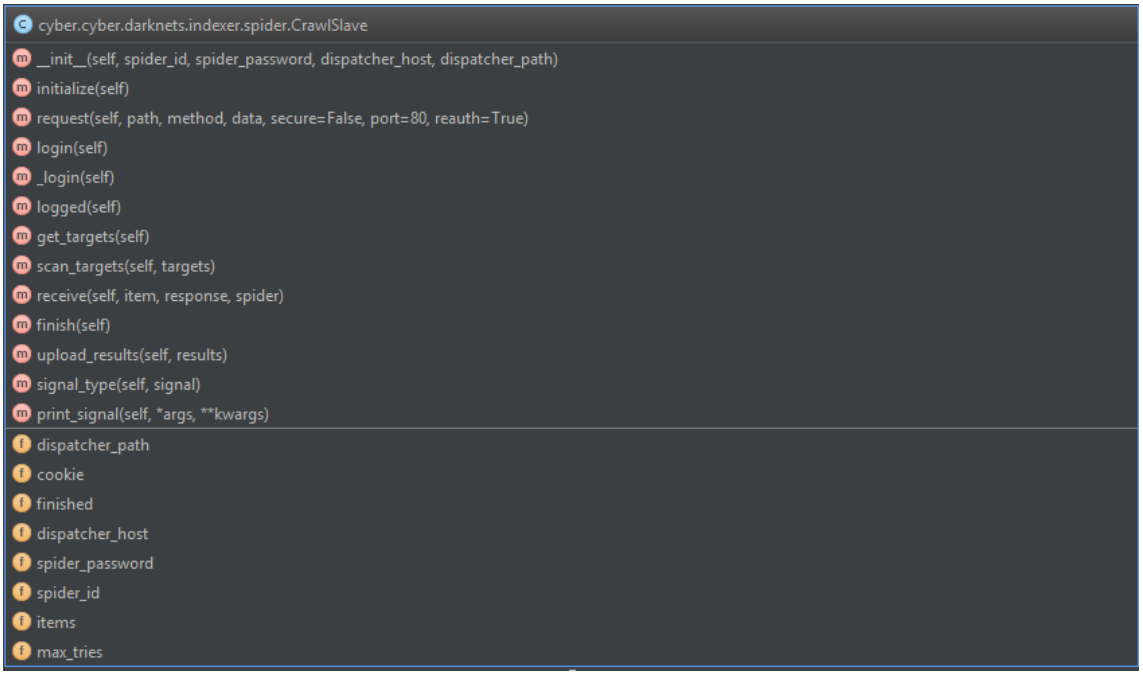


ILUSTRACIÓN 15 - UML, CRAWLSLAVE

Sus métodos son:

- initialize ()
  Primer *callback* a ejecutar por el programa. Actualmente, simplemente encadena con el método login ()
- login ()
  Llama a _login () y encadena con get_targets ().
- _login ()
  Utiliza el método request () para lanzar una petición de autenticación al maestro. En caso de error, levanta una excepción de autenticación.
- request (path, method, data, secure, port, reauth)
  Encargado de realizar peticiones HTTP al maestro. Recibe como parámetros a qué ruta (encadenada con la ruta de acceso almacenada en la clase), el método HTTP a usar, un flag booleano que indica si se debe usar HTTPS, el puerto de acceso y un flag booleano que indica si se debe intentar reautenticar contra el

maestro si la petición devuelve un código HTTP que requiera autenticación. Esto puede ocurrir si la cookie de sesión ha expirado porque se ha tardado más tiempo del previsto en completar las tareas de arañado asignadas. Si se debe reautenticar, se utilizará el método _login ().

- logged ()
  Simplemente comprueba que el valor del campo *cookie* no sea vacío. Cuando se hace una petición vía request (), se actualiza el valor de dicho campo. Si se detecta que la sesión ha expirado, se deja en blanco.
- get_tatgets ()
  Obtiene objetivos de arañado del maestro utilizando request (). Si la petición es satisfactoria y no se ha recibido una lista vacía de objetivos, encadena con scan_targets (). Si se ha obtenido una lista vacía, se espera 20 segundos antes de encadenar consigo mismo. Si se detecta un error de sesión, se llama a _login () y posteriormente se encadena con scan_targets () de nuevo.
- scan_targets ()
  Se encarga de inicializar la araña de Scrapy con los parámetros necesarios para su correcto funcionamiento. Así mismo, crea una nueva instancia de *ItemReceiver* (utilizado para recibir y almacenar los resultados devueltos por la araña, ver más adelante).
  Scrapy permite configurar manejadores ante algunos de sus eventos. Los dos de ellos que nos interesan son:
  - item_scraped: levantado cuando la araña ha finalizado de arañar y procesar un objetivo. Se configura el método *receive (ítem, response, spider)* como manejador de esta señal.
  - spider_closed: levantado cuando la araña ha terminado su ejecución. Lo encadenaremos al método *finished ()*.

  De este modo, podemos controlar los resultados obtenidos y continuar nuestro flujo de trabajo cuando se ha finalizado la tarea de arañado a los objetivos.
- receive (ítem, response, spider)
  Llamado por la araña de Scrapy cuando se obtiene un resultado. Recibe por su parte, además de un objeto de alguna subclase de scrapy.item.Item (en nuestro caso, *ResultItem*) conteniendo toda la información que se ha querido extraer del objetivo, una referencia al objeto de respuesta HTTP de la petición y a la propia araña que lo ha llamado.
  Hace de simple intermediario al método receive (ítem) del objeto ItemReceiver creado al inicio de la tarea de arañado actual.
- finish ()
  Llamado por la araña al finalizar su labor. Se encarga de encadenar con el método upload_results, pasando como argumento los ítems recogidos por *ItemReceiver*.
- upload_results (results)

Utiliza request () para enviar los resultados al maestro, teniendo en cuenta posibles errores debidos a la caducidad de la sesión. En caso de éxito, encadena con get_targets (), comenzando de nuevo todo el ciclo. En caso de error, vuelve a encadenar consigo mismo.

La lista de resultados es la devuelta por el método get_items () de *ItemReceiver*.

DARKSPIDER

Hereda de scrapy.spider.Spider. Es una araña básica de Scrapy, modificada para extraer contenido de varios tipos de archivo e ignorar cualquier tipo de archivo multimedia.

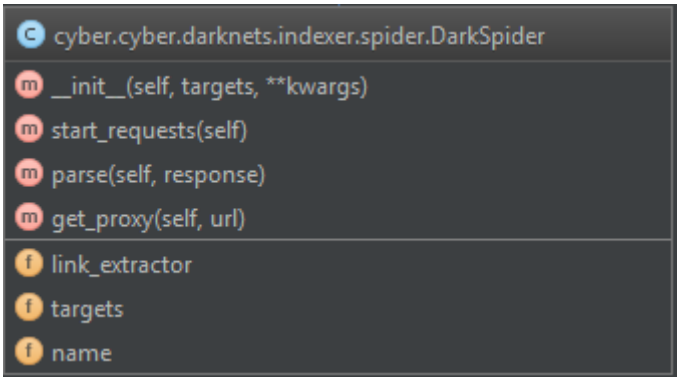


ILUSTRACIÓN 16 - UML, DARKSPIDER

Su constructor recibe la lista de objetivos a arañar. Se encarga de inicializar el extractor de enlaces (el dado por Scrapy) con los parámetros para ignorar:

- Enlaces fuera de las darknets soportadas.
- Enlaces con extensiones de tipo de archivos no deseados (multimedia).

Sus métodos son:

- start_requests ()
  Para cada enlace en *targets*, crea un objeto Request de Scrapy, asignando los parámetros necesarios para que el motor de acceso de Scrapy utilice el proxy correcto. Para ello se utiliza el método get_proxy (). Como manejador de la respuesta, se asigna el método parse ().
- get_proxy (url)
  A partir de una URL, utilizando una lista de pares (expresión regular, proxy), devuelve el proxy correcto para acceder a ella. Se utiliza para determinar el medio de acceso a cada tipo de darknet.
- parse (response)
  Es llamado por Scrapy cuando obtiene un objeto de clase Response HTTP al haber completado satisfactoriamente una descarga de contenido. Recibe el mismo objeto de tipo *Response*.

