UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FíSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE SISTEMA DISTRIBUIDO Y COLABORATIVO DE PETICIONES HTTP/S

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

FRANCISCO LEONARDO PULGAR ROMERO

PROFESOR GUÍA:
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
JOSÉ PINO URTUBIA
TOMÁS BARROS ARANCIBIA

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: FRANCISCO LEONARDO PULGAR ROMERO
FECHA: AGOSTO 2018
PROF. GUÍA: AIDAN HOGAN

DISEÑO E IMPLEMENTACIÓN DE SISTEMA DISTRIBUIDO Y COLABORATIVO DE PETICIONES HTTP/S

En la actualidad existen muchos computadores y dispositivos tecnológicos con capacidad computacional ociosa, con el potencial de ser usados. Es así como existen una gran cantidad de proyectos donde personas donan voluntariamente su poder computacional para ayudar en problemas tales como: renderización de animaciones 3D, correr simulaciones de experimentos, estudiar conjeturas matemáticas, optimización de variables y parámetros en *Machine Learning*, estudiar estructuras de proteínas y moléculas, clasificación de galaxias, predicción del clima, entre un sinfín de aplicaciones posibles tanto en el área de investigación como en el área empresarial.

Esa necesidad de poder de procesamiento y recursos computacionales ha llevado a crear tecnologías como la *computación grid* (o en malla), que consiste en un sistema de computación distribuido que permite coordinar computadoras de diferente hardware y software haciendo uso de estos para resolver en paralelo tareas en común.

La presente memoria tiene como fin la creación de un sistema distribuido en malla donde dispositivos tecnológicos se comunican con un servidor central para recopilar datos de internet; usando así la capacidad ociosa de dispositivos tecnológicos y brindando ayuda voluntaria a aquel que necesite recopilar datos de internet.

Durante el desarrollo de este trabajo se implementa un sistema de administración de usuarios y dispositivos tecnológicos realizado con *Django*, un sistema de distribución de consultas HTTP/S desarrollado con *Tornado* y un software que corre de lado de los dispositivos tecnológicos para resolver tareas y mandar resultados, hecho en *Python*. Estos tres sistemas se comunican entre ellos para lograr la distribución de las consultas HTTP/S, pero son independientes entre sí, ayudando a la escalabilidad y tolerancia a fallos del sistema general.

Finalmente se realizan pruebas y experimentos de los diferentes componentes para obtener datos relevantes que nos permitan estudiar el comportamiento del sistema, identificando ventajas y desventajas del uso del mismo. Los resultados obtenidos muestran que a medida que aumenta la cantidad de dispositivos tecnológicos que colaboran en una tarea, disminuyen los tiempos de resolución de éstas; además se demuestra una correlación directa entre el tiempo de respuesta de una consulta HTTP/S y la distancia física que existe entre el dispositivo que hace la consulta y el servidor web.

Agradecimientos

Quiero dar las gracias a mi familia que me ha acompañado en este largo camino, ofreciéndome todo su apoyo y comprensión.

A Aidan Hogan por compartir su conocimiento y orientarme en esta memoria, por su paciencia, dedicación y buena disposición siempre.

A mi novia Yanniré por acompañarme en este proceso.

A mis amigos David, Fernanda, Eduardo, Leonel, Axel, Daniel, Felipe, Ignacio, Sebastián y muchos otros que se me quedan en el tintero, por enseñarme y hacer más grato todo este proceso.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

El *crawling* es una técnica para obtener e indexar contenido de la red (particularmente de páginas web) de forma automatizada.

Un *crawler* es el software encargado de realizar el crawling. Este comienza generalmente con una lista de URLs, llamadas semillas, las que se visitan obteniendo más URLs, que pasan a ser las nuevas semillas, repitiendo recursivamente los pasos anteriores, deteniéndose en una cierta profundidad o nivel. Para evitar loops y semillas repetidas, cada una de las URLs visitadas se guardan en una base de datos.

El *scraping*, por otro lado, es una técnica utilizada para procesar información de sitios web y extraer información específica de ella. Esta técnica se enfoca más en la transformación de datos de formato HTML a estructuras relacionales, para poder almacenarlos en bases de datos y analizarlos posteriormente.

Tanto el crawling como el scraping son técnicas indispensables para el *análisis de datos*, *minería de datos*, *web intelligence*; entre otras disciplinas, que mediante análisis y procesamientos computacionales, alimentados de grandes volúmenes de datos, obtienen patrones, predicciones o correlaciones relevantes.

Actualmente el gran problema que se enfrenta es la lentitud al momento de obtener datos de sitios web mediante crawling y scraping; acelerar estos procesos es indispensable para que no se transformen en cuellos de botella, de esta manera, se podrá procesar y consumir de forma más fluida información de la web.

Por otro lado, las máquinas modernas son capaces de ejecutar miles de millones de instrucciones en un segundo. Sorprendentemente, la gran mayoría de los ordenadores personales están infrautilizados; la porción de aplicaciones que utilizan el 100 % de los recursos son muy pequeñas. Dicha tendencia no muestra signos de revertirse, de hecho, la capacidad computacional mejora día a día.

El presente trabajo tiene como fin la creación de un sistema que haga más eficaz y ágil la obtención y transformación de datos de internet, descentralizando las consultas HTTP/S usando la capacidad ociosa de dispositivos tecnológicos.

1.1. Justificación del trabajo de memoria

La motivación de crear este sistema nace de disminuir el tiempo de obtención de datos de páginas web, ayudando a analistas y/o empresas que requieren dentro de sus procesos hacer *crawling* y *scraping*. Por otro lado, hacer uso de la computación ociosa de dispositivos tecnológicos es un problema que merece ser atacado para poder hacer un mejor uso de los recursos tecnológicos el día de mañana.

Al atacar este problema se presentan interesantes desafíos; se debe disminuir los tiempos de cada consulta HTTP/S tanto de forma individual como en su conjunto (*delays*[1] entre consultas), se debe hacer un uso eficiente del ancho de banda de los dispositivos tecnológicos y evitar congestión en todos los componentes involucrados del sistema, se debe distribuir las tareas de forma eficiente y buscar patrones que permitan mejorar el desempeño.

Por último, este es el primer paso para en el futuro crear un sistema con más propósitos y funcionalidades, que permita realizar tareas distribuidas en paralelo y recibir los resultados para su posterior análisis.

1.2. Objetivos de la memoria

1.2.1. Objetivo general

El objetivo general de este trabajo es crear un sistema colaborativo para realizar consultas HTTP/S distribuidas donde los clientes financian el funcionamiento del sistema y los usuarios son remunerados por cantidad de consultas HTTP/S servidas.

1.2.2. Objetivos específicos

Para lograr el objetivo general, se identifican los siguientes objetivos específicos:

1. Diseñar e implementar una arquitectura capaz de administrar y soportar:
   (a) Usuarios y sus dispositivos registrados.
   (b) Esclavos conectados al sistema.
   (c) Consultas y respuestas HTTP/S de los clientes.
   (d) Procesamiento masivo de consultas HTTP/S.
2. Desarrollar parte del sistema distribuido y colaborativo de peticiones HTTP/S.
3. Evaluar el rendimiento y desempeño del sistema, estableciendo métricas y comparaciones.

[1]Un *delay* es un tiempo de atraso asignado entre consultas secuenciales.

1.3. Plan de trabajo

Con la finalidad de cumplir con los objetivos antes mencionados, se establece el siguiente plan de trabajo:

1.3.1. Diseño

- Diseñar arquitectura entre el servidor web (administración de usuarios) y el sistema distribuido de consultas HTTP/S.
- Diseñar software de recepción y solución de consultas HTTP/S.
- Diseñar servidor que reparte las consultas HTTP/S entre clientes y computadores voluntarios.

1.3.2. Implementación

- Desarrollar sistema de administración de usuarios y dispositivos.
- Crear API de consultas HTTP/S.
- Ajustar y optimizar la distribución de consultas HTTP/S.

1.3.3. Evaluación

- Crear conjuntos de datos para las diferentes pruebas.
- Definir parámetros para poder evaluar y medir el desempeño del sistema en cada experimento.
- Hacer experimentos para analizar el comportamiento del sistema en general.
- Evaluar los resultados y el rendimiento del sistema completo, en comparación a un escenario donde no existe este sistema.

1.4. Estructura del documento

A continuación se describe los tópicos que se tocarán a lo largo de este documento:

- En el **capítulo 2** veremos el **marco teórico** de este trabajo, donde se definen conceptos relevantes para el entendimiento del documento en su totalidad.
- En el **capítulo 3** se describe el **diseño y arquitectura** de la solución propuesta, además se justifican las diferentes decisiones que se tomaron para mejorar el desempeño del sistema. También se tocan temas como el financiamiento del sistema, sus aplicaciones, los posibles cuestionamientos éticos, la privacidad de los datos y la confianza en el sistema.

- En el **capítulo 4** se muestra la **implementación del sistema** descrito, donde se detalla con mayor precisión todos los componentes mencionados en el capítulo anterior y las tecnologías involucradas en el desarrollo de la propuesta.
- El **capítulo 5** está dedicado a los **experimentos y evaluación** de sus resultados, donde se podrá ver y estudiar el comportamiento del sistema con diferentes fuentes de datos.
- El **capítulo 6** es la **conclusión** del trabajo, donde se resumen y analizan los resultados más relevantes de la memoria vista.

Capítulo 2

Marco teórico

En este capítulo se describen los principales conceptos y tecnologías involucradas en el desarrollo de la memoria propuesta. También se hace un repaso por soluciones existentes y razones para buscar alternativas.

2.1. Computación distribuida

La computación distribuida [9] es el campo de la informática que estudia los sistemas distribuidos. Los componentes están ubicados en diferentes computadoras en red, que luego se comunican y coordinan sus acciones pasando mensajes entre sí. Los componentes interactúan entre sí para lograr un objetivo común. Tres características significativas de los sistemas distribuidos son: concurrencia de componentes, falta de un reloj global y fallo independiente de los componentes.

Los principales problemas de diseño que deben tenerse en cuenta al diseñar cualquier sistema de computación distribuida son los siguientes [16]:

- **Transparencia**: el sistema en su conjunto debe parecer un único sistema coherente para su usuario.
- **Escalabilidad**: el sistema distribuido debe ser capaz de hacer frente al aumento del número de nodos.
- **Heterogeneidad**: el sistema distribuido debe funcionar con un sistema heterogéneo, es decir, debe ser independiente de la plataforma.
- **Tolerancia a fallos**: el fallo de un nodo individual no debe afectar el logro del objetivo común del sistema distribuido.
- **Programación de tareas**: la política de distribución de tareas entre los distintos nodos debe seleccionarse de manera que se maximice el rendimiento.
- **Seguridad**: los datos importantes y/o confidenciales no deben ser revelados al usuario del nodo individual.

2.1.1. Estilos de arquitecturas distribuidas

Los estilos de arquitecturas distribuidas se formulan en términos de componentes, la forma en que estos se conectan entre sí, los datos intercambiados entre ellos y, por último, la forma en que estos elementos se configuran conjuntamente en un sistema. Un componente es una unidad modular con interfaces bien definidas, necesarias y proporcionadas, que puede sustituirse en su entorno.

Se identifican cuatro estilos importantes para los sistemas distribuidos [18], los cuales se presentan en la figura 2.1:

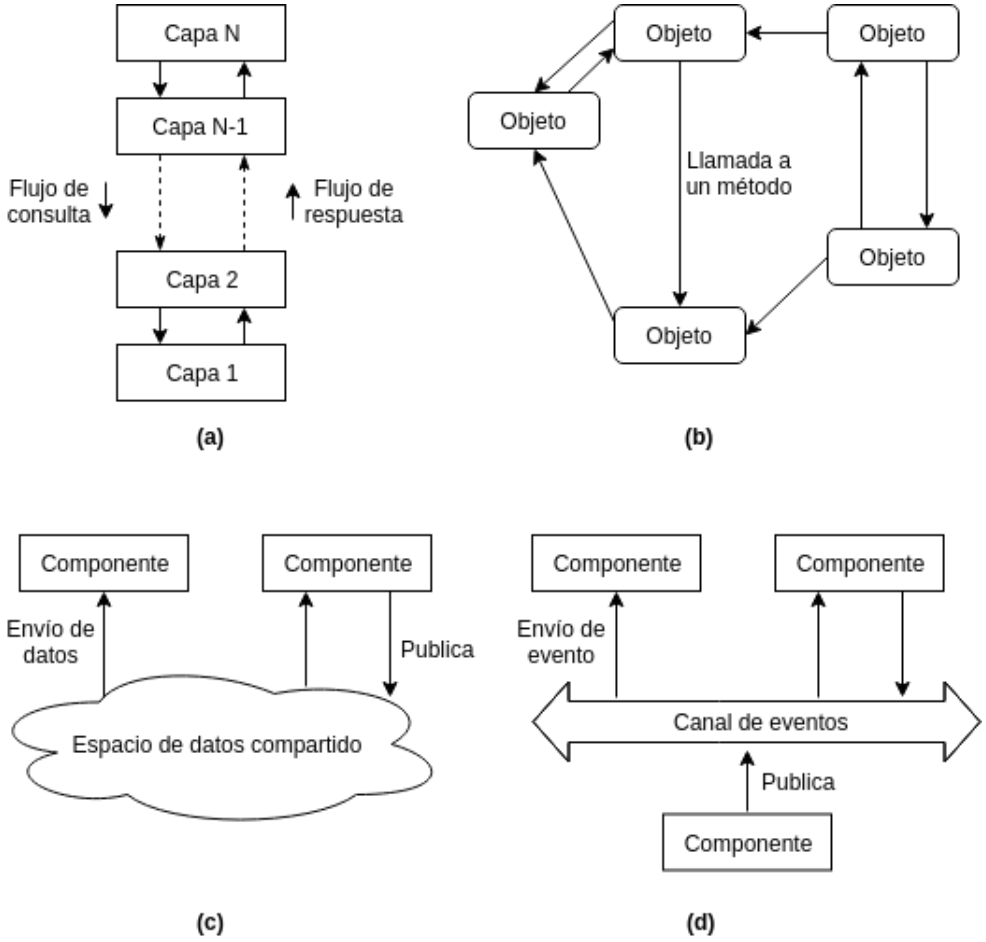

Figura 2.1: Estilos de arquitecturas distribuidas: (a) en capas (b) basada en objetos (c) centradas en datos (d) basadas en eventos.

- **Arquitecturas en capas**: los componentes están organizados en capas, donde cada capa puede comunicarse con la capa que lo antecede o que le sigue, ie. la capa L_{i} puede comunicarse con la capa $L_{\mathrm{i+1}}$ o la capa $L_{\mathrm{i-1}}$, pero no puede con $L_{\mathrm{i+2}}$ o $L_{\mathrm{i-2}}$. La comunicación fluye generalmente de capa en capa: las consultas bajan por las capas, mientras que los resultados fluyen hacia arriba, como podemos ver en la figura 2.1 (a).
- **Arquitecturas basadas en objetos**: cada objeto corresponde a un componente, y estos están conectados a través de un mecanismo de llamada (remoto), como podemos ver en la figura 2.1 (b). Los componentes son independientes y se pueden comunicar con

cualquier otro. Este estilo de arquitectura coincide con la arquitectura cliente-servidor que se describe más adelante.

- **Arquitecturas centradas en datos**: los diferentes componentes se comunican a través de un repositorio de datos común, como podemos ver en la figura 2.1 (c).
- **Arquitecturas basadas en eventos**: los componentes se comunican esencialmente a través de la propagación de eventos (condiciones que bloquean o permiten la ejecución de un componente), que opcionalmente también transportan datos. La idea básica es que los procesos publican eventos, de los cuales el *middleware*[1] asegura que sólo aquellos procesos que se suscribieron a esos eventos los recibirán.

La principal ventaja de los sistemas basados en eventos es que los procesos están acoplados, no es necesario que se comuniquen explícitamente entre sí; esto se denomina desacoplamiento en el espacio.

Las arquitecturas basadas en eventos pueden combinarse con arquitecturas centradas en datos, dando lugar a lo que se conoce como espacios de datos compartidos. La ventaja de los espacios de datos compartidos es que los procesos están desacoplados del tiempo (no es necesario que ambos estén activos cuando se produce la comunicación).

Como vimos anteriormente, los tipos de arquitecturas tienen como objetivo lograr la transparencia de la distribución, sin embargo, la transparencia de la distribución requiere sopesar el rendimiento, tolerancia a fallos, escalabilidad, etc. Como no existe una solución única que satisfaga todas las necesidades de las aplicaciones distribuidas, es que en la práctica se usan múltiples tipos de arquitecturas en una sola aplicación distribuida.

2.1.2. Tipos de arquitecturas distribuidas

En esta sección, nos referiremos a la arquitectura como un conjunto de patrones y componentes que definen una estructura, con la cual se comunican los diferentes componentes del sistema. Se definen arquitecturas distribuidas centralizadas y descentralizadas [18]:

Arquitecturas distribuidas centralizadas

Las arquitecturas distribuidas centralizadas se basan en el modelo **cliente-servidor** donde los procesos en el sistema se dividen en dos grupos (cliente y servidor). El servidor es un proceso que implementa un servicio específico (base de datos, sistema de archivos, etc). Un cliente es un proceso que solicita un servicio al servidor enviándole una petición y esperando la respuesta del servidor.

Un patrón de arquitectura muy popular es el **maestro-esclavo**, donde el servidor maestro es el encargado de mandar tareas a los esclavos, y estos a su vez, deben resolver las tareas para luego enviar un resultado al servidor maestro.

[1] Un *middleware* es el software que se sitúa entre un sistema operativo y las aplicaciones que se ejecutan en él.

Arquitecturas distribuidas descentralizadas

Las arquitecturas distribuidas descentralizadas se basan en la ausencia de un servidor central que proporcione un servicio o que gestione los recursos de red. En su lugar, todas las responsabilidades se dividen uniformemente entre todos los participantes o nodos (máquinas) que componen el sistema distribuido, teniendo cada nodo el rol de servidor como el de cliente.

Se identifican tres arquitecturas distribuidas descentralizadas [18], en la figura 2.2 podemos ver un ejemplo de cada una de ellas:

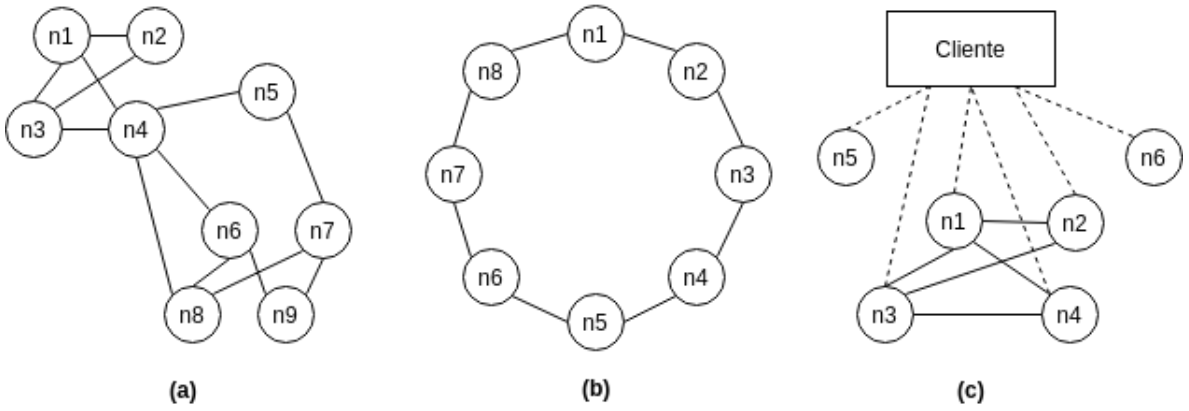

Figura 2.2: Ejemplo de arquitecturas P2P: (a) no estructurada (b) estructurada (c) híbrido.

- **Arquitecturas P2P no estructuradas**: no existe una estructura específica en esta arquitectura. Por esta razón, la escalabilidad de esta arquitectura es muy alta. Estos sistemas se basan en algoritmos aleatorios para seleccionar nodos vecinos al azar, como se puede ver en la figura 2.2 (a).
- **Arquitecturas P2P estructurada**: los nodos se organizan siguiendo una estructura de datos específica; en general, los nodos se forman en un anillo lógico. Cada red estructurada sufre intrínsecamente de una mala escalabilidad, debido a que la estructura se debe mantener. En la figura 2.2 (b) podemos ver una arquitectura P2P estructurada en un anillo lógico.
- **Arquitecturas P2P híbridas**: las arquitecturas híbridas a menudo se basan tanto en arquitecturas cliente-servidor como en redes P2P, como se puede ver en la figura 2.2 (c).

2.2. Cloud computing

Cloud computing [4] o computación en la nube es un paradigma de computación distribuida moderna que proporciona infraestructura *TI* y servicios esenciales, es decir, infraestructura como servicio (*IaaS*), plataforma como servicio (*PaaS*), software como servicio (*SaaS*), backend como servicio (*BaaS*), función como servicio (*FaaS*), etc. Actualmente la computación en la nube es un modelo muy utilizado e importante ya que permite el acceso a un conjunto de recursos computacionales compartidos, por ejemplo, redes, servidores, almacenamiento,

aplicaciones y servicios, que pueden ser rápidamente accedidos con un consumo muy bajo de recursos.

2.2.1. Utility computing

Utility Computing es un modelo de prestación de servicios en el que un proveedor de servicios pone a disposición del cliente los recursos informáticos y la gestión de la infraestructura según sus necesidades, y los cobra por un uso específico en lugar de una tarifa plana. Al igual que otros tipos de computación bajo demanda, el modelo de utilidad busca maximizar el uso eficiente de los recursos y/o minimizar los costos asociados.

Una forma de *Utility Computing* es *Serverless computing* que es un modelo de ejecución en *Cloud computing* en el que el proveedor de la nube actúa como servidor, gestionando dinámicamente la asignación de recursos de la máquina. En este modelo el desarrollador no ve directamente el servidor y en cambio el código corre en "ambientes de ejecución" que administran proveedores externos, quienes cobran por el tiempo de ejecución del código; mientras más rápido termine la función, menor es el costo que se paga.

2.3. Cluster computing

Cluster computing o la computación en clúster [11] es el paradigma de computación distribuida en el que se pueden conectar nodos (computadoras) en una red local para usarlos en conjunto con un fin común. Los componentes de un clúster suelen estar conectados entre sí a través de redes locales rápidas, con cada nodo (computador utilizado como servidor) ejecutando su propia instancia de un sistema operativo. En la mayoría de los casos, todos los nodos utilizan el mismo hardware y el mismo sistema operativo, aunque dependiendo de la implementación, se pueden utilizar sistemas operativos diferentes en cada ordenador o hardware diferente.