Su labor es extraer los datos oportunos de esta respuesta y devolver uno o varios objetos de alguna subclase de *Item* o *Requests* HTTP adicionales.
En nuestro caso, se utiliza para extraer la información del contenido descargado, sea del tipo que sea (siempre que contenga texto).
Se intenta hacer una detección inicial del tipo de archivo mediante la librería *magic*, similar al comando *file* de Linux. Si se detecta tipo HTML o XML se utiliza la librería lxml de Python para extraer, mediante la consulta XPath "//*[not(self::script or self::style)]/text()[normalize-space(.)]" , todo el contenido de texto del archivo, y mediante la consulta "//title/text()" el título, si es posible. Si es de cualquier otro tipo, se utiliza la librería *textract*, que a su vez utiliza varias librerías y programas externos para extraer el texto de archivos de gran cantidad de tipos distintos.
Para extraer los enlaces, se hace un primer intento con el extractor de enlaces de Scrapy. Si no se consiguen resultados, se busca con las expresiones regulares que cazan enlaces pertenecientes a las darknets soportadas en el contenido de texto extraído previamente.
Devuelve objetos de la clase ResultItem, conteniendo el contenido de texto, el título del documento (los primeros 50 caracteres en el contenido si no se ha encontrado un título explícito), y la lista sin repetidos de enlaces extraídos, la URL solicitada, y la URL a la que realmente se accedió (son distintas sólo en el caso de que la primera sea una redirección).

ITEMRECEIVER

Encargado de almacenar los *Items* encontrados por la araña. Extiende de la clase Item de Scrapy, debe ser generado por el método *parse* de una araña. Se pueden considerar los resultados de las labores de arañado.

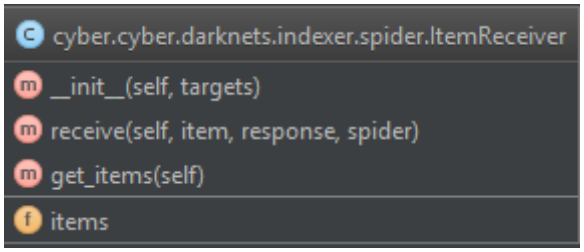


ILUSTRACIÓN 17 - UML, ITEMRECEIVER

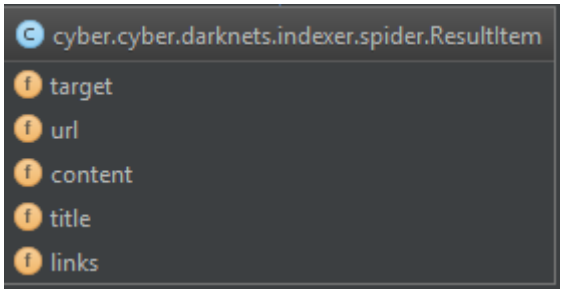


ILUSTRACIÓN 18 - UML, RESULTITEM

Se crea uno para cada tarea de arañado. Su constructor recibe la lista de objetivos, a partir de la cual crea un mapa de objetos con la forma:

```
{
    target : {
        "url": target,
        "target": target,
        "content": "",
        "title": "",
        "links": [],
        "available": False,
    },
    target2: { ... }
}
```

ILUSTRACIÓN 19 - ITEMRECEIVER, MAPA DE OBJETIVOS

Por defecto, se considera que una URL no estará disponible. Cuando la araña finaliza con éxito la descarga de un objetivo, tras pasar por parse () se crea un objeto de tipo *ResultItem*, que será reenviado al método receive () de *ItemReceiver*. Este obtendrá el objeto correspondiente del mapa a partir de la URL, y actualizará los campos *url*, *content*, *title*, *links* y *available*.

El método get_items() simplemente extrae los valores del mapa de objetivos, devolviendo la lista de resultados obtenidos.

PANEL DE CONTROL

Es la interfaz de uso de la herramienta. Incluye el panel de definición de *brands*, con toda su configuración según lo descrito en la especificación funcional, la visualización de los resultados obtenidos para cada uno de ellos, y un panel de búsqueda para explorar manualmente el índice de texto.

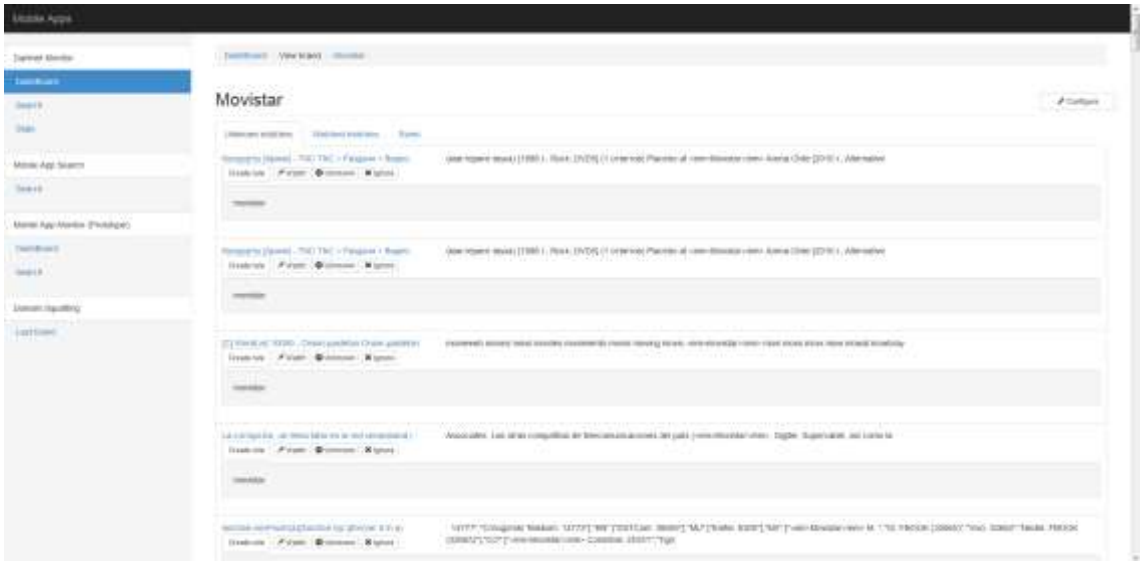
Se ha desarrollado con Python (mediante Flask, de nuevo) para el backend, y se ha usado el framework Bootstrap para el frontend. La ventaja de utilizar bootstrap es que permite crear interfaces web visualmente agradables y adaptativas con muy poco esfuerzo, ya que se trata de un conjunto de plantillas CSS que definen el comportamiento de los elementos contemplados por bootstrap, más un conjunto de scripts JS opcionales para algunas de las funcionalidades más avanzadas.

El panel muestra inicialmente un resumen de todos los *brands* dados de alta, en el que se indica el número de resultados desconocidos, controlados, y el número de consultas definidas:

El proceso para dar de alta un nuevo *brand* es el siguiente:

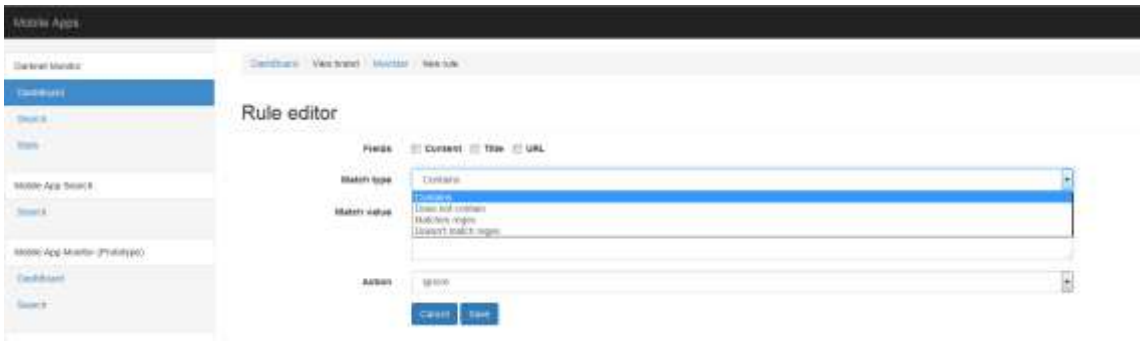
- Añadir el *brand* a la lista.
- Acceder a la configuración del *brand*.
    - Añadir consultas. Utilizar el buscador para una vista previa de los resultados.
    - Añadir emails de notificación a los que enviar un mensaje cuando aparezcan resultados nuevos.

- Visualizar los resultados obtenidos. Se muestra el título con el enlace al resultado, un extracto del contenido relevante que ha levantado el resultado como *match*, y la consulta que lo ha localizado. Se permite clasificar de manera manual cada resultado de manera individual.