Por lo general, los clústeres se despliegan para mejorar el rendimiento y la disponibilidad en comparación con un solo equipo, mientras que normalmente son mucho más rentables que los equipos individuales de alta velocidad o disponibilidad [2]. Tienen una amplia gama de aplicaciones y despliegues, que van desde clústeres en pequeñas empresas con pocos nodos hasta algunos de los superordenadores más rápidos del mundo.

2.3.1. Herramientas de cluster computing

Actualmente existen muchas herramientas como *frameworks*, *kernels* y librerías que pueden ser utilizadas para gestionar clústeres. En el presente se analizarán 5 herramientas, las cuales se detallan a continuación:

- **Apache Mesos**[2]**:** es un kernel que se ejecuta en cada máquina y proporciona aplicaciones (por ejemplo, Hadoop, Spark, Kafka, Elasticsearch) con API's para la gestión de recursos y programación en entornos de centro de datos y cloud.
- **Kubernetes**[3]**:** es un sistema para la administración de contenedores Docker (es una aplicación de virtualización utilizado para desplegar y ejecutar aplicaciones distribuidas sin lanzar una máquina virtual completa para cada aplicación). Se encarga de gestionar las cargas de trabajo en función de parámetros definidos por el usuario.
- **Amazon ECS**[4]**:** servicio que soporta contenedores Docker y permite ejecutar aplicaciones en un clúster en instancias de Amazon EC2.
- **Apache Helix**[5]**:** es un framework de gestión de clústeres genérico utilizado para la gestión automática de recursos particionados, replicados y distribuidos alojados en un clúster de nodos.
- **Swarm**[6]**:** proporciona nativamente la funcionalidad de clúster para contenedores Docker, lo que permite convertir un grupo de nodos Docker en un único motor Docker virtual.

A continuación se describen 10 características relevantes para sistemas de *cluster computing*:

1. **Código abierto:** modelo de desarrollo de software basado en la colaboración abierta, lo que significa que su código fuente es de acceso público.
2. **Gratis:** a pesar de que un software sea de código abierto, se puede cobrar por su uso o servicios asociados. La gratuidad implica que a medida que un proyecto escala, el costo en este ítem no escala, lo que permite distribuir el presupuesto en otras áreas del proyecto.
3. **Interfaz gráfica:** tener acceso a una visualización de las tareas y nodos conectados al sistema.
4. **Balanceador de carga:** como su nombre lo indica, asigna o balancea las solicitudes que llegan de los clientes a los nodos usando algún algoritmo (*round-robin*, *FIFO*, *LIFO*, etc.)
5. **Chequeo de los nodos:** chequear el estado de los nodos es importante para saber si se puede repartir tareas o recursos a estos. En este proyecto es un aspecto crítico ya que los dispositivos pueden no tener un comportamiento estable.
6. **Tolerancia a fallos:** es la capacidad del sistema de disminuir la cantidad de fallos con el aumento de carga en tareas y recursos.
7. **Adaptabilidad:** nivel de dificultad para adaptar, modificar o configurar soluciones personalizadas usando como base la herramienta.
8. **Nodos soportados:** cantidad promedio de nodos soportados por un servidor maestro.
9. **Comunidad:** cantidad de usuarios que aportan y dan soporte a la comunidad de una herramienta. Muy importante al momento de tener algún problema difícil de solucionar.

[2]Apache Mesos (http://mesos.apache.org)
[3]Kubernetes (https://kubernetes.io)
[4]Amazon ECS (https://aws.amazon.com/ecs)
[5]Apache Helix (http://helix.apache.org)
[6]Swarm (https://docs.docker.com/engine/swarm)

10. **Lenguajes de programación:** es importante saber en que lenguaje de programación se puede desarrollar e implementar una solución.

En la tabla comparativa 2.1 podemos ver las deferentes características por cada herramienta de *cluster computing*.

Tabla 2.1: Tabla comparativa frameworks de cluster computing

	Mesos	Kubernetes	Amazon ECS	Helix	Swarm
Código abierto	✓	✓	✓	✓	✓
Gratis	✓	✓	✗	✓	✓
Interfaz gráfica	✓	✓	✓	✗	✗
Balanceador de carga	✓	✓	✓	✓	✓
Chequeo de los nodos	✓	✓	✓	✓	✓
Tolerancia a fallos	✓	✓	✓	✓	✗
Adaptabilidad	✓	✓	✓	✓	✓
Nodos soportados	10000+	5000+	1000	∼1000	∼1000
Documentación	Mucha	Media	Media	Pequeña	Grande
Lenguajes	Cualquiera	Go	Cualquiera	Java	Cualquiera

2.4. Grid computing

Grid computing [3] o computación en malla es una forma de conectar recursos informáticos para compartir su potencia de cálculo usando las redes informáticas que permiten el acceso a los recursos informáticos desde muchas ubicaciones diferentes. Estos recursos informáticos incluyen capacidad de almacenamiento de datos, potencia de cálculo, sensores, herramientas de visualización, entre varios.

La malla puede considerarse como un sistema distribuido con cargas de trabajo independientes entre sí, que procesan una gran cantidad de información para después reunir los resultados individuales. La computación en malla se diferencia de los sistemas de computación distribuida convencionales de alto rendimiento, como los clústeres, en que los nodos están configurados para realizar tareas diversas, con diferentes recursos y en general están geográficamente dispersos.

Los proyectos más destacados de *grid computing* involucran la computación voluntaria [10], donde los recursos computacionales son proporcionados voluntariamente para hacer uso de su poder. Generalmente este tipo de computación se usa para resolver problemas que pueden descomponerse en subproblemas de menor dimensión y procesarse de forma independiente (en paralelo); la descomposición debe realizarse de tal manera que cada subproblema elemental pueda ser resuelto incluso por un nodo de computación débil.

Dentro de un sistema de computación voluntario, hay cuatro roles que interactúan entre sí [14]:

1. **Voluntarios**: aquellos que ofrecen voluntariamente sus ordenadores.
2. **Presentadores**: aquellos que "presentan" tareas para ser calculadas o resueltas.
3. **Desarrolladores**: aquellos que desarrollan el código que se ejecuta en las computadoras de los voluntarios.
4. **Facilitadores**: el ambiente de trabajo que conecta a los desarrolladores, presentadores y voluntarios.

Existen una infinidad de proyectos donde personas donan voluntariamente el poder de sus computadores o dispositivos móviles[19] para ayudar en problemas tales como: renderización de animaciones 3D, correr simulaciones de experimentos, estudiar conjeturas matemáticas, optimización de variables y parámetros en *Machine Learning*, estudiar estructuras de proteínas y moléculas, clasificación de galaxias, predicción del clima, entre un sinfín de aplicaciones posibles en el área de investigación.

2.4.1. Plataformas existentes

Los conceptos de computación distribuida y de computación voluntaria se introdujeron hace mucho tiempo y existen varios sistemas que los utilizan para proporcionar una plataforma para crear aplicaciones que se ejecutarán en nodos de voluntarios independientes. Dos plataformas de *grid computing* muy poderosas son *BOINC* y *WCG*.

BOINC

BOINC (*Berkeley Open Infrastructure for Network Computing*) [1] es un sistema que facilita a los científicos la creación y operación de proyectos informáticos con recursos de participantes voluntarios. Soporta diversas aplicaciones, incluyendo aquellas con grandes requerimientos de almacenamiento o comunicación. Los propietarios de computadores pueden participar en múltiples proyectos BOINC, y pueden especificar cómo se asignan sus recursos entre estos proyectos.

Los objetivos de BOINC son:

- Reducir las barreras de entrada a la computación voluntaria.
- Compartir recursos entre proyectos autónomos.
- Soportar diversas aplicaciones, adaptarse a una amplia gama de aplicaciones, proporcionar un mecanismo flexible y escalable para la distribución de datos.
- Recompensar a los participantes.

Proyectos de investigación que actualmente funcionan y se ven beneficiados gracias a BOINC y a los voluntarios que ofrecieron sus computadores son:

- **SETI@home**: proyecto cuyo propósito es analizar señales de radio, buscando señales de inteligencia extraterrestre, y como tal es una de las muchas actividades emprendidas como parte del esfuerzo mundial del SETI.

- **Climateprediction.net**: proyecto para investigar y reducir las incertidumbres en la modelización climática.
- **Cosmology@Home**: proyecto cuyo objetivo es comparar los modelos teóricos del universo con los datos medidos hasta la fecha y buscar el modelo que mejor se adapte a ellos.
- **Einstein@Home**: proyecto que busca señales de estrellas de neutrones giratorias en datos de los detectores de ondas gravitacionales LIGO, de grandes radiotelescopios y del Telescopio Espacial de Rayos Gamma Fermi.
- **MilkyWay@home**: proyecto cuyo objetivo es generar modelos dinámicos tridimensionales precisos de corrientes estelares en las inmediaciones de la Vía Láctea.
- **Quake-Catcher Network**: proyecto que tiene como objetivo el uso de acelerómetros computarizados para la detección de terremotos.
- **Rosetta@home**: proyecto cuyo objetivo es la predicción de estructuras proteicas.

WCG

WCG (*World Community Grid*) [16] es un esfuerzo por crear la red de computación pública más grande del mundo para abordar proyectos de investigación científica que beneficien a la humanidad. Lanzado el 16 de noviembre de 2004, es coordinado por IBM y hace uso de la plataforma BOINC. World Community Grid permite a cualquier persona que disponga de un dispositivo tecnológico donar su potencia de computación no utilizada para usarla en investigación científica de vanguardia; los investigadores utilizan WCG para abordar en gran medida problemas de salud; analizado aspectos del genoma humano, VIH, dengue, distrofia muscular, cáncer, influenza, ébola, entre otros.

2.5. Crawling/scraping

Crawling y *Scraping* son técnicas para obtener e indexar contenido de la red (particularmente de páginas web) de forma automatizada [5].

Un *crawler* es el software encargado de realizar el crawling. Este comienza con una lista de URLs a visitar, llamadas semillas. A medida que el *crawler* visita estas URLs, identifica todos los enlaces de la página actual y los agrega a la lista de URLs a visitar, repitiendo recursivamente los pasos anteriores, deteniéndose en una cierta profundidad o nivel [13].

El crawling se diferencia del scraping en que se centra en obtener e indexar el contenido de páginas web de forma automatizada. En cambio, el scraping se enfoca en procesar información de sitios web y extraer información específica de ellos, transformando los datos de formato HTML a estructuras relacionales, para poder almacenarlos en bases de datos y analizarlos posteriormente.

Múltiples centros de investigación y empresas utilizan estas técnicas para obtener datos de diferentes fuentes públicas y poder procesarlas localmente.

2.5.1. Políticas de crawling

El comportamiento de un *crawler* es el resultado de una combinación de políticas [7]:

- **Política de selección** que indica las páginas a descargar. Esto requiere una métrica de importancia para priorizar las páginas web. La importancia de una página depende de su calidad, de su popularidad en términos de enlaces o visitas. El diseño de una buena política de selección tiene una dificultad añadida: debe trabajar con información parcial, ya que no se conoce el conjunto completo de páginas web durante el *crawling*.
- **Política de re-visitas** que indica cuándo comprobar si hay cambios en las páginas. Debido a que muchas páginas web son dinámicas o su contenido es actualizado continuamente, en el momento en que un *crawler* ha terminado de realizar su trabajo, muchos eventos podrían haber ocurrido, incluyendo creaciones, actualizaciones y eliminaciones.
- **Política de cortesía** que establece cómo evitar sobrecargar los sitios web. Los *crawlers* pueden recuperar datos mucho más rápido que los usuarios humanos, por lo que pueden tener un impacto en el rendimiento de un sitio.
- **Política de paralelización** que establece cómo coordinar los *crawlers* web distribuidos. Un *crawler* paralelo es un *crawler* que ejecuta múltiples procesos en paralelo. El objetivo es maximizar la tasa de descarga mientras se minimiza la sobrecarga de la paralelización y evitar consultas repetidas de la misma página.

Tanto la **política de selección** como la **política de re-visitas** pueden ser definidas por el encargado de una página web en su archivo *robots.txt* (explicado en breve), pero la **política de cortesía** y la **política de paralelización** deben ser implementadas por el creador del *crawler*.

2.5.2. Robots.txt

Dado que un sitio web necesita tratar con muchos *crawlers* (en su mayoría motores de búsqueda) usando sus recursos, necesita regular su comportamiento. El protocolo de exclusión de robots[7] es una solución parcial a este problema de regulación. Para utilizar este protocolo, un sitio web normalmente especificará sus reglas de protocolo en un archivo llamado *robots.txt*. Las reglas establecen que el sitio permita o rechace parte o todo su contenido a *crawlers* específicos [17].

2.6. Crawling distribuido

Hay dos componentes que hacen muy difícil recolectar e indexar datos de internet, el primero es el volumen de datos que crecen en todo momento y el segundo es la velocidad de cambio; el gran volumen implica que un *crawler* solo puede descargar una porción de las páginas web en un tiempo determinado, y la velocidad de cambio implica que para cuando

[7]http://www.robotstxt.org/robotstxt.html

un *crawler* haya descargado las últimas páginas de un sitio, es muy probable que se hayan añadido nuevas páginas, o que las páginas ya hayan sido actualizadas o incluso borradas del sitio.

Es por lo antes mencionado que el *crawling* a gran escala se realiza de forma distribuida, utilizando varias máquinas para indexar contenido de internet. La mayoría de los buscadores modernos utilizan esta técnica, como Google y Yahoo que utilizan miles de máquinas individuales para *crawlear* internet.

Existen dos políticas [8] de crawling distribuido:

- **Asignación dinámica:** un servidor central asigna nuevas URLs a diferentes *crawlers* de forma dinámica, esto permite por ejemplo, equilibrar dinámicamente la carga de cada *crawler*. Normalmente el servidor central puede convertirse en un cuello de botella, por lo que la mayor parte de la carga de trabajo debe transferirse a los *crawlers*.
- **Asignación estática:** hay una regla fija establecida desde el principio del *crawling* que define cómo asignar nuevas URLs a los *crawlers*; se puede utilizar una función de *hash*[8] para transformar URLs en un identificador que corresponda al índice del proceso de rastreo correspondiente.

En la actualidad muchos proyectos de *crawling* distribuido usan la colaboración, reclutando voluntarios para que presten el poder de sus computadores personales.

2.6.1. Trabajos de crawling distribuido

Existen muchos trabajos y proyectos relacionados al crawling distribuido, a continuación mencionaremos algunos de ellos:

- **Apache Nutch**[9]**:** es un proyecto de web crawling de código abierto altamente escalable. En conjunto con *Apache Hadoop*[10], se puede crear un sistema distribuido de web crawling usando el poder de varias máquinas en un clúster.
- **YaCy**[11]**:** es un motor de búsqueda gratuito, distribuido y construido sobre los principios de P2P. Puede ejecutarse en modo *crawling* y colaborar voluntariamente con la descarga de contenido, o ejecutarse en modo *proxy* indexando las páginas web visitadas por la comunidad de YaCy. El acceso a las funciones de búsqueda se realiza a través de un servidor web que funciona localmente y que proporciona un cuadro de búsqueda para introducir los términos de búsqueda, devolviendo los resultados de la búsqueda en un formato similar al de otros motores de búsqueda populares.
- **UbiCrawler[6]:** es un trabajo realizado el 2003 el cual abarca la creación de un crawler distribuido usando el lenguaje Java, se centra en la escalabilidad del sistema, la

[8]Una función de hash es un algoritmo criptográfico que transforma un bloque arbitrario de datos en una nueva serie de caracteres con una longitud fija.

[9]Apache Nutch (https://nutch.apache.org/)

[10]Apache Hadoop es un framework que permite el procesamiento distribuido de grandes conjuntos de datos a través de clústeres de computadores.

[11]YaCy (https://yacy.net/)

tolerancia a fallos y el balance de distribución en el proceso de *crawling*.

Otros proyectos que utilizan el *crawling* distribuido como elemento clave dentro de sus procesos son los conocidos buscadores comerciales, como Google, Yahoo, Bing, entre otros.

2.7. Frameworks de aplicaciones web

Los frameworks[12] de aplicaciones web son aquellos que hacen más fácil escribir, mantener y escalar aplicaciones web.

Proporcionan herramientas y bibliotecas que simplifican tareas comunes de desarrollo web, tales como manejo de bases de datos, enrutar URLs, manejo y administración de usuarios, seguridad en la programación.

Además la mayoría siguen patrones de diseño aceptados ampliamente por la comunidad informática, como por ejemplo, el patrón MVC, que es un patrón de arquitectura de software, que separa los datos, la lógica y las vista en tres componentes distintos, haciendo mucho más entendible y escalable un proyecto TI.

2.7.1. Django

Django[13] es un *framework* de desarrollo web de código abierto, escrito en *Python*, que sigue el patrón de diseño MVC.

Se destaca por su gran comunidad y su avanzando desarrollo. Se libera su código en 2005 y desde entonces a la fecha el proyecto se mantiene activo y con actualizaciones periódicas, hoy en día el proyecto se mantiene en manos de la fundación *Django Software Foundation* (fundada el 2008), que permanentemente mejoran el *framework*.

Django además de ofrecer una gran comunidad, un patrón de diseño MVC y propiciar las buenas prácticas en la programación, cuenta con soporte a múltiples motores de bases de datos, ofrece seguridad en la programación evitando errores comunes, rapidez y por su carácter modular es altamente escalable.

2.7.2. Tornado

Tornado[14] es un *framework* web de código abierto, escrito en *Python*, que cuenta con bibliotecas de soporte para redes asíncrona. Mediante el uso de E/S de red sin bloqueo,

[12]Un framework es un conjunto estandarizado de conceptos, prácticas, criterios y herramientas que sirven para resolver de forma más fácil problemas recurrentes.
[13]Django (https://www.djangoproject.com/)
[14]Tornado (http://www.tornadoweb.org/)

Tornado puede escalar a decenas de miles de conexiones abiertas, lo que lo hace ideal para aplicaciones que requieran múltiples conexiones o conexiones de larga duración.

Tornado es conocido por su alto rendimiento e intenta resolver el problema C10K [12] que es el problema de optimizar los *sockets*[15] de red para manejar un gran número de clientes al mismo tiempo, que afecta a millones de servidores en el mundo.

Tiene además soporte para bases de datos no relacionales y descentralizadas, lo que ayuda en rendimiento y rapidez general del sistema.

2.8. Resumen

En este capítulo se describieron los principales conceptos y tecnologías involucradas en el desarrollo de la memoria propuesta.

Se definió la computación distribuida y los principales desafíos que debe enfrentar: transparencia, escalabilidad, heterogeneidad, tolerancia a fallos, programación de tareas y seguridad. Se vio algunos estilos de arquitecturas distribuidas como las arquitectura en capas, basada en objetos, centrada en datos y basada en eventos; se detalló sobre estilos de arquitecturas distribuidas como las centralizadas donde destaca el modelo cliente-servidor y el estilo de arquitecturas distribuidas descentralizadas donde sobresalen las arquitecturas P2P no estructuradas, estructuradas e híbridas.

Se explicó el concepto de *cloud computing* y su importancia para diferentes tipos de infraestructuras como las *IaaS*, *PaaS*, *SaaS*, *BaaS*, *FaaS*, entre otros. Además se menciona el *utility computing*, un modelo de prestación de servicios en el cual se cobra al cliente solo por los recursos usados.

Se desarrolló el concepto de *cluster computing* donde computadores se conectan en una red local, y su importancia en el rendimiento y la disponibilidad de un sistema. Se comparó diferentes herramientas de *cluster computing* como *frameworks*, *kernels* y librerías.

Se expuso sobre *grid computing* y su forma de conectar recursos informáticos para compartir poder de cálculo, almacenamiento y ancho de banda. Se vio que los proyectos más destacados de *grid computing* involucran la computación voluntaria, donde interactúan cuatro roles entre sí; los voluntarios, los presentadores, los desarrolladores y los facilitadores. Además se presentan diferentes plataformas de *grid computing* y algunos proyectos que se llevan a cabo en ellas.

De tópicos sobre sistemas distribuidos, se pasa a explicar sobre *crawling* y *scraping* que son técnicas para obtener e indexar contenido de internet de forma automatizada. Asimismo se muestran políticas que debe seguir un *crawler* para no entrar en conflicto con la página web objetivo.

[15] Un *socket* es un mecanismo para la entrega de paquetes de datos provenientes de la tarjeta de red a los procesos correspondientes.

Con el estado del arte de los sistemas distribuido y el *crawling*, se introduce el *crawling* distribuido, para explicar cómo funciona y su alcance, incluyendo además importantes trabajos en el área.

Por último se cambia de tema, para hablar sobre frameworks de aplicaciones web, específicamente *Django* y *Tornado*, que son utilizados para la implementación del proyecto, cumpliendo un rol fundamental en el desarrollo de esta memoria.

Capítulo 3

Diseño y arquitectura

En este capítulo se se detalla el diseño y arquitectura del proyecto bautizado *Crawlsourcing*, con el cual se pretende crear un sistema distribuido de consultas HTTP/S para realizar *crawling* y *scraping*, con la particularidad de usar el potencial de dispositivos tecnológicos cotidianos para acoplarlos a una red colaborativa que hará las consultas HTTP/S distribuidamente.

Actualmente los computadores personales y dispositivos inteligentes pasan una gran parte del tiempo encendidos, donde se observa alta inactividad o muy poco uso (el caso más emblemático es el celular); poder aprovechar esa capacidad ociosa con un gasto energético menor, automatizadamente y basada en la filosofía de los incentivos (en la sección 3.2 se explica el financiamiento del sistema), crea un modelo colaborativo beneficioso tanto para el que necesita consultar datos de una web, como para los dispositivos tecnológicos que componen la red distribuida.

3.1. Discusión

Para lograr el objetivo y hacer uso de la capacidad ociosa de dispositivos tecnológicos, hay dos componentes que no son modificables:

1. **Clientes:** en el esquema de computación voluntaria son los presentadores, aquellos que necesitan *crawlear* o *scrapear*.
2. **Voluntarios:** al igual que en el esquema de computación voluntaria, son aquellos que ofrecen sus recursos voluntariamente, en este caso para resolver consultas HTTP/S.

Existe una infinidad de arquitecturas posibles que pueden ser usadas para resolver el problema de distribuir las tareas de los clientes entre los voluntarios de forma justa y eficiente, pero se decide analizar dos arquitecturas como las que mejor se adaptan al problema, las que podemos ver en la figura 3.1:

1. **Arquitectura P2P híbrida:** en esta arquitectura contamos tanto con voluntarios

que resuelven las consultas HTTP/S, como clientes quienes son los que proponen las consultas HTTP/S. Un servidor central almacena las peticiones de los clientes y las direcciones de nodos voluntarios con consultas HTTP/S resueltas; además gestiona al sistema de incentivos para que voluntarios participen del sistema. Un nodo puede ser cliente y voluntario a la vez, puede ayudar a otros a *crawlear* y a su vez proponer consultas HTTP/S. Los voluntarios son libres de resolver consultas HTTP/S obtenidas del servidor o de replicar la información de otros voluntarios.