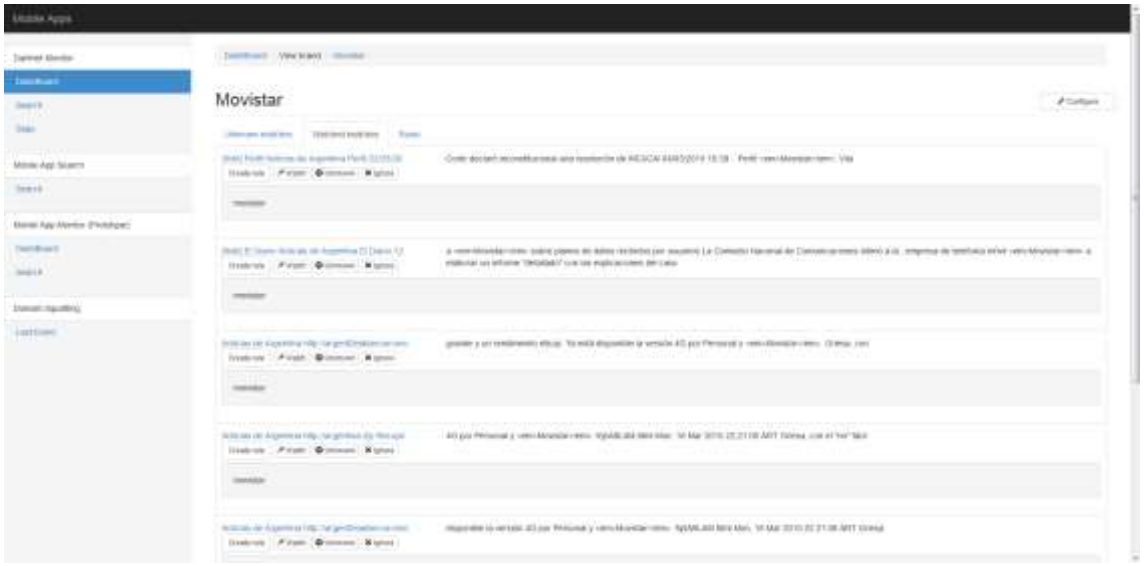
- Definir reglas para
    - Ignorar resultados que cumplan un patrón.
    - Añadir a la lista de control resultados que cumplan un patrón.

  Al definir una regla, su efecto se aplica tanto a los resultados que se encuentren en el futuro como a los ya existentes.

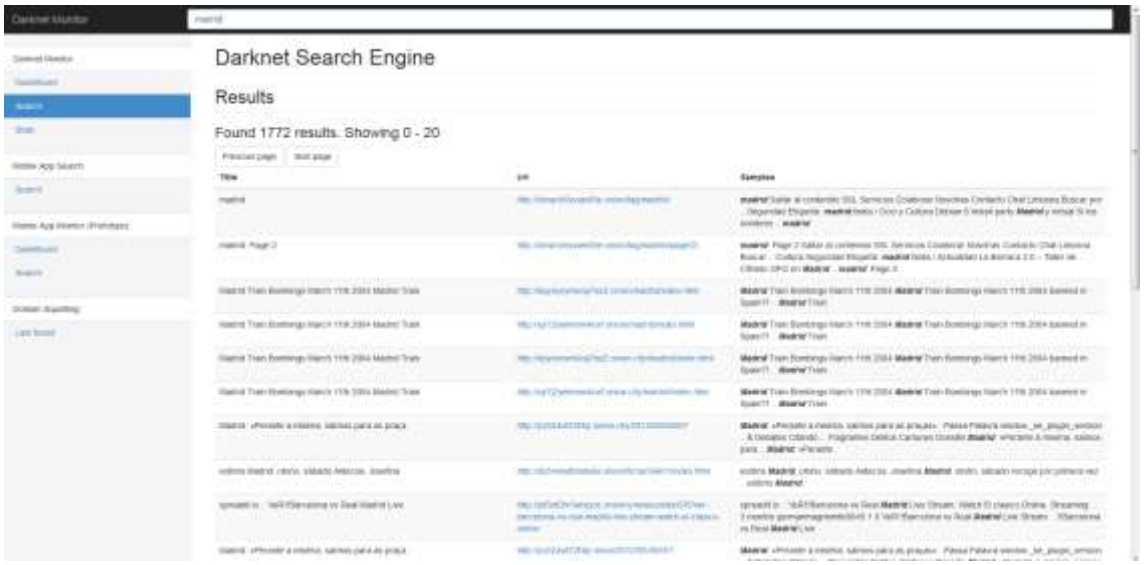


Por ejemplo, tras definir estas reglas:

Los resultados en el grupo *watched* son:



El panel de búsqueda es el siguiente:



Las consultas que permite son las mismas que permite ElasticSearch en sus “Query strings”. Su sintaxis es:

([OP][CAMPO:] <EXPRESIÓN>)+

OP es un modificador opcional. Puede tomar los valores:

- +: EXPRESIÓN debe cumplirse
- -: EXPRESIÓN no debe cumplirse

CAMPO es uno de los campos indexados en ElasticSearch. Permite, opcionalmente, limitar la condición de la consulta a uno de ellos. En nuestro caso, se encuentran contemplados los campos “title”, “url” y “content”.

EXPRESIÓN puede ser uno de los siguientes:

- Expresión regular: con la forma /expresión regular/
- Frase: con la forma “frase a buscar”. Permite definir el número máximo de palabras entre una palabra de la frase y otra añadiendo “~N” tras la frase. Si N==1, el resultado es el mismo que si no se especifica.
- Conjunto de EXPRESIONES: expresión booleana combinando otras EXPRESIONES. Se utilizan paréntesis para agrupar miembros, y los operadores AND y OR (o && y ||) para combinarlos.

Hay algunos caracteres especiales:

- * : significa “cualquier cosa”. Si se introduce en una palabra, ElasticSearch la expandirá a cualquier otra que mantenga lo que haya antes y después, con cualquier combinación sustituyendo el asterisco. No conviene utilizarlo, puesto que ElasticSearch hace una expansión real en memoria a lo largo del índice y es

muy costoso. Por ejemplo, para la búsqueda "r*", ElasticSearch buscaría primero todas las ocurrencias en el índice que comenzasen por la letra "r".

- ? : significa "cualquier carácter": Sirve para reemplazar un único carácter de una palabra con cualquiera, generando los resultados de búsqueda apropiados. Para la búsqueda "c?sa", por ejemplo, podría expandir a "casa" y "cosa".
- ~N tras una palabra: permite hacer búsquedas *fuzzy*, es decir, permitir errores tipográficos. Añadir éste operador a una palabra hace que ElasticSearch busque en su índice todas aquellas que son idénticas salvo por N caracteres.

Los resultados mostrados son actualizados por un demonio que consulta periódicamente la base de datos de consultas almacenadas, ejecutándolas sobre el indexador de texto y teniendo en cuenta las reglas definidas. El flujo de trabajo es el siguiente:

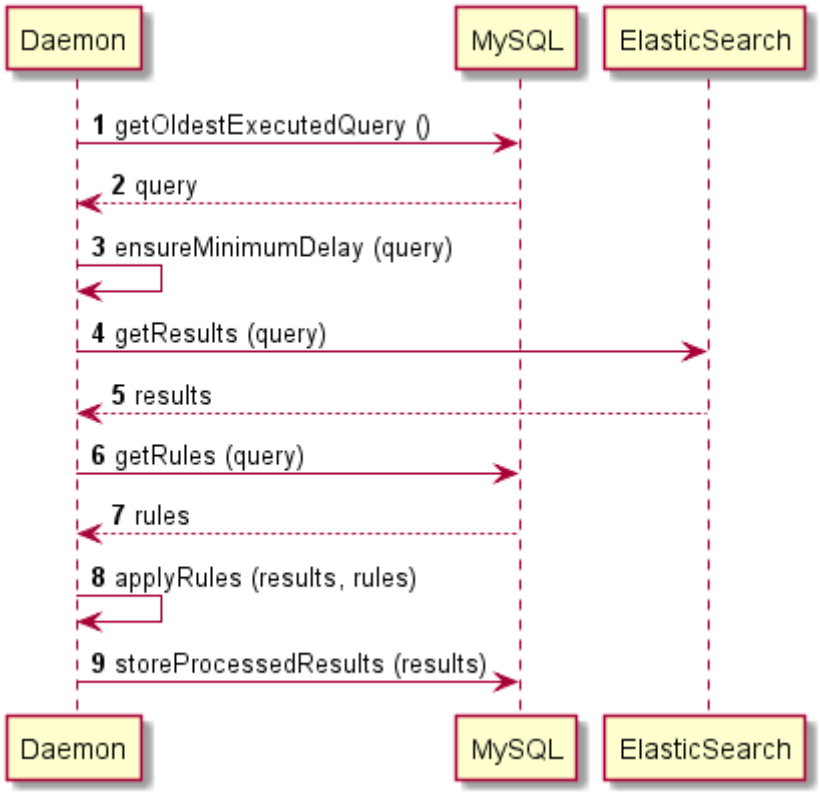


ILUSTRACIÓN 20 - UML SECUENCIA, DEMONIO DE ACTUALIZACIÓN DE RESULTADOS

Constantemente, el demonio pide a la base de datos de brands la consulta que fue ejecutada hace más tiempo. Comprueba que ha pasado un tiempo mínimo desde la última ejecución (esperando si es necesario). Lanza la consulta a ElasticSearch. Posteriormente, pide a la base de datos las reglas aplicables a la consulta realizada, pudiendo aplicar clasificaciones automáticamente a los resultados. Finalmente, se almacenan los resultados clasificados en la base de datos y se repite el ciclo.

INSTALACIÓN DEL NODO CENTRAL

En el nodo central deben instalarse y configurarse los siguientes servicios:

MONGODB

Se debe instalar una versión reciente de MongoDB, preferiblemente igual o superior a 3.0 por contar con el motor de almacenado WiredTiger (la preferencia se basa en cuestiones de eficiencia, no de funcionalidad necesaria).

El servidor, tal y como se tiene instalado en el sistema en funcionamiento, no requiere configuraciones específicas, más allá del uso del motor de almacenamiento WT.

Se recomienda configurar la autenticación de usuarios en el servidor.

Se debe crear un índice en el campo “domain” de la colección de datos de arañado. Desde la consola de mongo:

mongo> use scrap

mongo> db.crawl.ensureIndex({“domain”: 1})

Donde “scrap” es el nombre de la base de datos que se ha configurado en el programa coordinador, y “crawl” es el nombre de la colección de datos de arañado.

Para el almacenamiento de estadísticas de conteo de exploración (histórico de conteo de dominios encontrados y URLs descargadas con una frecuencia de media hora) se utiliza una *capped collection*, una colección limitada en tamaño (disco y elementos) que funciona de manera similar a una cola circular, asegurando el orden natural de los elementos en disco. Esto es provechoso siendo la información almacenada conceptualmente similar a un log, ya que permite una inserción muy rápida y un acceso similarmente rápido a los datos desde el final.

Se consideró dos meses un tiempo suficiente de almacenado de este dato histórico. Observando el tamaño de un documento de este tipo (con la forma {“date”: CurrentTimestamp, “domains”: Integer, “urls”: Integer }), se determinó que el tamaño de cada documento no es superior a 100 bytes. Por tanto, considerando meses de 30 días, se necesita almacenar 2 * 30 * 24 * 2 = 2880 documentos. Puesto que el límite real se establece en tamaño de disco, siendo el número de documentos un límite adicional, el límite se establece en 2880 * 100 = 288000 bytes. Desde la consola de mongo:

mongo> use scrap

mongo> db.createCollection(“stats”, { capped: true, size: 288000, max: 2880 })

Para la inserción de datos históricos, existe un script (update_stats.py). Mediante una entrada en crontab, se ejecuta cada media hora.

ELASTICSEARCH

Descargar una versión actual de ElasticSearch (se ha utilizado 1.6). No requiere más instalación que descomprimir el directorio en alguna carpeta. Se recomienda crear un usuario específico para la ejecución del demonio y configurar las rutas de datos y log en algún directorio accesible para el mismo.

Es necesario instalar el plugin *icu*. Para ello, desde la carpeta de Elastic:

$ bin/plugin install elasticsearch/elasticsearch-analysis-icu/2.6.0

Donde 2.6.0 es la versión correspondiente a la versión de ElasticSearch instalada.

De cara a contar con una interfaz de monitorización de estado y de realización de consultas de manera manual se ha instalado el plugin HQ. No obstatce, no es necesario para el funcionamiento del sistema.

La inicialización de los índices se realiza mediante curl, mediante el comando:

```
$ curl -XPUT "http://127.0.0.1:9200/crawl_1" -d '
{
	"settings": {
		"number_of_replicas": "0",
		"number_of_shards": "1",

		"analysis": {
			"filter": {
				"stop_filter": {
					"type": "stop"
				}
			},
			"analyzer": {
				"web_analyzer_search": {
					"type": "custom",
					"filter": [
					    "icu_normalizer",
					    "stop_filter",
					    "icu_folding"
					],
					"tokenizer": "icu_tokenizer"
				},
				"web_analyzer_index": {
				    "type": "custom",
				    "filter": [
				        "icu_normalizer",
				        "stop_filter",
				        "icu_folding"
				    ],
				    "tokenizer": "icu_tokenizer"
				 }
			}
		}
	},

	"mappings": {
		"web": {
		"_all": {
		    "enabled": false
		},
		"_id": {
		    "path": "url_hash"
		},
```

```
            "_source": {
                "includes": [
                        "url",
                        "content",
                        "title",
                        "firstFound",
                        "lastChecked"
                    ]
            },
            "properties": {
                "content": {
                    "type": "string",
                    "index_analyzer": "web_analyzer_index",
                    "search_analyzer": "web_analyzer_search",
                },
                "title": {
                    "type": "string",
                    "index_analyzer": "web_analyzer_index",
                    "search_analyzer": "web_analyzer_search",
                },
                "firstFound": {
                    "type": "date",
                    "format": "dateOptionalTime"
                },
                "lastCheck": {
                    "type": "date",
                    "format": "dateOptionalTime"
                },
                "available": {
                    "type": "boolean"
                },
                "url": {
                    "type": "string"
                },
                "url_hash": {
                    "type": "string",
                    "index": "no"
                }
            }
        }
      },
    "aliases": {
        "crawl_index": {},
        "crawl_search": {}
    }
}'
```

Donde “crawl_1” es el nombre del primer índice creado para almacenar los resultados. Según se llene, es necesario crear nuevos índices de modo similar al primero, actualizando los alias “crawl_index” y “crawl_search” tal y como se describió en el apartado dedicado al funcionamiento de ElasticSearch.

REDIS

Es necesaria una versión igual o superior a 2.8, ya que es la primera en introducir las notificaciones de eventos.

La instalación no requiere más configuración que la activación del envío de notificaciones. Para ello, modificar el archivo de configuración, añadiendo la línea:

```
notify-keyspace-events "Ex"
```

MySQL

Se ha utilizado una versión reciente de MySQL. No es necesaria ninguna funcionalidad especial. Se recomienda crear un usuario en la base de datos específico para este sistema, con permisos de lectura / escritura únicamente en las tablas de la base de datos del sistema.

Los comandos de inicialización de la base de datos son:

```
CREATE TABLE brands (
    brand_id    INTEGER PRIMARY KEY,
    name        VARCHAR(128) UNIQUE,
    enabled     BOOLEAN
);

CREATE TABLE mails (
    brand_id    INTEGER NOT NULL,
    mail        VARCHAR(256),

    PRIMARY KEY(brand_id, mail),
    FOREIGN KEY(brand_id) REFERENCES brands (brand_id) ON UPDATE CASCADE ON
DELETE CASCADE
);

CREATE TABLE queries (
    query_id    INTEGER PRIMARY KEY,
    query       TEXT NOT NULL,
    brand_id    INTEGER NOT NULL,
    last_execution  TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    enabled     BOOLEAN NOT NULL DEFAULT TRUE,

    FOREIGN KEY (brand_id) REFERENCES brands (brand_id) ON UPDATE CASCADE ON
DELETE CASCADE
);

CREATE TABLE rules (
    rule_id     INTEGER PRIMARY KEY,
    type        INTEGER UNSIGNED NOT NULL DEFAULT 0,
    action      INTEGER UNSIGNED NOT NULL DEFAULT 0,
    fields      VARCHAR(256) NOT NULL,
    match       TEXT NOT NULL,
    query_id    INTEGER NOT NULL,

    FOREIGN KEY (query_id) REFERENCES queries (query_id) ON UPDATE CASCADE ON
DELETE CASCADE
);

CREATE TABLE matches (
    match_id    INTEGER PRIMARY KEY,
    brand_id    INTEGER NOT NULL,
    query_id    INTEGER NOT NULL,
    title       VARCHAR(256) NOT NULL,
    snippet     VARCHAR(1024) NOT NULL,
    url         VARCHAR(512) NOT NULL,
    status      INTEGER UNSIGNED DEFAULT 0,

    FOREIGN KEY (brand_id) REFERENCES brands (brand_id) ON UPDATE CASCADE ON
DELETE CASCADE,
    FOREIGN KEY (query_id) REFERENCES queries (query_id)  ON UPDATE CASCADE ON
DELETE CASCADE
);
```