2. **Arquitectura Maestro/Esclavo:** en esta arquitectura existe uno o más servidores centrales, que llamaremos maestro, que recibe las consultas HTTP/S de los clientes y las envía a los voluntarios, que llamaremos esclavos. Este servidor distribuye las consultas HTTP/S y las asigna a esclavos según métricas de rendimiento u otra estrategia, una vez que el esclavo retorna la respuesta, el servidor maestro la procesa y envía al cliente.

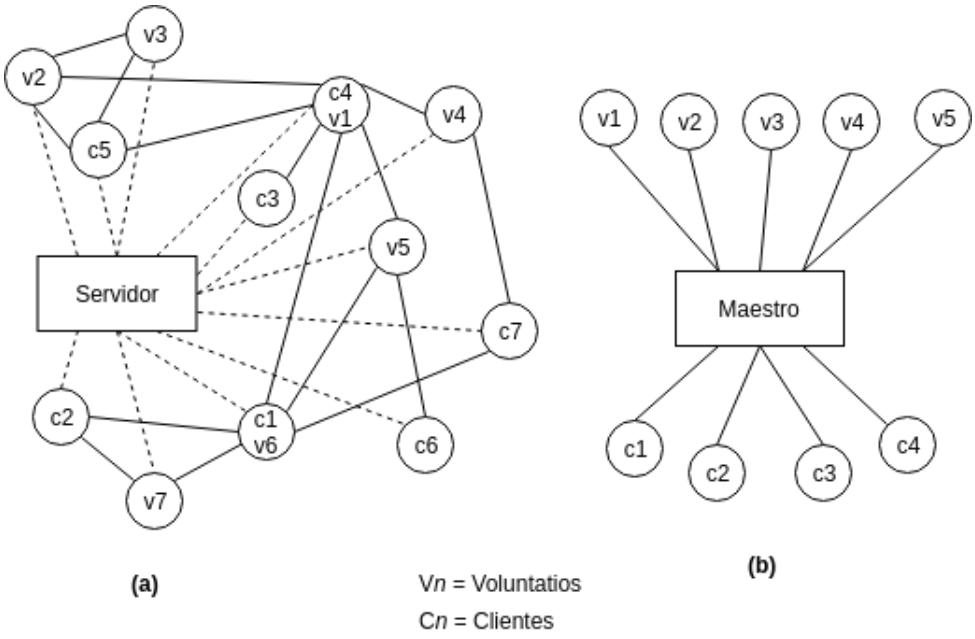

Figura 3.1: Posibles arquitecturas (a) Arquitectura P2P (b) Arquitectura Maestro/Esclavo.

En la tabla 3.1 se compara y analiza las ventajas y desventajas de las dos arquitecturas antes mencionadas, en ella se mencionan aspectos relevantes para el desempeño y requisitos del proyecto como la privacidad, el almacenamiento de los datos, la asignación de consultas HTTP/S, el manejo del *cheating* o detección de engaño y la sobrecarga en el servidor que presenta cada una de las arquitecturas.

3.2. Arquitectura

En la sección anterior se analizó dos posibles arquitecturas, en esta memoria se a decido por la arquitectura Maestro/Esclavo, ya que esta arquitectura nos permite realizar tareas de suma importancia para el funcionamiento del sistema en general, como lo es el manejo de *cheating* y tener el control de la asignación de consultas HTTP/S; otro aspecto importante

Tabla 3.1: Comparación de arquitecturas P2P híbrida vs Maestro/Esclavo.

	P2P híbrida	**Maestro/Esclavo**
Privacidad	No existe privacidad entre el cliente y el voluntario que resuelve la consulta. Se puede solucionar con un sistema de saltos entre nodos, los cuales funcionen como *proxy* para enmascarar el verdadero destino de las consultas HTTP/S (parecido al proyecto Tor[1]), pero eso aumentaría los tiempos y el rendimiento general del sistema.	La privacidad de las consultas tanto de los clientes como de los voluntarios quedan enmascaradas por el servidor maestro (asumiendo confianza en el servidor maestro por parte de los clientes).
Almacenaje	La principal ventaja de esta arquitectura es que el servidor central no tiene que almacenar los resultados de las consultas HTTP/S, delegando el almacenamiento a los nodos voluntarios.	El servidor maestro es el encargado de almacenar todas las consultas HTTP/S y sus respuestas.
Asignación de consultas	La asignación de consultas se hace en base al algoritmo *Gnutella*[15], ampliamente usado en distribución de archivos entre pares. No se pueden usar estrategias en base al rendimiento de los voluntarios ya que esos datos no son administrados por el servidor.	En esta arquitectura se pueden crear estrategias en base al comportamiento de los voluntarios, de esta forma se pueden ir asignando las consultas HTTP/S de manera conveniente según diferentes parámetros.
Manejo de *cheating*	Tanto voluntarios como clientes pueden mentir, por esto es que cualquier proceso que ocurra de lado de ellos puede ser modificable. Por lo tanto, el servidor debe ser el encargado de verificar la consistencia de los datos, pero para hacerlo debe descargarlos y verificarlos, rompiendo con su principal ventaja que es evitar cuellos de botella.	Se pueden crear diferentes tipos de estrategias para verificar la consistencia de los datos (en la sección 3.8 se habla más de esto).
Sobrecarga en el servidor	El rol del servidor es simplemente administrar las consultas HTTP/S de los clientes y entregar la lista de tareas a los voluntarios. El servidor no resuelve las consultas ni almacena los resultados por lo cual su consumo de recursos solo depende de las conexiones establecidas por los clientes y los voluntarios, para saber a quién consultar contenido o que tarea realizar.	Todas las consultas HTTP/S hechas por los clientes como las respuestas hechas por los voluntarios deben pasar por el servidor maestro, por lo cual genera un gasto tanto en ancho de banda como en procesamiento para lograr su cometido.

es la privacidad tanto de los clientes, como de los voluntarios; y un punto importante es que la arquitectura elegida es más sencilla para lograr el objetivo propuesto y de implementar.

La arquitectura Maestro/Esclavo ayuda además a realizar menos consultas HTTP/S de las necesarias, ya que al centralizar las respuestas de las consultas, se pueden guardar en un sistema de *caching* (proceso de almacenar datos en caché) para resolver de forma más rápida consultas populares y/o repetitivas de sitios web estáticos. De esa forma las consultas ya almacenadas no se necesitarán hacer más de una vez. Esto ayudará a descongestionar las consultas hechas a un sitio web objetivo.

Existen tres componentes claves en el funcionamiento del sistema, como podemos ver en la figura 3.2:

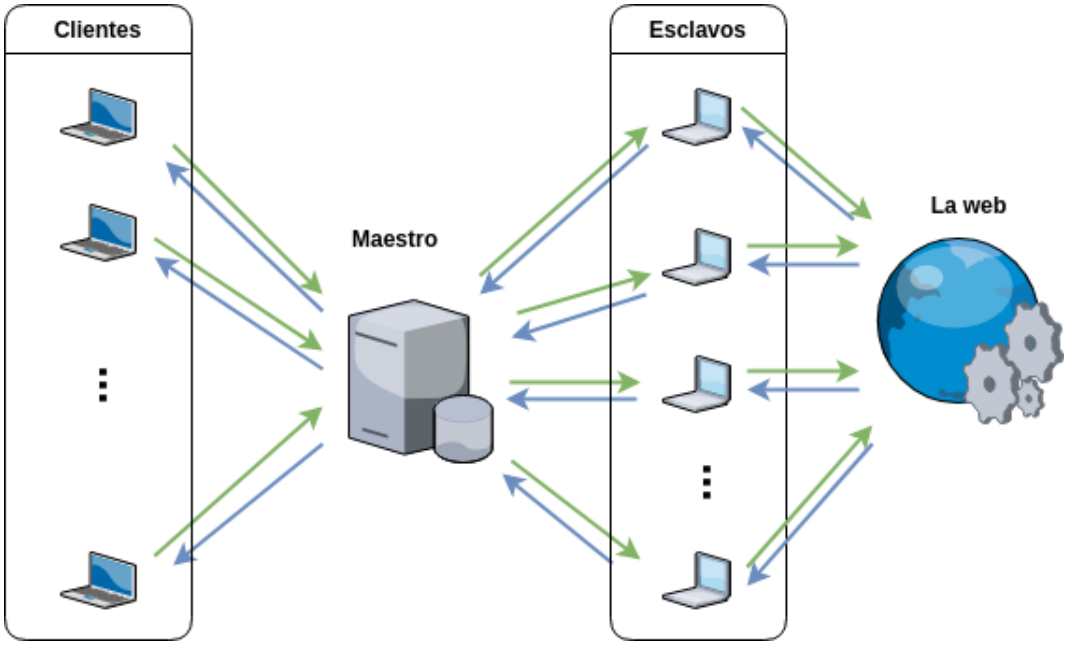

Figura 3.2: Arquitectura de Crawlsourcing

1. **Clientes:** al igual que en la sección anterior son aquellos que necesitan *crawlear* o *scrapear*, usando *Crawlsourcing* a través de una API.
2. **Dispositivos tecnológicos:** (servidores esclavos de ahora en adelante) son aquellos que reciben consultas HTTP/S del servidor maestro, donde las consultas se resuelven y cuyo resultado es devuelto al servidor maestro, para que este las envíe al cliente original.
3. **Servidor maestro:** (en el esquema de computación voluntaria es el desarrollador y el facilitador) encargado de recibir, administrar y distribuir las consultas HTTP/S.

Para mejorar el rendimiento del sistema en general, los esclavos servirán un bloque de consultas HTTP/S y retornarán una respuesta compactada de ellas. El camino que toma el bloque de consultas HTTP/S desde que la hace el cliente hasta que recibe la respuesta, se detalla en la figura 3.3:

1. El cliente manda consultas HTTP/S al servidor maestro.
2. El servidor maestro recibe las consultas HTTP/S del cliente, hace un paquete de con-

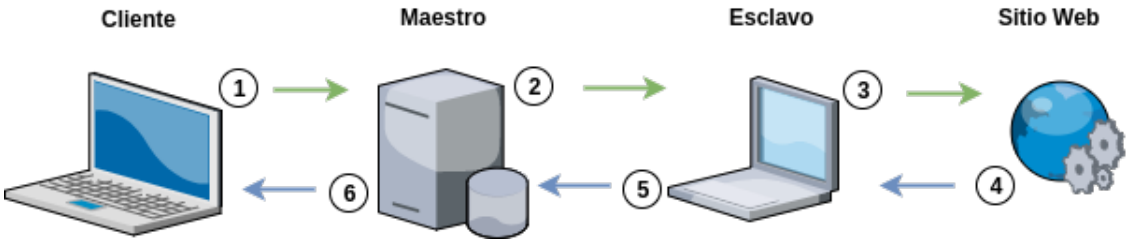

Figura 3.3: Flujo de una consulta HTTP/S a través de Crawlsourcing

sultas HTTP/S y la distribuye a un esclavo o más.

3. Un esclavo recibe el paquete de consultas HTTP/S y hace las consultas HTTP/S a los sitios web objetivos.
4. El sitio web recibe la consulta HTTP/S individual y envía la respuesta de vuelta al esclavo.
5. El esclavo recibe las respuestas, las comprime y se las envía al servidor maestro en un solo paquete.
6. El servidor maestro recibe la respuesta y retorna el resultado de las consultas a origen (cliente).

Las posibles ventajas para el usuario final (el cliente) con respecto a otras soluciones existentes, son una mejora en el tiempo de consultas y un menor costo monetario (ya que la ganancia de los esclavos se pueden ajustar para disminuir el costo para el cliente; en la sección 3.2 se especifica el modelo de financiamiento) para realizar el trabajo de scraping y crawling.

La mayor ventaja para los esclavos y servidor maestro es participar de un sistema colaborativo donde pueden obtener beneficios económicos por el uso de sus dispositivos conectados a internet con un gasto muy menor; más adelante se especifica mejor la forma en la que el servidor maestro y los esclavos ganan.

Por el lado de las páginas web se busca hacer un *crawling* ético, por lo cual se respetará su archivo *robots.txt* (como se ve en la sección 3.4), se tiene un máximo global de consultas que se pueden hacer en simultáneo al sitio y se cuenta con un sistema de *caching* que impide que consultas populares o repetidas se vuelvan a realizar en el servidor web.

3.3. Requerimientos

Para que funcione correctamente el sistema planteado, las responsabilidades del **servidor maestro** son:

- Administrar credenciales de clientes.
- Administrar credenciales de usuarios (red colaborativa).
- Administrar esclavos.
- Recibir consultas HTTP/S de clientes.

- Distribuir consultas HTTP/S a esclavos.
- Recibir respuesta de consultas HTTP/S.
- Enviar respuesta de consultas HTTP/S al cliente.

El servidor maestro se encarga también de distribuir las consultas HTTP/S limitándose a un máximo simultáneo de peticiones a un mismo sitio web, para evitar que el sitio web se vea sobrepasado por la cantidad de peticiones.

Las consultas se distribuyen por prioridad; cada esclavo contiene indicadores y métricas como el tiempo de respuesta, consultas realizadas con éxito y consultas en cola de espera. Así se puede elegir qué esclavo resolverá qué consulta.

Otra consideración importante es el hecho de que el sistema se rige bajo las directrices impuestas por cada servidor web en su *robots.txt*, por ello, en cada consulta se debe verificar su correspondiente *robots.txt* para ver si se puede *crawlear* y cuál es el tiempo de retraso entre consultas.

Una de las características de los esclavos es que son dispositivos tecnológicos cotidianos, por tanto, pueden tener intermitencia en su disponibilidad. El sistema debe ser capaz de reasignar las tareas de un esclavo que se desconecte, muera o haga un trabajo muy lento.

Los **esclavos**, propiedad de los usuarios conectados a la red de *Crawlsourcing*, deben:

- Recibir consultas HTTP/S del servidor maestro.
- Hacer las consultas HTTP/S.
- Enviar las respuestas de consultas HTTP/S al servidor maestro.

Los esclavos deben conectarse con el servidor maestro tanto para tomar tareas como para enviar el resultado; en el caso de que no haya tareas, para evitar una conexión abierta por cada esclavo esperando por tareas, solo se mantendrán conectados algunos esclavos y a otros se les pedirá que vuelvan a conectarse tiempo después para constatar si hay tareas nuevas.

Para hacer más eficiente la comunicación y descongestionar el sistema en general, a los esclavos se les asignará bloques (una lista) de consultas para que las resuelvan y luego envíen los resultados compactados en un solo paquete, así se disminuyen la cantidad de conexiones del esclavo para pedir tareas, y por ende mejora el desempeño en general.

Tanto servidor maestro como los servidores esclavos tienen un umbral de peticiones; de esta forma el esclavo no supera un umbral local de peticiones a un sitio web en particular (para evitar ser bloqueado), y el servidor maestro evita un DDoS global (denegación de servicio distribuido que causa que un servicio o recurso sea inaccesible a los usuarios legítimos) por todos los servidores esclavos haciendo las consultas simultáneamente.

El **cliente** por su lado solo debe hacer la consulta HTTP/S a través de la API que se le proporciona, usando el sistema de Crawlsourcing.

3.3.1. Servidor maestro

El principal objetivo del servidor maestro es comunicar las tareas de los clientes con los esclavos, y los resultados de los esclavos con los clientes. También debe administrar a los clientes, usuarios, esclavos, entre otros. Sus funcionalidades se describen a continuación:

- **Procesar tareas**: los clientes envían las consultas que desean resolver y las ingresa al sistema.
- **Asignar las tareas a un esclavo**: una vez que la tarea fue ingresada, esta se le asigna a un esclavo usando un algoritmo que se explica más adelante; la tarea se almacena en una cola *FIFO* [2], para que cuando el esclavo se conecte las realice en orden de entrada.
- **Guardar respuestas**: se guardan las consultas resueltas en un sistema de caching para que sea más rápido resolver consultas duplicadas y disminuir la carga en los sitios web objetivos.
- **Administrar credenciales de usuarios**: los usuarios se registran y pueden tener uno o varios esclavos en su poder.
- **Administrar esclavos**: se almacenan datos esenciales de los esclavos para el funcionamiento del sistema y poder asignar las tareas de forma más eficiente. Los datos más importantes son la cantidad de tareas resueltas, el tiempo que demora en resolverlas y el tamaño de estas.
- **Enviar resultados**: cada consulta que envía un cliente es una conexión que se mantiene activa en el servidor maestro hasta que sea devuelto el resultado; el servidor maestro se encarga de administrar las conexiones internas para asignar de forma correcta los resultados entregados por esclavos y la respectiva consulta del cliente.

3.3.2. Esclavo

El software que corre de lado del computador voluntario (esclavo) está diseñado para:

- **Conectar a maestro**: al conectarse se marca como disponible para recibir y ejecutar tareas asignadas por el servidor maestro.
- **Pedir tareas**: cuando el esclavo ya está conectado pide tareas a maestro.
- **Procesar tareas**: las tareas están asignadas en bloques (grupos de consultas HTTP/S); el esclavo debe ejecutar las tareas que se puedan en paralelo y las que no en serie (cuando existe delay entre consultas del mismo dominio).
- **Empaquetar respuesta**: los resultados se comprimen para que al mandarlos al maestro se demore menos.
- **Enviar resultados**: se conecta al servidor maestro y carga los resultados empaquetados.
- **Pedir nuevas tareas**: este proceso se hace en paralelo mientras se envía los resultados al servidor maestro para que se aproveche de mejor manera los recursos del esclavo.

[2]FIFO, del inglés *first in, first out* (primero en entrar, primero en salir), significa que el primer elemento que se deposita, es también el primer elemento en ser sacado, manteniendo un orden de entrada y salida.

3.4. Financiamiento

El servidor maestro es un componente crítico, que debe tener un alto poder de procesamiento y un ancho de banda capaz de soportar las consultas hechas por los clientes y las respuestas de los usuarios.

El costo de un servidor que atienda dichos requisitos es variable y escalable según demanda. Por ello es indispensable hacer un modelo de ingresos que obtenga ganancias dependiendo de la demanda, en este caso cantidad de consultas resueltas con éxito.

Otro componente crítico es la participación de los usuarios, quienes voluntariamente disponen de sus dispositivos tecnológicos para hacer consultas HTTP/S a través de ellos.

Basándonos en la teoría de la motivación o incentivos[3], los usuarios estarían dispuestos a participar en esta red distributiva ya que sus dispositivos con un costo energético menor generarían ganancias. El único esfuerzo inicial es registrar los dispositivos e instalar una aplicación de redirección de consultas HTTP/S en ellos.

El costo que paga el cliente se distribuye en las partes involucradas; un porcentaje al servidor maestro para costos operativos y el resto se divide de forma proporcional a las consultas resueltas por los servidores de consultas (esclavos).

En la figura 3.4 se muestra en colores las consultas de cada cliente y como el servidor maestro las distribuye a los esclavos. Los clientes pagan una suma establecida por cierta cantidad de consultas, $28 entre todos los clientes por ejemplo, 30 % del total se queda para *Crawlsourcing* ($8,40) y el resto se distribuye proporcionalmente a la cantidad de consultas servidas a los esclavos (70 % del total ó $19,60).

3.5. Aplicaciones

Crawlsourcing está enfocado en mejorar los tiempos de obtención de datos de páginas web mediante crawling y scraping, pero puede ser usada para más aplicaciones.

Por ejemplo, en el caso hipotético de que se aproxime un Cyber-Monday y un cliente necesite testear una aplicación web para saber si será capaz de soportar una cantidad alta de usuarios (cada usuario que no logre entrar a la plataforma puede significar una venta perdida); actualmente no existe ningún sistema que pueda simular un ambiente cercano a la realidad para poder evaluar el desempeño real de una plataforma frente a carga de usuarios.

Lo que pareciera ser una desventaja de *Crawlsourcing* (sobrecargar un sistema con consultas HTTP/S y generar un DDoS), puede ser usado como una ventaja. Como *Crawlsourcing* contará con sistemas que evalúen las condiciones de un sitio web para establecer el máximo global de consultas, estos mismos indicadores pueden ser usados para testear la confiabilidad

[3]Teoría que estipula el valor de los incentivos al momento de la ejecución de una tarea o conducta, el incentivo más común es una recompensa.

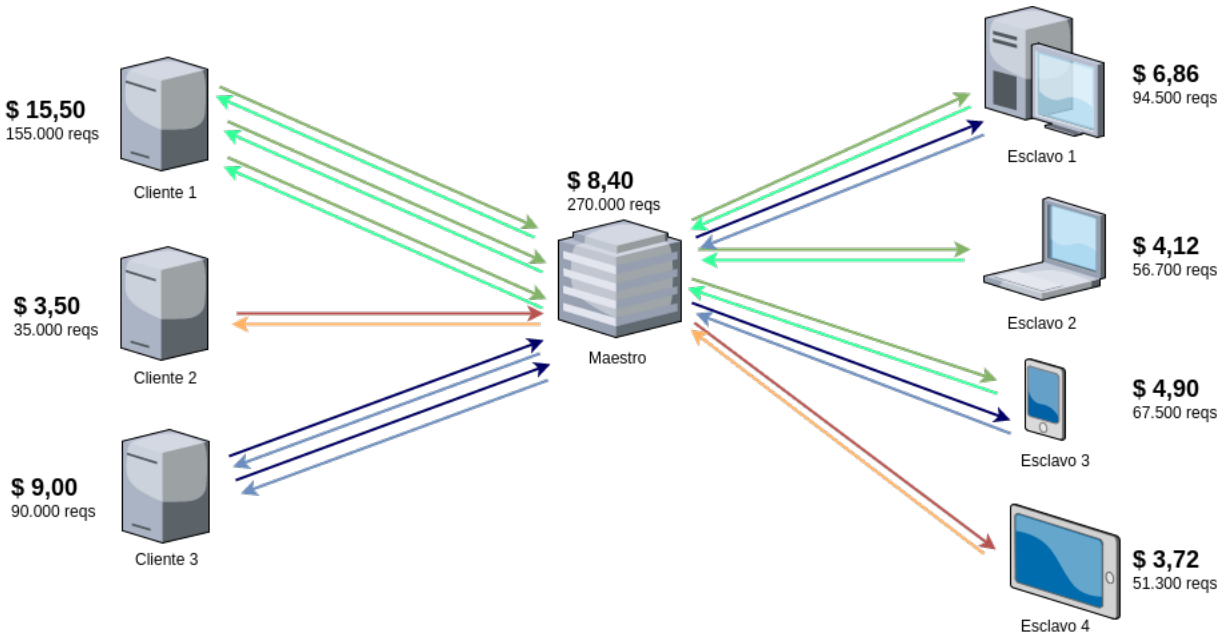

Figura 3.4: Esquema funcionamiento y financiamiento de Crawlsourcing.

y resistencia de una plataforma web.