DEPENDENCIAS DE PYTHON

El sistema utiliza los siguientes módulos:

TABLA 1- DEPENDENCIAS PYTHON, NODO COORDINADOR

bson	pickle
datetime	pymongo
elasticsearch	qr
flask	Queue
hashlib	redis
json	rq
mandrill	urllib
mysqldb	urllib2

Todos deben ser instalados vía *pip*. Si se desea, puede ser conveniente utilizar un entorno virtual de python para el sistema completo.

UWSGI

Las aplicaciones Flask se levantan mediante la aplicación Uwsgi, encargado de mantener un número de procesos web constante y comunicarse con el servidor Nginx que hace de frontal.

La configuración se hace individualmente por cada aplicación Flask. Hay dos en el sistema, con sus respectivas directivas de configuración. Los parámetros más relevantes son el socket de comunicación para enviar y recibir de Nginx, la ruta base de donde importar las aplicaciones Flask, y el directorio de log.

- Panel de control (darknetindexer.ini):

  ```
  socket      = %d/sockets/indexer.sock
  module      = cyber.darknets.web.indexer_web:app
  pythonpath  = /home/javi/cyber/
  callable    = app
  processes   = 2
  threads     = 2
  chdir       = /home/javi/cyber/cyber/darknets/web/
  master      = true
  daemonize   = false
  logto       = logs/darknet_indexer.log
  logto2      = logs/darknet_indexer.log
  ```

- Dispatcher de tareas (darknetdispatcher.ini):

  ```
  socket      = %d/sockets/dispatcher.sock
  module      = cyber.darknets.indexer.dispatcher:app
  pythonpath  = /home/javi/cyber/
  ```

```
callable        = app
processes       = 2
threads         = 2
chdir           = /home/javi/cyber/cyber/darknets/indexer/
master          = true
daemonize       = false
logto           = logs/darknet_dispatcher.log
logto2          = logs/darknet_dispatcher.log
```

Para lanzar los procesos:

```
$ uwsgi darknetdispatcher.ini
$ uwsgi darknetindexer.ini
```

NGINX

Las interfaces web del sistema no son accedidas directamente, sino que se encuentran detrás de un servidor Nginx que reenvía las peticiones vía uwsgi. La configuración para el panel de control es:

```
location /darknets/ {
        root /home/javi/cyber/cyber/darknets/web;
        try_files $uri @darknets;
}
location @darknets {
        include             uwsgi_params;
        uwsgi_param  SCRIPT_NAME /darknets/;
        uwsgi_modifier1     30;
        uwsgi_pass   unix:/home/javi/cyber/uwsgi/sockets/indexer.sock;
        proxy_redirect      off;
}
```

Para la api de comunicación con los nodos esclavos:

```
location /darknet_dispatcher/ {
        client_max_body_size      0;
        client_body_timeout 0;

        root /home/javi/cyber/htdocs;
        try_files $uri @darknet_dispatcher;
}
location @darknet_dispatcher {
        include             uwsgi_params;
        uwsgi_param  SCRIPT_NAME /darknet_dispatcher/;
        uwsgi_modifier1 30;
        uwsgi_pass   unix:/home/javi/cyber/uwsgi/sockets/dispatcher.sock;
        proxy_redirect      off;
}
```

En ambos casos, la primera sección define simplemente la ruta de acceso bajo el servidor web en la que el contenido estará accesible. Para cualquier petición, primero se trata de obtener un archivo existente con el nombre solicitado. Si no se encuentra, se pasa al segundo bloque (directiva try_files). En el caso de la API, es importante eliminar el límite por defecto en el tamaño de las peticiones HTTP, ya que el contenido

enviado por una araña, al contener todo el contenido de texto de una cantidad arbitraria de URLs, puede ser bastante grande.

La segunda sección define el comportamiento cuando el archivo no se ha encontrado, por lo que se envía la petición a la aplicación back-end. Además de declarar las directivas para el funcionamiento correcto de esta comunicación (incluir el archivo de directivas uwsgi_params, deshabilitar proxy_redirect, y el parámetro uwsgi_modifier1, según dicta la documentación), se define a qué socket se reenviará la petición (uwsgi_pass). Su valor debe ser la misma ruta del socket creado en la configuración de uwsgi.

DEMONIO DE ACTUALIZACIÓN DE RESULTADOS DE CONSULTAS

Basta con lanzar el programa en background. Para ello:

```
$ nohup python run_darknet_monitor_daemon.py > daemon.log &
```

NODO ARAÑA

Cada nodo araña precisa de la instalación de los clientes de las darknets a explorar y, si es necesario, un proxy HTTP que haga de intermediario. Otra posibilidad es utilizar un proxy externo que dé acceso a las darknets.

CLIENTES DE DARKNETS

Si se opta por acceso local desde el nodo araña a las darknets, es necesario instalar sus respectivos clientes.

No se ha necesitado realizar ninguna configuración adicional, por lo que se obvia el proceso de instalación. Ver la guía de usuario de cada una de ellas.

TMPFS

Durante la extracción de texto de algunos tipos de archivo, la araña escribe en disco su contenido completo. Para optimizar el tiempo de lectura / escritura, es conveniente crear una carpeta montada sobre memoria RAM. La araña viene configurada para utilizar el directorio */mnt/ramdisk*.

Para crearlo:

```
# mkdir /mnt/tmpfs
# mount –t tmpfs –o size 256M tmpfs /mnt/ramdisk
```

Y añadir la línea en /etc/fstab:

```
tmpfs /mnt/ramdisk tmpfs   nodev,nosuid,noexec,nodiratime,size=256M   0 0
```

PRIVOXY

Se utiliza Privoxy como intermediario en los casos en los que el cliente de darknet provea únicamente de acceso vía proxy SOCKS, ya que Scrapy es compatible únicamente con proxys HTTP. Este es el caso, por ejemplo, del cliente de TOR.

Se ha utilizado el archivo de configuración por defecto, con la siguiente línea:

forward-socks5 / 127.0.0.1:9060 .

Para forzar todo el tráfico a través de TOR. Dadas las limitaciones de conectividad del entorno en el que se ha desarrollado el proyecto, no se han hecho pruebas con el resto de darknets contempladas en el alcance.

DEPENDENCIAS DE PYTHON

La araña utiliza las siguientes librerías:

TABLA 2 - DEPENDENCIAS PYTHON, NODO ESCLAVO

json	re
lxml	scrapy
magic	textract
mimetypes	urllib
Queue	urllib2

Textract, por su parte, requiere el siguiente software instalado:

TABLA 3 - DEPENDENCIAS TEXTRACT, NODO ESCLAVO

libxml2-dev	flac
libxslt1-dev	ffmpeg
antiword	lame
poppler-utils	libmad0
pstotext	libsox-fmt-mp3
tesseract-ocr	sox

CONFIGURACIÓN

Es necesario configurar en el script spider.py los siguientes parámetros:

- ramfs: directorio del sistema de archivos montado en memoria
- dispatcher_host: dirección IP del nodo maestro
- dispatcher_port: puerto en el que se encuentra escuchando el servidor HTTP del nodo maestro
- dispatcher_path: ruta en el servidor HTTP del maestro en la que se encuentra la API de distribución de objetivos
- spider_id: nombre identificador de la araña
- spider_password: clave de autenticación de la araña

EJECUCIÓN

Basta con lanzar la araña en background:

```
$ nohup python spider.py > spider.log &
```

MANTENIMIENTO: RESOLUCIÓN DE PROBLEMAS

Salvo que se especifique lo contrario, los comandos mencionados deben ser lanzados con el usuario dueño de los archivos del proyecto.