Crawlsourcing al contar con muchos dispositivos tecnológicos cotidianos reales, crea el ambiente más cercano a la realidad que se puede simular. De esta forma se puede ofrecer el servicio de test de esfuerzo y estrés a una aplicación web para poder estimar que número de usuarios se soportarán interactuando con el sistema.

Para poder ofrecer este servicio, primero debe existir una verificación de dominio que puede ser:

- Archivo en carpeta raíz del dominio.

- Registro TXT en el DNS con una función de *hash* de identificación.

- Entre otros.

Al ser un servicio voluntario y con pasos de verificación, *Crawlsourcing* puede dejar de lado la regla de evitar DDoS en sistemas, para poder ofrecer el servicio.

Otra aplicación posible para *Crawlsourcing* es sacar ventaja de las ubicaciones de los esclavos y crear un sistema de vigilancia para la neutralidad de la red. Se debe crear listas de páginas web que sean visitadas desde diferentes ubicaciones para verificar que la información no sea censurada o modificada dependiendo de la localización desde la cual se consulte.

3.6. Cuestionamientos éticos

Primero, se debe enfatizar que los datos a los que se quiere acceder con Crawlsourcing son de acceso público y están a disposición de cualquiera con acceso a internet.

Existe la posibilidad de que algún sistema no quiera ser crawleado o scrapeado, es por ello que se seguirán las directrices establecidas en el archivo *robots.txt*, por lo tanto, un sitio web que desee bloquear el sistema *Crawlsourcing*, puede hacerlo usando el *User-agent* $CrawlsourcingBot$.

```
User-agent: CrawlsourcingBot
Disallow: /
```

O en el caso que tenga bloqueado el acceso total a los *bots*, Crawlsourcing no actuará en él.

```
User-agent: *
Disallow: /
```

Uno de los principales desafíos que tiene este proyecto es evitar la posibilidad de generar un DDoS en el sistema que está siendo consultado (página web que recibe las consultas HTTP/S); es por eso que se pretende distribuir las consultas HTTP/S limitándose a un máximo simultáneo de peticiones a un mismo sitio web, tanto global (la sumatoria de consultas entre esclavos) como localmente (cada esclavo individualmente), como se especifica en la sección 4.4.

En el mismo sentido, un administrador de sistemas que quiera probar sus aplicaciones web frente a consultas simultáneas, podrá usar el servicio para verificar el comportamiento del sistema bajo estrés, pudiendo así crear métricas y heurísticas para mejorar su sistema frente a estos casos.

Se debe tocar el caso particular en el que se use el sistema de Crawlsourcing para evadir sistemas de bloqueo geográficos o para realizar ataques a otros sistemas (por ejemplo ataques de inyección SQL). Para estos casos, Crawlsourcing debe contar con heurísticas que le permitan detectar malos usos, los cuales deberán ser bloqueados y notificados.

3.7. Privacidad

Las consultas HTTP/S hechas por los clientes son resueltas por los servidores esclavos; estos al ser nodos terminales deben saber la consulta y la web objetivo (página web de dónde se obtienen los datos).

Por el lado de los servidores esclavos, el sistema no puede resguardar el contenido de las consultas HTTP/S de los clientes, pero sí resguarda el origen de las peticiones.

Siendo el servidor maestro el único que sabe de dónde viene una consulta y quién la

resuelve, se protege la identidad de los clientes, ya que los servidores esclavos saben las peticiones, pero no saben de cuál cliente es, como se puede ver en la figura 3.5:

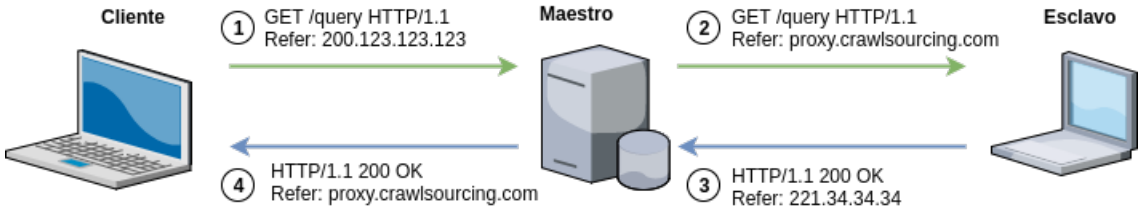

Figura 3.5: Esquema de una consulta a través de Crawlsourcing

3.8. Confianza

El servidor esclavo al hacer la consulta puede mentir en el resultado. Esto generaría problemas de consistencia y confiabilidad por el lado de los clientes. Para mitigar esta eventualidad, se debe crear un sistema de "desconfianza" donde se hagan consultas aleatorias a esclavos hacia sitios web estáticos (aleatorios también para que los esclavos no puedan reconocerlos) cuyos resultados son conocidos y se pueden comparar; si se descubre a un esclavo mintiendo se tomarán medidas a establecer. En la medida que un esclavo sirva bien (buenos tiempos de respuestas y no ser detectado mintiendo) se le harán cada vez menos consultas de "desconfianza".

Este mismo sistema es el que evalúa la tolerancia a fallos de los esclavos usando métricas por establecer; un esclavo puede ser expulsado o bloqueado temporalmente por mal desempeño (tiempo largos de respuesta, consultas falladas, conexión inestable, etc.).

En ese mismo sentido, el cliente también tendrá poder para denunciar consultas mal servidas.

3.9. Desafíos de diseño

Los principales desafíos que se presentan para lograr los objetivos propuestos, se desglosan a continuación:

- Al ser un sistema distribuido centralizado, el servidor maestro recibe conexiones tanto de los esclavos como de los clientes; esto puede generar denegaciones de servicios que afecten la funcionalidad del sistema.
- El sistema administrará múltiples consultas HTTP/S y múltiples esclavos; elegir de forma óptima para asignar las consultas a cada esclavo es un problema difícil de solucionar.
- El sistema debe ser resistente a la intermitencia y a la desconexión sorpresiva de los esclavos.

- Los esclavos al realizar el trabajo, pueden mentir para obtener mayor beneficio monetario; detectar conductas tramposas es un problema desafiante.
- Los esclavos deben realizar el trabajo, pero se debe evitar sobrecargarlos más allá de su rendimiento.
- Las consultas HTTP/S deben ser enviadas en bloques, ya que hacer las consultas una a una (de manera secuencial) es muy ineficiente; se debe calcular el número de consultas que habrán por bloque.

Capítulo 4

Implementación

Diseñado el mecanismo de distribución de consultas HTTP/S, es necesario construir el software que lleve a cabo todos los requerimientos y funcionalidades, para poder preservar las características mínimas con las cuales debe contar un sistema distribuido, poniendo especial énfasis en la escalabilidad y su tolerancia a fallos.

El sistema a implementar se compone de un servidor maestro que recibe y distribuye las consultas HTTP/S, y esclavos, los cuales se conectan al servidor maestro para obtener tareas y posteriormente entregarlas resueltas.

4.1. Servidor maestro

El servidor maestro se separa en dos para dividir operaciones de administración. Para que el sistema funcione correctamente, se debe administrar a los usuarios que ofrecen voluntariamente sus dispositivos tecnológicos (esclavos) y las consultas que resuelven cada uno para su posterior recompensa.

El manejo de los datos relacionales y la administración de usuarios y esclavos, se delegan al *framework* de desarrollo web de código abierto, que sigue el patrón de diseño MVC, *Django*, ideal para crear modelos relacionales e implementar una API[1] que se comunique con otros sistemas.

Para llevar a cabo el sistema distributivo de consultas HTTP/S, se opta por un *framework* que cuenta con un servidor web escalable y asíncrono, que permite múltiples conexiones simultáneas, *Tornado*.

[1]Una API es un conjunto de funciones y procedimientos con la capacidad de comunicación entre componentes externos al sistema.

4.2. Django

4.2.1. Modelo entidad relación Django

En la figura 4.1 se muestra el modelo entidad relación[2] que se usó para la administración de usuarios y esclavos.

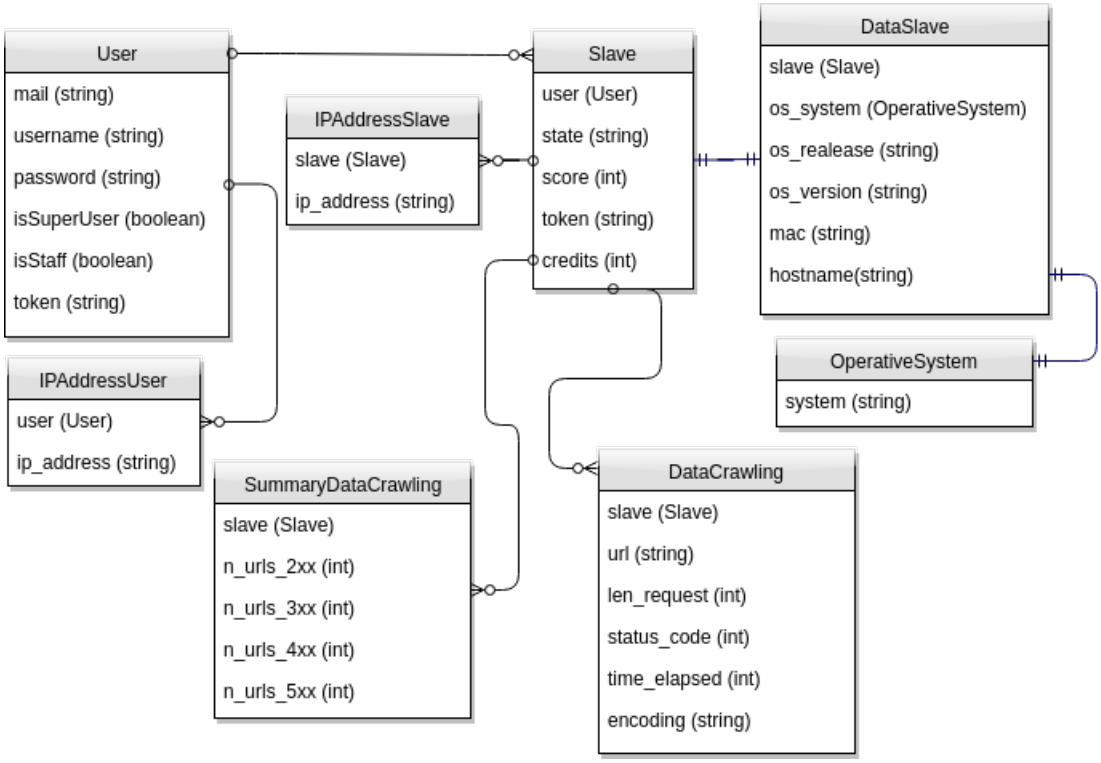

Figura 4.1: Modelo entidad relación Django.

Una descripción más detallada de cada entidad y sus atributos se encuentra en el apéndice sección A.1.

Las entidades **Slave** y **DataSlave** podrían unirse en una sola entidad y no generaría redundancia en los datos, pero se decide separarlos, ya que en **Slave** se mantiene los datos mínimos necesarios para que el sistema pueda gestionar a los esclavos; en cambio, en **DataSlave** a futuro se le pueden agregar o borrar atributos sin quebrantar el funcionamiento general del sistema.

Las entidades **IPAddressUser** y **IPAddressSlave**; encarcagadas de guardar la dirección IP[3] de usuarios y esclavos, podrían ser una sola y agregar un atributo que diferencie entre esclavo o usuario, pero se decide separarlos ya que la consulta para obtener la dirección IP es mucho más directa y menos costosa si se mantiene separadas, por ejemplo, al estar

[2]Un modelo entidad relación es una representación de las entidades relevantes de un sistema de información así como sus interrelaciones y propiedades.

[3]Una dirección IP es un número que identifica de manera lógica a una interfaz en red de un dispositivo (computadora, celular, etc) que utilice el protocolo TCP/IP.

separadas se puede consultar directamente y relacionalmente la dirección IP de un esclavo y un usuario; en cambio, al estar unidas, para obtener la dirección IP se debe hacer una consulta de búsqueda secuencial hasta encontrar el registro que cumpla con las condiciones.

La entidad **SummaryDataCrawling** es una redundancia de la entidad **DataCrawling** ya que haciendo consultas de conteo sobre **DataCrawling** se puede obtener los mismos datos que se almacenan en **SummaryDataCrawling**. Pero se decide hacerlo así por dos motivos:

1. El sistema maneja miles de consultas HTTP/S; hacer una consulta a la base de datos para contar la cantidad de consultas HTTP/S cada vez que se tenga que obtener el desempeño de un esclavo es muy costoso y poco escalable.
2. En la práctica solo se usa la entidad **SummaryDataCrawling** ya que Django solo maneja la administración y no las operaciones; por tanto, sólo necesita saber la cantidad de consultas hechas con éxito y las que no. En **DataCrawling** sólo se guarda un respaldo del trabajo realizado por un esclavo.

Cada una de las entidades mostradas anteriormente guardan además de sus atributos, una fecha de creación y una fecha de modificación de la entidad, pero se omitieron para hacerlo más entendible y menos repetitivo.

4.2.2. API Django

El sistema desarrollado en Django debe comunicarse con el sistema que administra las consultas HTTP/S, en este caso Tornado. Es por ello que se crea una API para que ambos sistemas conversen. La API cuenta con 4 *endpoints* (URLs con sus respectivas funcionalidades con la cual se obtiene los datos de la API) como se puede ver en la figura 4.2:

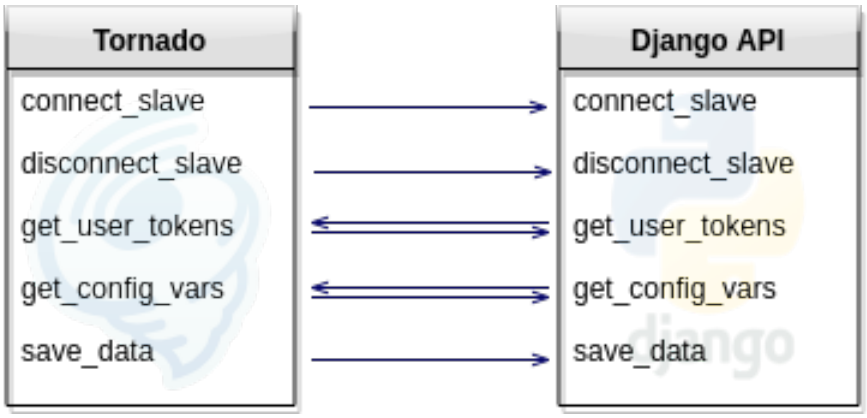

Figura 4.2: Interacción entre la API de Django y Tornado.

- **connect_slave**: *Endpoint* encargado de recibir los datos de los esclavos cuando se conectan a Tornado; cambia el estado del esclavo en Django a *online*.
- **disconnect_slave**: *Endpoint* encargado de recibir la notificación cuando un esclavo se desconecta de Tornado; cambia el estado del esclavo en Django a *offline* .
- **get_user_tokens**: *Endpoint* que responde una lista con todos los *tokens* de los usuarios registrados en Django.

- **get_config_vars**: *Endpoint* que retorna variables de configuración guardadas en Django; estas variables son (se explican con mayor detalle su usos y funcionalidades más adelante):
  - **SLAVE_ASSIGNMENT**: Modo asignación de tareas a esclavos.
  - **max_time_save_cache**: Tiempo que se guarda en caché los resultados.
  - **max_size_block_task**: Cantidad máxima de tareas por bloque.
  - **timeout_task**: Máximo tiempo de espera de una tarea.
  - **max_number_slaves_connected_waiting**: Número máximo de esclavos conectados simultáneamente esperando tareas.
- **save_data**: *Endpoint* encargado de recibir los datos que va almacenando Tornado.

Como método de seguridad, la API está diseñada para que solamente Tornado pueda acceder a ella y de forma local (deben correr en el mismo servidor Django y Tornado).

4.2.3. Interfaz web de administración

El sistema de administración cuenta con una interfaz web como podemos ver en la figura 4.3, donde cada usuario puede registrar y administrar sus dispositivos tecnológicos[4].

4.3. Tornado

Tornado es el framework que se usó para implementar el sistema de distribución de consultas HTTP/S entre los clientes y los esclavos. Esto se logra creando cinco *handlers*, que en el contexto de Tornado, son clases bien definidas que son invocadas a través de métodos POST o GET y que tiene su propia URL; cada *handler* es independiente y no bloquea el funcionamiento de los demás *handlers*. Cada vez que se invoca a un *handler*, se crea un nuevo *thread*[5], lo que permite que el sistema soporte múltiples esclavos y clientes conectados al mismo tiempo.

Como se dijo anteriormente, los esclavos reciben múltiples tareas en una sola llamada, para hacer más efectivo su trabajo, realizando las tareas en paralelo. A este conjunto de tareas le llamaremos *bloque*.

La implementación del sistema cuenta con cinco *handlers* con los cuales se comunican los esclavos y clientes con Tornado, sus funcionalidades se describen a continuación:

[4]En el apéndice sección A.2 se puede ver una ampliación de la imagen 4.3.

[5]Un *thread* es la unidad más pequeña en un proceso, que se ejecuta en paralelo con los demás threads.

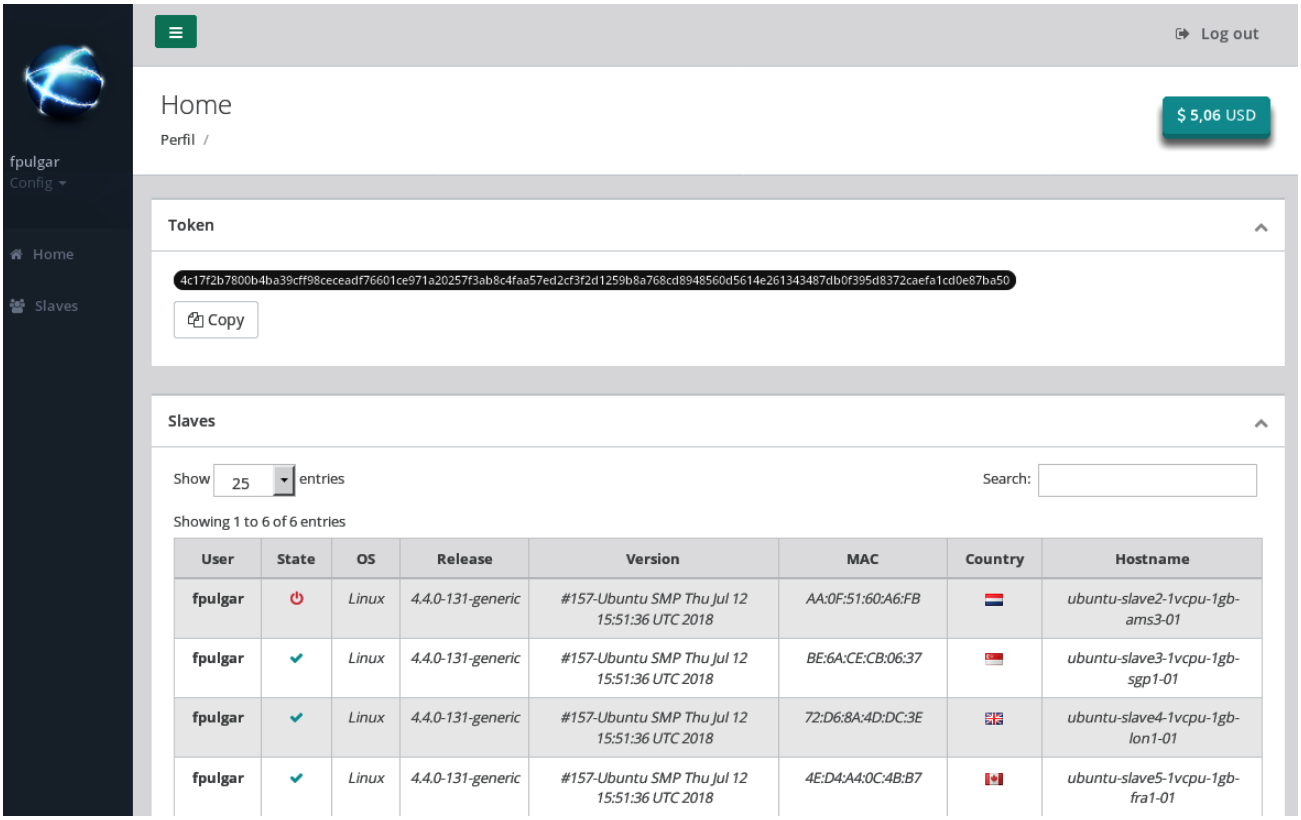

Figura 4.3: Interfaz gráfica de administración de usuarios y sus dispositivos tecnológicos.

4.3.1. ConnectSlaveHandler

El esclavo en su primera conexión debe avisar que está disponible al sistema, para que este pueda asignarle tareas. Este *handler* recibe el *token* del usuario dueño del esclavo para verificar que pertenezca al sistema; si pertenece, se le envía un mensaje de aprobación, de lo contrario uno de error.

4.3.2. DisconnectSlaveHandler

Cuando el esclavo se desconecta, manda un mensaje al sistema para que las tareas que tenía asignadas sean repartidas entre los demás esclavos.

En caso de que el esclavo tenga una caída repentina y no notifique su desconexión, el sistema cuenta con un *timeout* por cada esclavo; si se excede este *timeout*, el esclavo es declarado muerto y se desconecta del sistema. El *timeout* está definido según la cantidad de tareas que esté realizando el esclavo actualmente; mientras más tareas esté realizando, más largo su *timeout*.

4.3.3. PutTaskHandler

Este es el *handler* con el cual interactúan los clientes que mandan sus consultas HTTP/S. Su funcionamiento es muy sencillo y se detalla a continuación:

1. El cliente manda una consulta HTTP/S al sistema.

2. El sistema la recibe y bloquea la conexión del cliente con un "candado" (lo cierra).

3. El sistema asigna la tarea a un esclavo conectado al sistema.

4. Cuando el esclavo retorna la respuesta de la consulta HTTP/S, abre el "candado" y desbloquea la conexión del cliente.

5. El cliente desbloqueado obtiene la respuesta.

Como podemos notar cada llamada a este *handler* crea una conexión nueva y crea un "candado" para esperar a que la consulta sea resuelta.