EL NÚMERO DE RESULTADOS NO AUMENTA

Puede tener varias causas:

1- La araña no tiene enlaces nuevos que visitar, habiendo sido visitados previamente todos los que tiene encolados.
2- Los esclavos se encuentran parados
3- El punto de acceso HTTP del coordinador está parado
4- Alguno de los servicios de base de datos se encuentra parado o en estado de error
5- No hay conectividad entre los nodos esclavos y el coordinador

El diagnóstico recomendado es el siguiente:

1- Comprobar que haya nodos esclavo corriendo.
   $ ps auxf | grep spider
2- Si hay esclavos corriendo, comprobar su log. Los errores más comunes son:
   a. El esclavo es incapaz de autenticar / comunicarse con el coordinador. En este caso, el log mostrará gran cantidad de errores HTTP seguidos.
      Solución:
      - Comprobar que el coordinador sea accesible desde la máquina donde se ejecuta el esclavo.
      - Comprobar que los datos de autenticación configurados en el esclavo sean los presentes en la base de datos de esclavos del coordinador.
      - Si los dos puntos anteriores son correctos, el esclavo debería volver a funcionar reiniciándolo según el procedimiento “Reinicio dela araña en un nodo esclavo”.
   b. El esclavo es incapaz de acceder a las darknets y por tanto de obtener resultados nuevos.
      Solución: comprobar manualmente que el acceso es posible desde la ubicación del nodo esclavo utilizando el proxy de acceso que se le ha configurado.
3- Comprobar que el punto de acceso HTTP del coordinador está funcionando. Lanzar una consulta manual con un navegador a <ip.donde.corre.el.coordinador>/<ruta/base/del/punto/de/acceso>/

   Debería devolver rápidamente un error 403. De no ser así, llevar a cabo el procedimiento “Reinicio del dispatcher”.

4- Comprobar el estado de Redis. En condiciones de alta carga y poca disponibilidad de memoria, Redis puede volverse inestable. Si no tiene memoria suficiente para duplicar su proceso y asegurar un correcto volcado de su base de datos en disco, deja de aceptar operaciones de escritura. Esta situación puede diagnosticarse de varias maneras:
   a. Visualizando el log de alguno de los scripts que interacciona con Redis, por ejemplo, los procesos rqworker.
   b. Visualizando el log de Redis
   c. Accediendo al servidor Redis vía redis-cli y tratando de escribir una clave no existente, por ejemplo, "set noexisto prueba".

   Solución: seguir el procedimiento descrito en "Reinicio del servidor Redis".

5- Comprobar el estado de los trabajadores de las colas de tareas en el nodo coordinador:
   $ ps auxf | grep rqworker
   Debería haber mínimo tres procesos, uno por cada cola de tareas (links_tasks, index_tasks, stats_tasks).

6- Comprobar el estado de MongoDB. Se ha dado el caso de que por carga excesiva del sistema se haya vuelto inestable y haya dejado de aceptar conexiones o se haya apagado.
   a. Comprobar que se encuentra en ejecución: $ ps auxf | grep mongo
      Si no se encuentra en ejecución, comprobar el log para localizar la causa de su apagado. Una vez solucionado, seguir el procedimiento "Reinicio del servidor MongoDB".
   b. Si se encuentra en ejecución, probar a realizar una conexión contra el servidor mediante el comando "mongo".
      Si la conexión falla, examinar el mensaje de error. Una vez solucionado, seguir el procedimiento "Reinicio del servidor MongoDB".
   c. Si permite conexiones, probar a acceder a la base de datos de la araña (testscrap en el momento de escritura de este documento) y a las distintas conexiones. Lanzar operaciones de lectura y escritura.
      Si esto falla, reiniciar el sistema completo, siguiendo el procedimiento "Reinicio del sistema completo".

LOS NODOS ESCLAVO VISITAN SIEMPRE LOS MISMOS SITIOS

Los objetivos de arañado se deciden según la cola de prioridad de URLs. La cola está ordenada según la puntuación asignada, calculada a partir de los datos históricos de visita a sitios web almacenados en la base de datos MongoDB.

Estos datos son actualizados por los trabajadores de la cola de tareas links_tasks.

Se han observado los siguientes problemas:

- Las tareas asignadas a una araña expiran antes de que devuelva resultados, siendo reencoladas con su prioridad original. Es probable que sean, por tanto:
    - Asignadas de nuevo a la misma araña una vez devueltos los resultados, en cuyo caso la tarea se repetirá una vez.
    - Asignadas a la misma araña, que no ha logrado reportar los resultados de vuelta y por tanto no ha actualizado la puntuación de los objetivos. La tarea puede repetirse durante un tiempo prolongado.
    - Asignadas a arañas distintas. Mismas posibilidades.
- Se ha extraído una gran cantidad de enlaces de un grupo de objetivos asignados. Esto puede aumentar de manera exponencial el número de tareas de cálculo de prioridad y adición a la cola de objetivos.

  Para observar el estado de las colas, levantar el dashboard de rqworker, mediante la orden:

  $ rq-dashboard –P 6667

  Acceder a la URL mostrada en la salida del comando anterior. Este dashboard permite observar el estado de los trabajadores y de las colas de trabajo. La situación típica causada por este problema es que la cola links_tasks tenga un gran número de elementos y el resto se encuentren casi vacías.

  Si se observa que hay muchas tareas con enlaces del mismo sitio, se pueden limpiar sin perder información relevante. Para ello, utilizar el script clean_tasks.py en la carpeta cyber/darknets/indexer/.

  Este script recibe como primer argumento la cola a limpiar. Se ha comprobado que funciona con la cola links_tasks, dada su estructura. El segundo argumento es la subcadena a limpiar. Si, por ejemplo, se quieren limpiar los enlaces que contengan la cadena “block”, típica en enlaces extraídos de mirrors del blockchain de bitcoin, la ejecución sería:

  $ python clean_tasks.py links_tasks block

  Mostrará el número de tareas afectadas y la cantidad de enlaces eliminados. Es posible que los cambios no se vean reflejados en la información mostrada por el dashboard, pero se debería observar un aumento de velocidad de disminución del número de tareas pendientes.

PROCEDIMIENTOS

REINICIO DE LA ARAÑA EN UN NODO ESCLAVO

```
$ kill -9 $(ps auxf | grep spider.py | awk '{print $2}')
$ nohup spider.py > spider.log &
```

REINICIO DEL PORTAL WEB

Si se encuentra en ejecución, pararlo mediante la secuencia:

```
$ for pid in $(ps auxf | grep -v grep | grep darknetindexer | awk
'{print $2}'); do kill -9 $pid; done
```

Asegurarse de que no ha quedado ningún proceso huérfano relacionado con el portal, mediante la orden:

```
$ ps auxf | grep -v grep | grep darknetindexer
```

Si queda alguno, repetir la primera orden para terminarlo.

Para levantar el servicio, desde la carpeta de configuraciones de uwsgi del proyecto lanzar la orden:

```
$ uwsgi darknetindexer.ini
```

Comprobar que se encuentra accesible accediendo desde un navegador a la ruta del panel. Si no responde, repetir el procedimiento.

REINICIO DEL DISPATCHER

Si se encuentra en ejecución, pararlo mediante la secuencia:

```
$ for pid in $(ps auxf | grep -v grep | grep darknetdispatcher | awk
'{print $2}'); do kill -9 $pid; done
```

Asegurarse de que no ha quedado ningún proceso huérfano relacionado con el dispatcher, mediante la orden:

```
$ ps auxf | grep -v grep | grep darknetdispatcher
```

Si queda alguno, repetir la primera orden para terminarlo.

Para levantar el servicio, desde la carpeta de configuraciones de uwsgi del proyecto, lanzar la orden:

```
$ uwsgi darknetdispatcher.ini
```

Comprobar que se encuentra accesible accediendo desde un navegador a la ruta del dispatcher. De no responder en un tiempo aceptable, repetir el procedimiento.