4.3.4. GetTaskHandler

Este es el *handler* al que se conecta un esclavo para obtener tareas. En caso de que no hayan tareas, este espera conectado a que se le asigne una tarea. En caso de que hayan, se procede de la siguiente manera:

1. El esclavo pide tareas al sistema.

2. Se crea un bloque con consultas HTTP/S:

   (a) Por cada consulta HTTP/S se verifica con su respectivo *robots.txt*, si no está en el sistema, se agrega al mismo bloque.

   (b) Las consultas HTTP/S son agrupadas según sus políticas de robots.txt.

Aquí podemos ver un ejemplo de un bloque:

```
{
  "0": {
    "threads": "3",
    "delay": "0",
    "urls": {
      "0": "https://www.microsoft.com/",
      "1": "https://facebook.com/",
      "2": "https://terra.com/"
    }
  },
  "1": {
    "threads": "1",
    "delay": "1",
    "urls": {
      "3": "https://twitter.com/status/990923727272636417",
      "4": "https://twitter.com/status/990235860728610816",
      "5": "https://twitter.com/status/989839580605177856",
      "6": "https://twitter.com/status/989839580605177856"
    }
  }
}
```

En la figura 4.4 podemos ver el esquema de ejecución de este ejemplo.

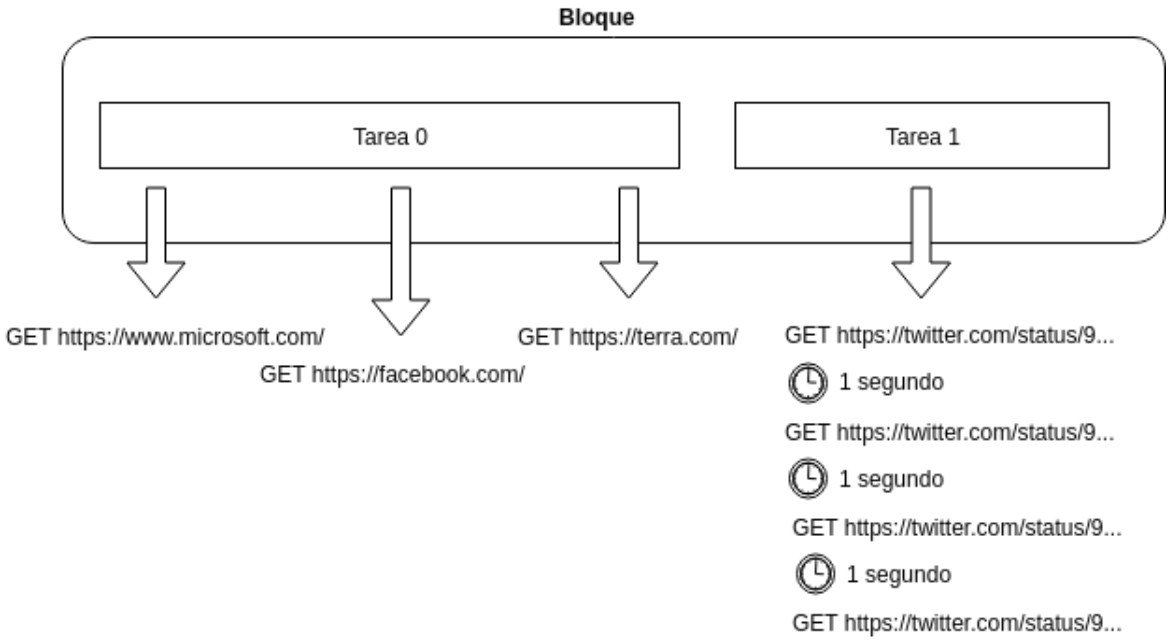

Figura 4.4: Esquema de ejecución de un bloque.

Bloques de enlaces

Para que el esclavo tenga un mejor rendimiento y también para hacer una menor cantidad de conexiones al servidor maestro, se deben realizar las tareas en grupos de enlaces. Se define un bloque como un conjunto de enlaces, este conjunto de enlaces se le manda al esclavo para que él luego devuelva un bloque con las respuestas al servidor maestro.

4.3.5. PutResponseHandler

En este *handler* el esclavo pone el bloque de respuestas. El funcionamiento es :

1. El esclavo manda la repuesta comprimida.
2. La respuesta se descomprime.
3. Por cada respuesta, se guardan en la memoria caché del sistema las consultas HTTP/S.
4. Por cada *robots.txt* en el bloque de respuestas, este se procesa y guarda.
5. Se desbloquea el proceso del cliente que hizo la consulta HTTP/S.
6. El cliente recibe la respuesta.

4.4. Asignación de tareas

Cada esclavo en *Tornado* tiene su propia cola FIFO donde se van asignando las tareas. Cuando el esclavo solicita tareas, el sistema le entrega un bloque de tareas con una cantidad definida de consultas HTTP/S que debe resolver, como podemos ver en la figura 4.5.

Para asignar las tareas (consultas HTTP/S a resolver) a los esclavos, se implementaron tres estrategias que se describen a continuación:

- **democratic**: se le asignan la misma cantidad de tareas a todos los esclavos.
- **less_busy**: las tareas se van asignando al esclavo que tenga la menor cantidad de tareas en su cola de tareas.
- **geography**: por cada consulta HTTP/S se obtiene su dirección IP y luego con la dirección IP se obtiene el país al que pertenece el servidor web. La consulta HTTP/S se le asigna al esclavo que esté más cerca del servidor web objetivo. En el caso de que un esclavo se conecte y no tenga tareas asignadas, este le "roba" tareas al esclavo que tenga más cerca y tenga tareas asignadas.

Independiente a cada estrategia, el sistema cuenta con un sistema de conteo, donde se registra la cantidad de esclavos trabajando en un mismo dominio. Este es un parámetro para evitar la sobrecarga de un sitio web por la paralelización del *crawling*; una vez que se supere una cierta cantidad de esclavos trabajando sobre un mismo sitio web, una nueva tarea que ingrese de un sitio web que sobrepasó el límite quedará a la espera de que un esclavo termine de realizar las tareas en ese dominio.

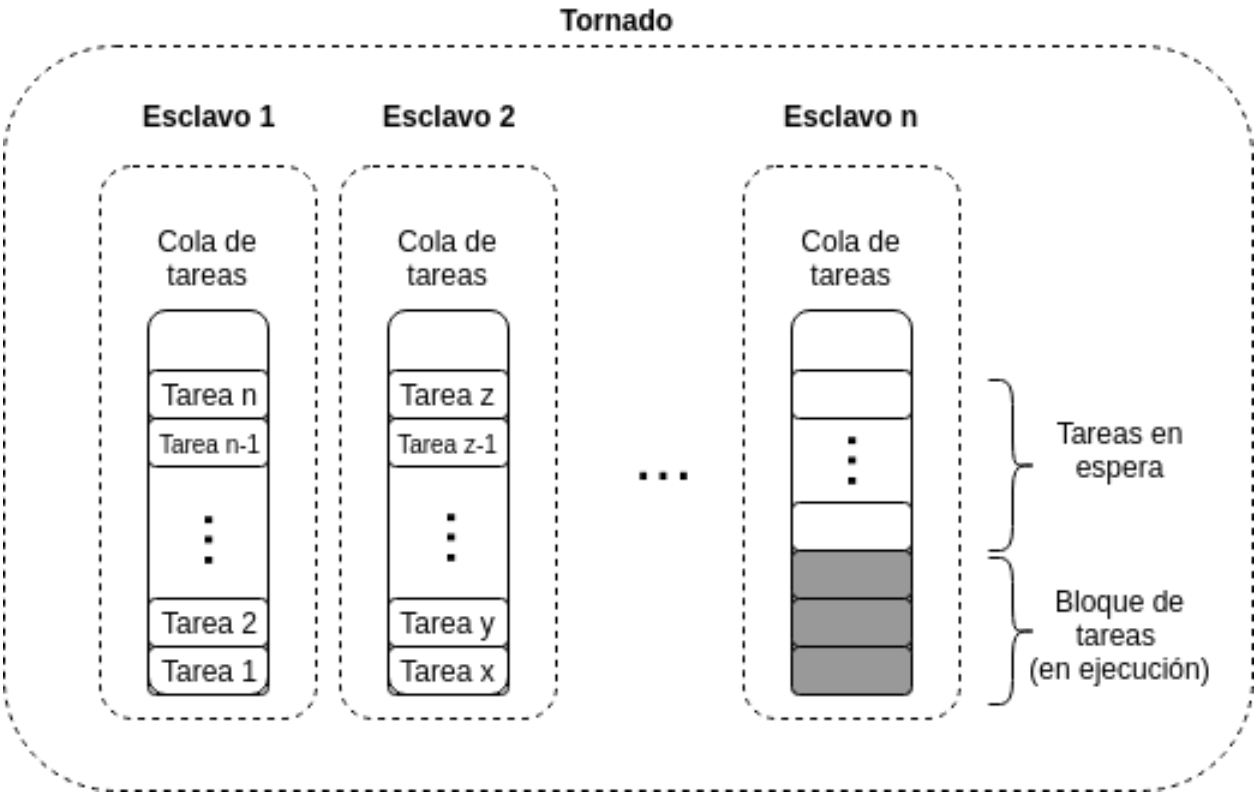

Figura 4.5: Esquema asignación de tareas en Tornado.

Por otro lado, existe un sistema de reasignación de tareas que se puede ejecutar en 3 ocasiones:

- El esclavo se desconecta y deja tareas sin resolver.
- El esclavo demora mucho en responder y las tareas se asignan a otro esclavo.
- Un esclavo no tiene tareas asignadas y otro esclavo tiene tareas sin resolver.

Para la reasignación se usan las mismas estrategias definidas anteriormente, a excepción de la estrategia **democratic**, ya que la idea de esta estrategia es no beneficiar a ningún esclavo por sobre otro, por ello, todos realizan la misma cantidad de tareas sin importar su desempeño. En el caso de la estrategia **less_busy**, las tareas son sacadas del esclavo que tenga la mayor cantidad de tareas sin resolver.

Para entender mejor cada estrategia se mostrará un pseudocódigo de cada una de ellas, cada estrategia cuenta con tres métodos:

- **get_task**: método que devuelve una tarea.
- **put_task**: método que pone tareas en la cola de un esclavo.
- **steal_task**: método para "robar" tareas a otro esclavo.

4.4.1. Pseudocódigo estrategia *democratic*

```
get_task(esclavo):
    if esclavo.queue.size() > 0
        tarea <- esclavo.queue.get()
        return tarea
    else:
        wait()
```

e*sclavo.queue.size*() obtiene la cantidad de tareas en la cola de un esclavo.

e*sclavo.queue.get*() saca la tarea más antigua de la cola de un esclavo.

```
put_task(tarea):
    esclavo <- get_least_tasks_slave()
    esclavo.tareas_totales <- esclavo.tareas_totales + 1
    esclavo.queue.put(tarea)
```

El método *get_least_tasks_slave*() recorre todos los esclavos del sistema y retorna aquel que tiene menor cantidad total de tareas asignadas.

El método *steal_task*() en esta estrategia retorna vacío ya que todos los esclavos deben trabajar lo mismo.

e*sclavo.queue.put*(*tarea*) pone en la cola del esclavo una tarea.

4.4.2. Pseudocódigo estrategia *less_ busy*

```
get_task(esclavo):
    if esclavo.queue.size() > 0
        tarea <- esclavo.queue.get()
    else
        steal_task(esclavo)
```

e*sclavo.queue.size*() obtiene la cantidad de tareas en la cola de un esclavo.

e*sclavo.queue.get*() saca la tarea más antigua de la cola de un esclavo.

```
put_task(tarea):
    esclavo <- get_least_busy_slave()
    esclavo.queue.put(tarea)
```

El método *get_least_busy_slave*() recorre todos los esclavos del sistema y retorna aquel que tiene menor cantidad de tareas asignadas sin resolver en ese momento.

e*sclavo.queue.put*(*tarea*) pone en la cola del esclavo una tarea.

```
steal_task(esclavo):
    mayor_cantida_de_tareas <- 0
    esclavo_seleccionado <- {}
    for each e in esclavos do
        if e.queue.size() > mayor_cantida_de_tareas:
            esclavo_seleccionado <- e
            mayor_cantida_de_tareas <- e.queue.size()
    esclavo.put(esclavo_seleccionado.get())
```

e.*queue.size*() obtiene la cuantiad de tareas en la cola del esclavo e.

e*sclavo.put*(e*sclavo_seleccionado.get*()) saca una tarea a e*sclavo_seleccionado* para ponerla en el e*sclavo* que la requiere.

4.4.3. Pseudocódigo estrategia *geography*

```
get_task():
    if esclavo.queue.size() > 0:
        tarea <- esclavo.queue.get()
    else:
        steal_task(esclavo)
```

e*sclavo.queue.size*() obtiene la cantidad de tareas en la cola de un esclavo.

e*sclavo.queue.get*() saca la tarea más antigua de la cola de un esclavo.

```
put_task(tarea):
    ubicación <- get_position(tarea)
    esclavo <- get_most_near_slave(ubicación)
    esclavo.queue.put(tarea)
```

El método *get_position*(*tarea*) obtiene la posición (latitud y longitud) de la tarea mediante la dirección IP del dominio de la consulta HTTP/S y luego se obtiene el país al que pertenece el servidor web. El método *get_most_near_slave*(*ubicacion*) recorre todos los esclavos y busca el mas cercano a la ubicación.

e*sclavo.queue.put*(*tarea*) pone en la cola del esclavo una tarea.

```
steal_task(esclavo):
    esclavo_seleccionado <- {}
    distancia <- 99999999
    for each e in esclavos do
        distancia_entre_esclavo = get_distance(e, esclavo)
        if e.queue.size() > 0 and distancia_entre_esclavo < distancia:
            esclavo_seleccionado <- e
            distancia <- distancia_entre_esclavo
    esclavo.put(esclavo_seleccionado.get())
```

e.*queue.size*() obtiene la cuantiad de tareas en la cola del esclavo e.

Al igual que en el pseudocódigo anterior e*sclavo.put*(e*sclavo_seleccionado.get*()) saca una tarea a e*sclavo_seleccionado* para ponerla en el e*sclavo* que la requiere.

4.5. Esclavo

El software que corre del lado de los esclavos está escrito en Python sin librerías externas; esto se decidió para evitar la dependencia externa y así permitir que en cualquier sistema que cuente con Python pueda correr sin problemas.

Para aumentar el desempeño del esclavo, este corre 3 subprocesos independientes, como podemos ver en el *BPMN* (*Business Process Model and Notation*, es una representación gráfica de procesos en un modelo)[6] de la figura 4.6:

1. Proceso que pide tareas al servidor maestro.
2. Proceso que realiza las tareas.
3. Proceso que empaqueta los resultados y los envía al servidor maestro.

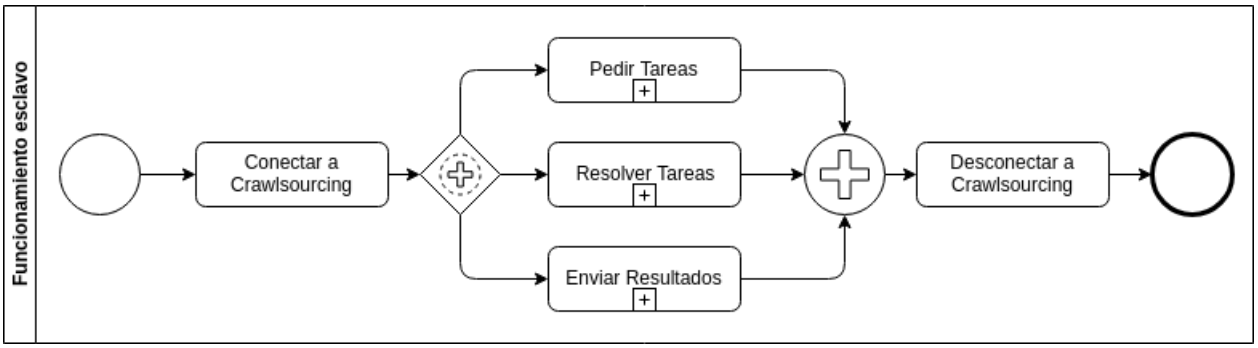

Figura 4.6: Funcionamiento de un esclavo en BPMN.

El primer subproceso pide x tareas al servidor maestro; cuando las recibe se empiezan a ejecutar. Cuando la cantidad de tareas resueltas y son mayor o igual $x/2$, se le pide al

[6]En el apéndice sección A.3 se puede ver la descripción de los símbolos y notación usada en los BPMNs.

servidor maestro $x - y$ tareas, si no hay se espera t segundos, donde t es un número aleatorio entre 5 y 30 segundos (para evitar el caso en el cual todos los esclavos intenten pedir tareas sincronizadamente al mismo tiempo), como se detalla en el BPMN de la figura 4.7.

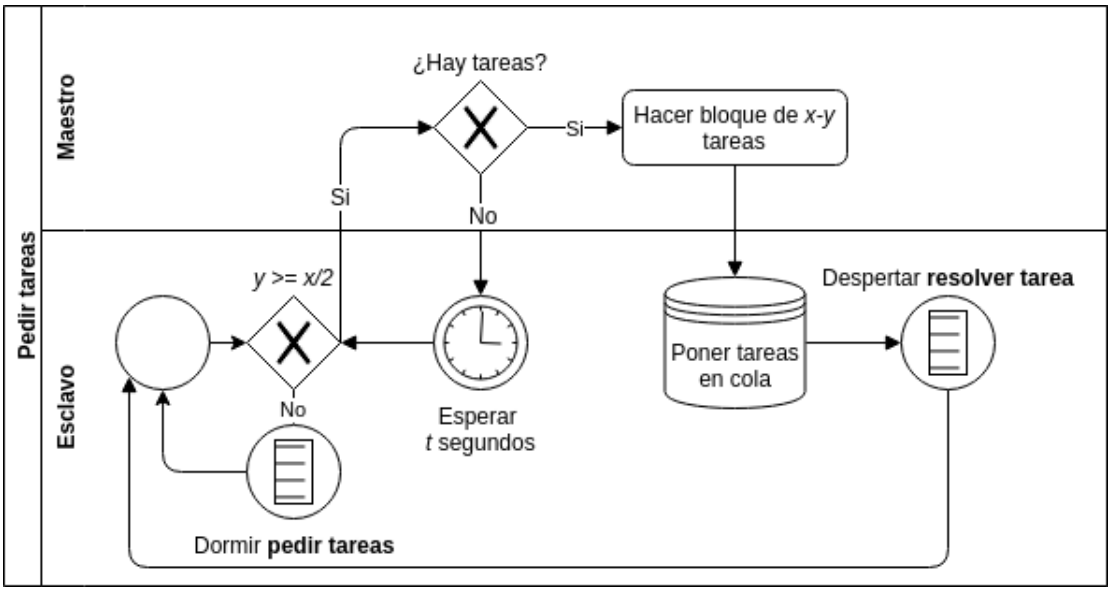

Figura 4.7: Subproceso pedir tareas en BPMN.

El subproceso que realiza las tareas simplemente sigue las indicaciones de cada bloque según el esquema que se puede ver en la figura 4.4, el flujo del proceso se describe en la figura 4.8

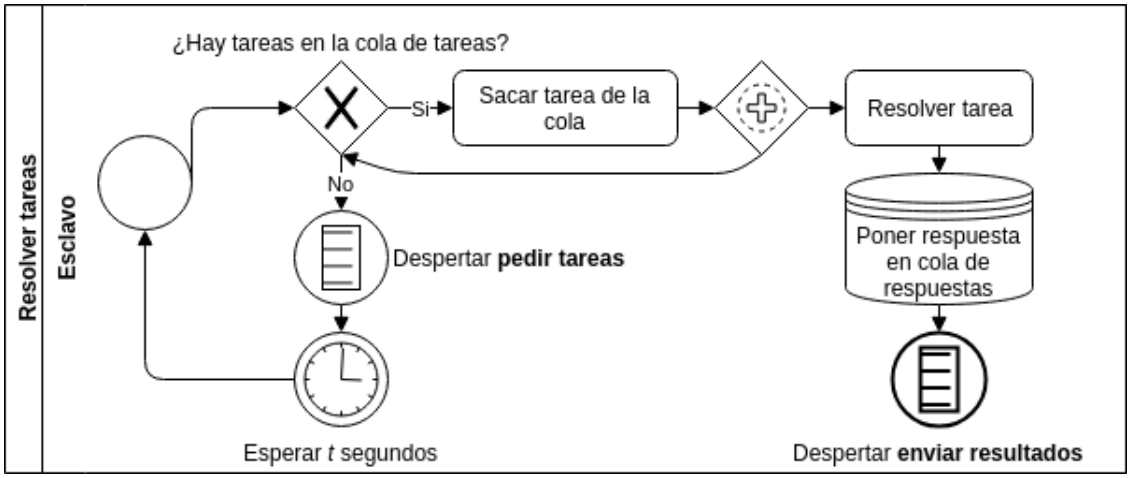

Figura 4.8: Subproceso resolver tarea en BPMN.

Todas las consultas resueltas se van guardando en una cola de resultados; cuando el tamaño de esta cola es superior o igual a $x/2$, siendo x la cantidad de tareas pedidas originalmente al servidor maestro, se comprimen los resultados y se envían al servidor maestro, como se especifica en la figura 4.9.

Con este diseño, los 3 subprocesos de la figura 4.6 no se bloquean entre sí y permite que el subproceso de resolver tareas (el que más tiempo demora) se realice sin interrupciones,

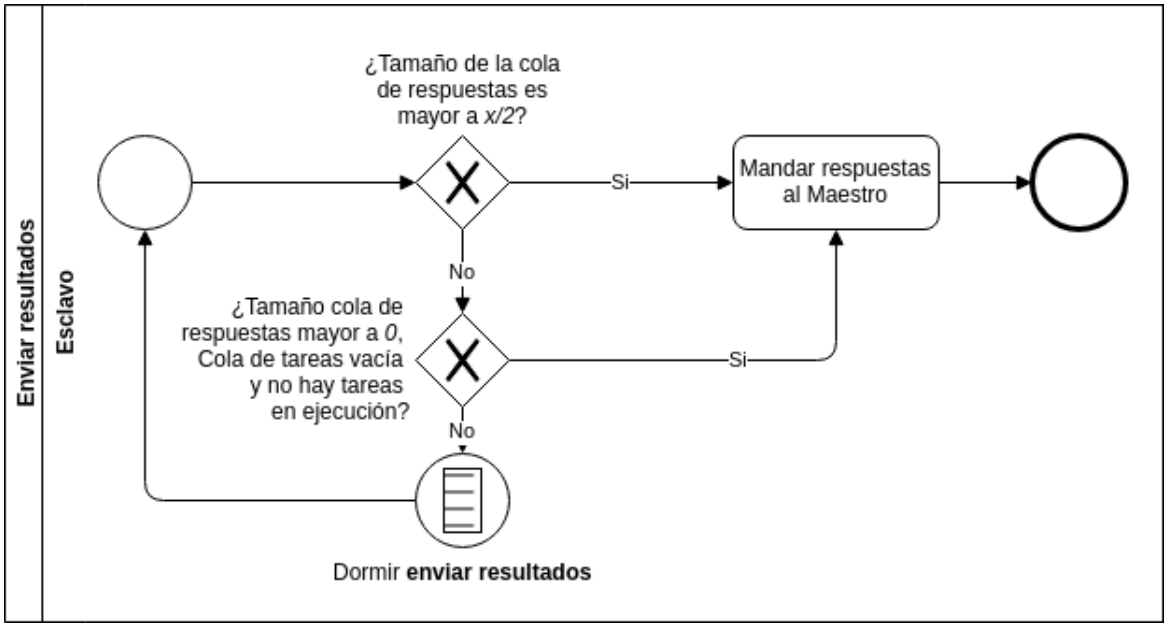

Figura 4.9: Subproceso enviar resultados en BPMN.

mientras los otros dos subprocesos piden tareas y envían resultados.

Una consideración importante es que el maestro no realiza *crawling*, por lo tanto, cada vez que se tiene una consulta HTTP/S cuyo dominio no ha sido *crawleado* antes (por lo cual el servidor maestro no tiene almacenado su *robots.txt*), el esclavo recibe la tarea con sus parámetros vacíos; es así como entiende que debe obtener el *robots.txt* del dominio y verificar si se puede o no realizar el *crawling*. Tanto el resultado de la consulta como el *robots.txt* del dominio se mandan al servidor maestro para que las almacenen.

Capítulo 5

Experimentos y evaluación

En este capítulo se exponen y analizan los resultados obtenidos de los experimentos realizados al sistema diseñado en el presente trabajo. El capítulo se divide en 4 secciones, las que abarcan el conjunto de datos utilizados, los parámetros que son relevantes en los resultados de los experimentos, el diseño de los experimentos y un análisis de éstos.

Las principales interrogantes que se pretenden responder con los experimentos son:

- ¿Cómo varía el rendimiento cuando los *threads* (tareas en paralelo) varían al realizar *crawling*?
- ¿Cuánto es lo que mejora el desempeño del sistema a medida que aumentan los esclavos conectados a él?
- ¿Cuál es la cantidad de enlaces por bloque ideal para que el esclavo tenga un mejor rendimiento?
- ¿Qué estrategia de asignación de tareas se demora menos en obtener los resultados?
- ¿Influye la ubicación de las páginas web y los esclavos en la asignación de tareas?
- ¿Cómo se comporta el sistema con diferentes tipo de conjuntos de datos?

Los experimentos detallados a continuación pretenden responder estas preguntas y ahondar en el comportamiento general del sistema y sus componentes.

5.1. Conjunto de datos

Para llevar a cabo los experimentos se han construido 3 conjuntos de enlaces, cada conjunto tiene 10.000 enlaces:

Conjunto 1 (*dataset Twitter*): el primer conjunto de datos es una lista de estados de *Twitter*[1]; todos los estados son únicos. La importancia de este conjunto es que Twitter en su *robots.txt* tiene especificado que el retraso entre consultas debe ser de 1 segundo.

[1]Twitter (https://twitter.com/)

Extracto del conjunto 1:

```
https://twitter.com/i/web/status/992100596512534528
https://twitter.com/i/web/status/992586589355233280
https://twitter.com/i/web/status/992332429728911360
https://twitter.com/i/web/status/992640143667449857
https://twitter.com/i/web/status/992640136822378496
https://twitter.com/i/web/status/992640126944624640
https://twitter.com/i/web/status/992640105218236416
https://twitter.com/i/web/status/992640064726368256
https://twitter.com/i/web/status/992640060217585665
https://twitter.com/i/web/status/992640047185780739
...
```

Conjunto 2 (*dataset mixto*): el segundo conjunto se obtuvo de la extracción aleatoria de 10.000 páginas web del primer millón de páginas web en el ranking *Alexa*[2] (sitio web que clasifica páginas web según categorías y las posiciona en un lugar mundial según un algoritmo propietario). No se repite ningún dominio.

Extracto del conjunto 2:

```
https://mcommunicator.ru
https://uro-forum.ru
https://mos-tour.moscow
https://yourdailydance.com
https://yellowstub.com
https://twitter.com
https://rk-rio.ru
https://xstamper.com
https://young.scot
https://y2mate.org
...
```

Conjunto 3 (*dataset por país*): el tercer conjunto es una lista de enlaces que se divide en 500 enlaces por país, con un total de 20 países. Se construyó usando los primeros 500.000 sitios web del ranking Alexa, antes mencionado, y de cada dominio se obtiene su dirección IP, para luego buscar a qué país corresponde en la base de datos pública *GeoLite2 Country* de *MaxMind*[3].

Teniendo los países a los que pertenecen los primeros 500.000 sitios web, se ordenan por cantidad de sitios web y se seleccionan los 20 países que tienen más, como se puede ver en la figura 5.1. De cada país se extraen 500 sitios web.

Extracto del conjunto 3:

[2]Ranking Alexa (https://www.alexa.com/topsites)
[3]MaxMind Dataset GeoLite2 Country (https://dev.maxmind.com/geoip/geoip2/geolite2/)

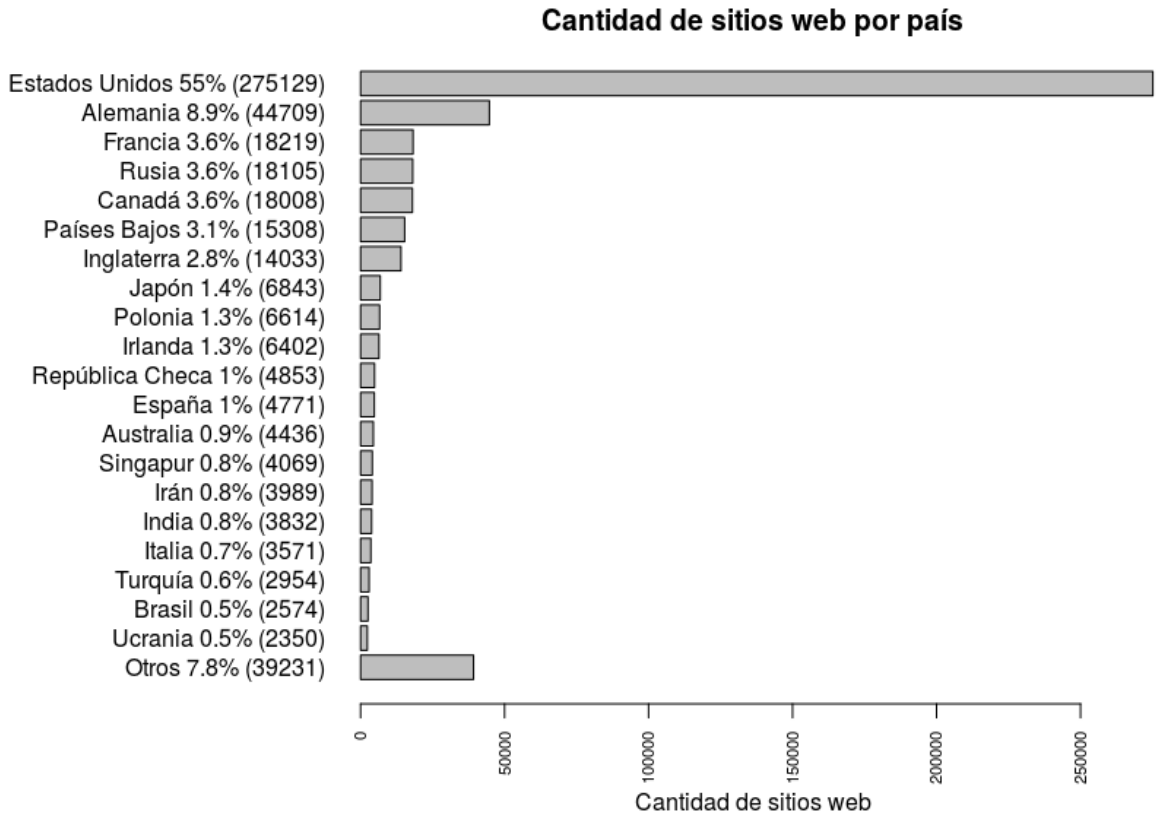

Figura 5.1: Gráfico de cantidad de sitios web por país obtenidos de los primeros 500.000 sitios web del ranking *Alexa*.

```
https://kandaharlodge.com
https://edtengineers.com
https://becextech.com.au
https://ptichka.moscow
https://friendstamilmp3.net
https://join4films.com
https://kinopooh.net
https://sharronsenter.com
https://turkeymacedonia.wordpress.com
https://tubeflix.online
...
```

5.2. Máquinas y recursos computacionales

5.2.1. Esclavos

Para realizar las pruebas se puso especial énfasis en elegir máquinas que tuvieran capacidades y recursos lo más similares posible, para disminuir y evitar sesgos de los recursos computacionales y centrarnos en el rendimiento general del sistema.

Por ello se decide usar un servicio ofrecido por *DigitalOcean*[4], empresa que ofrece el servicio de arriendo de máquinas virtuales según ciudad (como podemos ver en la imagen 5.2) y recursos (memoria RAM, cantidad de CPUs, almacenaje y sistema operativo).

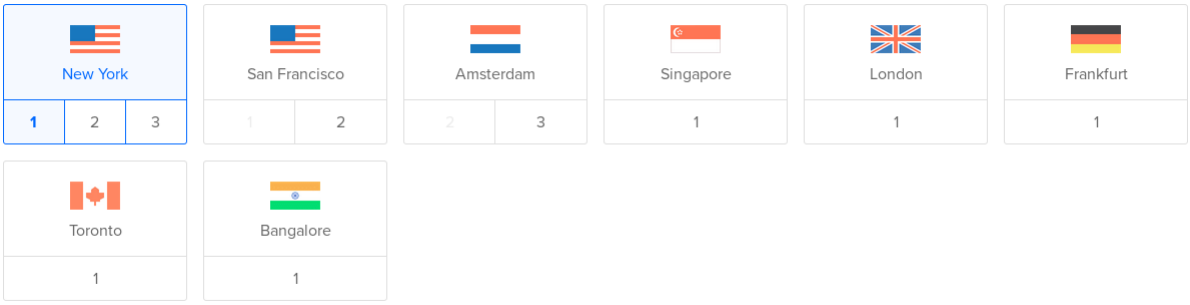

Figura 5.2: Ciudades de las máquinas virtuales ofrecidas por *DigitalOcean*.

Para los esclavos se decide utilizar una máquina virtual de cada país ofrecido, dando un total de 7 esclavos: **Estados Unidos** (Nueva York), **Países Bajos** (Ámsterdam), **Singapur**, **Inglaterra** (Londres), **Alemania** (Frankfurt), **Canadá** (Toronto), **India** (Bangalore).

Todos los esclavos tienen los mismos recursos y características, que se detallan a continuación:

- **Memoria RAM:** 1GB
- **Número de CPUs:** 1
- **Almacenaje:** 25 GB
- **Sistema Operativo:** Ubuntu 16.04.5 x64

Con el propósito de este trabajo y para obtener resultados más confiables se decide usar las máquinas virtuales con las mismas capacidades y recursos, pero en el futuro se espera utilizar el sistema con dispositivos tecnológicos de diferentes capacidades y recursos.

5.2.2. Servidor maestro

Para el servidor maestro se utiliza el mismo servicio de *DigitalOcean* explicado anteriormente, pero se decide utilizar una máquina virtual que tenga más cantidad de CPUs, ya

[4]DigitalOcean (https://www.digitalocean.com)

que el sistema requiere trabajar con muchas tareas y esclavos en paralelo. Empíricamente notamos que el sistema diseñado no consume mucha memoria RAM y a mayor cantidad de CPUs ayuda a procesar las tareas más rápido, es por ello que los recursos y características del servidor maestro son:

- **Memoria RAM:** 1GB
- **Número de CPUs:** 3
- **Almacenaje:** 25 GB
- **Sistema Operativo:** Ubuntu 16.04.5 x64

5.2.3. Adversario

Llamamos adversario a la máquina que realiza las mismas pruebas de *crawling* que el sistema de *Crawlsourcing* en los experimentos que lo requieran, pero de forma directa y sin ayuda del sistema. Sus restricciones son las mismas para ambas partes y deben respetar los archivos *robots.txt* de cada sitio web. De esta forma cada experimento se puede comparar con el caso en el que se decida hacer *crawling* con y sin el sistema *Crawlsourcing*.

Los recursos y características del adversario son exactamente los mismos que el de los esclavos.

5.3. Conjunto de parámetros

Para el correcto funcionamiento del sistema, existen dos parámetros críticos que influencian el rendimiento del sistema general. El primero es la cantidad de *threads* o cantidad de tareas que puede hacer una máquina en paralelo. El segundo parámetro es la cantidad de enlaces que tiene cada bloque que se envía al esclavo; como se vio en la sección 4.3.4, el tamaño de este bloque puede disminuir el tiempo general de obtención de resultados, pero puede aumentar el tiempo de obtención individual de cada resultado, por ello es fundamental encontrar un tamaño óptimo.

5.3.1. Cantidad de *threads*

Para fijar la cantidad de *threads* con los que trabajarán los esclavos y el adversario, se realiza un experimento simple. Con el conjunto de datos mixto se programa al adversario para descargar los datos con n cantidad de *threads* donde $n = 1, 2, 4, 8, 16, 32, 64, 128$.

Dados los resultados que podemos ver en la figura 5.3 y en la figura 5.4, se decide fijar la cantidad de *threads* tanto para los esclavos como para el adversario en 16, ya que tiene un buen desempeño y además no se busca colapsar la CPU de los esclavos y de las máquinas que participen del sistema.

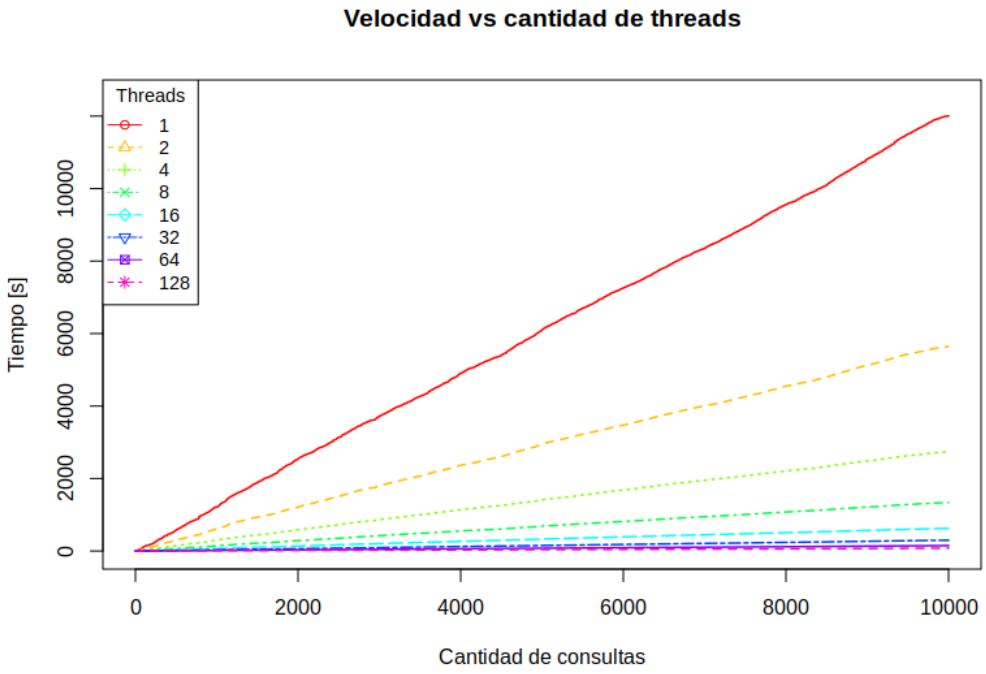

Figura 5.3: Gráfico velocidad vs cantidad de threads.

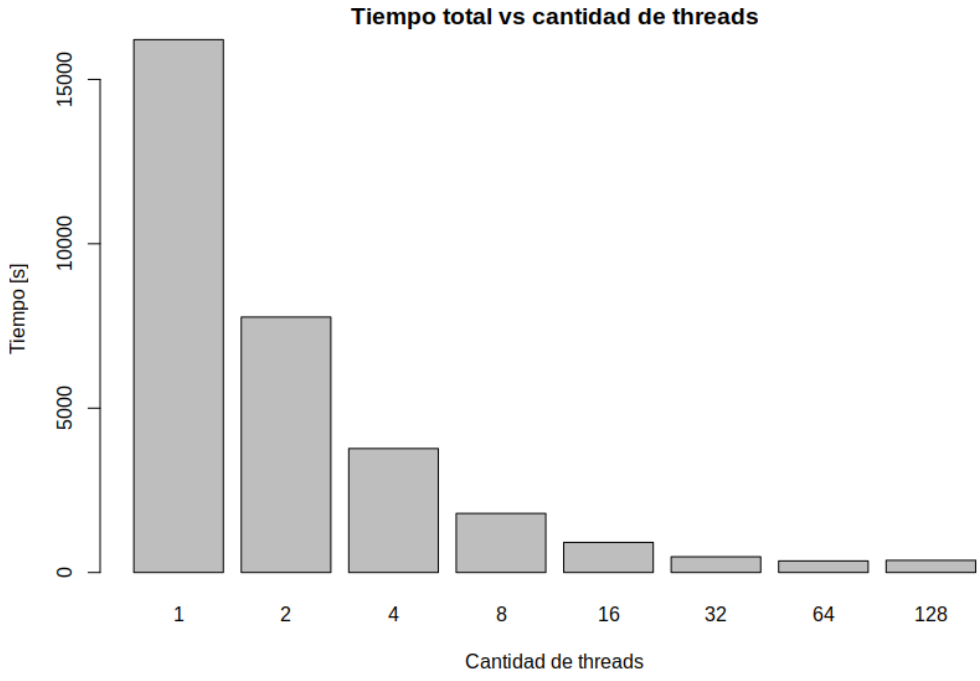

Figura 5.4: Gráfico tiempo vs cantidad de threads.

5.3.2. Cantidad de enlaces por bloque

Como se dijo anteriormente la cantidad de enlaces por bloque pueden influir tanto en el tiempo total de obtención de datos, como en la consulta individual; por ello puede influir mucho en el desempeño del sistema.

El experimento consiste en tomar 1000 enlaces del conjunto de datos mixtos; estos enlaces se resuelven con un solo esclavo, con n diferentes tamaños de bloques, donde $n = 1, 2, 4, 8, 16, 32, 64$.

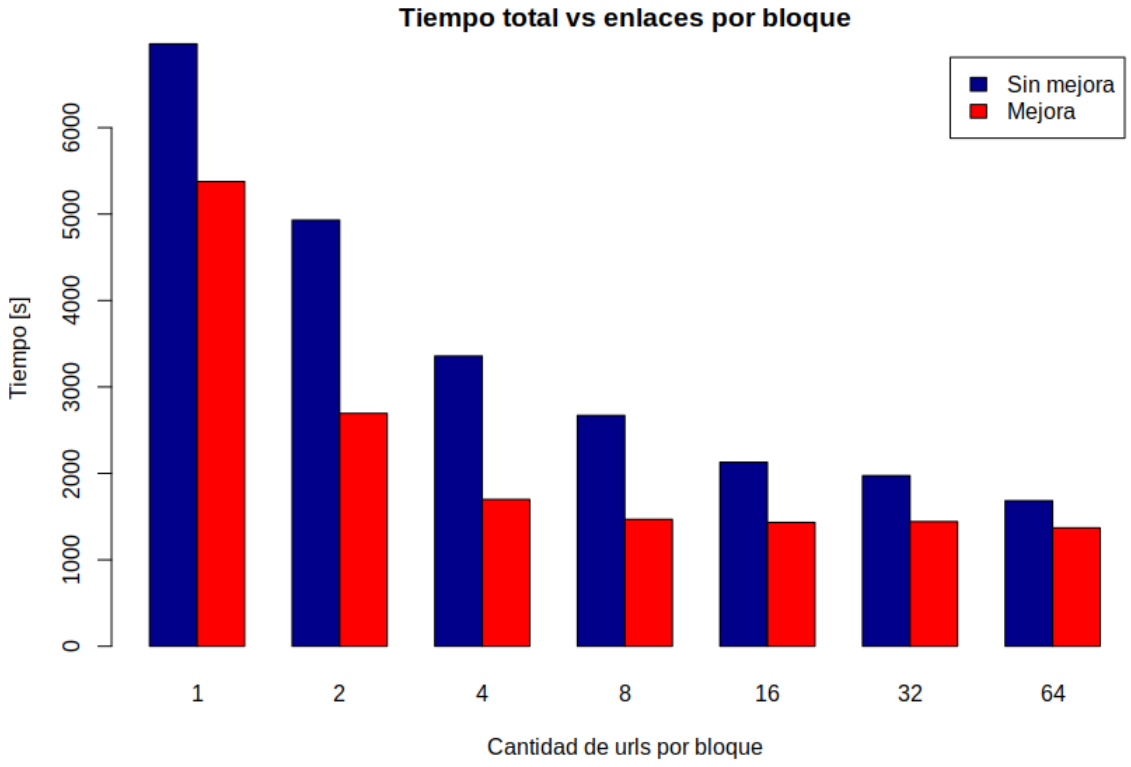

Figura 5.5: Gráfico tiempo total vs enlaces por bloque.

En la figura 5.5 se muestran los tiempos totales del experimento, y se contrastan con la primera versión del esclavo, en donde los 3 procesos descritos en la sección 4.5 se ejecutaban secuencialmente, provocando que el proceso de resolución de consultas se pausara en el proceso de mandar los resultados y en el proceso de pedir las tareas al servir maestro.

En la figura 5.6 podemos ver la velocidad con la que se resuelven las tareas dependiendo de la cantidad de enlaces por bloque, donde se ve el panorama global del comportamiento del sistema. En la figura 5.7 podemos ver en más detalle la incidencia que tiene el tamaño del bloque en el tiempo de respuesta de una consulta individualmente.

Ante los resultados obtenidos, se decide por un tamaño de 16 enlaces por bloque, ya que 16 enlaces fue el último valor que mostró una mejora en el desempeño global y el tiempo de respuesta de consultas individuales en promedio no fue mayor a 5 segundos.