REINICIO DEL MONITOR DE TAREAS EXPIRADAS

Si se encuentra en ejecución, detener el proceso mediante la orden:

```
$ kill -9 $(ps uaxf | grep expiremonitor | grep –v grep | awk ‘{print $2}’)
```

Para levantar el programa, desplazarse a la carpeta del proyecto (/home/javi/cyber a la escritura de este documento). Lanzar el proceso mediante la orden:

```
$ nohup python -m cyber.darknets.indexer.expire_monitor > logs/expire_monitor.log &
```

REINICIO DEL DEMONIO DE CONSULTAS

Si se encuentra en ejecución, detener el proceso mediante la orden:

```
$ kill -9 $(ps auxf | grep darknet_monitor_daemon | grep –v grep | awk ‘{print $2}’)
```

Para levantar el demonio, desplazarse a la carpeta del proyecto y lanzar la orden:

```
$ nohup python run_darknet_monitor_daemon.py > monitor_daemon.log &
```

REINICIO DEL SERVIDOR REDIS

Si el servidor está en ejecución, pararlo vía kill -9. Se trata de un servicio crítico del que depende el resto de actividad, por lo que es necesario detener varios procesos:

- Parar los procesos de los trabajadores según lo descrito en el proceso “Reinicio de los trabajadores”, sin volver a levantarlos.
- Parar los procesos de arañado en los nodos esclavos según lo descrito en el proceso “Reinicio de la araña en un nodo esclavo”, sin volver a levantarlos.

Para levantarlo, ir al directorio de trabajo del servidor Redis del proyecto. Lanzar el comando:

```
$ redis-server redis.conf
```

Si es preciso, levantar de nuevo los procesos detenidos al comienzo de este procedimiento.

REINICIO DEL SERVIDOR MONGODB

Si el servidor está en ejecución, pararlo vía kill -9.

Para levantarlo, lanzar como root el comando:

```
$ mongod
```

REINICIO DEL SERVIDOR ELASTICSEARCH

Si el servidor está en ejecución, pararlo vía kill -9.

Para levantarlo, basta con lanzar el programa. Ir a la ruta donde esté descargado y lanzar el comando:

```
$ bin/elasticsearch
```

Los servicios que dependen de ElasticSearch deberían restaurar las conexiones de forma automática.

REINICIO DE LOS TRABAJADORES

Si se encuentran en ejecución, terminarlos mediante el comando:

```
$ for pid in $(ps auxf | grep rqworker | grep –v grep | awk ‘{print $2}’); do kill -9 $pid; done
```

Para levantar un trabajador por cada cola de trabajo, lanzar los comandos:

```
$ rqworker -P /home/javi/cyber/ -u redis://127.0.0.1:6667 index_tasks
$ rqworker -P /home/javi/cyber/ -u redis://127.0.0.1:6667 links_tasks
$ rqworker -P /home/javi/cyber/ -u redis://127.0.0.1:6667 stats_tasks
```

Siendo /home/javi/cyber/ la ruta donde se encuentra el código del proyecto (contenido en el subpaquete cyber dentro de esa carpeta) y 127.0.0.01:6667 la IP y puerto donde se encuentra accesible el servidor Redis.

REINICIO DEL SISTEMA COMPLETO

Realizar, en el orden escrito, las siguientes acciones:

1- Parada de las arañas en los nodos esclavos, según lo descrito en el procedimiento “Reinicio de la araña en los nodos esclavo”.
   a. Si se considera necesario, parada de los clientes de las darknets configuradas en los nodos esclavos. Este paso es opcional, por no ser servicios que interaccionen con el resto del sistema.
   b. Si se considera necesario, parada del servidor Privoxy. Este paso es opcional, por no ser un servicio que interaccione con el resto del sistema una vez realizados los pasos anteriores.
2- Parada del dispatcher según lo descrito en el procedimiento “Reinicio del dispatcher”.
3- Parada del portal web según lo descrito en el procedimiento “Reinicio del portal web”.
   a. Si se considera necesario y no interfiere con otros servicios corriendo en la misma máquina, parar el servidor nginx que hace de frontal. Este paso es opcional, por no ser un servicio que interaccione con el resto del sistema una vez que los puntos anteriores han sido completados.

4- Parada de los trabajadores según lo descrito en el procedimiento “Reinicio de los trabajadores”.
5- Parada del demonio de actualizaciones de consultas de brands según lo descrito en el procedimiento “Reinicio del demonio de consultas”.
6- Parada del servidor Redis según lo descrito en el procedimiento “Reinicio del servidor Redis”.
7- Parada del monitor de tareas expiradas según lo descrito en el procedimiento “Reinicio del monitor de tareas expiradas”.
8- Parada del servidor MongoDB según lo descrito en el procedimiento “Reinicio del servidor MongoDB”.
9- Parada del servidor ElasticSearch según lo descrito en el procedimiento “Reinicio del servidor Elasticsearch”.

Para levantar el sistema, levantar los componentes en el orden inverso a la parada, con el procedimiento correspondiente a cada punto.

RESULTADOS

Tras dos meses de ejecución de la araña distribuida, con un único nodo esclavo, los resultados son:

- 30000 dominios localizados en TOR
- 1 200 000 URLs descargadas
- 18GB de espacio ocupados por el índice en ElasticSearch
- 300MB de espacio ocupados por la base de datos de arañado en MongoDB
- Uso total de la memoria del servidor (8GB), uso de swap (3.5GB)
- Picos de uso de CPU por parte de los trabajadores de las colas de tareas

La estimación de coste en disco del prototipo era de 35KB por URL indexada, habiendo alcanzado 10GB con 300000 URLs. El coste actual es de 15KB por documento, por lo que se ha mejorado considerablemente al pasar a ElasticSearch.

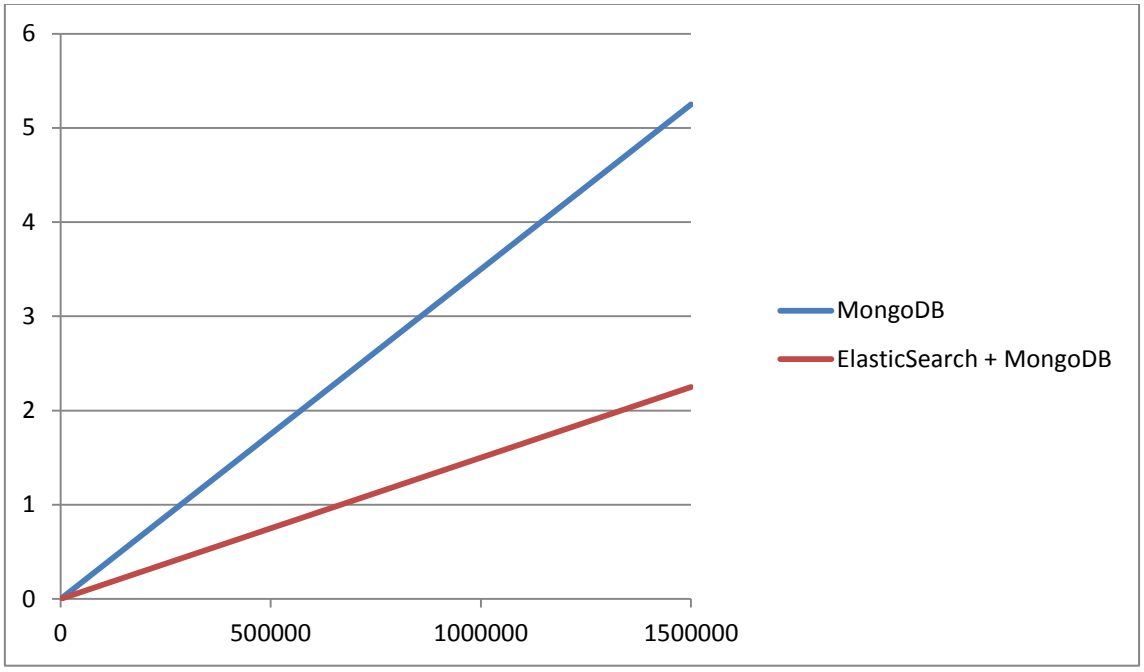


ILUSTRACIÓN 21 – EVOLUCIÓN PREVISTA DE TAMAÑO (Nº DE DOCUMENTOS / GB)

En ciertos casos de mucha actividad, se han observado fallos del servicio debido a la imposibilidad de Redis de operar en condiciones mínimas de memoria disponible, rechazando operaciones de escritura.

En una implementación real, se sugiere que ElasticSearch, MongoDB y Redis se ejecuten independientemente en al menos un nodo cada uno, dados los errores mencionados. Las tres están pensadas para trabajar con su conjunto de datos en memoria, por lo que su rendimiento empeora progresivamente según el tamaño del conjunto aumenta fuera del límite marcado por la memoria disponible en el sistema.