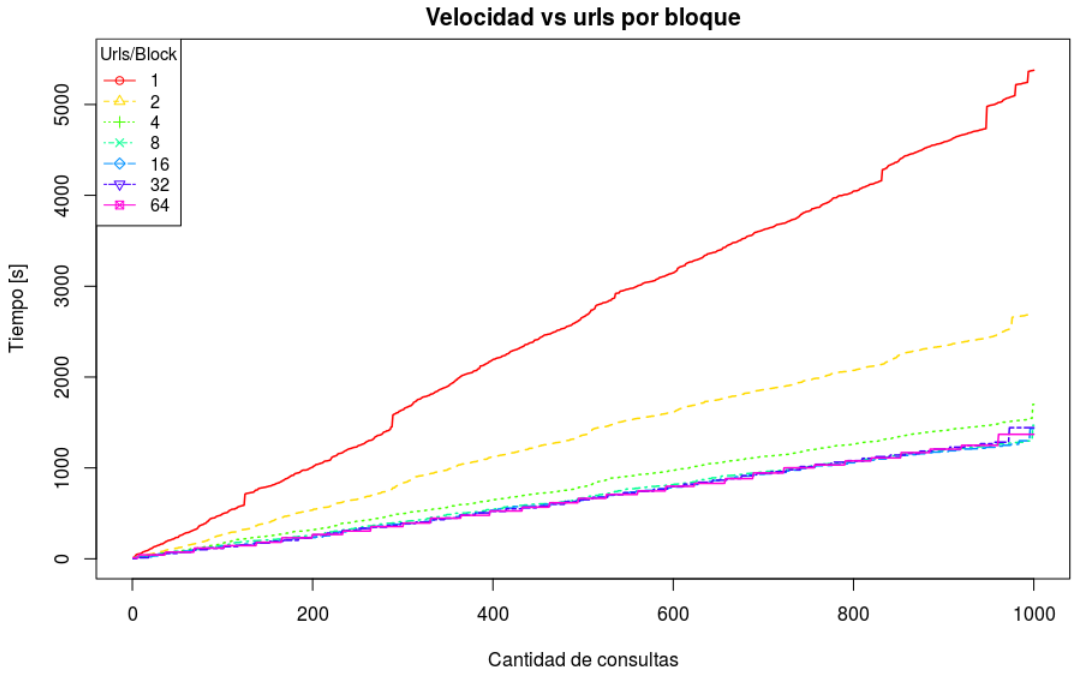

Figura 5.6: Gráfico velocidad vs enlaces por bloque.

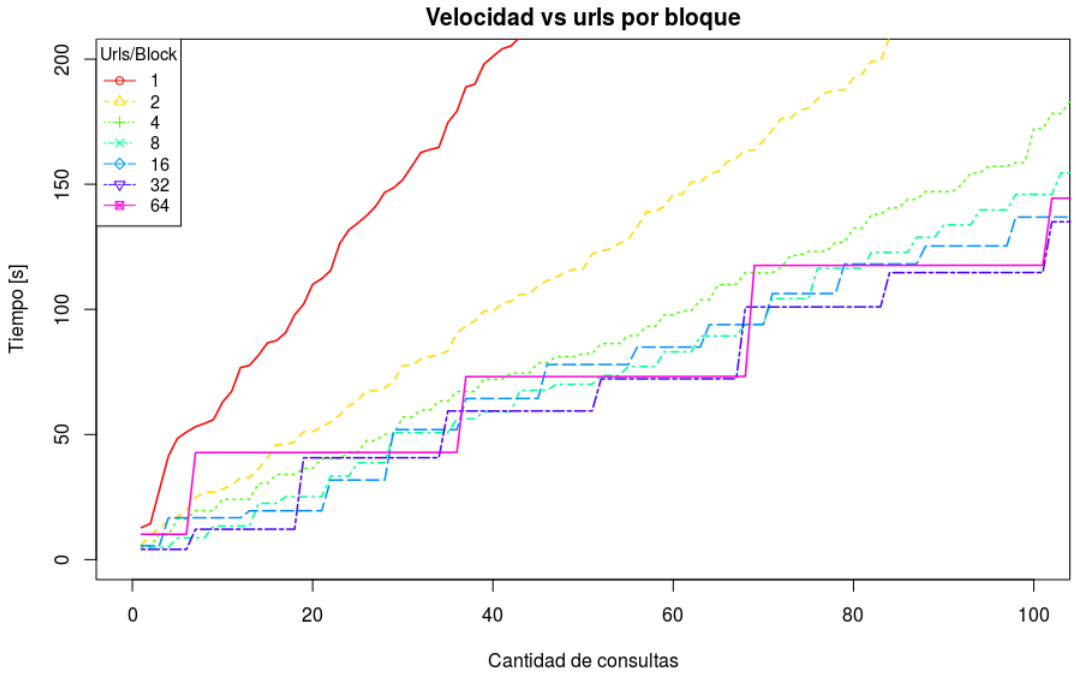

Figura 5.7: Acercamiento a gráfico de velocidad vs enlaces por bloque.

5.4. Experimentos

En esta sección se detallan 3 experimentos para analizar el comportamiento y desempeño del sistema implementado. El primer experimento es para ver y comparar cuánto mejora el

desempeño en la medida que se tienen más esclavos conectados al sistema. En el segundo se hace un análisis del desempeño de los esclavos según su cercanía con el servidor web objetivo. Por último, se realiza un experimento que busca responder cuál de las estrategias descritas en la sección 4.3.3 es la mejor y ver su comportamiento para los diferentes conjuntos de datos.

5.4.1. Experimento cantidad de esclavos

En este experimento se usan los conjuntos de datos 1 (dataset Twitter) y el conjunto de datos 2 (dataset mixto).

Cada conjunto de datos se manda a resolver al sistema n veces, y en cada vez se aumenta en uno la cantidad de esclavos conectados al sistema, llegando a un total de 5 esclavos. Se usa además la estrategia de asignación de tareas *less_ busy*.

En los gráficos 5.8, 5.9, 5.10 y 5.11 la leyenda que corresponde al adversario es *0 esclavos*.

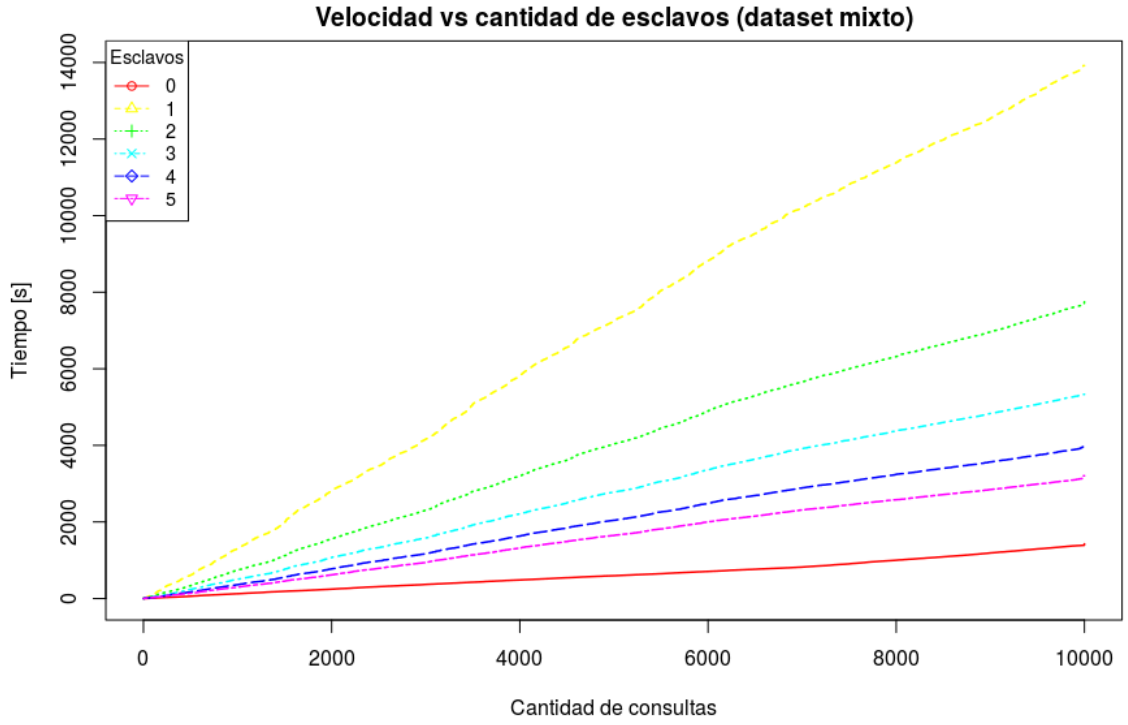

Figura 5.8: Gráfico velocidad vs cantidad de esclavos para el conjunto de datos 2 (mixto).

Como podemos ver en los resultados de la figura 5.8 y en la figura 5.9, por cada esclavo conectado al sistema se disminuye el tiempo de resolución de las consultas en general, pero aún con 5 esclavos no se pudo superar al adversario

En el caso del conjunto de datos de Twitter podemos ver en los resultados de la figura 5.10 y la figura 5.11 que al segundo esclavo se supera la velocidad del adversario. Esto se debe en gran medida a que en el *robots.txt* de Twitter se especifica que debe haber un retraso de 1 segundo por cada consulta.

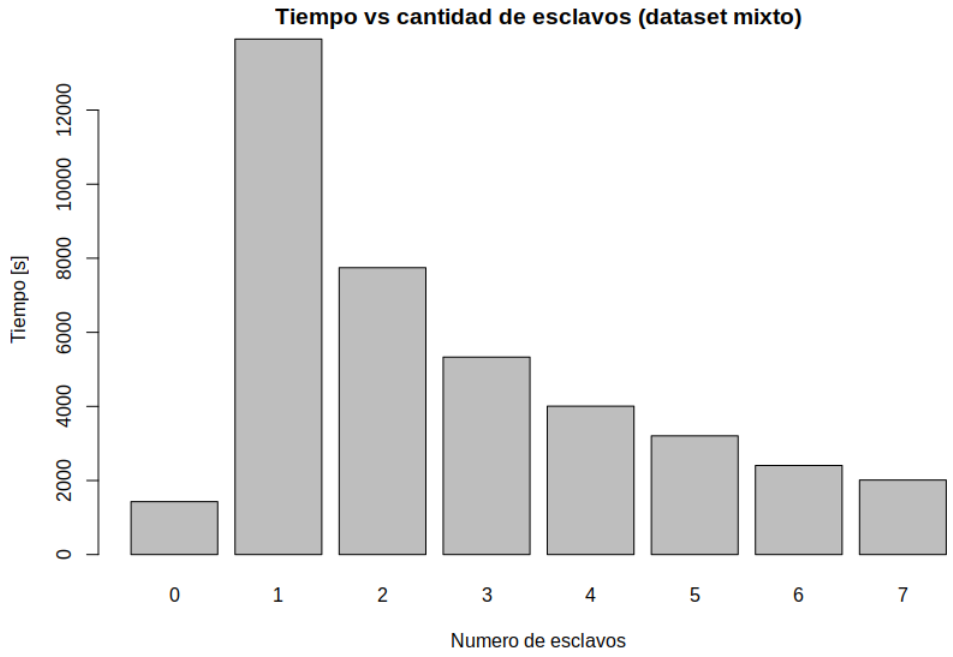

Figura 5.9: Gráfico tiempo total vs cantidad de esclavos para el conjunto de datos 2 (mixto).

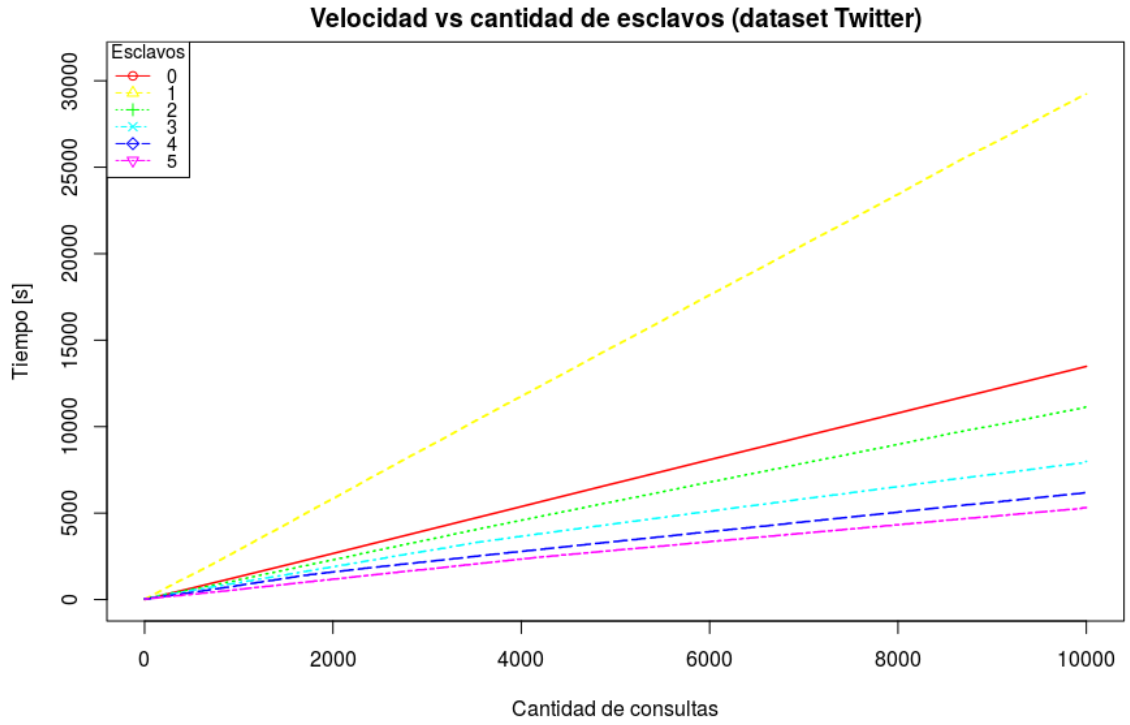

Figura 5.10: Gráfico velocidad vs cantidad de esclavos para el conjunto de datos 1 (Twitter).

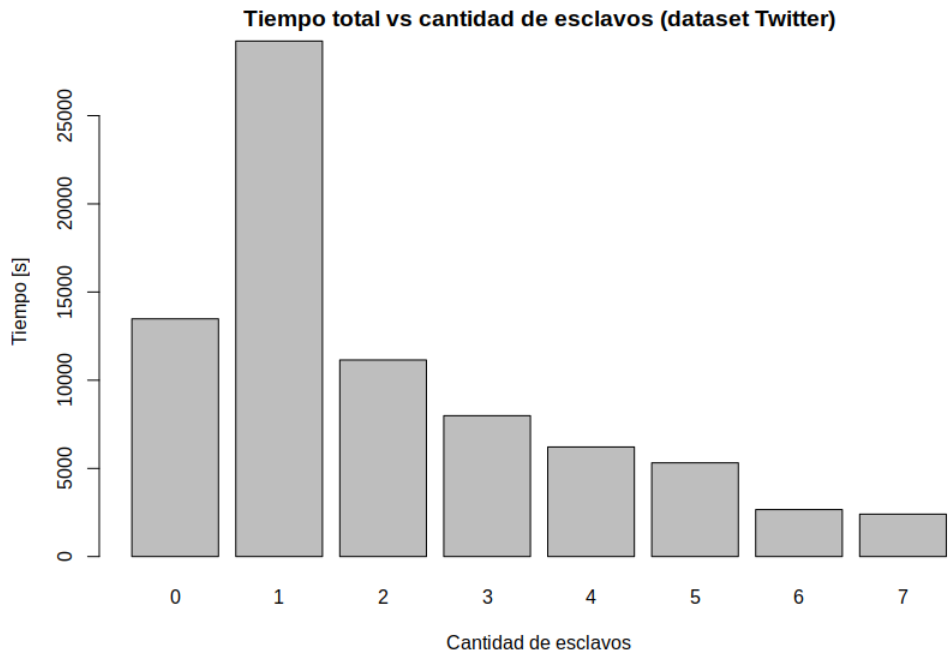

Figura 5.11: Gráfico tiempo total vs cantidad de esclavos para el conjunto de datos 1 (Twitter).

5.4.2. Experimento desempeño por país

Con este experimento se busca sacar provecho al sistema implementado, y poder hacer experimentos de desempeño según distancia a los servidores web, para ver si existe alguna correlación entre tiempo de descarga y distancia física al servidor web.

Este experimento requiere de un solo esclavo y se le asignan los enlaces del conjunto de datos 3 (dataset por país), con la estrategia *geography* se recolectan los datos de ubicación de los enlaces y el tiempo que demoran en descargarse según la ubicación del esclavo.

Los datos se recolectan y tabulan para poder ver y evaluar los resultados.

Como se puede ver en los datos de la tabla 5.1 y en el mapa de calor 5.12, se logra demostrar que existe una correlación entre: (1) la distancia física del servidor web y la máquina que hace la consulta y (2) el tiempo de respuesta de las consultas.

Para poder ver matemáticamente la correlación entre el tiempo de descarga y la distancia física usaremos el coeficiente de Kendall τ que mide el grado de correlación entre dos variables X y Y. Intuitivamente, el coeficiente de Kendall τ entre dos variables será alto cuando los valores tengan un rango similar entre las dos variables, y bajo cuando las observaciones tienen un rango diferente (o totalmente diferente para una correlación de -1) entre las dos variables.

Sean $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ un conjunto de observaciones de las variables aleatorias X y Y respectivamente, de manera que todos los valores de (x_i) y (y_i) son únicos, y (x_j, y_j), donde i $< j$, se dice que son concordantes si las filas de ambos elementos coinciden: es decir,

Tabla 5.1: Tiempo promedio de respuesta de los esclavos por países.

URLS/Esclavos	EEUU	P. Bajos	Singapur	Inglaterra	Alemania	Canadá	India
EEUU	1,3459	1,5573	5,1589	1,2205	1,4278	1,4030	2,2871
P. Bajos	0,9965	0,6902	1,7137	0,6842	0,7677	1,1619	1,5127
Singapur	2,1104	2,1121	1,2066	2,2223	1,9470	2,2331	1,6574
Inglaterra	2,3453	1,8286	3,0631	1,0220	1,8194	2,4916	2,8761
Alemania	1,5192	0,7252	1,7396	0,6713	0,7624	1,2646	1,9430
Canadá	0,6569	1,3271	1,7931	1,0002	1,1566	0,7935	2,1469
India	1,8816	2,5577	1,5134	2,0634	2,1613	2,6015	1,3816
Australia	1,9365	2,1940	1,2860	2,1368	2,1341	2,2989	1,9903
Brasil	1,6202	2,6326	3,5221	2,0314	1,8524	1,9757	4,6393
Rep. Checa	1,2602	0,7719	2,1008	0,8049	0,8259	1,4200	1,7349
España	1,4768	1,0424	2,7132	1,0459	1,0607	1,5801	2,0183
Francia	2,9132	1,0852	2,5849	0,9250	1,3386	1,5522	2,1044
Irlanda	1,0061	0,5751	1,5642	0,4813	0,5604	1,0788	1,5274
Irán	3,1883	2,5982	18,7609	14,4037	42,0339	16,7602	20,1032
Italia	1,4981	1,2970	2,3881	1,3273	1,1928	1,8095	2,0900
Japón	1,5387	2,2889	1,0620	1,6241	1,7285	2,0084	1,3982
Polonia	1,3510	1,2850	2,1763	0,7977	0,7894	1,6628	1,8340
Rusia	7,1380	1,4440	9,4644	0,8121	1,0087	1,5613	2,0466
Turquía	1,7287	1,0859	1,2066	1,1059	0,8836	2,2110	2,1511
Ucrania	1,6163	1,0749	2,2761	1,1426	1,0193	1,6047	2,3482

si $x_{\mathrm{i}} > x_j$ y $y_{\mathrm{i}} > y_j$, o si $x_{\mathrm{i}} < x_j$ y $y_{\mathrm{i}} < y_j$. Se dice que son discordantes, si $x_{\mathrm{i}} > x_j$ y $y_{\mathrm{i}} < y_j$, o si $x_{\mathrm{i}} < x_j$ y $y_{\mathrm{i}} > y_j$. Si $x_{\mathrm{i}} = x_j$ o $y_{\mathrm{i}} = y_j$, el par no es ni concordante ni discordante.

Definición 5.1 *Kendall* τ

$$\tau = \frac{(\textit{cant}\text{id}\text{ad de }\textit{pares concordantes}) - (\textit{cant}\text{id}\text{ad de }\textit{pares discordantes})}{n(n-1)/2}.$$

En otras palabras, si ponemos los datos de las variables en dos columnas creando tuplas $(x_{\mathrm{i}}, y_{\mathrm{i}})$, si al ordenar las columnas por separado queda la misma composición de tuplas pero en diferente orden, entonces Kendall τ retornará un coeficiente 1.

En este caso nuestra variable X = {Conjunto de tiempo promedio de descarga por país} obtenidos de la tabla 5.1, y nuestra Y = {Conjunto de distancia entre países}.

Como podemos ver en la tabla 5.2, existe una correlación positiva entre la distancia física de las páginas web y la ubicación del que hace la consulta.

5.4.3. Experimento estrategia de asignación de tareas

Este experimento busca comparar el desempeño de las diferentes estrategias con diferentes conjuntos de datos; para ello se toman los 3 conjuntos de datos y se resuelve cada uno con

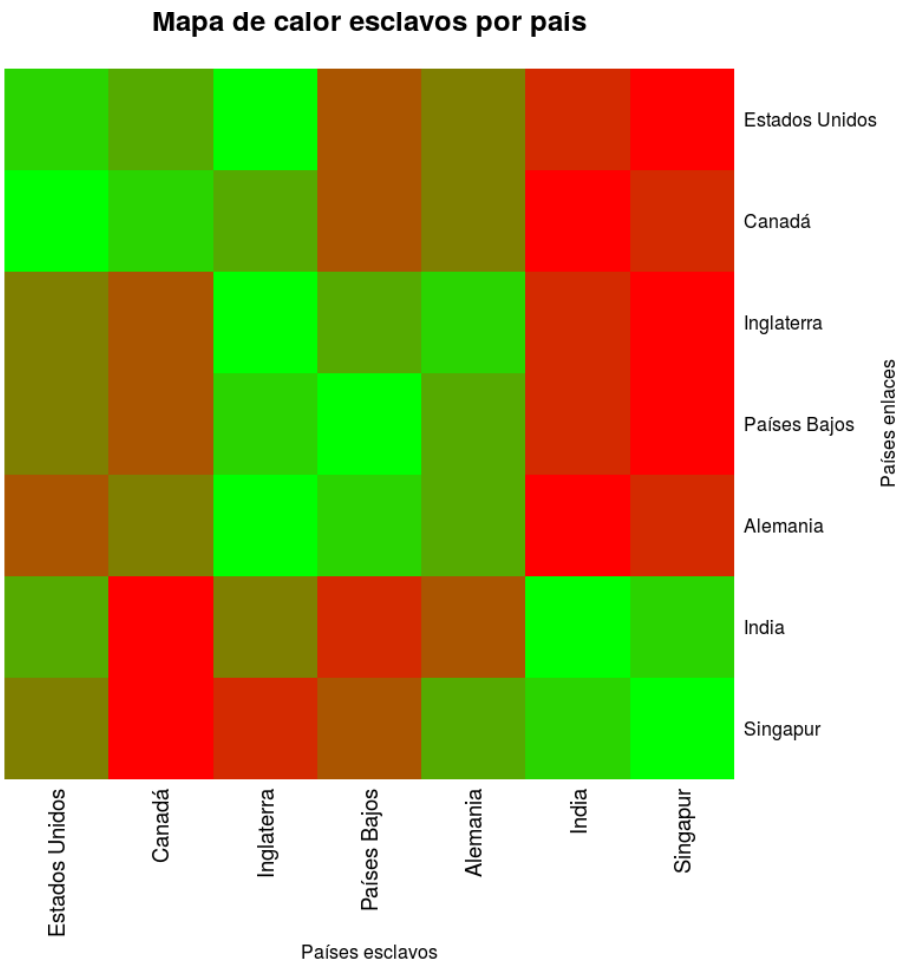

Figura 5.12: Mapa de calor de esclavos por país.

Tabla 5.2: Coeficientes de Kendall τ según tiempo de descarga promedio por países versus distancia entre ellos.

Países Esclavos	τ	p
Estados Unidos	0.453	0.0058198
Países Bajos	0.568	0.00051749
Singapur	0.364	0.027288
Londres	0.611	0.00019062
Alemania	0.526	0.0013182
Canadá	0.663	0.000050068
India	0.221	0.18345

cada una de las 3 estrategias.

Como podemos ver en la figura 5.13 la estrategia que siempre tuvo mejor desempeño para todos los conjuntos de datos fue la estrategia *less_ busy*, seguido de la estrategia *geography* y por último la estrategia *democratic*.

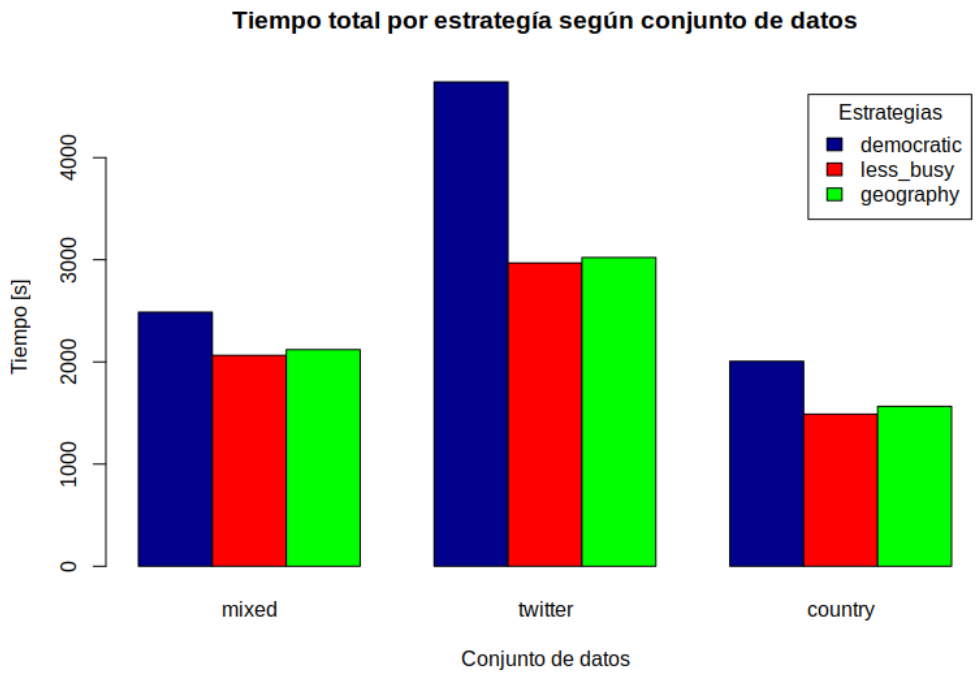

Figura 5.13: Gráfico tiempo total por estrategia según conjunto de datos.

5.5. Análisis de los resultados

Después de presentar los resultados de los experimentos, podemos ver como el sistema mejora en la medida que aumentan los esclavos conectados a él. Pero con 7 esclavos no se logró superar al adversario en el caso del conjunto de datos 2 (dataset mixto), pero se logró una mejora significativa cuando se hizo la prueba con el conjunto de datos 1 (dataset Twitter), necesitándose solamente dos esclavos para superar al adversario. Esto se debe en gran medida ya que Twitter no permite consultas en paralelo desde un mismo computador.

Es interesante notar que efectivamente existe una correlación entre el tiempo de carga de una página web y la distancia física hacia ella. Haciendo una inspección visual en la figura 5.12 podemos ver una relación positiva entre los continentes y el desempeño en la descarga de datos, creando grupos entre Estados Unidos y Canadá (Norte América); Inglaterra, Paises Bajos y Alemania (cercanías a Europa); y Singapur e India (Asia).

Por lo anterior se esperaba que de las estrategias; *geography* tuviera un mejor desempeño, pero el costo de obtener por cada consulta HTTP/S su dirección IP, para luego obtener el país de procedencia hace que sea superada por la estrategia que *less_ busy*. Otro factor por el cual tiene un menor desempeño, es la cantidad de reasignaciones que se hacen; en el caso de *less_ busy* se hacen muchas menos reasignaciones ya que se auto-balancea la carga a medida que se asignan las tareas. La estrategia *democratic* fue siempre peor a las demás estrategias, y era un resultado esperable debido a que los esclavos no tienen los mismos desempeños individuales, por lo que el sistema se demorará tanto como el más lento de sus componentes.

Capítulo 6

Conclusión

A continuación se presentan las conclusiones obtenidas a lo largo del trabajo de memoria realizado.

6.1. Objetivos logrados

En este trabajo de memoria se implementó un sistema distributivo de consultas HTTP/S con un enfoque colaborativo y voluntario. Se desarrolló el sistema que recibe, administra y asigna las consultas HTTP/S a los diferentes dispositivos tecnológicos conectados al sistema y que disponen de su capacidad ociosa. Por el lado de los dispositivos tecnológicos que resuelve las consultas, se logró identificar y paralelizar las tareas más importantes; obtener tareas, resolver las consultas HTTP/S y entregar los resultados. Además se creó un sistema para administrar usuarios y sus dispositivos tecnológicos conectados a él, el cual sirve para almacenar resultados y estadísticas de desempeño.

En total se crean tres piezas de software, que en su conjunto crean un sistema completo, donde cada uno de sus componentes son independientes y no interfieren entre ellos, logrando una asincronía entre procesos y una comunicación fluida entre los elementos del sistema.

6.2. Resultados obtenidos

De los experimentos realizados se logró establecer la cantidad de tareas en paralelo que deben realizar los dispositivos tecnológicos para resolver las consultas HTTP/S; además, se determinó la cantidad de enlaces por bloque de tareas que se le debe asignar a un esclavo para que este tenga el mejor desempeño en comparación con la velocidad de respuesta individual de una consulta HTTP/S.

Los principales resultados obtenidos de los experimentos hechos al sistema utilizando todos sus componentes son:

- La cantidad de esclavos resolviendo tareas tiene una incidencia directa en el tiempo global de resolución de consultas HTTP/S. A mayor cantidad de esclavos, menor es el tiempo que se demora el sistema en resolver las consultas, pero por cada esclavo agregado al sistema se logra una mejora menos significativa que el esclavo anterior agregado; es decir, el desempeño del sistema es creciente a tasas decrecientes con respecto a la cantidad de esclavos.
- El desempeño por distancia al servidor web objetivo muestra una relación directa con el tiempo que se demora en hacer la consulta.
- Las estrategias de asignación de tareas tienen una influencia en el comportamiento y el desempeño general del sistema. La estrategia que tiene peor desempeño es aquella en la cual a todos los esclavos se le asigna el mismo número de tareas, lo que genera que esclavos terminen antes que otros sin tener en cuenta el desempeño de cada esclavo individual. La estrategia que tuvo mejores tiempos fue aquella que asignaba las tareas a los esclavos que tuvieran la menor carga actual de tareas, esto beneficiaba a los esclavos con mejor desempeño y perjudicaba a los esclavos más lentos, generando un mejor desempeño global del sistema. Por último se probó una estrategia que asignaba a las tareas por distancia física, esta era un poco más lenta que la anterior descrita y se debe al sobrecosto de obtener por cada consulta HTTP/S su ubicación y encontrar el esclavo más cercano a ella.

6.3. Conclusiones de los resultados

De los resultados se puede deducir que si bien, a medida que se usan más esclavos para resolver las consultas, menor es el tiempo global de la resolución de las consultas, pero también menor es el aporte proporcional que se consigue por cada esclavo agregado. Esto se evidencia en el hecho de que el sistema al probarse con el conjunto de enlaces mixtos y con siete esclavos no fue capaz de superar al adversario.

Una de las conclusiones que recoge esta memoria, es la correlación que existe entre el tiempo de respuesta y la distancia física entre el cliente y el servidor. Empíricamente se demuestra que a menor distancia física, menor es el tiempo de respuesta y vice versa.

A pesar de los favorables resultados geográficos, al poner a prueba la estrategia que asignaba las tareas a los esclavos que estuvieran más cerca físicamente de los sitios web objetivos, esta no consigue superar a la estrategia que asigna las tareas a los esclavos que tienen menor carga. Esto se debe en gran medida a que el costo de obtener por cada enlace su ubicación física es muy alto en comparación al beneficio de asignar la tarea al esclavo más cercano, además, al asignar tareas por distancia no se toma en cuenta el desempeño individual de cada esclavo, lo que genera al sistema otro sobrecosto, esta vez, para reasignar las tareas cuando existen esclavos sin tareas asignadas.

Dado lo anterior, asignar tareas por distancia es una buena opción cuando se puede conseguir la distancia de forma rápida y no genere retrasos en la ejecución global del sistema, también es una muy buena alternativa para combinar con otras estrategias y usarla para decidir empates por ejemplo.

6.4. Trabajo futuro

Como trabajo futuro quedan varios desafíos abiertos, acordes a la magnitud de un sistema distribuido. Dentro de los desafíos más interesantes nos encontramos con:

- Lograr la privacidad de las consultas hechas por el cliente. En otras palabras, impedir que el esclavo pueda saber la consulta que hace el cliente, pero aún así resolverla.
- Detectar cuando un esclavo miente, ya que tiene el incentivo de realizar la mayor cantidad de tareas en la menor cantidad de tiempo.
- El servidor maestro tiene un sistema de caché para guardar el resultados de las consultas y los *robots.txt*, pero ese caché está alojado en memoria primaria (RAM). Crear un sistema de caché en memoria secundaria (disco) que sea de rápido acceso.
- Crear más estrategias de asignación de tareas para poder encontrar la manera de asignar tareas que le saquen el mayor provecho a los esclavos y que no estrese al servidor maestro.
- Generalizar el problema y no solamente resolver consultas HTTP/S, sino también tareas que requieran el uso de CPU, además de buscar más aplicaciones al sistema distribuido y colaborativo.
- Realizar experimentos con dispositivos cotidianos con diferentes capacidades y recursos, en ambientes reales con las intermitencias de conexión que ello signifique.
- Hacer un sistema que balancee o distribuya las consultas HTTP/S entre más servidores maestros.
- Crear una mejora en la selección de threads y cantidad de enlaces por bloque, haciendo que en base a métricas del sistema se puedan ajustar automáticamente.

Bibliografía

[1] David P. Anderson. BOINC: A system for public-resource computing and storage. In *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 4–10, 2004.

[2] David A Bader and Robert Pennington. Cluster computing: Applications. *The International Journal of High Performance Computing Applications*, 15:181–185, 2001.

[3] Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and grid technologies for wide-area distributed computing. *Softw., Pract. Exper.*, 32(15):1437–1466, 2002.

[4] Amid Khatibi Bardsiri and Seyyed Mohsen Hashemi. Cloud computing vs. grid computing. *ARPN journal of systems and software*, 2:188–194, 2012.

[5] Geoff Boeing and Paul Waddell. New insights into rental housing markets across the united states: Web scraping and analyzing craigslist rental listings. *Journal of Planning Education and Research*, 37:457–476, 2017.

[6] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Softw., Pract. Exper.*, 34(8):711–726, 2004.

[7] Carlos Castillo. Effective web crawling. *SIGIR Forum*, 39(1):55–56, 2005.

[8] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii, USA*, pages 124–135, 2002.

[9] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2002.

[10] Muhammad Nouman Durrani and Jawwad A. Shamsi. Volunteer computing: requirements, challenges, and solutions. *J. Network and Computer Applications*, 39:369–380, 2014.

[11] Rakesh Kumar and Shilpi Charu. Comparison between cloud computing, grid computing, cluster computing and virtualization. *International Journal of Modern Computer Science and Applications*, 3(1):42–47, 2015.

[12] Dong Liu and Ralph Deters. The reverse C10K problem for server-side mashups. In *Service-Oriented Computing - ICSOC 2008 Workshops, ICSOC 2008 International Workshops, Sydney, Australia, December 1st, 2008, Revised Selected Papers*, pages 166–177, 2008.

[13] Julien Masanès. *Web archiving*. Springer, 2006.

[14] Adam McMahon and Victor Milenkovic. Social volunteer computing. In *Journal on Systemics, Cybernetics and Informatics (JSCI)*, volume 9, pages 34–38, 2011.

[15] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *1st International Conference on Peer-to-Peer Computing (P2P 2001), 27-29 August 2001, Linköping, Sweden*, pages 99–100, 2001.

[16] Ajit Sull, Vishal Razdan, Satyendra Tiwari, and Rupali Pashte. Survey on distributed computing platform. *International Research Journal of Engineering and Technology*, 2016.

[17] Yang Sun, Ziming Zhuang, and C. Lee Giles. A large-scale study of robots.txt. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 1123–1124, 2007.

[18] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.

[19] Davide Vega, Roc Messeguer, Sergio F. Ochoa, José A. Pino, Felix Freitag, Esunly Medina, and Dolors Royo. Sharing hardware resources in heterogeneous computer-supported collaboration scenarios. *Integrated Computer-Aided Engineering*, 20(1):59–77, 2013.

Apéndice

A.1. Descripción entidad y atributos del modelo entidad relación de Django

A continuación se describe brevemente cada entidad y sus atributos del modelo entidad relación visto en la sección 4.2.1:

- **User**: entidad para los datos del usuario registrado en el sistema. Sus atributos vienen definido por el *framework* Django y son los datos de identificación y estado que se usan comúnmente en un sistema de administración de usuarios; el único atributo agregado en la implementación es el **token**:
  - **mail**: el correo electrónico del usuario.
  - **username**: un *string*[1] único y entendible para identificar a un usuario.
  - **password**: la contraseña de seguridad.
  - **isSuperUser**: un valor *booleano*[2] que se usa para definir si un usuario tiene permisos de administrador.
  - **isStaff**: un valor *booleano* que se usa para definir si un usuario tiene permisos de miembro privilegiado.
  - **token**: es un *string* aleatorio, único y difícil de adivinar que identifica a un usuario.
- **Slave**: entidad para un esclavo conectado al sistema y sus estados:
  - **state**: el estado actual del esclavo; puede ser *online* (en linea) u *offline* (desconectado).
  - **score**: dato que se utiliza para calificar el desempeño de un esclavo.
  - **token**: es un string aleatorio, único y difícil de adivinar que identifica a un esclavo.
  - **credits**: son los créditos acumulados por el trabajo realizado.
- **DataSlave**: entidad para los datos que caracterizan a un esclavo; cada entidad **Slave** tiene una unica instancia de esta entidad; sus atributos son:
  - **os_system**: es el sistema operativo del esclavo (Windows, Linux, MacOS, etc.)
  - **os_release**: es la versión actual del Kernel del sistema.
  - **os_version**: es la versión del sistema operativo. (Windows 10, 7, Vista, etc.)

[1] tipo de dato alfanumérico.
[2] tipo de dato que representa dos valores (1 ó 0) o estados (verdadero o falso).

  - **mac**: es la dirección MAC, un identificador (que se presume único) para la tarjeta de red del terminal.
  - **hostname**: es el nombre del terminal.
- **OperativeSystem**: entidad para los diferentes nombres de sistema operativo para evitar datos duplicados (Linux, Windows, MacOS).
- **IPAddressUser**: entidad para guardar la dirección IP del usuario. Cada entidad **User** puede tener varias instancias a esta entidad (cada nueva conexión guarda la dirección IP).
- **IPAddressSlave**: entidad para guardar la dirección IP del esclavo. Cada entidad **Slave** puede tener varias instancias a esta entidad (cada nueva conexión guarda la dirección IP).
- **DataCrawling**: entidad usada para guardar en detalle cada una de las consultas HTTP/S echas por cada esclavo. Sus atributos son:
  - **url**: es la dirección de la consulta HTTP/S.
  - **len_request**: es el largo de la respuesta a la consulta HTTP/S.
  - **status_code**: códigos de estado HTTP, que sirven para saber si la consulta fue hecha correctamente o hubo algún problema; en la siguiente entidad se explica más en detalle los estados.
  - **time_elapsed**: es el tiempo entre que se hizo la consulta HTTP/S hasta que se recibió por completo la respuesta.
  - **encoding**: es la codificación de la respuesta HTTP; existen múltiples codificaciones diferentes dependiendo del alfabeto o idioma que se utilice.
- **SummaryDataCrawling**: entidad usada para guardar un resumen de las consultas hechas por los esclavos; cada uno de los atributos es un contador de la cantidad total de respuestas agrupadas por sus códigos de estados HTTP:
  - **n_urls_2xx**: cantidad de consultas HTTP/S que tuvieron como respuesta un código de estado entre 200 y 208 (consultas correctas).
  - **n_urls_3xx**: cantidad de consultas HTTP/S que tuvieron como respuesta un código de estado entre 300 y 308 (redirecciones).
  - **n_urls_4xx**: cantidad de consultas HTTP/S que tuvieron como respuesta un código de estado entre 400 y 451 (errores del cliente).
  - **n_urls_5xx**: cantidad de consultas HTTP/S que tuvieron como respuesta un código de estado entre 500 y 512 (errores del servidor).

A.2. Interfaz gráfica de administración

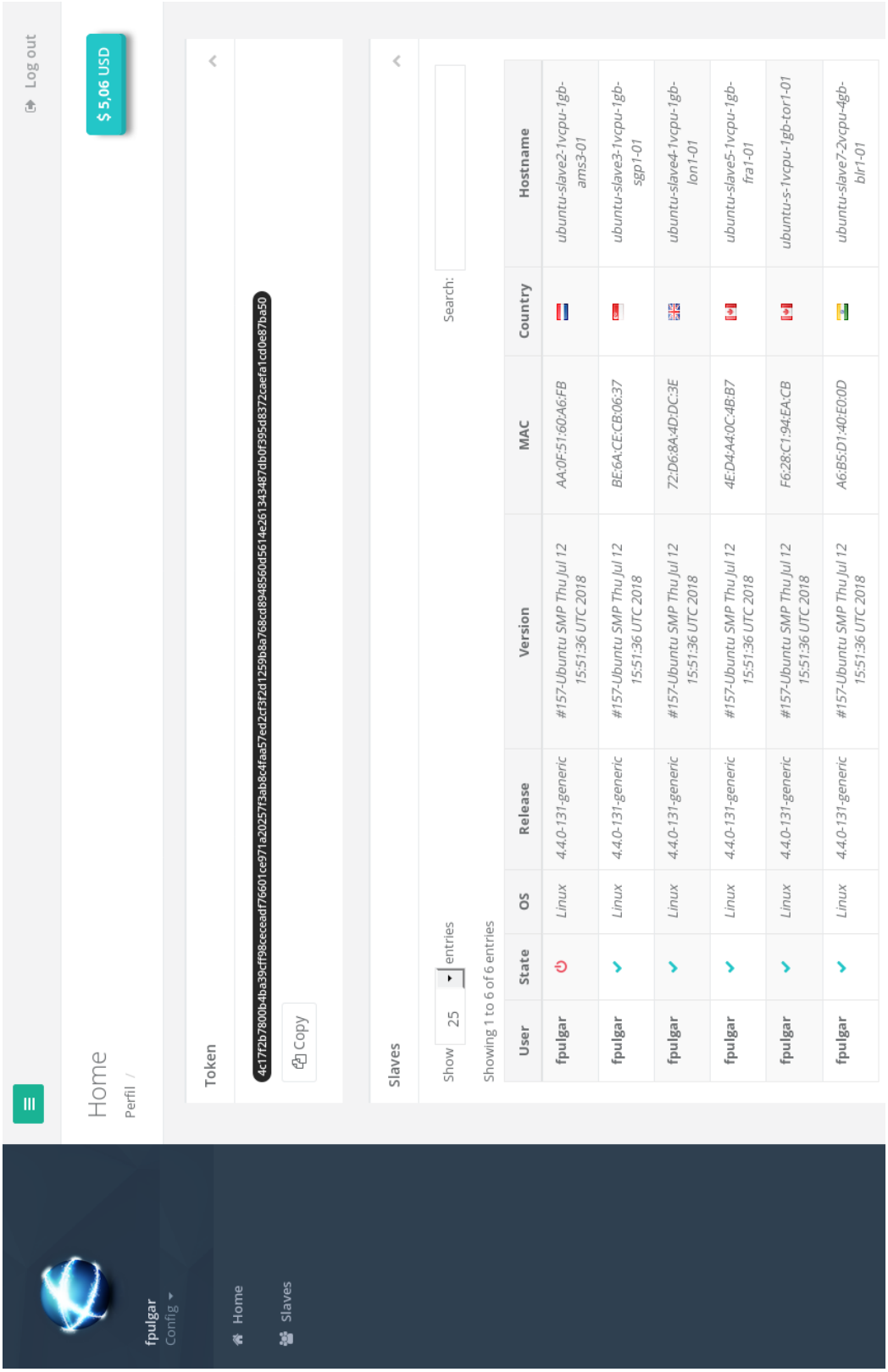

Figura A.1: Interfaz gráfica de administración de usuarios y sus dispositivos tecnológicos.

A.3. Símbolos y notación de diagramas BPMN

Figura A.2: Símbolos y notación de diagramas BPMN.