Es conveniente que la latencia entre ellos y el nodo en el que se ejecute el programa coordinador sea mínima, así como entre ellos y el nodo en el que corren los trabajadores de las colas de tareas.

Conclusiones

A partir de los resultados experimentales obtenidos de la ejecución del proyecto descrito en este documento durante dos meses, se concluye que la indexación de una parte considerable de la denominada red oculta es viable.

Según las estimaciones del proyecto TOR, existen entre 25000 y 35000 dominios distintos activos simultáneamente[2]. Por tanto, la exploración realizada (30000 dominios) se acerca bastante a dicha métrica, y puede considerarse que mantener actualizado con un retraso aceptable el índice es cuestión de recursos (en forma de nodos araña que repasen periódicamente las URLs conocidas).

Dadas las condiciones de la puesta en práctica (red con ancho de banda limitado y un único nodo araña) y los resultados obtenidos mencionados anteriormente, se cree posible mantener un índice relativamente actualizado del estado de TOR (aceptando varios días de retraso) con un número no muy grande de nodos araña. Con el entorno actual, se descargan 20000 URLs al día. Suponiendo un entorno con conectividad con enlaces de TOR rápidos y un nodo maestro correctamente dimensionado, mantener el índice actualizado con un desfase de tres días necesitaría acceder a 400000 URLs diarias. Asumiendo 20000 URLs / araña, podría realizarse con 20 arañas corriendo simultáneamente. Con 100 arañas, se podrían actualizar los resultados completos cada 14 horas aproximadamente.

[2] https://metrics.torproject.org/hidserv-dir-onions-seen.html?graph=hidserv-dir-onions-seen&start=2015-03-17&end=2015-06-15

Trabajos futuros

Aunque el proyecto es funcional en su estado actual, hay ciertos puntos que son susceptibles de ser mejorados, así como funcionalidades que podrían incorporarse. Algunos de ellos son:

- Mejora del algoritmo de cálculo de prioridad para los objetivos:
  El algoritmo actual hace que las puntuaciones incrementen indefinidamente. Podría ser conveniente diseñar un algoritmo que normalizase la puntuación en un rango determinado.
- Separación de tareas de arañado para sitios nuevos y para sitios ya conocidos:
  Actualmente, el coordinador no distingue entre tareas de reindexado y tareas de exploración. Una posible funcionalidad interesante podría ser la gestión de tareas on-demand por parte de los operadores. Puede simularse en la implementación actual añadiendo objetivos con prioridad cero, pero no puede asegurarse su completitud.
- Gestión de estadísticas de arañado:
  Actualmente, el coordinador no guarda información respecto a las arañas, más allá de sus credenciales de autenticación. A la hora de analizar el funcionamiento de la herramienta, podría ser interesante tener un histórico completo de las tareas asignadas a cada araña, las completadas y no completadas, y el tiempo que les ha llevado.
- Grafo de relaciones entre sitios en darknets:
  Un estudio interesante podría ser el análisis del camino que lleva de un sitio en una darknet a otra, es decir, cómo se ha descubierto un sitio dado.
- Control de acceso en panel de control y separación de vistas:
  Actualmente, el panel de control no cuenta con medidas de control de acceso integradas (e la implementación, se ha utilizado autenticación HTTP a nivel del servidor nginx tras el que se encuentra). Dado que la herramienta está pensada para ser utilizada por un grupo de operadores controlando la presencia de un conjunto de *brands* en las darknets indexadas, es conveniente que cada operador se autentique contra el panel, y pueda ver y configurar únicamente los *brands* que tiene asignados.
- Fuentes de objetivos externas:
  La araña, en su estado actual, toma enlaces únicamente de los sitios web arañados, añadiéndolos iterativamente a su base de datos. Una posible mejora puede ser el desarrollo de un conjunto de monitores de sitios web conocidos en los que se sabe que ocasionalmente aparecen enlaces a sitios en darknets, como pueden ser 4chan, pastebin, o foros de discusión. Estos monitores estarían encargados de arañar periódicamente dichos sitios, extrayendo

enlaces que apunten a sitios dentro de alguna de las darknets para las que se tiene alcance, y alimentar al nodo coordinador con dichos enlaces.

- Gestión de credenciales para sitios cerrados:
  Hay sitios web dentro de las darknets que requieren autenticación para ser accedidos, siendo, no obstante, de libre registro. Se podría crear un repositorio de credenciales en el nodo coordinador, controlado desde el panel de control, con un mapa { sitio_web => { credenciales de acceso } }. El coordinador, al enviar tareas de arañado a un esclavo, si detectase que uno de los sitios que está enviando requiere credenciales, se las enviaría. Sería necesario definir un protocolo de descripción del proceso de autenticación, además de las propias credenciales (ruta en la que se debe lanzar la petición de autenticación, método, parámetros HTTP, etc.). También hay que considerar casos especiales, como pueden ser captchas u OTP enviados a correo electrónico.
- Detección y evasión de sinks:
  Entiéndase como sink una página pensada para evitar el correcto funcionamiento de una araña web, generalmente creando enlaces infinitos dentro de un conjunto de sitios controlados por el autor. De este modo, se intenta que la araña quede atrapada y no continúe su exploración.
  En la implementación actual esto queda parcialmente paliado por el algoritmo de prioridad: la araña puede quedar atrapada durante un periodo de tiempo, hasta que los sitios sink alcancen una puntuación de prioridad superior a las siguientes más prioritarias. No obstante, se mantienen los siguientes problemas:
    - Un sink que ha salido del grupo de objetivos más prioritarios volverá a entrar en él, cuando los siguientes más prioritarios incrementen su puntuación.
    - La base de datos queda contaminada por un número previsiblemente alto de sitios falsos.
- Utilización de la DHT (Tabla Hash Distribuida) de TOR para obtener objetivos
- Asignación de objetivos controlada, pudiendo una araña solicitar objetivos solo para un subconjunto de las darknets soportadas.
- Detección de idioma y clasificación de contenido mediante análisis de texto.

Bibliografía

[1] D. Manning, C., Raghavan, P. y Schütze, H. (2008). *Introduction to Information Retreival, Primera edición, Cambridge University Press*.

http://nlp.stanford.edu/IR-book/pdf/20crawl.pdf

[2] Castillo, C. (2004). *Effective Web Crawling*.

http://chato.cl/papers/crawling_thesis/effective_web_crawling.pdf

[3] Shkapenyuk, V. y Suel, T. (AÑO). *Design and Implementation of a High Performance Distributed Web Crawler*.

http://cis.poly.edu/suel/papers/crawl.pdf

[4] Nielsen, M. (2012). *How to crawl a quarter billion webpages in 40 hours*.

http://www.michaelnielsen.org/ddi/how-to-crawl-a-quarter-billion-webpages-in-40-hours/

[5] TOR Project. *TOR Documentation.*

https://www.torproject.org/docs/documentation.html.en

[6] TOR Project. *TOR Specification.*

https://gitweb.torproject.org/torspec.git/tree/

[7] Equipo de desarrollo de I2P. *I2P Documentation.*

https://geti2p.net/en/docs

[8] Equipo de desarrollo de Freenet. *Freenet Documentation.*

https://freenetproject.org/documentation.html

[9] Equipo de desarrollo de Scrapy. *Scrapy Documentation.*

http://doc.scrapy.org/en/latest/

[10] Peticolas, D. (2009). *Twisted Introduction.*
http://krondo.com/?page_id=1327

[11] Driessen, V. *RQ: Easy job queues for python*.
http://python-rq.org/docs/

[12] Nyman, Ted. *QR: Queues, stacks, queues and priority queues with Redis.*
https://github.com/tnm/qr

[12] Ronacher, A. (2013). *Flask Documentation.*
http://flask.pocoo.org/docs/latest/

[13] Gormley, C. y Tong, Z. (2014). *ElasticSearch: The Definitive Guide.*
https://www.elastic.co/guide/en/elasticsearch/guide/current/index.html

[14] MongoDB Inc. *The MongoDB 3.0 Manual.*
http://docs.mongodb.org/manual/

[15] *Redis documentation*
http://redis.io/documentation

[14] Twitter. *Bootstrap documentation*
http://getbootstrap.com

[15] Desarrolladores de Privoxy. *Privoxy 3.0.23 User Manual.*
http://www.privoxy.org/user-manual/index.html

[16] Unibit. *uWSGI documentation.*
https://uwsgi-docs.readthedocs.org/en/latest/

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá