ÁREA DE ARQUITECTURA Y TECNOLOGÍA DE COMPUTADORES

DEPARTAMENTO DE INFORMÁTICA

PROYECTO FIN DE CARRERA

ADQUISICIÓN Y ANÁLISIS DE INFORMACIÓN GEOLOCALIZABLE DE UNA RED SOCIAL, APOYADO EN PROCESAMIENTO PARALELO

Autor: Beatriz Rodríguez Ruiz

Tutor: Francisco Javier García Blas

Leganés, junio de 2011

Título: Adquisición y análisis de información geolocalizable de una red social, apoyado en procesamiento paralelo.

Autor: Beatriz Rodríguez Ruiz.

Tutor: Francisco Javier García Blas.

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

AGRADECIMIENTOS

El primer lugar es para mis padres, José y Fe, porque siempre han creído firmemente en mí, incluso en aquellos momentos en los que ni yo misma lo hice.

A mi hermano David, porque le he aleccionado lo mejor que he podido en mis frikismos varios y mola un montón (porque se parece a mí, por supuesto).

A mi tutor Javier, por las largas y numerosas conversaciones vía GMail a horas intempestivas. Los profesores como tú son los que hacen que los alumnos tengamos inquietudes, y eso no tiene precio.

A mis compañeros del último curso de carrera, porque han puesto el broche de oro a mi paso por la Universidad: Moguel, Elena, Gonzalo, Almudena y Daniel. Lástima que solo haya sido un año, pero lo habéis convertido en inolvidable.

A mis amigas de toda la vida, Beatriz, Cristina y Sagrario, que me recuerdan delante de un ordenador hasta donde les alcanza la memoria.

A mis 'sosios', Lina, Luis y Sara, porque siempre están dispuestos a sacarme una sonrisa y a hacerme más llevaderas las horas de trabajo.

Al mejor tesoro que me llevo de la Universidad: Juanjo, sin ti no hubiera sido lo mismo. Gracias por estar en los buenos momentos, pero sobre todo, gracias por estar en los malos. Eres el mejor compañero que podría haber tenido.

Y en definitiva, a todos aquellos que me aprecian y me quieren. No olvidéis que aunque no estén todos vuestros nombres aquí escritos, sí que están en mi pensamiento.

RESUMEN

Este Proyecto Fin de Carrera estudia las diferentes áreas geográficas de incidencia de una persona, empresa, asociación, etcétera, a través de información susceptible de ser geolocalizada y que ha sido previamente obtenida de una red social. En concreto, la red social utilizada es Twitter y se emplean las relaciones de *following* (siguiendo) y *follower* (seguidor) para establecer los vínculos entre aquella persona (o más ampliamente, cuenta de Twitter que represente a cualquier entidad, sea cual sea esta) que será el origen del estudio, y aquellas otras sobre las que incide.

Los datos sobre los que se ha orientado el estudio han sido tres: el campo ‘localización’ del perfil de usuario de Twitter, la etiqueta 'geo' que se encuentra adjunta a cualquier *tweet* (mensaje) que se publica y utilizar el texto de los *tweets* para realizar búsquedas sobre él.

La cantidad de información que se deberá recoger de la red social es muy elevada, por lo que se propone para su tratamiento un enfoque que explote las bondades del procesamiento paralelo. Haciendo uso de una arquitectura paralela tipo clúster y utilizando un modelo de programación acorde a él, se estudiará el rendimiento de este método frente a una aproximación clásica no paralela.

El procesamiento paralelo se ha desarrollado bajo un modelo maestro-esclavo, en el que el primero se encarga de crear una porción de trabajo que después asigna a un esclavo que se encuentre libre. El esclavo será el que procese estos datos y sea capaz de obtener resultados parciales relativos a esa asignación de trabajo.

Finalmente, todos los esclavos devolverán los resultados parciales al maestro, quien calculará los resultados definitivos.

Palabras clave: red social, geolocalización, procesamiento paralelo, computación paralela

ABSTRACT

This Project aims to study the geographical areas in where a person, a company, an association, etc, has influence by using information capable of being localized and which has been previously gathered up from a social network. More specifically, the social network is Twitter, and its *following* and *follower* relationships are used to establish links between that person (or more generically, a Twitter account which represents to any entity, whichever it might be) who is the root of the study, and the other ones related to them.

The study is focus on three pieces of data: the ‘location’ field of a Twitter user profile, the 'geo' tag attached to any *tweet* (message) published and by searching on the *tweet* text itself.

The quantity of information that is required to be gathered up from the social network is very high, so for its processing it will be used a parallel computing approach. By using a parallel computing architecture such as a cluster and a parallel programming model, the performance of this method will be studied, and it will prove that it is better than a classic non-parallel approach.

The parallel processing has been developed following a master-slave model: the first one is responsible for creating a piece of work which later will be assigned to an idle slave. The slave will process the piece of work and will get partial results related to this assignment.

Finally, all the slaves will return their partial results them obtained to the master, who will calculate the definitive results.

Keywords: social network, geolocalization, parallel processing, parallel computing

Todo el mundo es un genio.
Pero si juzgas a un pez en su habilidad de escalar un árbol,
vivirá toda su vida creyendo que es un estúpido.

Albert Einstein

Just one more thing...
The people and the friends that we have lost,
or the dreams that have faded... Never forget them.

Yuna (Final Fantasy X)

TABLA DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

El objetivo de este capítulo es presentar el trabajo realizado en este Proyecto Fin de Carrera y describir la estructura del presente documento. Con esta introducción, se describirá brevemente el problema, los motivos que han llevado a su desarrollo, la solución propuesta y los objetivos que se desean alcanzar.

1.1 Motivación

El auge de las redes sociales es, a día de hoy, un fenómeno conocido por todos. Ya sea en mayor o menor medida, prácticamente cualquier persona conoce el término 'red social' y hay quienes las han convertido en parte de su rutina diaria.

Las redes sociales son servicios o plataformas online, cuya función consiste en construir y reflejar relaciones sociales establecidas entre personas, por ejemplo, entre aquellas que comparten un cierto interés o afinidad. Actualmente las redes sociales más conocidas son Facebook, Twitter, Tuenti, LinkedIn y MySpace, pero existen una innumerable cantidad de ejemplos.

La información que generan estas redes sociales no ha pasado desapercibida para investigadores de todo el mundo. Las poblaciones de las redes sociales y la información que generan son modelos reales sobre los que se pueden realizar estudios estadísticos y sociológicos. Estudiantes de campos muy diversos han realizado estudios abarcando diferentes temas y aproximaciones: conocer el perfil típico de un usuario de una red social concreta (1), analizar los sentimientos (2) o realizar análisis psicológicos basándose en la auto descripción de un perfil (3).

Las redes sociales también pueden ser utilizadas como medio de publicidad y marketing-viral. En esta línea, el artículo "Measuring User Influence in Twitter: The Million Follower Fallacy" (4) demuestra que el tamaño de la audiencia de un usuario de Twitter no está necesariamente relacionado con la influencia que este ejerce sobre ellos,

entendiendo esta influencia como no ser *retweeteado* (reenviar un *tweet* de otra persona) de forma significativa: los usuarios con más *followers* (seguidores) no son necesariamente los más influyentes a la hora de difundir sus mensajes.

Pero no sólo se queda ahí. Las redes sociales también se han utilizado por las fuerzas de la ley de diferentes países: Facebook fue usado por la policía del Reino Unido para descubrir actividades criminales, identificando a personas portadoras de armas (5). La misma red social fue utilizada en el caso de atropello y fuga de Carlee Vinos, un estudiante de la Universidad de Conneticut. La policía utilizó Facebook para establecer una conexión entre el conductor sospechoso, Anthony Alvino de Nueva York, con la universidad: Alvino tenía una relación con Michele Hall, una estudiante de la Universidad de Connecticut (6).

'Geolocalización' es un término de reciente creación, que oculta una de las áreas de mayor expansión tecnológica en los últimos años. La geolocalización, también conocida como georreferenciación, se refiere al posicionamiento con el que se define la localización de un objeto en un sistema de coordenadas concreto (7).

El lanzamiento de Google Earth en junio de 2005, marcó uno de los primeros hitos en el área de los Sistemas de Información Geográfica, permitiendo a cualquier persona ver en la pantalla de su dispositivo aquel lugar del mundo que desee. Google Earth permite explorar el espacio tridimensional que describe nuestro planeta y los datos que anidan en cada uno de sus recovecos (8).

Las posibilidades de la geolocalización son muy diversas. Según el Instituto Cartográfico Nacional, el 80% de la actividad humana requiere conocer su posición sobre un mapa. Por ejemplo, los servicios de emergencia pueden utilizar la geolocalización para detectar rutas rápidas que les conduzcan a determinados lugares, o se puede conocer el tiempo que tardará en llegar el próximo autobús a una parada, o localizar un móvil extraviado o robado (9).

En el congreso Mobile World Congress del año 2011, celebrado en Barcelona, el presidente de Google, Eric Schmidt, afirmó que "ahora, nadie se pierde", haciendo referencia a los últimos avances en telefonía móvil y geolocalización (9) .

Unos 500 millones de personas en todo el mundo acceden hoy como usuarios habituales a servicios de geolocalización. La previsión es de crecimiento, puesto que faltan por incorporarse a esa cifra muchos países en desarrollo (9).

La convergencia entre las redes sociales y la geolocalización es una realidad: Google Latitude o Foursquare son redes sociales que permiten conocer dónde se encuentran tus amigos.

1.2 Objetivos

El objetivo fundamental de este Proyecto Fin de Carrera consiste en identificar la presencia que tiene un usuario popular de una red social (entendiéndose esto como ser una personalidad destacada, una compañía, una asociación, un medio de prensa, etcétera) en diferentes zonas geográficas, con el fin de conocer cuáles son sus áreas de mayor actuación o relevancia.

Al estudiar un partido político, se podrá saber cuáles son las zonas en las que tiene un mayor número de afines. Al estudiar una compañía, se podrá conocer donde se localiza la gente interesada en sus productos, siendo estos compradores potenciales. Un actor podría conocer donde se concentran la mayoría de sus fans.

La manera de definir o medir la presencia se hará utilizando las relaciones de *following* o siguiendo de Twitter. A través de ellas, los usuarios de Twitter establecen vínculos con otros usuarios. Las relaciones de *following* son unidireccionales y no implican necesariamente una amistad sino un interés por parte de un usuario hacia otro.

El Proyecto Fin de Carrera presentará el desarrollo de una aplicación que permita recoger los datos generados por una población de usuarios de Twitter (compuesta por usuarios que tienen una relación de *following* hacía un usuario común) para posteriormente procesarlos haciendo uso de diversas aproximaciones, y así lograr información útil que permita conocer dónde se localiza cada uno de ellos.

La fuente de estos datos serán los propios usuarios de Twitter. Twitter es una red social que permite a sus usuarios publicar cualquier cosa que deseen escribir (una noticia, su estado de ánimo, algún evento, etcétera) en sus páginas personales a través de mensajes de texto de tamaño reducido (hasta 140 caracteres), de tal modo que otros usuarios puedan leerlos de forma asíncrona. Ser *following* de un usuario implica estar suscrito a los mensajes que dicho usuario publica.

Los datos se recogerán haciendo uso de la API desarrollada por Twitter y la aplicación debe tratarlos de la forma más rápida y eficiente posible.

El presente Proyecto Fin de Carrera deberá alcanzar los siguientes objetivos secundarios:

- Compatibilidad con la API de Twitter, puesto que los datos de entrada deben obtenerse a través de ella.
- Capacidad de obtener datos susceptibles de ser geolocalizados.
- Tratamiento de una forma rápida y eficaz de los datos obtenidos, creando estadísticas y recopilando toda la información útil que de ellos se pueda obtener.

1.3 Estructura del documento

El presente documento está dividido en varios capítulos, agrupados de forma lógica de tal modo que a través de lectura ordenada se consiga la comprensión paulatina y global de la aplicación y su dominio:

En el capítulo 1, INTRODUCCIÓN, se establece el marco del presente proyecto, un resumen del mismo, la presentación de sus objetivos y una tabla de los acrónimos utilizados en este documento .

En el capítulo 2, ESTADO DEL ARTE, se presenta una visión global de todas las tecnologías y otros elementos importantes que recoge el proyecto, y que son necesarios para su comprensión.

En el capítulo 3, ANÁLISIS, se ofrece una amplia reflexión sobre qué debe hacer la aplicación, dando lugar a los requisitos.

E el capítulo 4, DISEÑO E IMPLEMENTACIÓN, se desarrolla de manera más técnica la aplicación, explicando las herramientas software utilizadas y los puntos más interesantes del desarrollo y la implementación.

En el capítulo 5, EVALUACIÓN, se presenta el entorno de evaluación, los parámetros utilizados en las pruebas, los resultados logrados y las conclusiones obtenidas, tanto de los resultados obtenidos por la aplicación como del rendimiento del procesamiento paralelo.

En el capítulo 6, PRESUPUESTO Y LÍNEAS FUTURAS, se describen las mejoras propuestas sobre las que se deberían trabajar para completar con elementos adicionales el Proyecto, así como una exposición de la metodología utilizada y el presupuesto.

En el capítulo 7, BIBLIOGRAFÍA, se enumera toda la literatura utilizada como referencia durante la elaboración de este documento.

En el ANEXO I, se incluye un pequeño manual de usuario que permitirá conocer los requisitos hardware y software de la aplicación así como los parámetros necesarios para su correcta configuración y ejecución.

1.4 Acrónimos

Se recopilan en esta sección los acrónimos más importantes que han sido utilizados a lo largo del presente documento.

Acrónimo	Significado
ANSI	American National Standard Institute
API	Application Programming Interface
BCPL	Basic Combined Programming Language
DICT	DICTionary Network Protocol
FIFO	First Input First Output
FILE	

FTP/FTPS	File Transfer Protocol / File Transfer Protocol Secure
HTTP/HTTPS	Hypertext Transfer Protocol / Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IP	Internet Protocol. Generalmente, referido a una dirección IP.
IMAP/IMAPS	Internet Message Access Protocol / Internet Message Access Protocol Secure
JSON	JavaScript Object Notation
LDAP/LDAPS	Lightweight Directory Access Protocol / Lightweight Directory Access Protocol Secure
LIFO	Last Input First Output
MIT	Massachusetts Institute of Technology
MPI	Message Passing Interface
NSS	Mozilla's Network Security Services
PBS	Portable Batch System
POP3/POP3S	Post Office Protocol version 3 / Post Office Protocol version 3 Secure
POSIX	Portable Operating System Interface for uniX
REST	REpresentational State Transfer
RFC	Request For Comment
RSS	Really Simple Syndication
RTMP	Real Time Messaging Protocol
RTSP	Real Time Streaming Protocol
SCP	Secure CoPy
SMS	Short Message Service
SFTP	Secure File Transfer Protocol
SMTP / SMTPS	Simple Mail Transfer Protocol / Simple Mail Transfer Protocol Secure
SSH	Secure SHell
TELNET	TELecommunication NETwork
TFTP	Trivial File Transfer Protocol
URL	Uniform Resource Locator
XML	eXtensible Markeup Language

2 ESTADO DEL ARTE

2.1 Redes sociales

Los sitios web de redes sociales (en inglés, *social network sites*) también conocidos sencillamente como redes sociales (término que se empleará de ahora en adelante) pueden definirse como aplicaciones basadas en web que ofrecen servicios que permiten a las personas (10):

1. Construir un perfil público o semipúblico dentro de un sistema limitado.
2. Articular una lista de otros usuarios con los que se comparte una conexión.
3. Ver y atravesar sus listas de conexiones y aquellas hechas por otros dentro del sistema. La naturaleza y nomenclatura de estas conexiones pueden variar en cada red social.

Lo que hace únicas a las redes sociales no es que permiten a los individuos conocer a extraños, sino que las personas pueden articular y hacer visibles sus relaciones sociales. En muchas de las redes sociales más grandes, los participantes no están necesariamente buscando conocer gente nueva, sino que, están comunicándose con gente que ya conocen.

No todas las redes sociales comenzaron siendo tal. QQ comenzó como un servicio de mensajería instantánea chino, Cyworld como una herramienta de discusión tipo foro y Skyrock (anteriormente Skyblog) era un servicio de *blogging* antes de añadir características de redes sociales.

Las diferentes redes sociales implementan una amplia variedad de características técnicas, pero la base de todas ellas consiste en perfiles visibles que muestran una lista de amigos, quienes a su vez son usuarios del sistema. Los perfiles son páginas únicas asociadas a un usuario y donde este puede describirse a sí mismo. Después de unirse a una red social, la persona es instada a completar un formulario. El perfil es generado

utilizando las respuestas dadas en el formulario e incluyen típicamente edad, lugar y intereses. Es frecuente que el perfil cuente con espacio para añadir una foto. Algunas redes sociales permiten a los usuarios mejorar sus perfiles añadiendo contenido multimedia o modificando su apariencia visual (por ejemplo, modificando sus colores). Otras como Facebook, permiten a los usuarios añadir módulos (llamados "aplicaciones").

La visibilidad de un perfil varía dependiendo de la red social y considerando las opciones que el dueño de perfil establezca. Por defecto, los perfiles en Friendster son recuperados por mecanismos de búsqueda haciéndolos visible a cualquier persona, sin importar si esta posee o no cuenta en dicha red social. LinkedIn controla lo que los usuarios pueden ver basándose en si tiene una cuenta de pago. MySpace permite a los usuarios decidir que contenido del perfil es público. Las variaciones estructurales en la visibilidad y el acceso son una de las principales maneras en las que las redes sociales se diferencian unas de otras.

Después de unirse a una red social, los usuarios pueden identificar a otros usuarios dentro del sistema con el que tienen una relación. La etiqueta de estas relaciones difiere dependiendo de la red. Términos populares incluyen "amigos", "contactos" y "fans". La mayoría de las redes sociales requieren confirmación bidireccional para establecer el vínculo, pero algunas no. Estos enlaces unidireccionales son a veces etiquetados como "fans" o "followers" (del inglés, *seguidores*), pero la mayoría de las redes sociales los llaman "amigos" también. El término "amigos" puede resultar engañoso porque la conexión no implica necesaria amistad en su sentido más estricto y las razones por las que las personas están conectadas son variadas.

Mostrar públicamente las conexiones es un componente crucial de las redes sociales. La lista de "amigos" contiene enlaces a los perfiles de cada persona con la que se está enlazado, permitiendo a los usuarios cruzar el gráfico de la red que se forma, haciendo clic a través de la lista de "amigos". En la mayoría de los sitios, la lista de "amigos" es visible a cualquiera que pueda ver el perfil de un usuario, aunque hay excepciones.

La mayoría de las redes sociales proporcionan un mecanismo para dejar mensajes en los perfiles de sus "amigos". Esta característica típicamente consiste en dejar "comentarios" aunque otros sitios utilizan una nomenclatura diferente para ella. Además, las redes ofrecen a menudo mensajería privada, muy similar al *webmail*.

Más allá de perfiles, "amigos", comentarios y mensajería privada, las redes sociales varían enormemente en sus características y usuarios. Unas permiten compartir fotos o vídeo y otras permiten *bloggear*. Hay redes sociales específicas para móviles (la desaparecida Dodgeball), y otras redes basadas en web también soportan interacciones a través de móviles (Facebook, MySpace, Twitter). Muchas redes sociales tienen como objetivo regiones geográficas específicas o grupos lingüísticos (como Tuenti en España), aunque esto no siempre determina a los inscritos en ella. Orkut fue lanzada en Estados Unidos con una interfaz únicamente inglesa, pero los brasileños portugueses

parlantes se hicieron rápidamente el grupo dominante. Otros sitios son diseñados considerando elementos que definen la identidad de una persona: etnia, religión, orientación sexuales o política.

Las redes sociales se diseñan para ser ampliamente accesibles, pero al comienzo suelen atraer a usuarios homogéneos por lo que no es poco frecuente encontrar grupos que utilizan los sitios para clasificarse ellos mismos por nacionalidad, edad, nivel educacional u otros factores que son segmentadores de la sociedad, incluso si esa no fue la intención de los diseñadores.

2.2 Twitter

En este punto se definirá y describirá la red social Twitter. En este apartado se detallará con más detalle esta red social debido a que este Proyecto Fin de Carrera se centra en esta red en concreto.

2.2.1 ¿Qué es Twitter?

Según se explica en la página de soporte del propio sitio, Twitter es un mecanismo que *ayuda a la gente a comunicarse y a mantenerse en contacto a través de un intercambio rápido y frecuente de mensajes de hasta 140 caracteres* (11).

Twitter es una red de información de tiempo real, que permite a sus usuarios conectarse a lo que consideran interesante: basta con encontrar cuentas de personas que publican contenido que les gusta y seguir sus conversaciones (12).

Twitter tiene su sede en San Francisco, California, pero es utilizado por personas de cualquier país del mundo. Twitter se ofrece en inglés, francés, alemán, italiano, japonés y español. Los usuarios pueden cambiar su preferencia de idioma en su configuración de usuario si así lo desean.

Twitter se basa en pequeños trozos de información llamados *tweets* (del inglés, *piar, grojear*). Cada *tweet* tiene un límite de 140 caracteres de longitud. Conectado a cada *tweet* hay un panel de detalles que permite añadir información adicional, mayor contexto y contenido multimedia embebido.

No es necesario *twittear* para disfrutar de Twitter: algunas personas nunca escriben *tweets*, sino que simplemente usan Twitter para obtener la información más actualizada sobre aquellos temas que les interesan.

Twitter también funciona en dispositivos móviles a través de aplicaciones gratuitas para iPhone, iPad, Blackberry, Windows Mobile 7 y Android, e incluso se pueden enviar y recibir actualizaciones utilizando mensajes de texto de móviles (SMS) (en España el servicio de SMS no está disponible (13)).

Twitter tiene 175 millones de usuarios registrados, se escriben 95 millones de tweets al día y cuenta con 300 trabajadores (12).

2.2.2 Terminología básica de Twitter

En Twitter, cada usuario está identificado de forma unívoca a través de un nombre de usuario. Todo usuario puede escribir mensajes que aparecen publicados en su página personal (también llamada página de inicio, perfil público, historial, cronología o línea temporal). Estos mensajes cortos tienen un máximo de 140 caracteres y se conocen con el nombre de *tweets*.

Tweet se utiliza también como un verbo, como por ejemplo en la frase "Yo twitteé esta mañana antes de salir de viaje". En este caso, el verbo "twittear" hace referencia a escribir y enviar un *tweet*, por lo que sinónimos serían "publicar" y "actualizar".

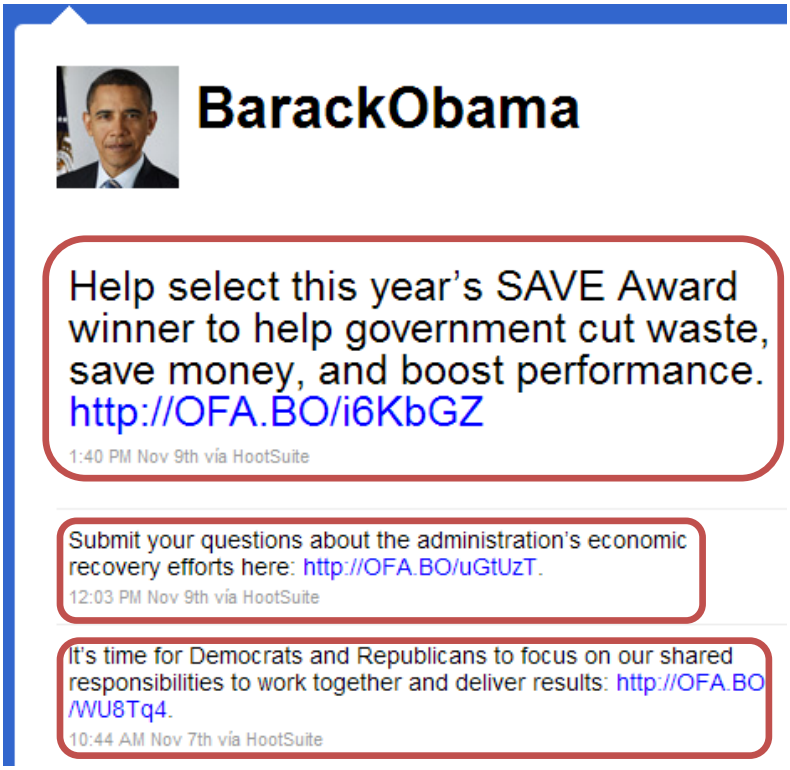


Ilustración 1: tweets de Barack Obama, presidente de los Estados Unidos

Cualquier usuario puede seguir los *tweets* que son publicados por otros usuarios, generalmente por considerar interesantes sus mensajes. Esto es conocido como 'estar siguiendo' (en inglés, *following*).

Naturalmente, existe el rol recíproco en el que una usuario es seguido por otro. En este caso, el término utilizado para denominar esta situación es tener 'seguidores' (en inglés, *followers*). Los seguidores son personas que reciben los *tweets* de otra.

En resumen, si un usuario A sigue a otro usuario B:

- el usuario A se convierte en "seguidor" de B, y se incrementa el contador de "siguiendo" de A.
- el usuario B incrementa su contador de "seguidores".

- las actualizaciones del usuario B aparecerán en la página personal del usuario A.
- el usuario B puede enviar mensajes directos a A (el concepto de mensaje directo se explica más adelante).

En Twitter, seguir a un usuario no es mutuo: un usuario puede seguir a otro sin que este siga al primero y viceversa. Además, Twitter permite administrar a quién se sigue, pudiendo seguir o dejar de seguir a un usuario en cualquier momento.

La lista de *following* y *followers* (siguiendo y seguidores) de cada usuario aparece en su página personal. Es en ella donde un usuario puede comenzar a seguir a otro.

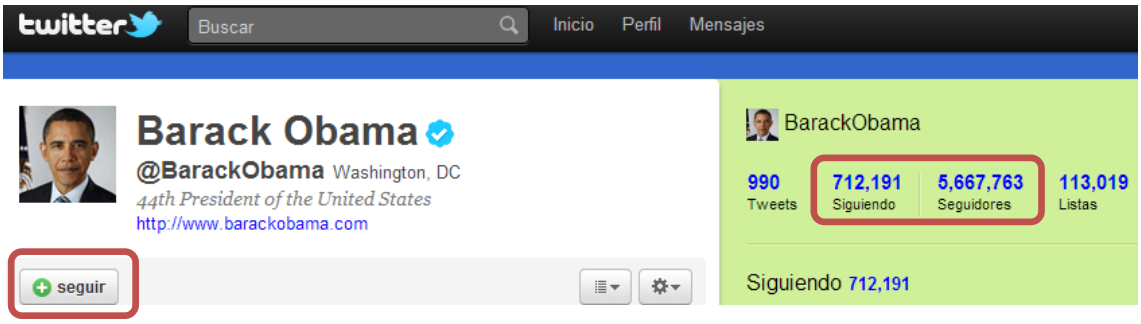


Ilustración 2: página personal en Twitter de Barack Obama

Twitter permite a sus usuarios restringir quienes les siguen o ven sus actualizaciones. Para ello, existe la opción de proteger el perfil de tal modo que cualquier persona que desee seguir a otra debe mandar una petición. Solo los seguidores que sean aprobados podrán ver los tweets. Además, si un perfil está protegido, los tweets asociados a ella no aparecerán en los resultados de búsqueda (14).

Una de las funcionalidades de Twitter es el intercambio de mensajes públicos con usuarios individuales: basta con comenzar un *tweet* con @nombredeusuario de la persona con la que se desea comunicarse. Si el destinatario está siguiendo la cuenta del emisor del mensaje, este aparecerá en su página de inicio. Si no, el mensaje aparecerá en su carpeta de menciones de usuario. Todas las personas que siguen a quien hace una mención también verán el mensaje en su página principal de Twitter. La mención es una característica adicional de la publicación de un *tweet* por lo que el mensaje que contiene la mención aparecerá en los resultados de búsqueda y se verá como un *tweet* más en la página personal del usuario que lo escribió (15).

Otra característica interesante es que las menciones se convierten automáticamente en un vínculo a la cuenta de dicha persona, ayudando así a la gente a descubrirse.

Ilustración 3: mención al usuario Pe_barrero escrita por el usuario redenzor

Los mensajes directos (en inglés, *Direct Messages*, *DM*) son el canal de mensajería privada de Twitter. Estos *tweets* no aparecen en la línea de tiempo ni en los resultados de búsqueda. Nadie excepto el destinatario puede ver un DM. Los DM se pueden enviar sólo a personas que te siguen y solamente pueden ser recibidos por gente que estás siguiendo.

Para ayudar a difundir "buenos" mensajes a través de Twitter, un usuario puede volver a publicar un mensaje que originariamente fue escrito por otra persona. Esto es conocido como *retweeting* o RT. Es frecuente su utilización junto con las menciones.

Para clasificar los *tweets* en temas, *twittear* en grupos o añadir datos extra, se utilizan etiquetas. Las etiquetas (también conocidas como *hashtags*) están compuestas por el símbolo almohadilla (#) antecediendo a la palabra que categoriza, como por ejemplo, #España. Éstas pueden aparecen en cualquier lugar del *tweet*.

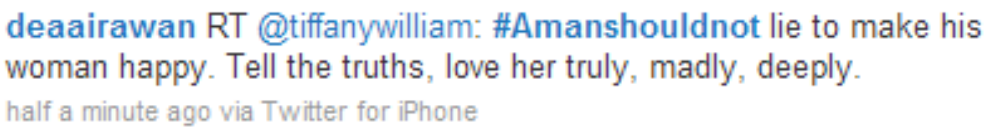


Ilustración 4: tweet del usuario deaairawan que incluye un retweet (RT) y una mención al usuario @tiffanywilliam y una etiqueta #Amanshouldnot

Cada usuario tiene asociado un perfil al cual puede añadir información complementaria, como su nombre real, su ubicación, una imagen, una dirección web asociada por ejemplo a un *blog*, y un cuadro de texto para escribir una pequeña biografía.

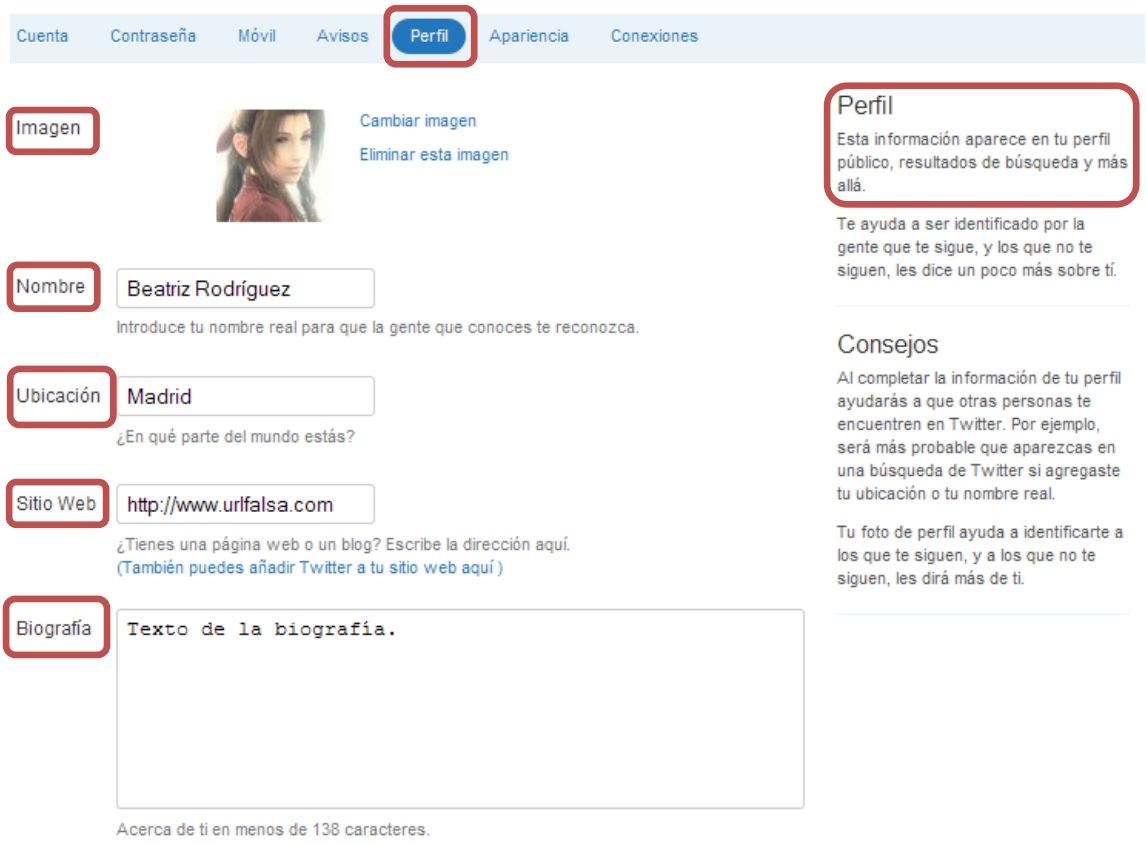


Ilustración 5: perfil del usuario yunnyloire

No se debe confundir el perfil, que es la información complementaria que un usuario de Twitter puede añadir, con el perfil público o línea temporal, que es la página personal de todo usuario de Twitter. Es en el perfil público donde aparece la información extra añadida en el perfil.

Ilustración 6: perfil público, página personal o línea temporal del usuario yunnyloire

2.2.3 Twittea con tu ubicación

"Twittea con tu ubicación" (en inglés, *tweet with your location*) es una nueva característica de Twitter que permite a los usuarios añadir información sobre su localización en cada *tweet* (16).

Por cuestiones de privacidad, es una característica que está desactivada por defecto y necesita ser activada si se desea utilizar. Para ello, basta con acceder a la configuración de la cuenta y activar la opción asociada. Una vez que esta opción está habilitada se puede añadir información individual de localización a cada *tweet* cuando este es escrito, ya sea desde Twitter.com o cualquier otra aplicación o dispositivo móvil que soporte esta característica.

Toda información de geolocalización comienza como una localización exacta, definida por coordenadas de latitud y longitud, la cual es enviada por el navegador o un dispositivo móvil. Basándose en esta información, los *tweets* pueden ser complementados con información sobre la localización exacta (las coordenadas), el lugar (como un vecindario o una ciudad) o ambas. Esta información es pública y puede ser vista por los demás usuarios, por lo que es conveniente ser cauteloso con los datos que se publican.

Twitter, en su afán por mantener la privacidad del usuario, permite controlar qué información es compartida y cuándo. Aún si la funcionalidad "Twittea con tu ubicación" está activada, el usuario puede elegir el nivel de precisión de la información adjuntada como su localización, elevándolo a uno más general (de coordenadas a calle, vecindario, ciudad...) o incluso no añadir información de localización a un *tweet* concreto. Twitter también permite la desactivación total de la funcionalidad "Twittea con tu ubicación", de tal modo que se pueden borrar los *tweets* que tienen datos de localización o únicamente, borrar la localización de aquellos *tweets* que la tengan.

"Twittea con tu ubicación" está disponible en varios países, entre ellos Estados Unidos y España, y se prevé su expansión progresiva a todos los países en los que Twitter opera(17).

Ilustración 7: tweet del usuario redenzor enviado desde Downers Grove, Illinois.

2.2.4 La API de desarrollo de Twitter

Twitter ofrece una completa API para desarrollar aplicaciones integradas con Twitter.

Esta API está formada por tres grandes bloques: REST API, Search API y Streaming API. Cada una de ellas tiene unas funcionalidades específicas (18).

La REST API proporciona las funcionalidades básicas de Twitter, como escribir un *tweet*, seguir a alguien, obtener la información de un usuario, etc. Los métodos de la Search API proporcionan a los desarrolladores una forma de realizar búsquedas en tiempo real y conocer los temas de moda. La Streaming API proporciona acceso en tiempo real a un gran volumen de información de *tweets* de una forma filtrada y ordenada.

Para poder utilizar las APIs de Twitter es necesario crear una cuenta en Twitter (con su correspondiente nombre de usuario y contraseña asociados) y además, registrar una aplicación de cliente (19). Cada aplicación de cliente tiene asociados una *consumer key*, que es un identificador único de la aplicación en formato alfanumérico, y un *consumer secret*, que evita que se puedan suplantar *consumer keys* en ataques de fuerza bruta. Estos elementos son utilizados en la autenticación OAuth que Twitter emplea para utilizar sus APIs.

Obtener datos de la API de Twitter se traduce en una operación de lectura. Para realizarla, basta con establecer una conexión HTTP y enviar una petición de tipo GET al recurso deseado. Se ofrece un ejemplo (20):

1. Se desea recoger la información de un status (sinónimo de *tweet*), especificando su identificador, (por ejemplo, el 13762161921). Esta funcionalidad está recogida por la URL de la REST API (21):

```
http://api.twitter.com/version/statuses/show/:id.format
```

2. Basándose en la URL anterior, se crea una cadena que incluya los parámetros adecuados. En este caso, la versión de la API, el id del status y el formato de retorno de la petición:

```
http://api.twitter.com/1/statuses/show/13762161921.xml
```

3. Por último, se lanza la solicitud utilizando GET como método de la petición. Esto puede hacerse utilizando un navegador de Internet (y se copia la URL en la barra de direcciones) o utilizando cURL.

La información protegida requiere de autenticación por parte del solicitante para que la petición sea aceptada. La mayoría de las peticiones a la API tipo GET pueden realizarse sin estar autenticado (como la mostrada en el ejemplo anterior). Sin embargo, Twitter destaca que la información protegida no será mostrada si no existe previa autenticación. Además, las peticiones tipo POST requieren necesariamente de autenticación. La autenticación se convierte por tanto un elemento imprescindible si se desea enviar información a Twitter, a la vez de lograr un mayor aprovechamiento de las APIs con ella, incluyendo un aumento en el número de peticiones que se pueden realizar a la API en un espacio de tiempo.

Ilustración 8: petición a la REST API de Twitter en formato XML utilizando el navegador Firefox

Para evitar abusos o ataques maliciosos, Twitter establece unos límites en el número de peticiones que pueden realizarse a la API en el periodo de una hora (22).

La tasa límite por defecto para las llamadas de la API REST varía dependiendo del método de autenticación usado y de si el recurso solicitado requiere o no autenticación:

- Las llamadas anónimas se basan en la IP y se permiten hasta 150 peticiones por hora.
- Las llamadas utilizando autenticación OAuth están limitadas a 350 por hora.

Los límites aplican a peticiones de información utilizando el comando HTTP GET. Generalmente, los métodos de la API que hacen uso de HTTP POST para enviar datos a Twitter no están limitados.

2.2.5 Usurpación de identidad en Twitter

Twitter controla las usurpaciones de identidad en su servicio. Se considera una usurpación de identidad el intento de engaño, o simplemente con fines lúdicos, al fingir

ser otra persona o empresa. Se considera violación de la normativa de Twitter la usurpación de identidad no paródica (23).

Los casos de usurpación de identidad pueden ser denunciados, del mismo modo que los datos de una cuenta que pertenecen a algún personaje popular o empresa pueden ser enviados a Twitter para que este certifique la cuenta como 'Cuenta verificada'.

Una cuenta que tiene este distintivo está acompañada de un *tick* color azul y blanco al lado del nombre de usuario y tiene el reconocimiento de Twitter de ser una cuenta válida, que representa a la persona o empresa de forma fidedigna.

Ilustración 9.- Cuenta verificada de Twitter del actor Aston Kutcher

2.3 Protocolo OAuth versión 1.0

OAuth es un protocolo abierto que permite la autorización segura de manera simple y estándar, y que puede ser utilizada en aplicaciones de escritorio y en aplicaciones web (24).

OAuth proporciona a los usuarios un método para otorgar acceso a recursos de terceras partes sin compartir sus credenciales, y además, proporcionar un mecanismo para dar accesos limitados (ámbito, duración, etc.) (25).

OAuth define tres roles: cliente (en inglés, *client*), servidor (en inglés, *server*) y el propietario del recurso (en inglés, *resource owner*). Estos tres roles están presentes en cualquier transacción de OAuth; en algunos casos, el cliente es también el propietario del recurso. La versión original de la especificación de OAuth (conocida como The OAuth Core 1.0, (26)) utilizaba unos términos diferentes para estos tres roles: consumidor (el ahora cliente), proveedor de servicios (el ahora servidor) y el usuario (el ahora propietario del recurso).

En el modelo tradicional de autenticación cliente-servidor, el cliente usa sus credenciales para acceder a sus recursos almacenados por el servidor. El servidor sabe que el secreto compartido usado por el cliente pertenece al cliente, por lo que realmente no le importa de dónde procede o si el cliente está actuando en nombre de otra entidad. Mientras que el secreto compartido se ajuste a lo que el servidor espera, la petición es procesada.

Ilustración 10: modelo de autenticación tradicional cliente-servidor

En numerosas ocasiones, el cliente está actuando en el nombre de otra entidad. Esta entidad puede ser otra máquina o una persona. Cuando un tercer actor está involucrado (típicamente un usuario interactuando con el cliente) el cliente está actuando en nombre del usuario. En estos casos, el cliente no está accediendo a sus propios recursos pero sí a los del usuario, quien es el propietario de los recursos.

En lugar de utilizar las credenciales del cliente, el cliente está utilizando las del propietario del recurso para hacer las peticiones fingiendo así ser el propietario del recurso. Las credenciales de un usuario típicamente incluyen un nombre de usuario y una contraseña, pero los propietarios del recurso no están limitados a ser los únicos usuarios, sino que puede serlo cualquier entidad que controle los recursos del servidor (27).

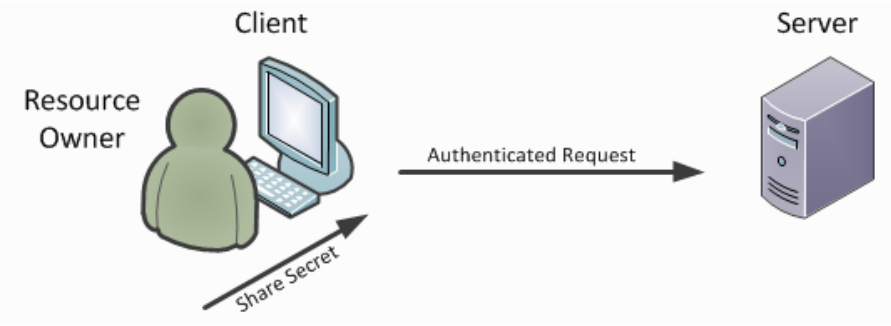


Ilustración 11: modelo de autenticación del protocolo OAuth

Un recurso protegido es un recurso almacenado o proporcionado por el servidor y que requiere de autenticación para acceder a él. Los recursos protegidos pertenecen o son controlados por el propietario del recurso. Cualquiera que solicite acceso a un recurso protegido debe ser autorizado a hacerlo por el propietario del recurso (es una condición impuesta por el servidor).

Un recurso protegido pueden ser datos (fotos, documentos, contactos) servicios (de *blogging*, transferencia de fondos) o cualquier recurso que tenga restricciones de acceso. OAuth solo está definido para recursos HTTP/HTTPS, aunque puede ser usado con otros protocolos de transporte.

Póngase un ejemplo: el usuario de una web (propietario del recurso) puede otorgar a un servicio de impresión (el cliente) acceso a sus fotos privadas almacenadas en un servicio de compartición de fotos (servidor), sin compartir su nombre de usuario ni contraseña con el servicio de impresión. En lugar de eso, el usuario se autentica

directamente con el servicio de compartición de fotos el cual emite credenciales específicas de delegación al sistema de impresión(28).

OAuth utiliza tres tipos de credenciales: las credenciales del cliente, credenciales temporales y credenciales *token*. En la especificación original(26), se utilizaban unos términos diferentes para denominar a estas credenciales: *consumer key y consumer secret* (credenciales del cliente), *request token y request secret* (credenciales temporales) y *access token y access secret* (credenciales de *token*).

Las credenciales del cliente son utilizadas para autenticar al cliente. Esto permite al servidor recoger información sobre como el cliente usa sus servicios, ofreciendo a algunos clientes un trato especial o proveyendo al propietario del recurso más información sobre los clientes.

Las credenciales de tipo *token* se usan en lugar del nombre de usuario y contraseña del propietario del recurso. En vez de que el propietario del recurso comparta sus credenciales con el cliente, este autoriza al servidor a emitir una clase especial de credencial al cliente, que representa el acceso que se le otorga al cliente por parte del propietario del recurso. El cliente usa este credencial de tipo *token* para acceder al recurso protegido sin tener que conocer explícitamente las credenciales (generalmente, nombre de usuario y contraseña) del propietario del recurso.

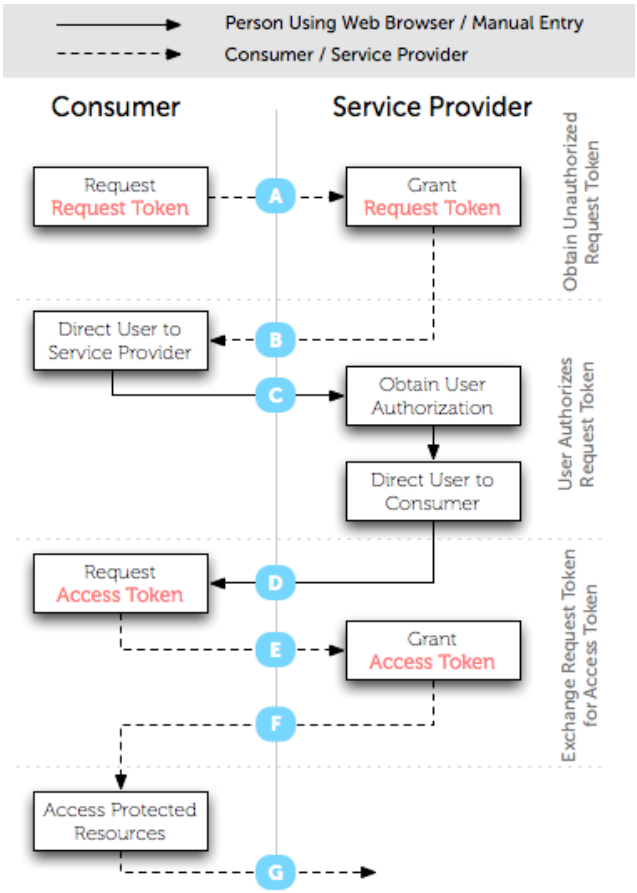


Ilustración 12: diagrama de flujo de la autenticación vía OAuth v1.0a

Las credenciales de tipo *token* incluyen un identificador del *token*, que generalmente es una cadena aleatoria de letras y números que es única, difícil de adivinar y está emparejada con un secreto para proteger al identificador del *token* de ser usado por partes no autorizadas.

El proceso de autorización de OAuth además utiliza un conjunto de credenciales temporales que son utilizadas para identificar la petición de autorización. y para ajustarse a diferentes tipos de clientes (basados en web, de escritorio, móviles, etc.), ofreciendo así flexibilidad y seguridad (27).

En el caso típico de OAuth (como es el presentado como ejemplo), el cliente, el servidor y el propietario del recurso forman las 3 partes involucradas en el proceso; esto se describe como "tres partes" o en inglés, *3-legged*. Cuando el cliente es también el propietario del recurso (esto es, que actúa en nombre de sí mismo), se denomina "dos partes" o *2-legged*.

La última versión de OAuth fue publicada en abril del 2010, como RFC 5849 (29). En la actualidad, se trabaja sobre una nueva versión, OAuth 2.0, pero es un protocolo completamente nuevo y no tiene retrocompatibilidad con versiones anteriores (30).

2.4 Lenguaje de programación C

C es un lenguaje de programación versátil, flexible y poderoso que fue diseñado y desarrollado en 1972 por Dennis Ritchie y Brian W. Kernighan. C incorporó muchas de las ideas de dos lenguajes que le precedieron: BCPL y B. BCPL (*Basic Combined Programming Language*) fue desarrollado en 1967 por Martin Richards. BCPL era un lenguaje de programación sin tipos de datos: cada dato ocupaba una palabra (en inglés, *word*) en memoria y la responsabilidad de darle tipo a las variables recaía sobre el programador. En 1970, Ken Thomson desarrolló B, que también era no tipado, y fue utilizado para crear las primeras versiones del sistema operativo UNIX en los laboratorios Bell. El lenguaje C fue el resultado de la evolución de estos lenguajes, llevada a cabo por Dennis Ritchie y Brian W. Kernighan . La evolución incorporó tipaje de datos y otras características que lo convirtieron en un lenguaje muy potente. C está muy relacionado con el sistema operativo UNIX ya que se desarrolló junto con él. El objetivo principal de C era crear un lenguaje para escribir sistemas operativos.

Ritchie y Kernighan escribieron un libro histórico, "The C Programming Language", que describía el C tradicional utilizado hasta el 1983. Varias mejoras fueron hechas al lenguaje original conduciendo a características no estándares y a algunas construcciones del lenguaje ambiguas. En 1983, el American National Standard Institute (ANSI) formó un comité para estandarizar el lenguaje haciéndolo no ambiguo y no dependiente de la máquina. Este estándar fue publicado en 1989 y se conoce como ANSI C.

El poder de C radica en el uso de punteros para manipular variables, *arrays* de variables y estructuras. Los punteros y su capacidad de reserva dinámica de memoria son sus más poderosas características. C tiene una rica colección de librerías que soportan funciones de E/S, funciones matemáticas, funciones de cadenas y de almacenamiento dinámico, haciéndolo muy útil en aplicaciones de ingeniería y científicas (31).

2.5 JSON

JSON se corresponde con las siglas *JavaScript Object Notation* y es un formato ligero para el intercambio de datos entre distintas partes. Es un lenguaje muy sencillo de leer y escribir por las personas, y fácil de ser analizado sintácticamente y ser generado por máquinas. Aunque es un subconjunto de la notación de JavaScript, JSON es un formato de texto completamente independiente de JavaScript o cualquier otro lenguaje de programación (32).

Los mensajes JSON se construyen siguiendo dos estructuras:

- Una colección de pares nombre/valor. En varios lenguajes, esto equivale a un *object*, *record*, *struct*, *dictionary*, *hash table*, *keyed list* o *associative array*.
- Una lista de valores ordenados. En muchos lenguajes, equivale a un *array*, *vector*, *list* o *sequence*.

En JSON, los mensajes toman la forma de *object* y *array* respectivamente, y se describen del siguiente modo:

Un *object* es un conjunto no ordenado de pares nombre/valor. Un *object* comienza con { (llave izquierda) y termina con } (llave derecha). Cada nombre es seguido por : (dos puntos) y el par nombre/valor separados por , (coma).

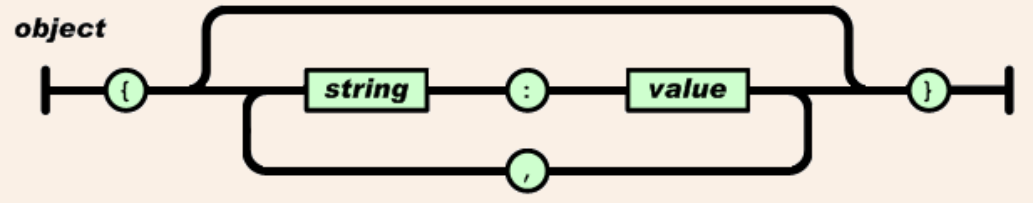


Ilustración 13: construcción de un JSON object

Un *array* es una colección ordenada de valores. Un *array* empieza con el carácter [(corchete izquierdo) y termina con] (corchete derecho). Los valores están separados entre sí por , (coma).

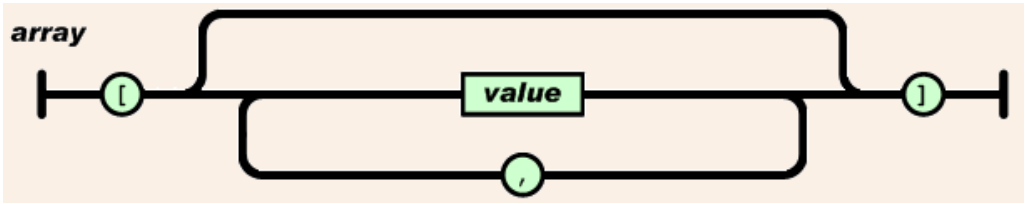


Ilustración 14: construcción de un JSON array

Los *objects* y *arrays* necesitan de *strings* y *values* para contener información útil. Se forman del siguiente modo:

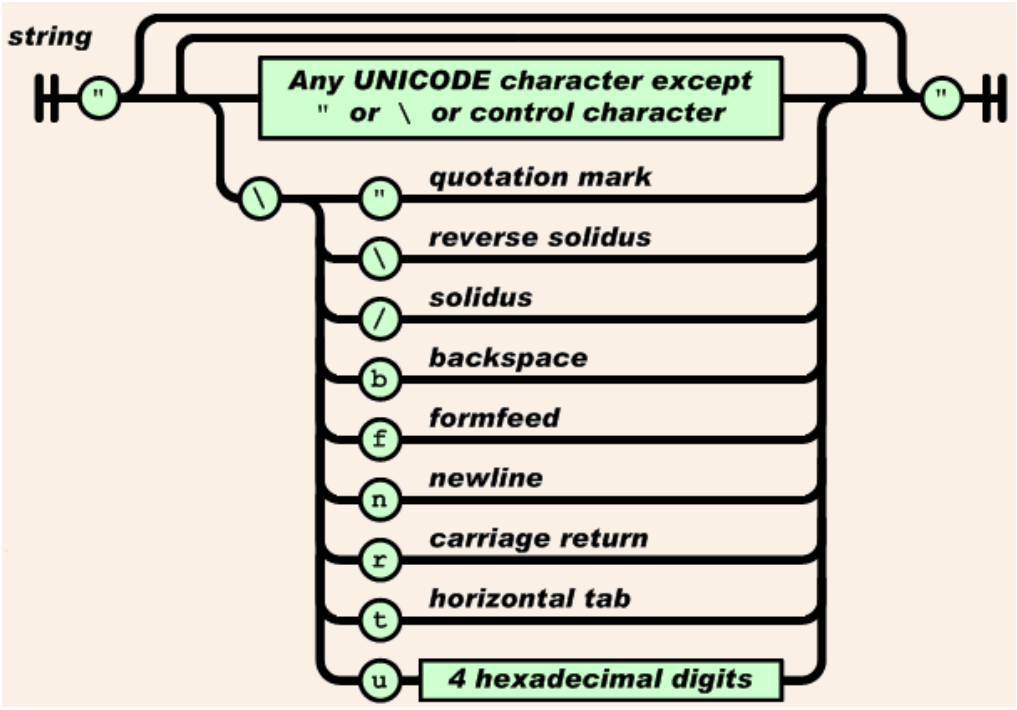


Ilustración 15: formación de un string en JSON

Un *string* es una colección de cero o más caracteres Unicode, envueltos entre " (comillas dobles). Un *string* de JSON es muy similar a uno de C o Java, pudiéndose utilizar escapes de barra invertida (\).

Un *value* puede ser un *string* entre " (comillas dobles), un *number*, *true* o *false* o *null*, un *object* o un *array*. Estas estructuras pueden anidarse.

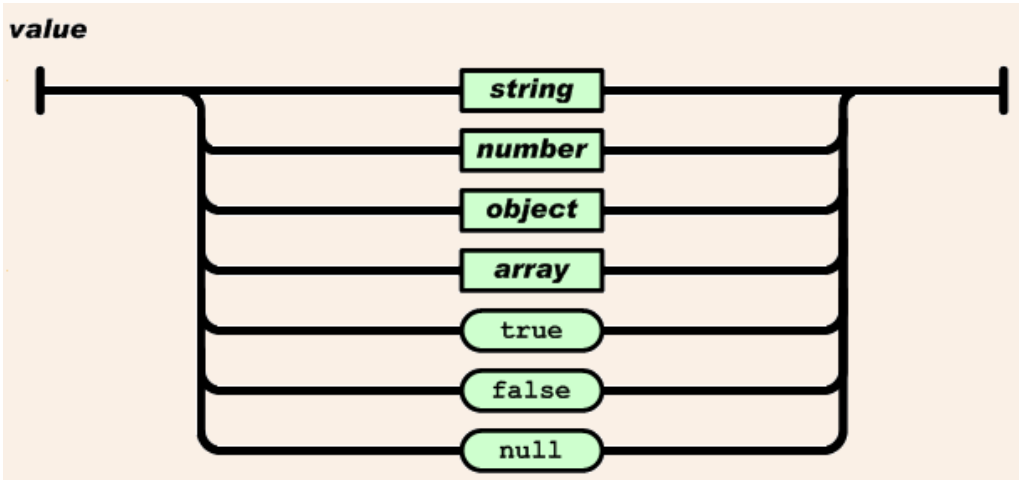


Ilustración 16: opciones válidas para la formación de un value en JSON

Un *number* es muy similar a un número de Java o C, exceptuando que el formato octal y hexadecimal no se utilizan.

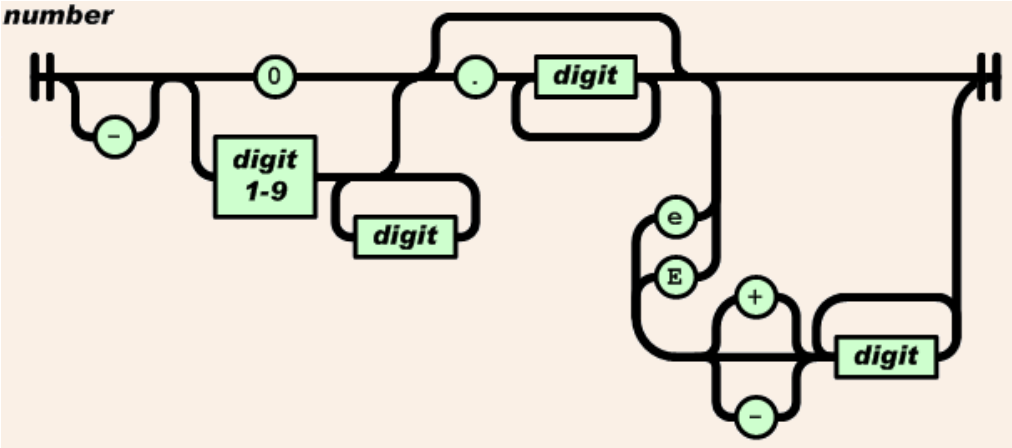


Ilustración 17: formato de un number en JSON

2.6 Computación paralela

La computación paralela consiste en *el uso de una computadora paralela para reducir el tiempo que se necesita para resolver un problema computacional* (33).

En la definición de computación paralela se utiliza el de computador paralelo. Un computador paralelo *es un sistema multiprocesador que soporta la programación paralela* (33). La programación paralela *es la programación en un lenguaje que permite indicar explícitamente cómo las diferentes partes de la computación deben ser ejecutadas por los diferentes procesadores* (33).

2.7 Arquitecturas paralelas

Siguiendo la definición dada por Almasi y Gottlieb, un computador paralelo es *una colección de elementos procesadores que se comunican y cooperan para resolver grandes problemas rápidamente*(34).

En el campo de la ciencia y la ingeniería, el aumento en los niveles del rendimiento que ofrecen los ordenadores es importantísimo, puesto que en estos campos con frecuencia se deben simular fenómenos físicos que son imposibles o muy costos de observar a través de métodos empíricos: son caros, difíciles, lentos o incluso peligrosos. Típicos ejemplos incluyen la modelización del cambio climático global a través de largos periodos de tiempo, la evolución de las galaxias, la estructura atómica de los materiales y el procesamiento de voz e imágenes.

Los modelos computacionales permiten realizar análisis en profundidad que pueden ser llevados a cabo de forma económica en hipotéticos diseños a través de la simulación por ordenador. Existe pues una correspondencia directa entre los niveles del rendimiento computacional y los problemas que pueden ser estudiados a través de la simulación: a mayor potencia de cálculo y capacidad de almacenamiento, mayores y más complejos problemas podrán ser simulados (35).

El rendimiento de una máquina está limitado (entre otros elementos) por el rendimiento del microprocesador. Para conseguir un alto rendimiento y mantener la

escalabilidad, los sistemas se han convertido en multiprocesadores. Los componentes que forman un sistema multiprocesador son (36):

- Un conjunto de procesadores.
- Una interconexión entre los procesadores y el resto del sistema.
- Los controladores de la E/S están conectados en un bus especializado (el bus de E/S).

Los multiprocesadores son arquitecturas paralelas y se dividen en dos grandes grupos (36):

- *Tightly-coupled* (del inglés, *fuertemente asociado*), *Symmetrical Multiprocessor* (SMP) (del inglés, *multiprocesador simétrico*) o sistemas de memoria compartida, en los cuales, los procesadores pueden ver y utilizar todos los recursos del sistema (memoria y dispositivos de E/S). El término "simétrico" hace referencia al hecho de que todos los procesadores son iguales en el sistema: cualquier cosa que un procesador es capaz de hacer, puede hacerla también cualquier otro, lo que implica también que cualquier procesador puede tomar cualquier rol. El término "fuertemente asociado" viene dado por el hecho de que existen conexiones muy cercanas entre los procesadores del sistema. Un SMP opera bajo el control de una única copia del sistema operativo, la cual gestiona todos los recursos del sistema. Para incrementar la capacidad de procesamiento basta con añadir un procesador (hasta el máximo que soporte el sistema), lo cual es una forma económica de aumentar el rendimiento.

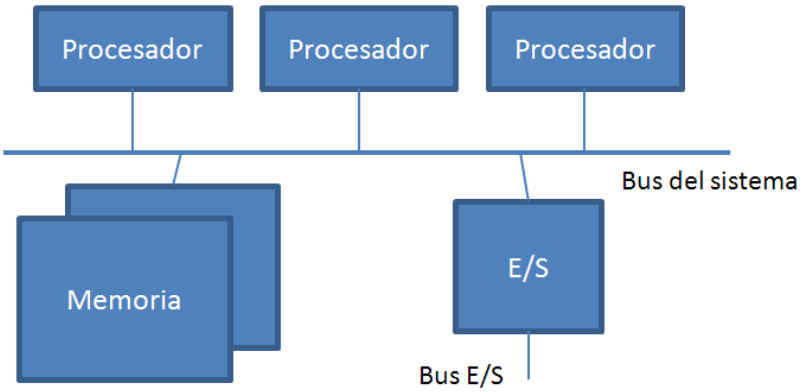


Ilustración 18: arquitectura de un SMP

- *Loosely-coupled* (del inglés, *débilmente asociado*) o sistemas de memoria distribuida, en los cuales el sistema paralelo se construye interconectando a través de una conexión de red local de alta velocidad un número cualquiera de sistemas independientes, en el que cada uno tiene sus propios recursos (procesadores, memoria y dispositivos de E/S) y funcionan bajo el control de sus propias copias del sistema operativo. Cada uno de estos sistemas independientes se denomina nodo. El término "débilmente asociado" hace referencia al hecho de que no existe hardware compartido entre los nodos (excepto por la conexión

de red y, en algunos de ellos, el subsistema de almacenamiento) y por tanto, un nodo no tiene acceso directo a la memoria de otro nodo ni tiene conexiones cercanas con procesadores de esos nodos. A través del software, el conjunto se presenta a los usuarios como un único sistema con una sustancial capacidad de procesamiento (potencialmente, la suma de las capacidades de procesamiento de cada nodo). Los *clusters* y los sistemas *Massively Parallel* (MPP) (del inglés, *masivamente paralelo*) pertenecen a esta categoría.

Estos sistemas son diferentes a los sistemas distribuidos. En un sistema de memoria distribuida:

- Los nodos son homogéneos.
- Los nodos se encuentran físicamente cerca entre ellos.
- Los recursos están, en general, agrupados (por ejemplo, un único sistema de ficheros) lo que hace que cada colección de recursos parezca única hacia sus usuarios (esto es conocido como "Single System Image", del inglés, *Imagen Única del Sistema*).

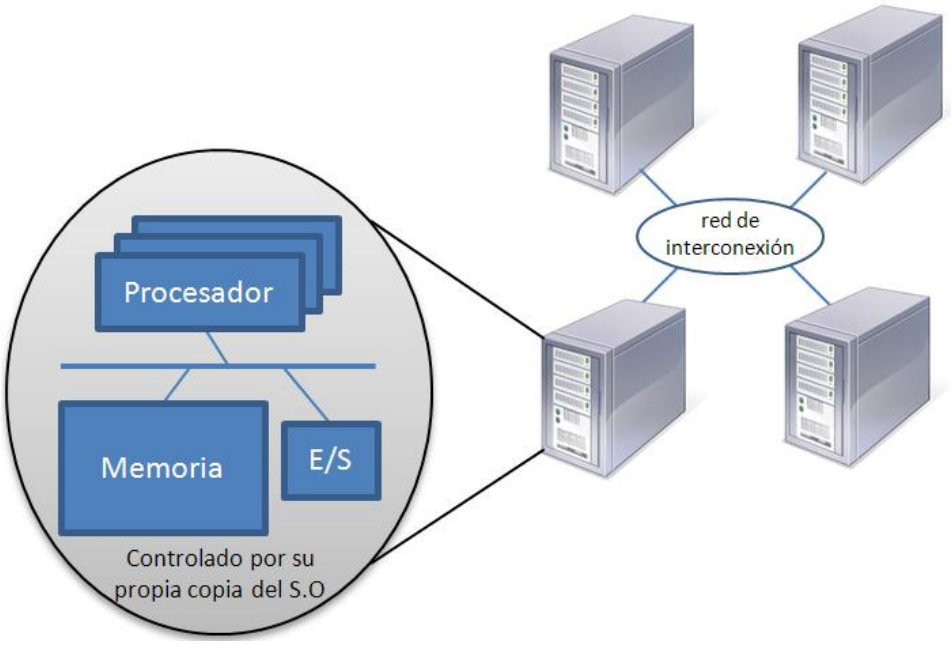


Ilustración 19: arquitectura de un sistema de memoria distribuida

Para que el software pueda beneficiarse de la arquitectura paralela, este debe estructurarse como una colección de unidades que puedan ser ejecutadas en paralelo, es decir, como múltiples procesos. Esto es definido en la arquitectura de comunicación.

Las arquitecturas paralelas pueden entenderse como una extensión de las arquitecturas convencionales que posibilitan la comunicación y la cooperación entre los elementos encargados del procesamiento. En esencia, las arquitecturas paralelas extienden el concepto habitual de arquitectura de computadores añadiendo una arquitectura de comunicación (35).

Las arquitecturas de comunicación definen las operaciones básicas de comunicación y sincronización. La capa más alta de la arquitectura de comunicación es el modelo de programación, que especifica cómo las partes del programa se ejecutan de forma paralela, cómo se comunica la información entre ellas y qué operaciones de sincronización están disponibles para coordinar sus actividades.

2.7.1 Modelo de programación de memoria compartida

La característica principal de este modelo es que los procesos comparten direcciones de memoria. La comunicación ocurre directamente como resultado de instrucciones de acceso a memoria convencionales (*loads* y *stores*) mediante las variables que comparten. La sincronización se realiza a través de mecanismos para el acceso exclusivo a los datos (exclusión mutua y sincronización de eventos).

Puede verse que este modelo de programación se ajusta perfectamente a las arquitecturas de memoria compartida o SMP, puesto que cualquier procesador puede acceder a cualquier posición de memoria y por tanto, la comunicación entre los procesadores y los procesos puede hacerse directamente a través de la memoria que comparten. Sin embargo, los modelos de programación también pueden aplicarse sobre una arquitectura diferente a la que le es "natural". En este caso, el modelo de programación de memoria compartida también se puede aplicar a arquitecturas de memoria distribuida que tienen soporte software de memoria compartida.

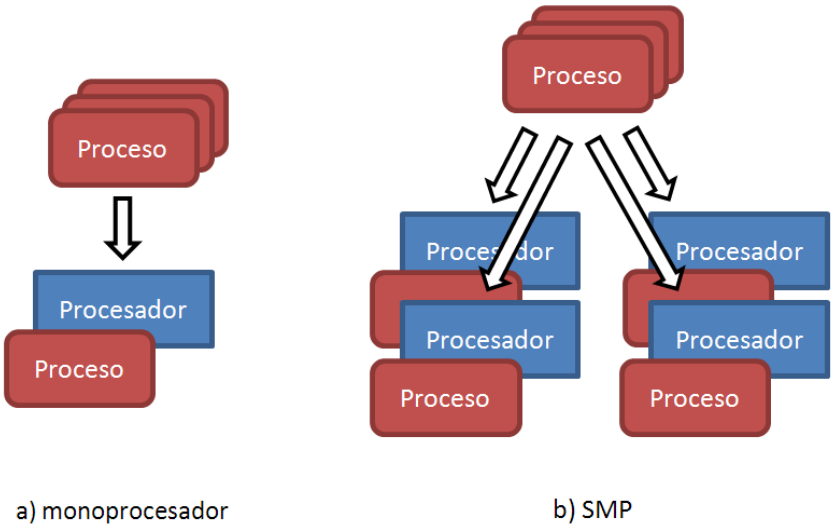


Ilustración 20: modelo de ejecución software para un SMP

2.7.2 Modelo de programación de paso de mensajes

En los sistemas de memoria distribuida, cada nodo soporta un número de procesos. Al no existir compartición explícita de memoria entre nodos, si procesos de diferentes nodos necesitan comunicarse no podrán hacerlo de forma directa a través de la memoria, sino que tendrán que pasarse un mensaje.

Las operaciones de paso de mensajes más comunes a nivel de usuario son *send* (del inglés, *enviar*) y *recieve* (del inglés, *recibir*). En su forma más simple, *send*

especifica un buffer de datos local para ser enviado a un proceso receptor que típicamente se encuentra en un procesador remoto. *Recieve* especifica un proceso emisor y un buffer local de datos en el que los datos transmitidos serán almacenados. Juntos, un *send* y un *recieve* hacen que los datos se transfieran de un proceso a otro logrando un evento de sincronización entre pares y una copia de memoria a memoria, donde cada uno especifica su dirección de memoria local (35).

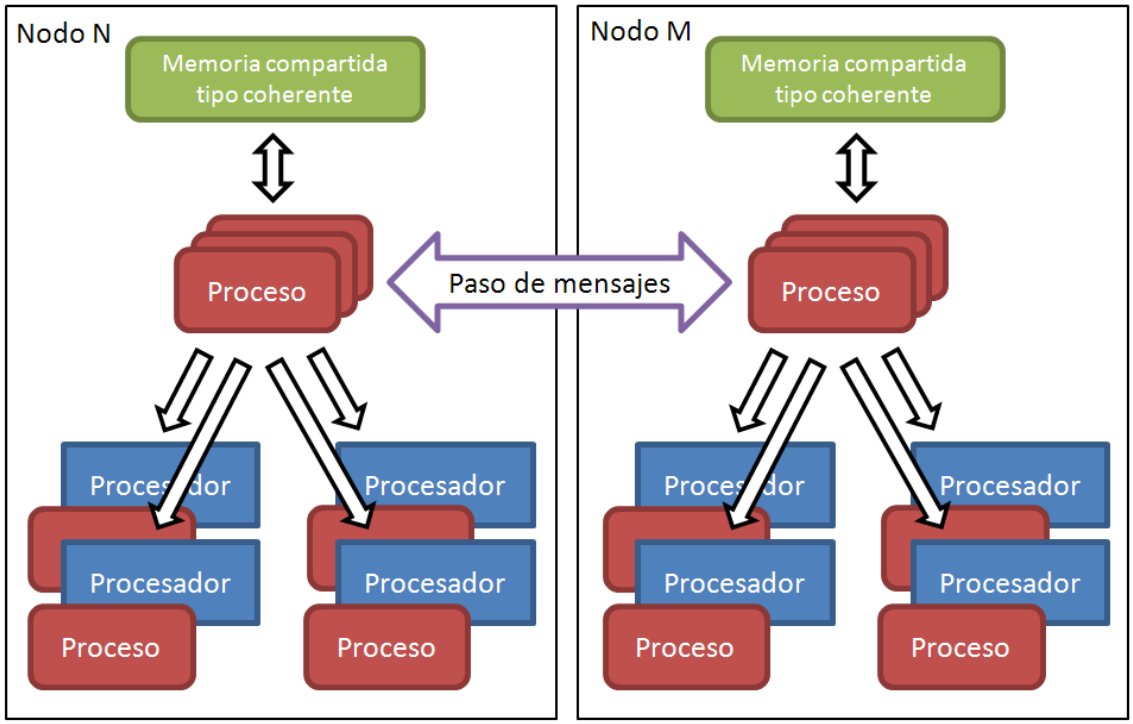


Ilustración 21: modelo de ejecución software para un sistema de memoria distribuida

2.8 Message Passing Interface

Message Passing Interface (MPI) es una especificación (y no una implementación) de una librería interfaz para el paso de mensajes. MPI apunta principalmente al modelo de paso de mensajes de programación paralela, en el cual, los datos se mueven desde el espacio de direcciones de un proceso hacia el de otro proceso a través de operaciones cooperativas en cada proceso (37).

2.8.1 Las metas de MPI

La meta global de MPI es desarrollar un estándar que sea ampliamente usado para desarrollar programas que utilicen paso de mensajes. Como tal, debe ser práctico, portable, eficiente y flexible (37).

Una lista completa de metas que alcanza MPI es (37):

- Diseñar una interfaz de programación de aplicaciones.

- Permitir una comunicación eficiente.
- Permitir implementaciones que puedan ser utilizadas en entornos heterogéneos.
- Permitir enlaces a la interfaz desde diversos lenguajes de programación como C, C++, Fortran-77 y Fortran-95.
- Asumir una interfaz de comunicación confiable: el usuario no necesita tratar con fallos de la comunicación. Dichos fallos son tratados por el subsistema de comunicación subyacente.
- Definir una interfaz que pueda ser implementada en muchas plataformas de vendedores sin cambios significativos en el sistema de comunicación subyacente y el software del sistema.
- La semántica de la interfaz es independiente del lenguaje.
- La interfaz está diseñada para permitir *thread safety*.

2.8.2 Plataformas aptas para una implementación de la interfaz MPI

El atractivo del paradigma de paso de mensajes se debe en parte a su amplia portabilidad. Programas escritos haciendo uso del paso de mensajes pueden ejecutarse en multiprocesadores de memoria distribuida, redes de *workstations* y combinaciones de todos ellos. Además, implementaciones de memoria compartida, incluyendo aquellas para procesadores multinúcleo y arquitecturas híbridas, son posibles. MPI sirve también con arquitecturas que combinen memoria compartida y memoria distribuida y no se verá afectado por incrementos en las velocidades de las redes de comunicación. Por lo tanto, es posible implementar este estándar en una gran variedad de máquinas, incluyendo aquellas que consisten en colecciones de otras máquinas, paralelas o no, conectadas por una red de comunicaciones (38).

3 ANÁLISIS

Con el fin de que un proyecto resulte exitoso, debe tenerse muy presente cual es el fin de este, así como haber profundizado en el dominio sobre el que versa. Estos dos factores son los que generan los requisitos del sistema, que son el comienzo de todo proyecto.

Así pues, para comenzar se presentará una descripción general de la aplicación diseña e implementada en este Proyecto Fin de Carrera, en la cual, ya se establecen los requisitos básicos de la misma, para seguir con una más detallada. Estas descripciones son el resultado del estudio realizado sobre el dominio e incluye las primeras decisiones, restricciones y limitaciones tomadas, todas ellas con el fin de definir distintos requisitos, lo más detallados posible.

A continuación se describirán de manera formal los requisitos de la aplicación, sin olvidar que estos juegan un papel muy importante en el desarrollo del resto del Proyecto Fin de Carrera, puesto que son el punto de referencia para alcanzar de la forma más satisfactoria posible el resultado deseado.

Finalmente mencionar que en el desarrollo del presente Proyecto Fin de Carrera se ha gozado de una gran libertad, por lo que ciertos requerimientos han surgido durante fases más avanzadas del desarrollo del proyecto, así como otros se han visto modificados o han sido adaptados.

3.1 Descripción general

La aplicación a desarrollar recogerá y almacenará en memoria secundaria un alto volumen de datos procedentes de la red social Twitter, los cuales habrán sido capturados de la información aportada por los propios usuarios de la red.

Los datos que interesen a esta aplicación serán aquellos que contengan información útil para localizar geográficamente a un usuario de Twitter. Se estudiará

por tanto la ubicación de un conjunto significativo de usuarios de Twitter relacionados entre sí, partiendo de un usuario semilla y a través de relaciones de tipo *followers* o "seguidores" propias de Twitter. Estos datos son accesibles a través de la REST API de Twitter.

El tratamiento automático de esta gran cantidad de datos debe ser lo más rápido y eficiente posible. Debido a que el tratamiento de la información llevará gran cantidad de tiempo, se optará por la paralelización del proceso. La solución paralelizada se basará en un modelo de paso de mensajes (MPI), utilizando la implementación MPICH2. La elección de este paradigma de programación paralela se debe a la escalabilidad de cómputo que ofrece esta solución. Finalmente, esta aplicación será ejecutada en un entorno cluster, con el fin de obtener un procesamiento en paralelo de los datos en un tiempo razonable.

La aplicación puede entonces dividirse en dos módulos con fines claramente diferenciados: el primero de ellos debe encargarse de recolectar y almacenar los datos obtenidos desde Twitter en memoria secundaria (disco duro); el segundo será el encargado de realizar el procesamiento de los datos almacenados y obtener los resultados tras el procesamiento.

3.1.1 Módulo recolector

El fin de este módulo será la recolección y el almacenaje de los datos. Uno de los requisitos fundamentales es que debe ser capaz de comunicarse con la REST API de Twitter, puesto que los datos solo pueden obtenerse a través de ella.

La API REST funciona con peticiones de tipo GET y POST, por lo que es necesario establecer una comunicación HTTP desde la aplicación hacia la API. Puesto que se desea recolectar la información de la forma más rápida posible y debido a las restricciones que impone el propio API en cuanto al número de peticiones por hora, la autenticación a través de OAuth permite que este límite sea menos restrictivo, por lo que la aplicación necesitará autenticarse. Además, la autenticación permite el acceso a perfiles de usuarios de Twitter que hayan sido bloqueados contra usuarios no autenticados, por lo que gracias a la autenticación se disminuirá el número de respuestas de tipo "no autorizado" obtenidas.

Otra de las consideraciones que debe tener en cuenta esta aplicación es qué datos debe solicitar y recoger. Para localizar geográficamente a los usuarios de Twitter solo serán relevantes datos que contengan información útil para ello, como por ejemplo, ciudades, países o barrios. Más concretamente, se realizará un estudio centrado en las ciudades. Los países son demasiado grandes para ser interesantes, siendo además obvio que dada una ciudad, se conoce inmediatamente a qué país pertenece.

Para comenzar a recoger la información, se parte de un usuario raíz al que se llamará semilla. Una vez recogidos los datos de la semilla, se obtienen los datos de sus *followers*, después los datos de los *followers* de los *followers* y así sucesivamente hasta un número en el nivel de relaciones que es configurable. Dependiendo del tamaño del

grupo que se quiera estudiar, se configurarán ambos parámetros: si se desea obtener un conjunto de datos *muy* grande, se debería utilizar una semilla con una gran cantidad de seguidores y un número alto en el nivel de relaciones a estudiar y viceversa.

La información que se recogerá por cada usuario tratado viene dada en tres puntos diferentes:

1. La información del perfil del usuario, dado que uno de sus campos se corresponde con la ubicación.
2. Los *n* últimos *tweets* publicados por cada usuario. Este valor será también configurable. Los *tweets* son utilizados frecuentemente para establecer el nuevo estado de un usuario, por lo que no será difícil encontrar *tweets* que contengan texto como "Ya estoy de vuelta en Madrid".
3. Los datos de geoposicionamiento asociados a cada *tweet* publicado. Twitter habilita la posibilidad de añadir datos de geolocalización a cada *tweet* que se publica, los cuales establecen una perfecta definición de la ubicación del usuario en el momento de escribir el mensaje.

La información recogida a través de las peticiones a la API va acompañada de otros datos que no serán de utilidad para este caso de estudio. Esta subaplicación no filtrará los datos no útiles, sino que los almacenará en bruto tal y como vienen dados por la respuesta de la API en formato JSON. Todos los datos se almacenarán en ficheros de salida, dónde cada uno de los cuales contendrá un gran número de objetos JSON.

Los datos de los puntos 2 y 3 se obtienen juntos en una misma petición (por cada usuario). Los datos del punto 1 requieren una petición adicional. Así pues, dos tipos de peticiones serán las que compongan los dos ficheros de almacenamiento que se crearán: cada fichero almacenará el mismo tipo de información, lo que se traduce en tener el mismo tipo de objetos JSON en cada fichero. Además, debería haber tantos objetos en cada fichero como peticiones se hayan invocado.

3.1.2 Módulo procesador

El fin de este módulo será el de procesar y tratar todos los datos almacenados en memoria secundaria. Esta parte de la aplicación es la que otorga valor a estos datos, puesto que sus resultados permitirán obtener conclusiones. El requisito fundamental de este módulo es que sea lo más rápido y eficiente posible, debido al gran volumen de datos con el que deberá lidiar.

Con tal objetivo, el desarrollo de una aplicación que trate los datos en paralelo se hace imprescindible. Esta se ha desarrollado haciendo uso del modelo de programación de paso de mensajes y a través de un paradigma de maestro/esclavo, donde el proceso maestro será el encargado de distribuir el trabajo entre los procesos esclavos así como de recoger la información útil que genere cada uno de ellos.

En términos generales y sin entrar en detalles concretos sobre la implementación del paso de mensajes (que se dejará para una explicación más detallada), esta

subaplicación es la encargada de tratar los ficheros con los datos dados en forma de objetos JSON. Dependiendo del objeto almacenado se deberá acceder al campo adecuado para obtener el dato necesario.

En el caso de la información de la ubicación del perfil, se obtiene una cadena sobre la que se busca alguna de las ciudades que la aplicación tiene almacenadas como ciudades de referencia. El problema más común que genera este campo es que es completado por el usuario de Twitter de forma libre, es decir, no existe ningún desplegable que acote sus posibles valores, por lo que es frecuente encontrar que el texto que contiene no se corresponde con una ciudad o sencillamente, que esté vacío.

Para los *tweets*, debe recogerse la cadena formada por cada uno de ellos: es sobre ella donde se buscará información apta para localizar al individuo en cuestión. Dado un *tweet*, no pueden buscarse en él únicamente nombres de ciudades porque se generaría un número elevado de falsos positivos. La aproximación utilizada consiste en emplear patrones en la búsqueda: no sólo se buscará una ciudad sino también una preposición que lo acompañe y que sugiera la idea de estar (o haber estado) localizado en ella. Por ejemplo, en el *tweet* "la capital de Madrid tiene más de tres millones de habitantes", la preposición "de" no indica la posibilidad de que el usuario se encuentre o haya encontrado en Madrid, a diferencia de "estuve en Madrid hace dos días", donde la preposición "en" sí sugiere esa idea.

Es obvio que en cada *tweet* pueden aparecer múltiples patrones y ciudades, por lo que se recogeran todas las que aparezcan en cada uno de ellos (sin repetir dentro del mismo *tweet*).

Para tratar los datos de geoposicionamiento asociados a cada *tweet*, bastará con leer la cadena de caracteres y compararla directamente con las ciudades que tiene almacenada la aplicación como ciudades a buscar. El problema que presenta esta información es que hasta el momento solo está habilitada en Estados Unidos, por lo que solo los *tweets* publicados en este país pueden contener información de geoposicionamiento. Además, en el caso de incluirla, es configurable por el usuario y puede contener información diferente a una ciudad, como un par de coordenadas, un barrio, una calle o un país entre otros.

Las ciudades de referencia que tiene almacenadas la aplicación son leídas al inicio de la ejecución de la misma desde un fichero de entrada que las almacena y que puede ser modificado para añadir o quitar las ciudades según se desee.

El procesamiento de los datos tiene como fin obtener un cómputo de las ciudades que aparecen así como también realizar una comparación entre los resultados arrojados por los diferentes datos de localización y métodos de tratar a los mismos. Cuando todos los objetos JSON de ambos ficheros hayan sido procesados, los procesos esclavo devolverán los resultados al maestro y finalizarán su ejecución, dejando al maestro la tarea de almacenarlos en un fichero de salida.

3.2 Definición de requisitos

Este apartado tiene como objetivo especificar todas las funcionalidades y restricciones asociados a la aplicación a desarrollar.

Los requisitos serán plasmados de modo que se complete por cada uno de ellos una plantilla, compuesta por los siguientes campos:

- Identificador (ID) : cadena identificadora única, asociada a un único requisito al cual hace referencia unívocamente. El ID seguirá la siguiente nomenclatura, R<tipo><número>, donde:
    - Tipo hace referencia a la clasificación del requisito, pudiendo ser este:
        - Requisito de capacidad, C.
        - Requisito de restricción, R.
    - Número, será un conjunto de cifras que comenzarán desde el 1 hasta el que sea necesario para identificar a todos los requisitos.
- Descripción: breve comentario textual que define al requisito.
- Necesidad: campo que indica si el requisito es imprescindible o no lo es. Toma uno de los siguientes valores: "esencial" o "no esencial".
- Prioridad: indica el nivel de preferencia que se le debe dar al requisito durante el desarrollo del sistema. Puede tomar el valor "baja", "media" o "alta".
- Estabilidad: se refiere a la probabilidad de que el requisito se vea alterado en un futuro por el cliente. Toma el valor "sin cambios" o "con cambios".
- Fuente: indica el origen del requisito. En el presente documento, todos los requisitos tienen como fuente al "tutor" o al "alumno".

3.2.1 Capacidad

ID	**RC1**
Descripción	La aplicación debe utilizar como fuente de los datos la red social Twitter, utilizando para ello su API basado en REST
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	**RC2**
Descripción	Los datos a recoger deben contener información susceptible de ser utilizada para localizar geográficamente a un individuo
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC3
Descripción	Los datos recogidos deben tener un formato de fácil tratamiento en términos computacionales
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC4
Descripción	Los datos recogidos deben poder ser almacenados en memoria secundaria
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC5
Descripción	El volumen de los datos recogidos debe ser modificable, permitiendo así recoger muestras del tamaño que se deseé
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Alumno

ID	RC6
Descripción	Se debe poder permitir cambiar el usuario utilizado como semilla
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC7
Descripción	Los datos deberán recogerse de la forma más rápida posible, utilizando para ello la autenticación OAuth que eleva el límite de peticiones por hora a la API
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Alumno

ID	RC8
Descripción	Se deberán minimizar las peticiones de datos cuyo resultado sea susceptible de "no autorizado", haciendo uso de la autenticación OAuth
Necesidad	No esencial
Prioridad	Media
Estabilidad	Sin cambios
Fuente	Alumno

ID	RC9
Descripción	El tipo de información de localización geográfica que se buscará, se corresponderá con ciudades (grano medio)
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC10
Descripción	Las ciudades que serán buscadas por la aplicación deben poder ser variables, siendo posible su configuración a través de un fichero de texto de entrada en formato ".txt"
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC11
Descripción	La aplicación funcionará bajo sistemas operativos Linux
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC12
Descripción	La aplicación funcionará desde la línea de comandos
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Alumno

ID	RC13
Descripción	Los datos deberán ser procesados de forma paralela, haciendo uso de MPI
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

ID	RC14
Descripción	Los resultados generados por el procesamiento deberán ser almacenados en un fichero de texto de salida ".txt" y siguiendo un formato fácilmente legible para una persona
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Tutor

3.2.2 Restricción

ID	RR1
Descripción	Para el correcto funcionamiento de la aplicación, los datos recogidos deben poder ser almacenados en el disco duro, por lo que debe existir suficiente espacio libre
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Sin cambios
Fuente	Alumno

4 DISEÑO E IMPLEMENTACIÓN

En este apartado se presentan las tecnologías utilizadas en el desarrollo de la aplicación así como una explicación a más bajo nivel de la misma, que abarca aspectos de diseño e implementación.

4.1 Herramientas de desarrollo

Este apartado está dedicado a la exposición de las diferentes herramientas de desarrollo que se han utilizado durante el diseño y la implementación de la aplicación. Fundamentalmente, se tratarán las tecnologías software. Ambos módulos han sido implementados utilizando el lenguaje de programación C.

4.1.1 cJSON

El procesamiento de las respuestas en formato JSON se realizará a través de cJSON. cJSON es un analizador sintáctico de JSON para C. Se caracteriza por ser ultraligero, portable, contenido en un único fichero y está simplificado al máximo, haciendo su uso muy sencillo. cJSON está desarrollado por Dave Gamble bajo una licencia de uso MIT(39).

4.1.2 liboauth y libcurl

El trabajo de la autenticación se delegará a la librería liboauth, que implementa OAuth para el lenguaje C.

liboauth es una colección de funciones de POSIX-C que implementan el estándar OAuth Core RFC 5849. liboauth proporciona funcionalidades para omitir y codificar parámetros de acuerdo a la especificación OAuth y ofrece funcionalidad a alto nivel para firmar peticiones o verificar firmas que sigan el protocolo OAuth, así como realizar peticiones HTTP.

liboauth depende de la librería OpenSSL o de NSS (*Mozilla's Network Security Services*), los cuales se utilizan para generar el resumen tipo hash y la firma. Opcionalmente, si se desean emitir las peticiones HTTP firmadas, la librería utilizada es libcurl (40). La aplicación desarrollada utilizará como método de envío el protocolo HTTP y por tanto, hará uso de esta parte adicional basada en libcurl.

libcurl es una librería utilizada para transferencias URL en el lado del cliente, libre y sencilla de usar. Soporta DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, TELNET y TFTP. Además permite, entre otras muchas funcionalidades, certificados SSL, HTTP POST, HTTP PUT, *proxies*, *cookies* y autenticación vía usuario + contraseña (41).

libcurl es altamente portable, se construye y se trabaja con ella del mismo modo en numerosas plataformas, incluyendo Solaris, NetBSD, FreeBSD, OpenBSD, Linux, Windows, OS/2, Mac OS X, DOS y más. Es gratis, *thread-safe*, compatible con IPv6, ampliamente soportada, rápida, documentada, llena de características y es utilizada por grandes compañías (Adobe, Apple, Cisco, Facebook, Google, Panasonic, Siemens, Sony entre otras muchas (42)) y numerosas aplicaciones.

libcurl está basada en C e incluye una API para él, pero existen más de 40 interfaces para otros lenguajes diferentes, realizadas por otros miembros de la comunidad del software libre (43).

4.1.3 MPICH2

MPICH2 es una implementación ampliamente portable y de alto rendimiento del estándar Message Passing Interface (MPI). La metas de MPICH2 son proporcionar una implementación de MPI que soporte eficientemente diferentes plataformas de comunicación y computación incluyendo *commodity clusters* (sistemas de escritorio, sistemas de memoria compartida, arquitecturas multinúcleo), redes de alta velocidad (10 Gigabit Ethernet, Infiniband, Myrinet, Quadrics) y sistemas de computación *high-end* propietarios (Blue Gene, Cray, SiCortex) (44).

4.1.4 MPE

MPE (MPI Processing Environment) trata de proporcionar a los programadores una serie de herramientas que permiten analizar el rendimiento de sus programas basados en cualquier implementación estándar de MPI, como MPICH2 y OpenMPI (45).

MPE utiliza una técnica de análisis *postmortem* vía *logs*, que permite analizar el rendimiento una vez ya ha finalizado la ejecución del programa paralelo. Estas herramientas incluyen un conjunto de librerías, un conjunto de programas y un conjunto herramientas de visualización gráfica.

El primer conjunto de herramientas que debe ser utilizado en un programa MPI es las librerías. Estas proporcionan una serie de rutinas que permiten crear ficheros de

traza o *log*. Estos log pueden crearse de forma manual, insertando llamadas a ciertas funciones de MPE dentro del programa MPI, automáticamente al enlazar con las librerías apropiadas de MPE o combinando los dos métodos. La librería de *logging* de MPE permite generar los *logs* a partir de la ejecución de programas MPI.

El conjunto de programas de MPE incluye conversores de formato de los ficheros de log, utilidades para la impresión de *logs* y visores de *logs*.

El conjunto de herramientas gráficas de MPE incluye 3 programas para la visualización (gráfica) de *logs*. En este conjunto se encuentra Jumpshot-4 (46).

4.1.5 Jumpshot

Jumpshot es una herramienta de visualización gráfica basada en Java que permite analizar el comportamiento *postmortem* de los programas paralelos. Utiliza un fichero de entrada que tiene registrados unos eventos que han sucedido en un momento concreto (un log) (47).

Cuenta con varias versiones, siendo la última Jumpshot-4, la cual proporciona un gran nivel de detalle gracias al uso de *logs* en formato SLOG-2 (48). (Referencia 6: 13 de abril)

4.2 Diseño detallado

Antes de poder desarrollar aplicaciones sobre Twitter, es necesario crear una cuenta de desarrollador y registrar una nueva aplicación.

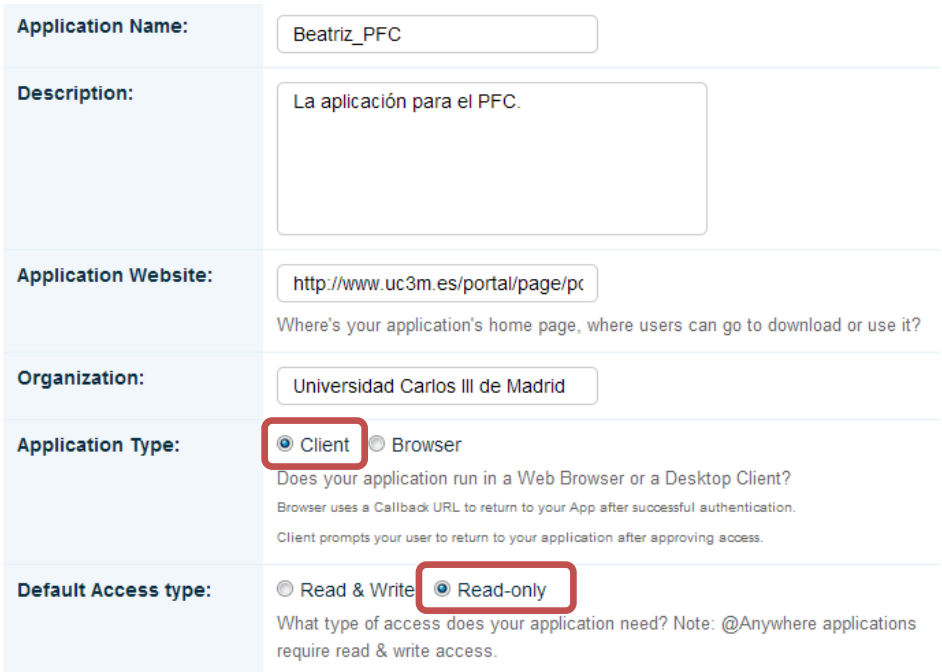


Ilustración 22: elementos configurables en una aplicación de Twitter

Cada aplicación dada de alta tiene unos parámetros configurables que deben establecerse de acuerdo al tipo de aplicación que se va a desarrollar. En el caso de la

aplicación a desarrollar, el tipo de acceso (*Default Access type*) será establecido a leer solo (*Read-only*) puesto que solo se hará uso de peticiones de tipo HTTP GET(49). Además, la aplicación será de tipo cliente de escritorio (*Desktop Client*).

Dar de alta una aplicación es un requisito imprescindible para trabajar con la API de Twitter de una forma más efectiva y rápida, puesto que las llamadas a la API de tipo GET que retornan contenido susceptible de ser privado y todas las llamadas tipo POST solo son permitidas a través de autenticación OAuth. Destáquese también que el límite de las peticiones autenticadas a la REST API es de 350 a la hora en contraposición a las 150 peticiones no autenticadas por hora.

Como parte de la autenticación siguiendo este protocolo, son necesarias unas claves (o *tokens*) para firmar las peticiones (en la ilustración se muestran sombreadas por cuestiones de seguridad)[1].

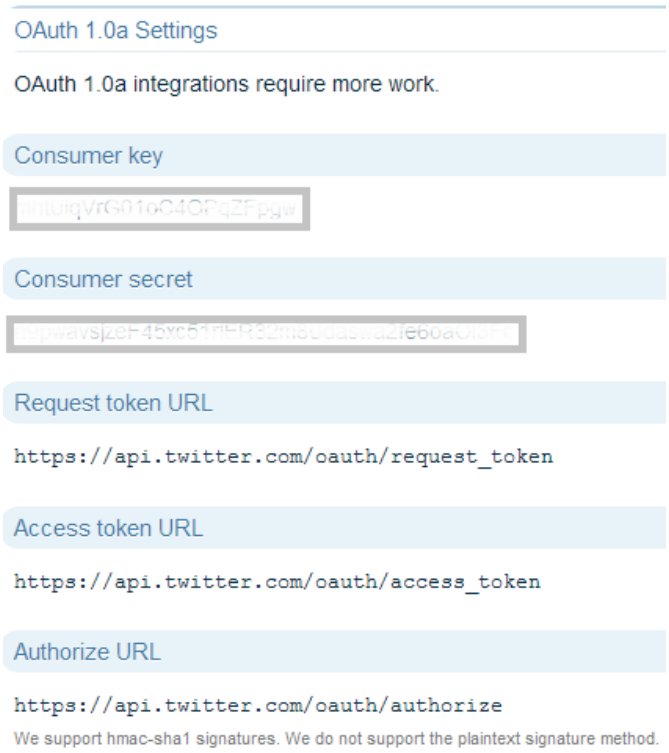


Ilustración 23: consumer key y consumer secret para una aplicación de Twitter

4.2.1 Módulo recolector

Procederá a explicarse el funcionamiento a más bajo nivel del módulo encargado de recoger los datos de Twitter, siguiendo el orden establecido por las tareas que debe desempeñar para llevar a cabo su función.

[1] Twitter aún hace uso de la versión OAuth Core 1.0 Revision A, la cual ha quedado obsoleta por la especificación RFC 5849. Recuérdese que en la nueva versión, el *consumer* (del inglés, *consumidor*) pasa a llamarse *client* (del inglés, *cliente*).

El módulo recolector tiene cinco parámetros de entrada cuyos valores deberán ser dados al ejecutarlo. Estos son, en orden de entrada:

- Nombre del usuario de Twitter utilizado como semilla. El nombre de usuario es utilizado para identificar las cuentas de Twitter, puesto que es único y no puede repetirse, y es solicitado durante el proceso de autenticación en Twitter. También es conocido como "screen name". Un ejemplo sería "BarackObama".

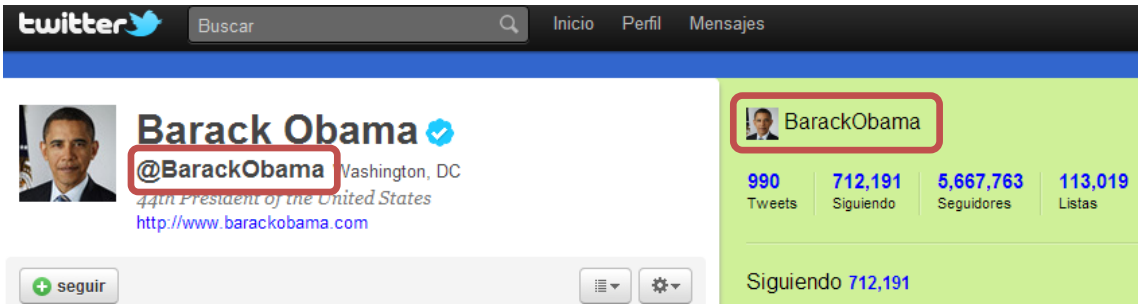


Ilustración 24: nombre de usuario de Barack Obama en Twitter

- Número de los últimos *tweets* a recoger de cada usuario tratado. Este parámetro debe tener un valor de entre 1 y 200.

- Profundidad a tratar en las relaciones de *followers* entre usuarios. Indica el número de generaciones que se estudiarán a partir de la semilla, considerando que la primera profundidad es la que relaciona a un usuario con sus *followers*, la segunda profundidad es la que une a un usuario con los *followers* de sus *followers*, y así sucesivamente. Este número debe ser mayor o igual a 1.

- Nombre de los ficheros de salida en los que almacenar los datos recogidos. Los datos generados por las peticiones se recogen en dos ficheros diferentes: el primero de ellos recogerá la información asociada a la ubicación contenida en el perfil; el segundo de ellos, recogerá los *tweets* y la información de posicionamiento suministrada por Twitter y que va ligada cada *tweet*. Por ejemplo, "salida1.txt" y "salida2.txt".

Una vez se tienen estos datos y antes de poder lanzar las peticiones a la API, es necesario autenticarse. Twitter hace uso del protocolo de autenticación OAuth. Este proceso sigue varios pasos(50) :

- Recuperar un *request token*.
- Solicitar autorización del usuario.
- Intercambiar el *request token* por un *access token*.

Sin embargo, la mayoría de las aplicaciones simples solo necesitan hacer uso de una única cuenta de usuario, puesto que sus casos de uso pertenecen siempre a un único y mismo usuario. Esta es la situación que da en la aplicación que se va a desarrollar por

lo que **no** será necesario implementar el flujo completo que requiere la autenticación OAuth. En lugar de ello, Twitter permite empezar desde el punto en el que se tiene un *token* de acceso (*access token*) para la única cuenta de usuario que va a ser utilizada en la autenticación (que será la del desarrollador) y realizar peticiones firmadas a los recursos de Twitter. Para obtener este *token* de acceso, basta con acceder a la gestión de las aplicaciones de la cuenta de desarrollador (51).

Para realizar las peticiones a la API es necesario componer la URL del recurso de la misma. La REST API tiene un gran número de recursos que pueden solicitarse, obteniéndose muy diversos tipos de información de cada uno de ellos. Del total de la REST API, esta subaplicación utilizará únicamente cuatro recursos diferentes:

1.- User resources: GET users/show (52):

Devuelve información extendida sobre un usuario dado, ya sea a través de su ID o su nombre de usuario. También será devuelto el status (*tweet*) más reciente del autor.

Esta petición permite acceder a la información del perfil de un usuario de Twitter para conocer su valor del campo ubicación. Todas las respuestas que se obtengan de cada petición users/show que se envíe serán almacenadas en uno de los ficheros de salida que creará la aplicación.

La URL base es "http://api.twitter.com/version/users/show.format". Soporta los formatos XML y JSON. Como parámetros requeridos se encuentran:

- user_id: el ID del usuario cuyos resultados se desean. Es útil como desambiguación cuando un ID válido es también un nombre de usuario válido.

- screen_name: el nombre de usuario de aquella persona de la cual se desean obtener datos.


```
{
   "description": "44th President of the United States",
   "verified": true,
   "profile_use_background_image": true,
   "profile_background_color": "3266CD",
   "url": "http://www.barackobama.com",
   "status": {
      "in_reply_to_status_id_str": null,
      "contributors": null,
      "in_reply_to_screen_name": null,
      "in_reply_to_status_id": null,
      "in_reply_to_user_id_str": null,
      "retweet_count": null,
      "created_at": "Fri Nov 19 23:04:53 +0000 2010",
      "geo": null,
      "in_reply_to_user_id": null,
      "source": "003Ca href=\"http://www.hootsuite.com\"
rel=\"nofollow\"003EHootSuite003C/a003E",
      "retweeted": false,
```

```
      "id_str": "5758439967948800",
      "coordinates": null,
      "place": null,
      "truncated": false,
      "id": 5758439967948800,
      "favorited": false,
      "text": "Send a message that you support giving students and service men and women a
path to citizenship through the DREAM Act: http://OFA.BO/kAu5MU."
   },
   "follow_request_sent": false,
   "lang": "en",
   "profile_background_image_url":
"http://a3.twimg.com/profile_background_images/57145311/twitterbamaNEW.jpg",
   "created_at": "Mon Mar 05 22:08:25 +0000 2007",
   "profile_text_color": "000000",
   "location": "Washington, DC",
   "notifications": false,
   "profile_background_tile": false,
   "profile_link_color": "0000ff",
   "id_str": "813286",
   "listed_count": 117825,
   "following": false,
   "followers_count": 5931447,
   "statuses_count": 1110,
   "profile_sidebar_fill_color": "e0ff92",
   "protected": false,
   "show_all_inline_media": false,
   "friends_count": 709741,
   "profile_image_url":
"http://a3.twimg.com/profile_images/784227851/BarackObama_twitter_photo_normal.jpg",
   "name": "Barack Obama",
   "contributors_enabled": false,
   "time_zone": "Eastern Time (US & Canada)",
   "favourites_count": 0,
   "profile_sidebar_border_color": "87bc44",
   "id": 813286,
   "geo_enabled": false,
   "utc_offset": -18000,
   "screen_name": "BarackObama"
}
```


Ilustración 25: ejemplo de respuesta JSON en una llamada a GET users/show

2.- Friends and followers resources: GET followers/ids (53):

Devuelve un array de IDs numéricos de cada usuario que es *follower* de un usuario dado.

Con esta petición, podrán conocerse los *followers* que tiene cualquier usuario de Twitter. A través de ella, se podrá conseguir avanzar en la profundidad de las relaciones. Los resultados obtenidos de estas peticiones son necesarios en la algoritmia de la aplicación. Debe tenerse en consideración que Twitter ofrece la posibilidad a sus usuarios de proteger sus datos, por lo que, para acceder a los followers de un usuario

protegido, el usuario que quiere estos datos debe autenticarse y tener permitido ver al usuario protegido.

La URL base es "http://api.twitter.com/version/followers/ids.format". Soporta los formatos XML y JSON. Los parámetros que necesita de forma obligada son los mismos que la petición GET users/show. Existe un parámetro opcional:

- cursor: hace que el resultado sea dividido en varias páginas que contengan no más de 5.000 IDs cada una. Para comenzar a paginar, el valor de cursor debe establecerse a -1. La respuesta dada por la API contendrá las variables previous_cursor y next_cursor para permitir la navegación entre páginas.

En la aplicación, este parámetro será siempre utilizado en la URL de la petición puesto que está enfocada a utilizar usuarios semilla con un gran número de *followers*.


```
{
   "next_cursor_str": "0",
   "next_cursor": 0,
   "ids": [
      95641375,
      88944547,
      39516096,
      24768718,
      15083331,
      50910571
   ],
   "previous_cursor_str": "0",
   "previous_cursor": 0
}
```


Ilustración 26: ejemplo de respuesta JSON en una llamada a GET followers/ids

3.- Timeline resources: GET statuses/user_timeline (54):

Devuelve los 20 últimos *tweets* publicados por el usuario autenticado. También es posible solicitar los tweets de otro usuario utilizando los parámetros screen_name o user_id. Los tweets de otros usuarios solo serán visibles si no están protegidos o si la solicitud de follower del usuario autenticado fue aceptada por el usuario protegido.

A través de esta petición pueden obtenerse los tweets de un usuario de Twitter, lo que equivaldría a acceder a un perfil de usuario en twitter.com. No todos los usuarios tendrán sus *tweets* disponibles, puesto que Twitter ofrece la posibilidad de protegerlos. Si están protegidos solo serán accesibles si, una vez autenticado, el usuario que desea acceder a los tweets es *follower* del usuario cuyos tweets están protegidos.

Toda la información obtenida a través de estas peticiones será almacenada en un fichero de salida creado por la aplicación, que completará los datos buscados sobre los tweets y sobre los datos de geoposicionamiento que incluye Twitter, puesto que van asociados individualmente a cada *tweet*.

La URL base es "http://api.twitter.com/version/statuses/user_timeline.format". Los formatos que soporta son XML, JSON, ATOM y RSS. Todos sus parámetros son opcionales pero para realizar la petición deseada, será necesario utilizar los siguientes:

- user_id o screen_name, al igual que en el recurso GET users/show.

- count: especifica el número de registros a recuperar. Debe ser menor o igual a 200.

En esta aplicación, se traduce en el número de tweets a recoger por cada usuario tratado.

- trim_user: cuando está establecida a true o a 1, cada *tweet* devuelto incluirá un objeto user que contiene solo el ID numérico del autor del *tweet*.

Los datos contenidos en el objeto user no son relevantes para esta aplicación, por lo que, con el fin de reducir el tamaño de la respuesta y por consiguiente, el tamaño del fichero de salida, se establecerá este parámetro a true para todas las peticiones que se realicen.


```
[
  {
    "in_reply_to_status_id_str": "1384559887257600",
    "contributors": null,
    "in_reply_to_screen_name": "Pe_barrero",
    "in_reply_to_status_id": 1384559887257600,
    "in_reply_to_user_id_str": "39516096",
    "retweet_count": null,
    "created_at": "Sun Nov 07 21:54:07 +0000 2010",
    "geo": null,
    "in_reply_to_user_id": 39516096,
    "source": "003Ca href=\"http://twitter.com\" rel=\"nofollow\"003ETweetie for
Mac003C/a003E",
    "retweeted": false,
    "id_str": "1391974695305216",
    "coordinates": null,
    "place": null,
    "user": {
      "id_str": "56458595",
      "id": 56458595
    },
    "truncated": false,
    "id": 1391974695305216,
    "favorited": false,
    "text": "@Pe_barrero Ok Penny"
  }
]
```


Ilustración 27: ejemplo de respuesta JSON en una llamada GET statuses/user_timeline

4.- Account resources: GET account/rate_limit_status (55):

Devuelve el número de peticiones a la API disponibles antes de que se alcance el límite de la hora actual. Las peticiones a este recurso *no* disminuyen el número de peticiones restantes. Si se proveen credenciales de autenticación, se devuelven las peticiones restantes del usuario autenticado. En otro caso, se devuelve el límite de peticiones para la dirección IP desde la que se están haciendo las mismas.

La URL base es "http://api.twitter.com/version/account/rate_limit_status.format". Los formatos soportados son XML y JSON y no tiene parámetros adicionales.

Este recurso será utilizado para controlar el número de peticiones que se realizan a la API durante toda la ejecución de la aplicación. Es imprescindible conocer cuándo se puede hacer una petición a la API porque Twitter confecciona una lista negra de usuarios e IPs en la que bloquea a aquellos que abusan de la API realizando más peticiones de las que son permitidas por hora.

```
{
  "remaining_hits": 150,
  "reset_time_in_seconds": 1277234708,
  "hourly_limit": 150,
  "reset_time": "Tue Jun 22 19:25:08 +0000 2010"
}
```

Ilustración 28: ejemplo de respuesta JSON en una llamada GET account/rate_limit_status

Las respuestas dadas por la REST API se ofrecen en varios formatos (XML, JSON, RSS y ATOM). Los únicos disponibles para todas las peticiones son XML y JSON, por lo que el resto quedan descartados. El formato JSON resulta más interesante que el XML, ya que es tan sencillo de procesar sintácticamente como él pero es más ligero (no utiliza etiquetas de inicio y fin), lo que se traduce en una reducción del tamaño de los ficheros de salida.

La parte de la aplicación que sabe acceder a un JSON para extraer o añadir datos porque conoce su lenguaje (las reglas sintácticas que definen la creación de un JSON correcto) se denomina *parser* (del inglés, *analizador sintáctico*).

La autenticación vía OAuth no es un proceso que se realice una única vez al comienzo del programa sino que se fundamenta en el hecho de que **cada** petición realizada debe ir firmada. Por tanto, antes de enviar una petición a la API debe ser previamente firmada haciendo uso del algoritmo adecuado. En este proceso se utilizarán (siguiendo la nomenclatura de OAuth Core 1.0 (26)) el *consumer key*, *consumer secret*, *access token key*, *access token secret* (que en este caso son siempre constantes) y la URL base de la petición (en este caso, las URLs de los recursos de la API de Twitter).

El módulo recolector hará uso del *parser* en dos ocasiones:

1. Una vez se ha obtenido una respuesta del recurso GET followers/ids, el módulo recolector necesita obtener cada uno de los IDs que identifican a los *followers* que un usuario de Twitter tiene. El acceso a esta información se hace a través

del *parser* y buscando el *tag* "ids" (como puede verse en la ilustración 26). Los IDs recogidos se almacenarán para poder ser utilizados como parámetros en futuras peticiones a la API, siguiendo la lógica del programa.

2. No todas las respuestas dadas por la API son correctas: estas deben ser analizadas con el *parser*. Se implementará un sistema de reintentos que hacen al módulo más tolerante a fallos, de modo que cuando se tiene una respuesta errónea o no válida, se espera un determinado tiempo (que es configurable) y se vuelve a realizar.

La algoritmia básica del módulo recolector se asemeja a la exploración de los nodos de un árbol. En una representación en forma de grafo de tipo arbóreo, el nodo raíz se corresponde con el usuario semilla que se utiliza como punto de inicio de la recolección de datos. Los *followers* de este usuario semilla formarían el segundo nivel del árbol, resultando un árbol de profundidad dos como este:

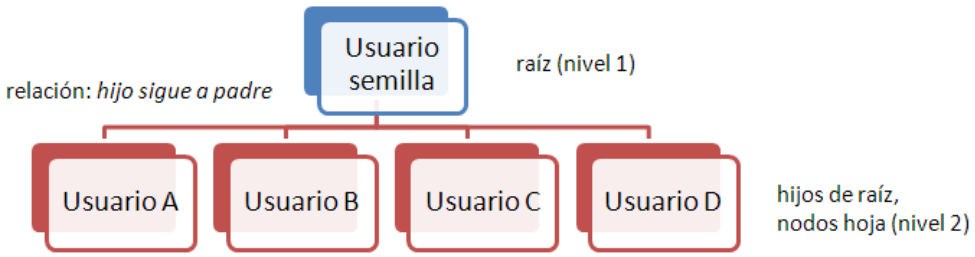


Ilustración 29: árbol de 2 niveles

La relación entre los nodos viene dada por los términos *follower*/*following*: el usuario semilla tiene como seguidor (*follower*) al usuario A, o lo que es lo mismo, el usuario A está siguiendo (*following*) al usuario semilla.

El árbol podría seguir expandiéndose tantos niveles como se desease al ir añadiendo los *followers* de los nodos hoja. Supóngase que se quiere un árbol de 4 niveles, se podría obtener uno como el representado en la Ilustración 29.

En este árbol, el usuario semilla tiene 4 *followers* (A,B,C y D), el usuario A tiene 2 *followers* (E y F) y el usuario E no tiene *followers*. Los usuarios K, L y M pertenecen al nivel 4 quedando como nodos hoja.

Al inicio de la ejecución, la aplicación solo conoce el nodo raíz o semilla, por lo que la generación y exploración del resto de nodos puede hacerse siguiendo métodos de expansión en amplitud o en profundidad.

Un recorrido en amplitud hace que primero se exploren todos los nodos de un nivel antes de descender al siguiente. Esta exploración del nodo implica obtener sus hijos (en nuestro dominio, serán sus *followers*) que deberán ser almacenados en una estructura FIFO hasta que se finalice con la exploración de todos los nodos del nivel

actual. Esta forma de exploración es inherentemente de tipo iterativa, y se realiza de forma 'horizontal'.

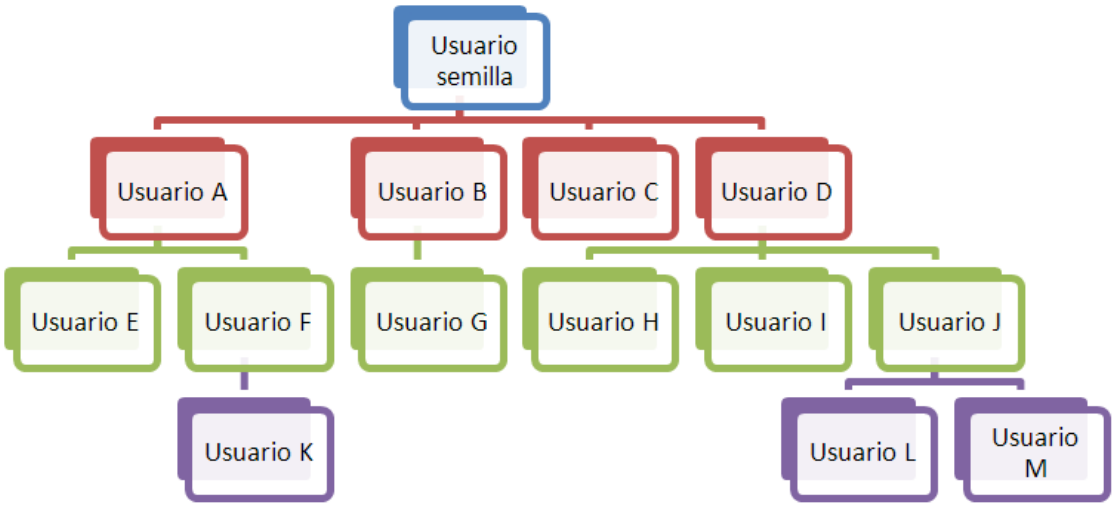


Ilustración 30: árbol de 4 niveles

En un recorrido en profundidad, se debe alcanzar la profundidad máxima de un camino (ya sea porque la profundidad máxima está definida o porque se alcanza un nodo sin descendencia u hoja) desde la raíz hasta el descendiente más lejano de su primer hijo. Se vuelve al segundo hijo, y se desciende de nuevo hasta la profundidad máxima y así sucesivamente. El recorrido en profundidad realiza búsquedas 'verticalmente' y es inherentemente recursivo (aunque también puede implementarse iterativamente con una estructura LIFO).

Volviendo a la representación arbórea, véase que podría darse la circunstancia de que un usuario Z siguiera y fuese seguido por un usuario X, o lo que es lo mismo, ambos usuarios se siguen mutuamente.

Ilustración 31: usuarios que se siguen mutuamente

Esto generaría que al explorar por primera vez al usuario X (nodo rojo) se obtuviese de nuevo al usuario Z (nodo verde) que ya ha sido tratado con anterioridad (nodo azul). Se entraría así en un bucle en el que se obtendría información repetida sobre dos usuarios (el Z y el X), haciendo que los datos obtenidos no fuesen útiles (no hay que localizar varias veces a un mismo individuo).

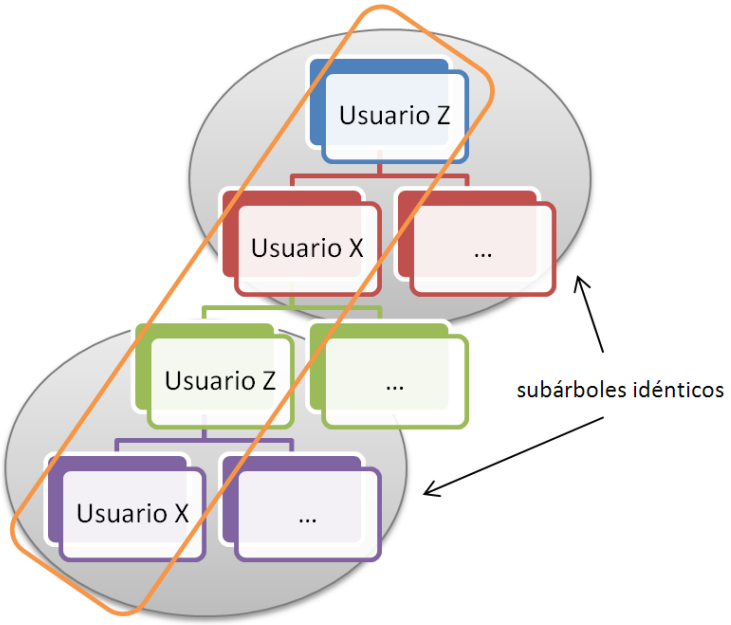


Ilustración 32: bucle encontrado durante la exploración de los nodos

Para que esto no suceda basta con podar la rama actual del árbol en el momento en el que se encuentre un usuario que ya ha sido tratado en pasos anteriores. Nótese además que este suceso solo puede producirse al explorar árboles de niveles mayores que 2: en un árbol de nivel 1, solo existe el nodo raíz; en un árbol de nivel 2, se tiene al nodo raíz y a sus *followers* (y es imposible ser *follower* de un usuario múltiples veces).

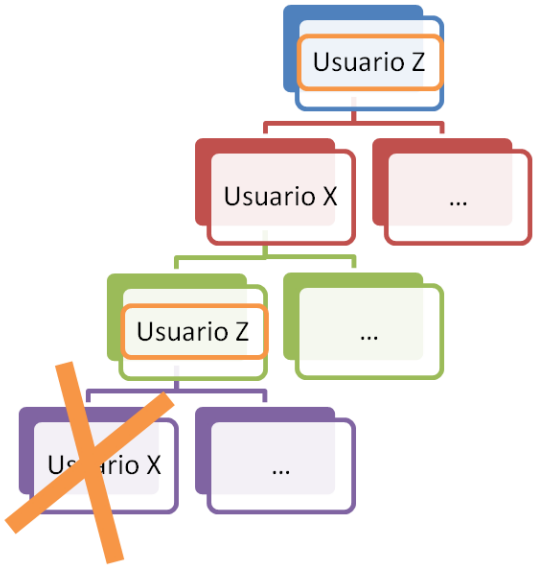


Ilustración 33: poda de una rama durante la exploración de los nodos

Para poder llevar a cabo las podas es necesario llevar un registro de aquellos usuarios que ya han sido tratados. Así pues, cada vez que se tiene un nodo candidato a ser explorado se debe asegurar que no ha sido tratado previamente. Para ello, se mantendrá una lista ordenada de usuarios que ya han sido tratados de tal modo que al llegar a un nuevo posible nodo a expandir, se busca primero en esta lista ordenada. Si

no se encuentra en ella el nodo debe ser expandido y además, añadido a la lista. Si el usuario se encuentra en la lista, el nodo candidato no debe ser explorado sino podado.

La clave de la lista ordenada será el ID del usuario. Este ID de usuario es único por cada cuenta de Twitter y su formato es numérico, haciéndolo idóneo para este propósito.

Además, se implementará también una recogida de datos que permite acotar el máximo número de usuarios cuya información se quiere recoger: dado un usuario que tiene 30 *followers*, puede acotarse la recogida de datos a un valor de por ejemplo, 4 *followers*. Así pues, de cada usuario se recogerían, como máximo, 2 *followers*, ignorando al resto. Este tipo de ejecución será útil cuando se quiera dar prioridad a descender por el árbol en una profundidad elevada puesto que la anchura del árbol que se genera es mucho menor a la que se obtendría en un recorrido completo.

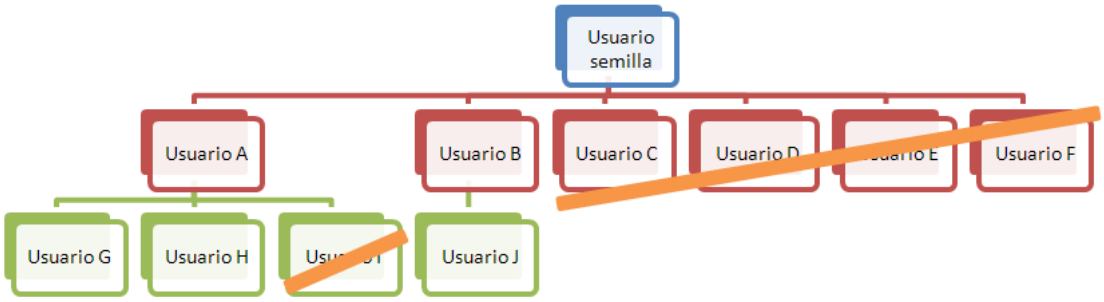


Ilustración 34: recogida parcial de 2 followers por nodo

En una ejecución de tipo completa, el número total de usuarios es complejo de estimar cuando la profundidad es mayor que 1. Para un valor en la profundidad de 1, el número total de información que se recoge se corresponde exactamente con el número de *followers* que tenga el usuario semilla. Para la profundidad 2, habría que añadir al valor anterior la suma del número de *followers* de sus *followers*. Si no se conoce el número de *followers* de cada usuario, las estimaciones no son sencillas de hacer.

En el caso de una ejecución parcial es más sencillo estimar el tamaño de la población que va a estudiarse para una profundidad igual o mayor que 1. En el caso de una ejecución parcial, para una profundidad igual a 1se recogen, como **máximo**, exactamente el número de usuarios que se indicó como parámetro en la entrada. Siguiendo el ejemplo mostrado en la Ilustración 33, este número sería 2. En la profundidad 2, por cada nodo se recoge la información de 2 nuevos usuarios. Como en esta profundidad se exploran 2 nodos, el resultado es de 2^2. El total de usuarios recogidos (cómo **máximo**) para la profundidad 2 es de $2^2 + 2$. En la profundidad 3 se exploran 4 nuevos nodos de los que se toman 2 nuevos usuarios, por lo que se recogen 2^3datos de usuarios nuevos, que sumados a los obtenidos hasta en nivel 2 resultan $2^3 + 2^2 + 2$.

En el caso genérico en el que se recojan n nodos a una profundidad máxima de p, la **cota máxima** de usuarios cuyos datos serán obtenidos se obtiene con la fórmula:

$$\sum_{x=1}^{p} n^x$$

Con todos los elementos explicados se puede desarrollar el algoritmo que recoge los datos a través de solicitudes a la API de Twitter para ser almacenados en memoria secundaria.

El primer paso consiste en conocer el número de llamadas que restan a la API a través del recurso GET account/rate_limit_status (sin olvidar que esta y todas las peticiones son autenticadas). De la respuesta JSON obtenida y a través del *parser*, se accede al *tag* `remaining_hits`, que indica el número restante de llamadas que se pueden hacer a la API antes de que sea necesario esperar a que se reinicie el valor. El resultado se almacenará en una variable, de modo que se evitará hacer una petición a este recurso cada vez que se quiera conocer el valor de llamadas restantes. El programa gana en velocidad (ahorra en llamadas a la API y en los posteriores tratamientos con el *parser* de las respuestas) pero por contra, debe encargarse de actualizar la variable cada vez que realice una petición, restando 1 después de hacer una de ellas. Obviamente, antes de realizar una petición deberá asegurarse de que puede hacerse (la variable tiene un valor mayor que 0). Siempre y cuando el número de llamadas restantes a la API sea mayor que 0, las peticiones se pueden realizar (exceptuando las llamadas al recurso GET account / rate_limit_status que pueden hacerse siempre que se desee porque **no** cuentan de cara al límite). Si el programa se encuentra en una situación en la que debe realizar una llamada pero el valor de las llamadas restantes es 0, debe dormir una hora para que se inicialice el contador. Al despertarse, podrá seguir trabajando desde el punto en el que lo dejó, y deberá restablecer el valor de la variable de llamadas restantes a su límite máximo.

Del nodo raíz, que se corresponde con el usuario semilla, solo se conoce su *screen name* (nombre de usuario) por lo que se debe obtener su ID numérico asociado. Para ello basta con realizar una petición al recurso GET users/show y acceder al *tag* `id`. Conocido el ID se añade a la lista ordenada y se procede a realizar otra petición a GET users/show almacenando el resultado en formato JSON en el primer fichero de salida, y otra petición a GET statuses/user_timeline, cuyo resultado va al segundo fichero de salida. Así se consigue que en el primer fichero se encuentre la información del perfil de todos los usuarios tratados, y en el segundo, sus *tweets*. El orden del fichero no es aleatorio sino que al insertar de esta forma, el JSON almacenado en la línea N del fichero de perfiles pertenece al usuario que escribió los *tweets* almacenados en el JSON número N del fichero de *tweets*. Esta característica será muy a tener en cuenta durante el diseño del módulo procesador.

Una vez que se han recogido los datos del usuario actual (el semilla) hay que empezar con la expansión del nodo a través de sus *followers*. Para conocer los *followers*

de un usuario se realizan tantas peticiones como sea necesario al recurso GET followers/ids. Cada petición devuelve un máximo de 5000 *followers*, por lo que si el usuario actual tiene más, deberán realizarse sendas peticiones hasta que todos sean recogidos.

El recorrido a implementar será de tipo en amplitud, puesto que la recursividad no resulta interesante en este módulo: haría que la ejecución fuese más lenta y además existe riesgo de desbordar la pila ante el gran número de llamadas recursivas que habría que hacer, teniendo que salvar un contexto con mucha información cada vez (por ejemplo, la lista ordenada de usuarios tratados).

Los *followers* obtenidos se añaden a una lista simple y se guarda el número de ellos, consiguiendo así poder realizar el susodicho recorrido en amplitud.

Una vez que se ha terminado con el nodo raíz, se incrementa la profundidad en la relación (que ahora sería igual a 1) y se pasa a explorar a sus hijos, que son los almacenados en la lista simple. La profundidad de la relación mínima admitida por el programa es de 1, lo que supone que el árbol más pequeño que se genera tiene dos niveles (árbol de profundidad 2).

El mismo proceso que se ha realizado con el nodo raíz se realiza ahora con cada usuario que se encuentra en la lista simple, teniendo siempre presente que antes de recoger la información de un nodo hay que conocer si ya ha sido tratado con anterioridad. Para ello, basta con buscarlo en la lista ordenada.

El caso explicado se corresponde con una exploración completa de los nodos. Para realizar una exploración parcial, el cambio radica en que cuando se solicitan los *followers* del nodo actual no se realizan peticiones a la API para pedir todos, sino solo para el número de ellos que se quiera. La información obtenida para estos usuarios se añadiría en la lista ordenada, y se continuaría con este proceso hasta alcanzar la profundidad deseada.

4.2.2 Módulo procesador

Se procede a continuación a explicar con más precisión y detalle el funcionamiento del módulo procesador.

El proceso de paralelización del algoritmo incluye identificar el trabajo que puede ser hecho en paralelo, determinando como se distribuye el trabajo y los datos entre los procesos, gestionando el acceso a datos necesarios, las comunicaciones y las sincronizaciones. Este trabajo incluye computación, acceso a datos y actividad de entrada/salida. El objetivo es obtener un alto rendimiento al mismo tiempo que se mantiene el esfuerzo en la programación y en los requisitos del sistema en niveles bajos. En particular, lo que se desea es obtener una mejora en el rendimiento con respecto al mejor programa secuencial. Esto requiere que se realice una distribución balanceada de trabajo entre los procesadores, reducir la comunicación entre los procesadores (puesto

que es costosa) y mantener bajos los costes asociados a las comunicaciones (en inglés, *overhead*), sincronizaciones y a la gestión general del paralelismo.

Los pasos en el proceso de crear un programa paralelo son cuatro (35):

1. Descomposición del trabajo de computación en tareas.
2. Asignación de las tareas a los procesos.
3. Orquestación de los datos necesarios, comunicaciones y sincronizaciones entre procesos.
4. Mapeo de los procesos a los procesadores.

Juntos, la descomposición y la asignación son llamados particionamiento porque dividen el trabajo hecho por el programa entre los procesos que cooperan.

Para comprender mejor estos pasos, se ofrece una definición de los términos tarea, proceso y procesador (35):

Una tarea es una porción de trabajo hecha por un programa y definida arbitrariamente. Es la unidad de concurrencia menor que el programa paralelo puede explotar. Una tarea individual es ejecutada por un único procesador y la concurrencia entre procesadores es solo alcanzada a través de tareas. Si la cantidad de trabajo de una tarea es pequeña, se denomina una tarea de grano fino. En caso contrario, es de grano grueso. En general, con un grano fino se consigue un buen balanceo de carga pero aumentan los costes en tiempo derivados de la necesidad de realizar más comunicaciones. Un grano grueso desequilibra la carga y limita el paralelismo, pero se reducen los tiempos de las comunicaciones.

Un proceso es una entidad abstracta que realiza tareas. Un programa paralelo está compuesto de múltiples procesos que cooperan, cada uno de las cuales ejecuta un subconjunto de tareas del programa. Las tareas se asignan a los procesos a través de algún mecanismo. Los procesos pueden necesitar comunicarse y sincronizarse con otros para realizar las tareas que tiene asignadas.

Un procesador es un recurso físico de una máquina, en el cual los procesos ejecutan sus tareas. La diferencia entre los procesos y los procesadores es importante en la paralelización: los procesadores son físicos; los procesos son entidades abstractas. Los programas paralelos se escriben en términos de procesos y no de procesadores, por lo que el número de procesos no tiene porque ser igual al número de procesadores disponibles al programa en una ejecución dada. Si hay más procesos que procesadores, estos son multiplexados entre los procesadores disponibles; si hay menos procesos, entonces algunos procesadores permanecerán inactivos.

Volviendo a los pasos del proceso de paralelización, estos son:

La **descomposición** consiste en dividir el programa en una colección de tareas. En general, las tareas se hacen disponibles dinámicamente a medida que el programa se ejecuta, y el número de tareas disponibles en un instante puede variar a lo largo de la

ejecución del programa. El objetivo de esta fase es encontrar suficiente concurrencia para mantener a los procesos ocupados en todo momento (35).

La tarea básica en el módulo que se desarrolla se define como el procesamiento de un par de JSON que están relacionados por pertenecer al mismo usuario de Twitter: uno es el JSON que contiene la información del perfil y el otro contiene los *tweets*.

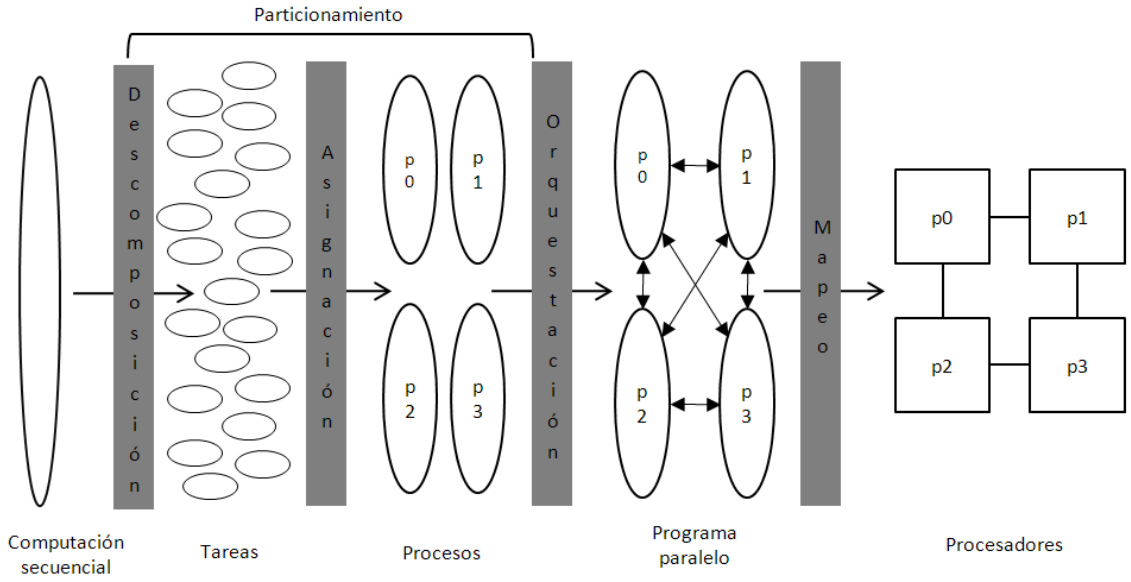


Ilustración 35: pasos del proceso de paralelización

Este procesamiento incluye varias subtareas a realizar en cada uno de los JSON.

Sobre el JSON del perfil, se debe obtener el valor del elemento *location*. El proceso almacenará si se ha obtenido una localización que existe en el fichero de ciudades que se pasa como entrada y cuál es, si existe una localización pero es desconocida (no se encuentra en el fichero de ciudades de entrada), o si el campo *location* está vacío.

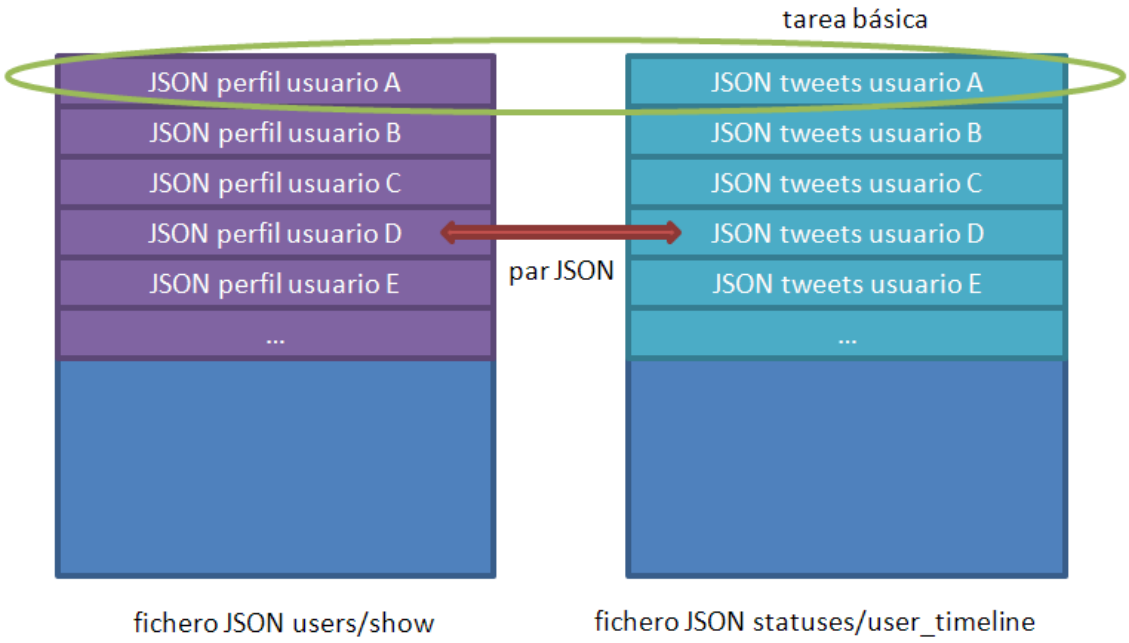


Ilustración 36: par de JSON y tarea básica

Sobre el JSON que contiene los tweets deben obtenerse por un lado el texto de cada *tweet* y por el otro, la información de geolocalización que puede existir en cada uno de ellos.

La información de geolocalización de un *tweet* viene dada en el objeto *place*. Se comprueba que el tipo de información que contiene es una ciudad a través del elemento *place_type* y si lo es, se obtiene su nombre (objeto *place->name*) y se coteja con las ciudades dadas a través del fichero de entrada. Los datos que este tratamiento genera son el número total de JSON tratados, el número total de tweets procesados (que no tiene porque ser igual al número de JSON tratados, ya que en cada JSON puede haber más de un *tweet*), el número de usuarios procesados que no tienen ningún *tweet* publicado, el número total de usuarios que tienen sus tweets protegidos, el número de tweets que no contienen data geo (es decir, aquellos cuyo objeto *place* es vacío) o que contienen datos que no son de ciudades, el número de ciudades válidas encontradas (si aparece varias veces la misma ciudad en los *tweets* de un único usuario, se contabilizan todas ellas), el número de ciudades diferentes por usuario que aparecen en sus *tweets* y el número de ciudades desconocidas (es decir, aquellas que no se encuentran en el fichero de entrada).

```
"place": {
  "name": "San Francisco",
  "country_code": "US",
  "country": "The United States of America",
  "attributes": {

  },
  "url": "http://api.twitter.com/1/geo/id/5a110d312052166f.json",
  "id": "5a110d312052166f",
  "bounding_box": {
```

```
      "coordinates": [
       [
        [
         -122.51368188,
         37.70813196
        ],
        [
         -122.35845384,
         37.70813196
        ],
        [
         -122.35845384,
         37.83245301
        ],
        [
         -122.51368188,
         37.83245301
        ]
       ]
      ],
      "type": "Polygon"
    },
    "full_name": "San Francisco, CA",
    "place_type": "city"
  }
```


Ilustración 37: detalle de objeto place de un JSON obtenido con el recurso statuses/user_timeline

El texto de cada *tweet* se extrae del elemento *text*. Una vez se tiene almacenado en una cadena, se realizan dentro de él diferentes tipos de búsquedas con el objetivo de encontrar algún patrón preposición más ciudad que sugiera la idea de que el usuario ha estado o está en dicha ciudad. Por cuestiones prácticas, ya que el inglés es un lenguaje ampliamente extendido a nivel global, se ha decidido que las búsquedas se realicen en este idioma. Ello afecta a la forma de los patrones: las preposiciones con las que trabaja la aplicación son "at", "in" y "on"; las ciudades del fichero de entrada estarán expresadas en inglés. Cada patrón resulta de la combinación de una preposición con una ciudad (por ejemplo, "at Madrid", "in Los Angeles", "on London").


```
[
  {
    "favorited": false,
    "in_reply_to_status_id_str": null,
    "in_reply_to_status_id": null,
    "contributors": null,
    "in_reply_to_screen_name": null,
    "in_reply_to_user_id_str": null,
    "created_at": "Sun Nov 21 18:46:33 +0000 2010",
    "geo": null,
    "retweet_count": null,
    "source": "003Ca href=\"http://www.hootsuite.com\"
rel=\"nofollow\"003EHootSuite003C/a003E",
    "id_str": "6418200967254016",
```

```
      "coordinates": null,
      "retweeted": false,
      "place": null,
      "user": {
         "id_str": "813286",
         "id": 813286
      },
      "in_reply_to_user_id": null,
      "truncated": false,
      "id": 6418200967254016,
      "text": "It is fundamental to America2019s national security that the Senate to approve the New START treaty this year.  http://OFA.BO/ML64vA"
   },
   {
      "in_reply_to_status_id_str": null,
      "contributors": null,
      "in_reply_to_screen_name": null,
      "in_reply_to_status_id": null,
      "in_reply_to_user_id_str": null,
      "retweet_count": null,
      "created_at": "Fri Nov 19 23:04:53 +0000 2010",
      "geo": null,
      "in_reply_to_user_id": null,
      "source": "003Ca href=\"http://www.hootsuite.com\" rel=\"nofollow\"003EHootSuite003C/a003E",
      "retweeted": false,
      "id_str": "5758439967948800",
      "coordinates": null,
      "place": null,
      "user": {
         "id_str": "813286",
         "id": 813286
      },
      "truncated": false,
      "id": 5758439967948800,
      "favorited": false,
      "text": "Send a message that you support giving students and service men and women a path to citizenship through the DREAM Act: http://OFA.BO/kAu5MU."
   }
]
```


Ilustración 38: elemento text en un JSON obtenido con el recurso statuses/user_timeline

La búsqueda se ejecuta de manera que se recogen todas las localizaciones que se encuentren en el *tweet* utilizando todos los patrones. En un *tweet* cuyo texto fuese "I was in Madrid yesterday but now I'm in Barcelona" (y suponiendo que tanto Madrid como Barcelona se encuentran en el fichero de entrada de ciudades), el proceso de búsqueda encontraría "in Madrid" e "in Barcelona" y sumaría uno al número de encontrados de cada una de las ciudades (uno a Madrid y otro a Barcelona). Si una misma ciudad se encontrase varias dentro del mismo *tweet*, solo se consideraría una vez.

Con este método puede obtenerse la cantidad de usuarios tratados que tienen algún *tweet*, el número de usuarios que no tienen *tweets*, el número de usuarios cuyos *tweets* están protegidos, el número de *tweets* que son null, el número de *tweets* útiles, el

total de búsquedas diferentes realizadas (teniendo en cuenta que un único *tweet* sufre numerosas búsquedas de diferentes patrones, el número de ciudades encontradas en todas las búsquedas (dado un *tweet*) y el número de ciudades diferentes encontradas (dado un *tweet*).

Una vez realizadas todas estas subtareas por cada par de JSON, se compararán los resultados arrojados por los diferentes métodos para encontrar información de localización: para cada par de JSON procesados se cotejarán los resultados obtenidos por todos los métodos entre sí, pudiendo así conocer cuáles encontraron los mismos resultados en términos de ciudades.

Todos estos procedimientos componen la tarea básica que debe desempeñar el programa, aunque existen otras tareas como son la lectura del fichero de ciudades y su almacenamiento en memoria primaria y la creación del fichero de salida que almacena los resultados.

La **asignación** consiste en especificar el mecanismo a través del cual las tareas serán divididas entre los procesos. Los principales objetivos son equilibrar la carga de trabajo de todos los procesadores, reducir las comunicaciones entre procesos y reducir los tiempos de *overhead*.

La asignación se realizará de forma estática de tal modo que el usuario de la aplicación decide cuántos pares de JSON desea enviar a un proceso, haciendo de este modo que el trabajo que ha de procesar tenga un grano del grosor deseado. El grosor menor se correspondería con el envío de un único par de JSON. Este valor es el mismo para todos los procesos, lo cual permite equilibrar la carga al distribuir el trabajo de igual modo entre todos.

La paridad de los JSON es importante, puesto que están relacionados uno a uno y es imprescindible que así sea para aumentar el rendimiento: si es el mismo proceso el que se encarga de tratar los dos JSON asociados a un usuario, se evita que se tengan que hacer envíos de información entre procesos y se reduce el número de comunicaciones. Por tanto, un número impar de JSON nunca será asignado como trabajo a un proceso sino que se podrán enviar un par (2 JSON), dos pares (4 JSON), tres pares (6 JSON), etc.

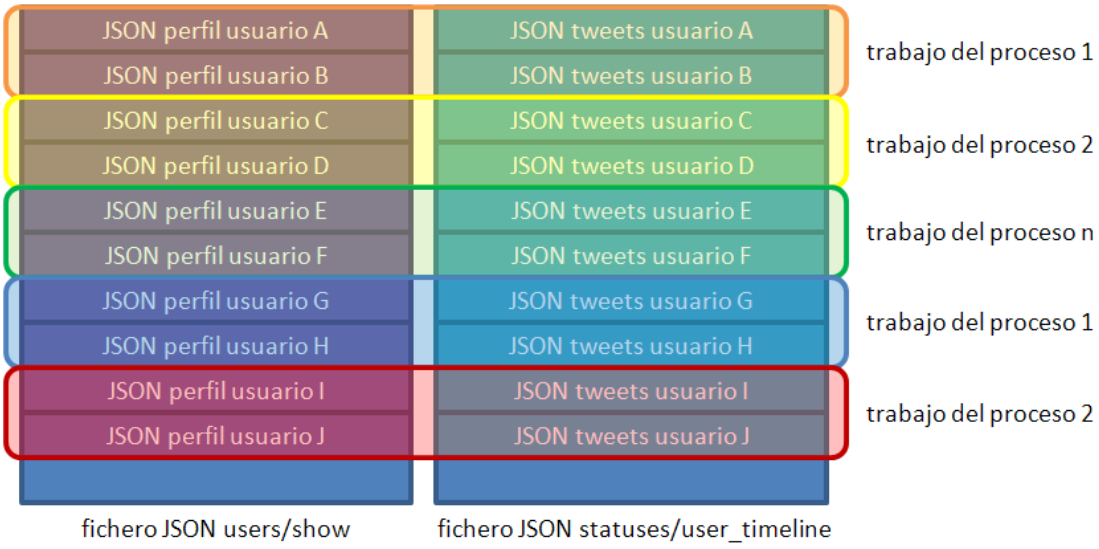


Ilustración 39: asignación de tareas a procesos

La **orquestación** trata de proveer a los procesos mecanismos para nombrar y acceder a los datos, cómo se deben comunicar los procesos y cuando se deben sincronizar. En esta etapa, la que la arquitectura y el modelo de programación juegan un papel muy importante (35).

La utilización de MPI I/O hubiera resultado muy beneficiosa ya que permitiría el acceso concurrente a los datos de los ficheros de entrada por parte de cada proceso de una manera autónoma, mejorando la velocidad de la aplicación y reduciendo las comunicaciones y sincronizaciones al mínimo. Para la obtención de los datos de un fichero, MPI I/O requiere la definición de un Datatype. Los Datatype que se pueden definir en MPI son estructuras de tamaño fijo y conocido, o al menos, capaces de ser descritas a través de desplazamientos de memoria y tamaño de variables. Los JSON que se recogen a través de la API de Twitter *no* son estructuras de tamaño fijo (gran cantidad de sus datos son cadenas de tamaño variable y sin un máximo definido), por lo que el uso de MPI I/O tuvo que ser descartado.

El problema de cómo acceden los procesos a los ficheros que contienen los JSON se resuelve mediante la utilización de un proceso maestro, que es el único capaz de acceder a los ficheros y que se encarga de preparar el trabajo de cada uno del resto de procesos, llamados procesos esclavos. Este proceso será también el encargado de crear el fichero de salida que contiene los resultados finales. Existe por tanto comunicación de tipo *send*/*recieve* entre el proceso maestro y los procesos esclavos.

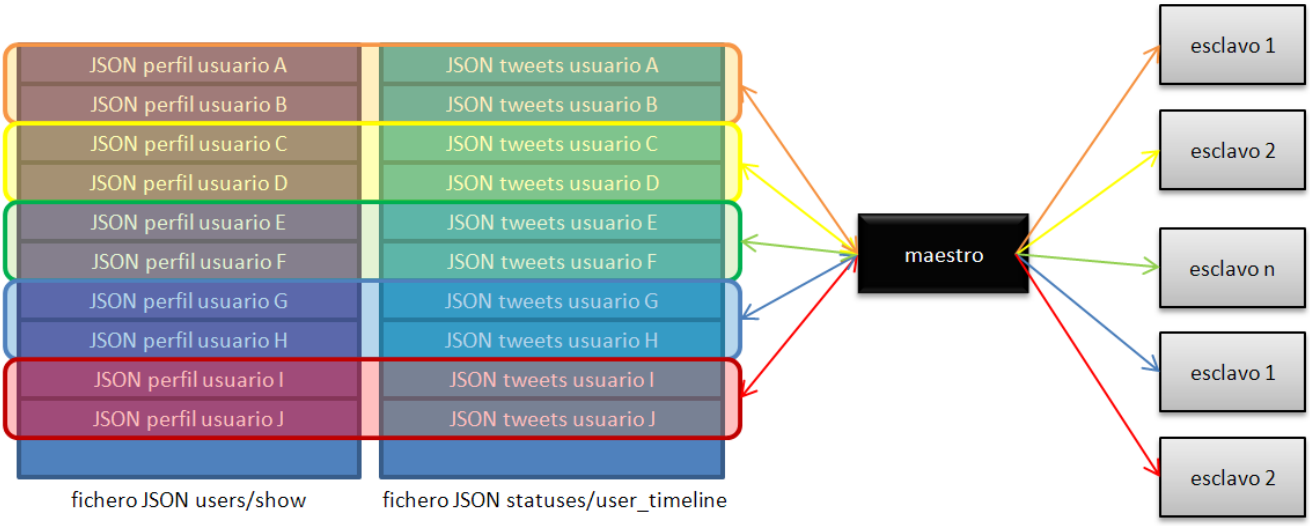


Ilustración 40: proceso maestro y procesos esclavos

Una de las claves será encontrar la mejor estructura para que el proceso maestro comunique a los procesos esclavos cual es su trabajo, de tal modo que se reduzcan las comunicaciones y que no sean de tamaños muy pequeños (puesto que la pérdida por *overhead* haría el proceso poco eficiente). Se explicará en detalle las decisiones tomadas sobre este aspecto en el siguiente apartado.

Puesto que serán varios procesos esclavos los que traten los JSON, los resultados se irán obteniendo parcialmente en cada esclavo. Una vez hayan sido procesados todos, los resultados parciales deben ser transmitidos al proceso maestro a través de operaciones de tipo *reduce*.

El **mapeo** consiste en la asignación de cada proceso a un procesador, es decir, en especificar dónde se ejecutará cada proceso. Esta fase tiene como objetivo minimizar el tiempo total de ejecución colocando procesos relacionados entre sí por comunicarse frecuentemente en un mismo procesador y explotando el principio de localidad de la topología de la red que une las máquinas (35).

En MPI, el número de procesos a utilizar es configurable. Considerando la aplicación desarrollada, esta no podría funcionar sin al menos 2 procesos: un maestro y un esclavo. Sin embargo, esta opción resultaría en una ejecución no paralela (secuencial) y de ella no se obtendría ninguna beneficio en términos de reducción en los tiempos de ejecución. Para que se obtuviese alguna mejora, el número mínimo de procesos debería ser 3.

MPICH permite controlar la asignación de los procesos a los procesadores físicos, permitiendo asignar un número concreto de procesos a un procesador. Si no se especifica, MPI realiza una asignación automática de procesos a procesadores, distribuyéndolos de modo uniforme según un algoritmo *round-robin*. En un entorno ideal, cada proceso se debería ejecutar en un procesador físico individual que emplease toda su potencia de cálculo en ese único proceso.

Esta fase depende en gran medida de la arquitectura en la que se ejecute la aplicación. Se detallará en el apartado de evaluación los detalles sobre la arquitectura utilizada y sobre el mapeo realizado en ella.

Una vez explicado el eje central de este módulo, se hará un breve resumen sobre los parámetros que requiere:

- Los dos ficheros de entrada que contienen los objetos JSON recogidos (y que son la salida generada por el módulo recolector).
- Un fichero de texto en formato .txt que contenga las ciudades a buscar.
- El nombre de un fichero de salida para almacenar los datos recogidos
- La cantidad de trabajo que será enviada en cada envío a cada proceso esclavo para que la trate (en número de pares).

4.3 Detalles de implementación

A partir de la descripción realizada en el apartado anterior, pueden obtenerse los elementos que necesitan ser modelados ya sea en forma de variables o de estructuras de datos. Se comentarán las decisiones más importantes realizadas y todos aquellos aspectos que se consideren relevantes .

Se explicarán por separado cada uno de los módulos para facilitar la lectura y la comprensión.

4.3.1 Módulo recolector

El código del módulo recolector se ha divido en una librería (`gatherlib.h`, `gatherlib.c`) y un programa principal (`gather.c`) (en inglés, *reunir información*) que hace uso de las funciones y estructuras definidas en ella.

En la librería `gatherlib.h` se definen los tipos de datos que permiten almacenar a los usuarios ya tratados y a los usuarios a tratar, así como las funciones que permiten su manejo y varias constantes declaradas vía `#DEFINE`.

Los usuarios ya tratados deben mantenerse en una lista cuyo tamaño irá aumentando durante la ejecución de la aplicación. A la hora de implementar esta lista entra en juego la eficiencia. El requisito que la lista presenta es que no debe mantener usuarios repetidos. Además, será una lista que aumentará de tamaño frecuentemente puesto que cada vez que se procese un usuario nuevo deberá ser añadido. ¿Cómo se sabe si se tiene un usuario nuevo? Ha de buscarse en esta misma lista: si la lista lo contiene es porque ya fue tratado en algún momento anterior de la ejecución y no debe volver a ser procesado porque se duplicaría su información. Existe entonces un conflicto entre inserciones y búsquedas: ambas operaciones van a ser muy frecuentes y las dos tendrán una proporción de ejecuciones muy similar puesto que cada usuario debe ser buscado y si no es encontrado, añadido.

Como ya se vio en el apartado anterior, si la aplicación se ejecuta con un nivel de profundidad en las relaciones igual a 1 (árbol de 2 niveles) se puede asegurar que todos los usuarios tratados no se repetirán. Puesto que las poblaciones de usuarios a tratar son de gran tamaño, cuando la profundidad sea igual a 1 **no** se creará esta lista porque se consumiría demasiada memoria y tiempo de ejecución de forma totalmente innecesaria.

En el caso de que la profundidad sea igual o mayor a 2, sí que debe estar presente para que los resultados no estén afectados por duplicidades en los datos.

La primera opción para su implementación consiste en crear un *array* ordenado sobre el que se pudieran realizar búsquedas binarias o dicotómicas. El coste de acceso a un elemento del *array* es de O(1) y el de la búsqueda binaria es de $O(\log_2 n)$. Suena muy atractivo pero el gran inconveniente está en las inserciones: el *array* crece de tamaño constantemente y para que las búsquedas funcionen, las inserciones tienen que ser ordenadas. Las inserciones provocan que el *array* tenga que ser reservado constantemente, siendo una función muy costosa. No sólo eso, sino que además tiene que hacerse suficiente espacio en memoria para insertar el nuevo elemento ordenadamente, lo que obliga (en la mayoría de los casos) a desplazar una cierta cantidad concreta de elementos una posición a la derecha. Para paliar el coste de las ampliaciones del espacio reservado en memoria, se podría considerar el reservar un gran número de espacios (en vez de uno cada vez) para evitar realizar reservas constantemente.

La segunda opción consistiría en implementar una lista enlazada, que evita el problema de "reallocar" la memoria, permitiendo crear nodos nuevos a medida que se necesitan. Mantener una lista ordenada se convierte ahora en el problema, puesto que los accesos a posiciones concretas de una lista enlazada tienen un coste de O(n) y los algoritmos de búsqueda requieren de múltiples accesos a elementos de la lista, multiplicando así el coste del algoritmo de ordenación. La mejor opción para mantener la lista enlazada sería realizar búsquedas sobre una lista no ordenada: las inserciones tendrían un coste O(1) y las búsquedas O(n).

Otras opciones serían implementar árboles binarios o de tipo B, pero no se considerará esta opción debido a que no se aconseja el estudio de poblaciones de profundidad mayor que 2 de tipo completo por las restricciones existentes en el uso de la API de Twitter.

Por tanto, se tomará la alternativa de la lista enlazada no ordenada que parece ofrecer un buen equilibrio entre eficiencia de uso de la memoria, coste de las inserciones y coste de las búsquedas. La estructura de datos se llama `Uids_llist`, y las funciones definidas sobre ella permiten insertar un elemento al final de la lista, buscar un usuario utilizando su ID, imprimir la lista y destruirla.

El elemento que define a cada usuario de Twitter será su ID. En la documentación ofrecida por Twitter de la API REST no aparece definido siguiendo

ningún tipo de datos o rango, pero en el análisis realizado se ha observado que se trata de un número positivo almacenable en una variable de tipo `int`. Puesto que será un tipo de datos importante, se ha definido un tipo propio llamado `Uid_type`.

```
typedef int Uid_type;
```

La siguiente estructura que se necesita es la que se encarga de recoger a los usuarios a tratar derivados de la exploración en amplitud del árbol de relaciones, haciendo que el algoritmo no sea recursivo. El recorrido en amplitud se apoya en una estructura FIFO y se utilizará la lista simple enlazada ya comentada, añadiendo las nuevas funciones propias para su manejo, como la extracción del primer elemento de la lista o el borrado de la misma.

```
struct node {
    Uid_type uid;
    struct node *next;
};

struct linked_list {
     struct node *first;
     struct node *last;
};

typedef struct linked_list Uids_llist;

int add_llist (Uids_llist *ll, Uid_type data);

Uid_type get_first_node_llist (Uids_llist *ll);

int search_llist (Uids_llist *ll, Uid_type uid);

void print_llist (Uids_llist ll);

void destroy_llist (Uids_llist *ll);
```

En `gatherlib.c` se incluyen todas las funciones para realizar las peticiones a la API REST, recoger las respuestas, tratarlas y almacenarlas en memoria secundaria.

Las peticiones se basan en una URL base que define al recurso sobre la que hay que concatenar ciertos parámetros y valores. Esta aplicación hace uso de cuatro recursos que están definidos como `FOLLOWERSIDS_URL`, `USERSSHOW_URL`, `USERTIMELINE_URL`, `RATELIMIT_URL`. Cada recurso, obviamente, devuelve un tipo de información, tal y como se explicó en el apartado de análisis.

```
//cadenas auxiliares utilizadas para construir URLs

static const char *SCREENNAME = "screen_name=";
static const char *USERID = "user_id=";
static const char *CURSOR = "cursor=";
static const char *COUNT = "count=";
```

```
static const char *TRIMUSER = "trim_user=";
static const char *ASKFORCURSOR = "-1";
static const char *TRUEVALUE = "1";
static const char *STARTPARAM = "?";
static const char *CONCATPARAM = "&";

//URLs que definen los recursos de REST API de Twitter

static const char *FOLLOWERSIDS_URL =
"http://api.twitter.com/1/followers/ids.json";

static const char *USERSSHOW_URL =
"http://api.twitter.com/1/users/show.json";

static const char *USERTIMELINE_URL =
"http://api.twitter.com/1/statuses/user_timeline.json";

static const char *RATELIMIT_URL =
"http://api.twitter.com/1/account/rate_limit_status.json";
```

Existen diversas funciones que se encargan de crear las URL y añadirles los parámetros según la necesidad del programa en el momento preciso de la ejecución. Por ejemplo, para conocer el ID que identifica al usuario raíz (supóngase, BarrackObama) es necesario realizar una petición al recurso `USERSSHOW_URL`. La URL base `USERSSHOW_URL` necesita como parámetro el nombre del usuario cuyo ID se quiere conocer; éste se le concatena utilizando `SCREENNAME` y `STARTPARAM`, generando la URL:

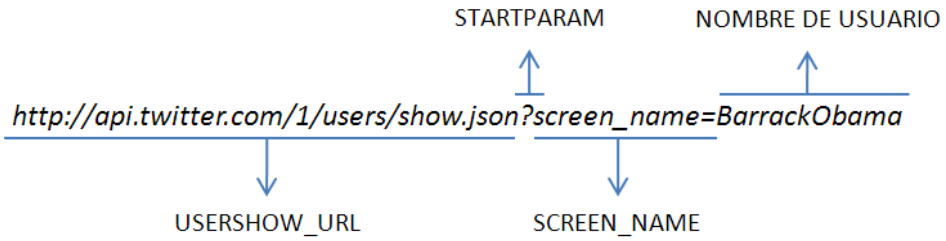


Ilustración 41: URL construida para realizar una petición al recurso users/show

La función encargada de esta tarea se denomina `get_usersshow_url_by_screen_name`.

Las diferentes funciones encargadas de crear las URL con sus parámetros para realizar las peticiones son:

```
char *get_usersshow_url_by_screen_name (const char
     *screen_name);

char *get_usersshow_url_by_user_id(const char *user_id);
```

Ambas construyen una URL para acceder al recurso USERSSHOW_URL, que es el que permite obtener el perfil de un usuario. La primera, añade el parámetro SCREENNAME, por lo que debe ser usada cuando se tiene el nombre del usuario de Twitter. La segunda puede utilizarse cuando se conoce el ID y añade el parámetro USERID.

```
char *get_usertimeline_url_by_user_id(const char *user_id, const
      char *num_status);
```

Esta función se encarga de formar la URL para acceder al recurso USERTIMELINE_URL, que permite obtener la línea de tiempo (los *tweets*) de un usuario dado su ID. Permite controlar el número de *tweets* a recoger, utilizando el parámetro COUNT y requiere también del parámetro USERID. Hace uso de un parámetro opcional, TRIMUSER, que hace que la respuesta devuelta sea más corta, eliminando ciertos datos.

```
char *get_followersids_url_by_user_id (const char *user_id,
      const char *cursor);
```

Esta función compone la URL que permite acceder al recurso que proporciona los *followers* de un usuario dado su ID, utilizando el parámetro USERID. Si el usuario en cuestión tiene más de 5.000 *followers*, la respuesta es dividida en páginas. Es aquí donde interviene el parámetro CURSOR, que permitirá ir realizando sendas peticiones al recurso para conseguir todos los followers de un usuario. En la primera llamada a este recurso para un usuario, el valor de CURSOR debe ser ASKFORCURSOR.

El único recurso que no necesita de parámetros adiciones es el definido por RATELIMIT_URL, por lo que no tiene función para añadirle parámetros y se usa directamente.

Volviendo sobre el ejemplo propuesto, una vez se tiene la URL lista para la petición puede lanzarse, pero antes debe ser firmada utilizando el protocolo OAuth y la librería sobre la que se delega esta tarea, liboauth. La función encargada de ello se llama oauth_sign_url2 y requiere el *consumer key*, el *consumer secret*, el *access token key* y el *access token secret* y utilizará el algoritmo de firma HMAC que es soportado por la API de Twitter.

req_url = oauth_sign_url2 (base_url, NULL, OA_HMAC, NULL, c_key, c_secret, t_key, t_secret);

Ahora sí se puede enviar la petición a la API, de nuevo utilizando la librería liboauth la cual a su vez de apoya en libcurl, enviando la URL por HTTP con la función oauth_http_get2.

```
json_text = oauth_http_get2(req_url, NULL, NULL);
```

La respuesta de esta petición es el JSON deseado (véase la Ilustración 25) contenido en una cadena. Este podrá ser tratado a través del *parser* cJSON para obtener el ID del usuario BarrackObama. El *parser* es muy sencillo de utilizar y permite obtener el valor de los elementos que se deseen dentro de un JSON. En el caso del ejemplo, se quiere obtener el objeto id que es de tipo numérico. Se analiza sintácticamente el texto JSON dado en una cadena utilizando la función `cJSON_Parse`.

```
root = cJSON_Parse(json_text);
```

Y se accede al objeto id para obtener su valor a través de la función `cJSON_GetObjectItem`.

```
user_id = cJSON_GetObjectItem(root,"id")->valueint;
```

No todas las respuestas dadas por la API son válidas: estas pueden contener errores o información no válida como denegaciones de servicio por saturaciones en los servidores o por exceder los límites. Existen dos grandes bloques de errores ante una petición:

1. Error de conexión, pertenece a la máquina sobre la que se trabaja.
2. Error de Twitter, es ajeno a la máquina.

En un error de conexión, es imposible realizar la petición porque no se dispone de salida a Internet. Este se detecta cuando la respuesta obtenida es NULL. El problema pertenece a la máquina sobre la que se está ejecutando la aplicación. La solución adoptada es implementar un reintento después de esperar un tiempo que es configurable.

Dentro de los errores de Twitter también se puede establecer una clasificación. En la documentación para desarrolladores no se explican cuáles son estos errores, por lo que después de realizar varios ensayos, los errores detectados fueron:

1. Errores de servicio o disponibilidad de Twitter: resultan cuando la API está saturada y es incapaz de atender a las peticiones. En este caso, la respuesta no es un JSON sino código HTML. De nuevo, la solución pasa por esperar y reintentar.
2. JSON con *tag* error: cuando se notifica un error, Twitter utiliza el *tag* error para contener una cadena de texto que explica en qué consiste el mismo. De nuevo,

dentro de los errores que se pueden obtener hay errores que obligan a tener que repetir la petición y otros que son errores válidos:

- Errores no válidos: dentro de este grupo se engloban aquellos errores que indican que se ha excedido el límite de peticiones máximas (aún cuando este se controla mediante una variable, y ello es porque Twitter se reserva el derecho a reducir este límite en función del estado de sus servidores), o que el *timestamp* generado durante la firma vía OAuth ha caducado. Para lidiar con estos errores, basta con utilizar de nuevo la técnica de los reintentos.
- Errores válidos: estos JSON contienen el *tag* error pero su información hace referencia a usuarios que han sido suspendidos, usuarios que han borrado sus cuentas o usuarios con tweets protegidos. Estos JSON sí son válidos y deben almacenarse en los ficheros de salida generados, puesto que es imposible obtener la información asociada a estos usuarios de ninguna forma. Si un error de este tipo se produce al solicitar los *followers* de un usuario se produce una poda en este nodo, porque imposibilita el poder seguir recogiendo usuarios a partir de él.
- Errores no clasificables: existen otros tipos de errores, cuyo contenido en el *tag* error no permite conocer a qué se ha debido o cómo podría solventarse y por lo tanto, se consideran como asumibles y de baja importancia.

Para controlar estos errores e implementar los sistemas de reintentos, se han creado las variables y constantes necesarias tanto en `gatherlib.c` como en `gatherlib.h`:

```
//mensajes del tag error no válidos

static const char *RATE_EXCEEDED = "Rate limit exceeded. Clients
may not make more than 350 requests per hour";

static const char *TIMESTAMP = "Timestamp out of bounds";

//variables globales

extern unsigned long long total_ok_api_calls;

extern unsigned long long total_twitter_errors;

extern unsigned long long total_connection_errors;
```

```
//máximo número de reintentos ante fallos de Twitter o conexión

#define MAXRETRIES_CONNECTIONFAIL 5760
#define MAXRETRIES_TWITTERFAIL 5760
```

```
//segundos a esperar para intentar de nuevo

#define WAITRETRY_CONNECTIONFAIL 30
#define WAITRETRY_TWITTERFAIL 30
```

Una respuesta válida obtenida tras realizar una petición, puede servir para recoger un dato de ella y continuar con la ejecución, o para almacenar directamente su resultado en fichero (en concreto, las respuestas dadas por `store_usersshow_data` y `store_usertimeline_data`).

```
int store_usersshow_data (char *user_id, FILE *fp);

int store_usertimeline_data(const char *user_id, FILE *fp);
```

Para ello, basta con usar la llamada `fwrite` sobre los ficheros pasados como parámetro y que deben estar previamente abiertos. Después de cada JSON llevado a fichero se inserta el carácter '\n'. Esto permitirá que la posterior lectura por parte del módulo procesador de cada JSON incluido en cada fichero sea más simple y rápida.

Uno de los aspectos importantes de este módulo es que debe controlar el número de llamadas efectuadas a la API puesto que exceder el límite supone que las peticiones devuelvan resultados erróneos y no solo eso, Twitter puede añadir la cuenta de desarrollador a una lista negra, impidiendo su uso de manera permanente y perdiendo así la validez del *consumer key*, *consumer secret*, *access token key* y *access token secret* imposibilitando la autenticación. Para controlar las llamadas, bastará con utilizar una variable que se encargue de almacenar el número de llamadas restantes y cada vez que se haga una llamada a la API, restarle uno. Por tanto, *toda* función que realice una llamada a la API y que, siguiendo la documentación de Twitter, cuente de cara al límite, deberá gestionar esta variable. Será una variable de tipo global definida como `remaining_calls`. El valor inicial de esta variable se consigue a través de una petición al recurso `RATELIMIT_URL` utilizando la función:

```
void get_remaining_calls(FILE *log);
```

Por ejemplo, la función `get_usersshow_url_by_screen_name` hace una llamada a la API de Twitter, por lo que debe controlar y gestionar la variable global `remaining_calls`.

```
Uid_type get_user_id_from_screen_name (const char *screen_name, FILE
*log)
{
     char *json_text = NULL;
     char *base_url = NULL;
     char *req_url = NULL;

     int nretry_connectionfail, nretry_twitterfail, ok, new_signature;
```

```
      int res = 0;

      Uid_type user_id;

      cJSON *root = NULL, *errortag = NULL;

      //se comprueba si se puede realizar la petición o se debe esperar a
su reinicialización
      while (remaining_calls == 0) res = wait_for_reset(log);

      if (res == (-1)) {
            fprintf(log,"Fatal error on wait_for_reset\n"); fflush(stdout);
            return -1;
      }

      //se crea la URL para hacer la petición
      base_url = get_usersshow_url_by_screen_name(screen_name);

      do {
            nretry_connectionfail = 0;
            nretry_twitterfail = 0;
            new_signature = 0;

            //se obtiene la URL firmada vía OAuth
            req_url = oauth_sign_url2(base_url, NULL, OA_HMAC, NULL, c_key,
c_secret, t_key, t_secret);

            //error en la obtención de la firma por un problema en la url: error
fatal

            if (req_url == NULL) {
                  fprintf(log,"FATAL ERROR: oauth_sign_url2 failed on
get_user_id_from_screen_name function. base_url = %s\n", base_url);
                fflush(log);
                  return -1;
            }

            nretry_connectionfail = 0;


            do {

                  //se envía la petición ya firmada vía GET
                  json_text = oauth_http_get2(req_url, NULL, NULL);

                //error en la petición por un problema de conectividad
                  if (json_text == NULL) {
                      nretry_connectionfail++;
                        wait_for_connection_fail(FUN_GETIDSCRNAM, log);
                        ok = 0;
                  }
               else {
                        //se ha obtenido la respuesta el formato JSON y se debe
analizar sintácticamente con el parser
                        root = cJSON_Parse(json_text);

                     //si el JSON resulta no válido según el parser, es porque
no contiene un JSON sino una cadena HTML, que indica un error interno de Twitter

                        if (root == NULL) {
                          //error de Twitter: reintentar
```

```
                              wait_for_twitter_fail(FUN_GETIDSCRNAM, log);
                              nretry_twitterfail++;
                              free(json_text); json_text = NULL;
                              ok = 0;
                        }

                        else {
                              //el JSON ha sido válido, pero se busca si tiene un
tag de error
                              errortag = cJSON_GetObjectItem(root,"error");

                              if ( (errortag != NULL) && (strcmp(errortag-
>valuestring,RATE_EXCEEDED) == 0) ) {
                                 //error: se excedió el límite de peticiones
                                    wait_for_twitter_fail(FUN_GETIDSCRNAM, log);
                                    nretry_twitterfail++;
                                    free(json_text); json_text = NULL;
                                    cJSON_Delete(root); root = NULL;
                                    ok = 0;
                              }

                              else if ( (errortag != NULL) && (strcmp(errortag-
>valuestring,TIMESTAMP) == 0) ) {

                                    //error: la firma expiró debido a tiempos de
espera: es necesario obtener una nueva
                                   free(req_url); req_url = NULL;
                                    free(json_text); json_text = NULL;
                                    cJSON_Delete(root); root = NULL;
                                    ok = 1;
                                    new_signature = 1;
                              }

                              else ok = 1;
                        }
                  }

            } while ( ((!ok) && (nretry_connectionfail <=
MAXRETRIES_CONNECTIONFAIL)) || ( (!ok) && (nretry_twitterfail <=
MAXRETRIES_TWITTERFAIL)) );

      } while ((new_signature) && (nretry_connectionfail <=
MAXRETRIES_CONNECTIONFAIL) && (nretry_twitterfail <= MAXRETRIES_TWITTERFAIL));

     //si se alcanza el límite de reintentos tras los fallos, se produce un error
      if (nretry_connectionfail > MAXRETRIES_CONNECTIONFAIL) {
      fprintf(log,"ERROR. All the retries failed.\n"); fflush(log);
            if (req_url != NULL) free(req_url);
            return -1;
      }
      else if (nretry_twitterfail > MAXRETRIES_TWITTERFAIL) {
            fprintf(log,"ERROR. All the retries failed.\n"); fflush(log);
            if (req_url != NULL) free(req_url);
            if (json_text != NULL) free(json_text);
            return -1;
      }

      if (root != NULL) cJSON_Delete(root);
      if (base_url != NULL) free(base_url);
      if (req_url != NULL) free(req_url);
```

```
        //se disminuye el número de llamadas a la API restantes
        remaining_calls--;

      //se aumenta el número de llamadas a la API hechas
      total_ok_api_calls++;

        //se obtiene el valor del objeto id de la respuesta dada
        user_id = get_user_id_from_json(json_text,log);
        if (json_text != NULL) free(json_text);

        return user_id;
}
```

Cuando la variable `remaining_calls` sea igual a 0, la aplicación deberá ponerse en espera para que se regenere de nuevo el límite. De esta tarea se encarga:

```
void wait_for_reset(FILE *log);
```

Para que la aplicación espere se utiliza un `sleep` con una duración de 3600 segundos (una hora). Este valor está definido en `SECONDS_FOR_RESET`, dentro de `gatherlib.h`. Al despertar, el proceso restaurará la variable a su valor máximo y continuará desde el punto en el que se puso a dormir.

En esta función pueden verse los sistemas de reintento mencionados en páginas anteriores.

Para los fallos provocados por falta de conectividad, la aplicación esperará dormida durante `WAITRETRY_CONNECTIONFAIL` segundos (que por defecto es 30) para después realizar la petición de nuevo. El número máximo de reintentos se puede modificar cambiando el valor de la constante `MAXRETRIES_CONNECTIONFAIL` (definida en `gatherlib.h`), que por defecto está establecida a 5760. Utilizando los valores predeterminados de ambas constantes, la aplicación espera hasta un máximo de 5760 * 30 = 172800 / 3600 = 48 horas.

Los errores de Twitter se tratan del siguiente modo. Lo primero consiste en realizar un análisis sintáctico con el *parser* cJSON sobre el resultado obtenido en cada llamada a la API. Si el *parser* detecta que el código no es un JSON válido, se considerará como que se ha presentado código HTML que explica un error de servicio o disponibilidad en Twitter. Dado este caso, se desecha la respuesta y la aplicación se pone en espera con el fin de dar a Twitter tiempo para recuperarse del fallo y a continuación, lanzar de nuevo la petición. Para implementar esta espera se definen en `gatherlib.h` las constantes `WAITRETRY_TWITTERFAIL`, (por defecto, 30 segundos) y `MAXRETRIES_TWITTERFAIL` (por defecto, 5760), que hacen una espera máxima de 5760 * 30 = 172800 / 3600 = 48 horas.

Si el análisis sintáctico es satisfactorio, se procede a comprobar si el JSON pertenece a la categoría de errores de Twitter de tipo *tag* error. Para ello, se comprueba

si existe este *tag*, y si existe, si la cadena de texto que contiene se corresponde con un error no asumible comparándola con las variables `RATE_EXCEEDED` y `TIMESTAMP`. Si es igual a alguna de ellas, de nuevo entra en escena el mecanismo de espera y reintento.

Cuando finaliza la ejecución del módulo recolector se crea un fichero llamado *stats.txt* que recoge el tiempo de ejecución del programa en segundos, la fecha y hora de inicio y fin de ejecución, el número total de llamadas a la API realizadas, el número de errores que Twitter ha arrojado al hacer una petición, el número de errores en la red y el número de perfiles y líneas de tiempo almacenados en los respectivos ficheros de salida.

Siguiendo las mismas pautas explicadas, puede codificarse el resto de funcionalidades y haciendo uso de ellas, el programa principal. Este se encuentra en `recolector.c` y empleando de las funcionalidades descritas, es capaz de recoger los perfiles y líneas de tiempo de tantos usuarios como son necesarios, sin duplicar la información de ninguno de ellos y sin sobrepasar los límites en el número de peticiones a la API. Se presenta en forma de pseudocódigo su implementación. El código presentado es una versión simplificada del desarrollado en la que no se ha incluido cómo se obtienen los resultados del fichero *stats.txt*, para facilitar así su comprensión.

```
profundidad_actual = 0;

profundidad_total = N; //se obtiene a través de un parámetro de entrada

id = obtener_id_desde_nombre_usuario(usuario_semilla);

//no se recogerán los datos de la semilla, en caso de que esta apareciese de
nuevo

si (profundidad_total <> 1)
      añadir_lista_tratados(lista_tratados, id);

nfol_añadidos = obtener_followers(id, lista_followers);

profundidad_actual++;

repetir {

      desde 0 hasta nfol_añadidos hacer {

            id = obtener_primer_nodo(lista_followers);

            si (profundidad total == 1) {

           //no se gestiona lista de tratados

                  almacenar_perfil(id);

                  almacenar_lineadetiempo(id, ntweets);

            }
            si no {

                  //sí se gestiona lista de tratados
```

```
                encontrado = buscar(id, lista_tratados);
                si no encontrado {
                  almacenar_perfil(id);
                  almacenar_lineadetiempo(id, ntweets);
                  si (profundidad_actual < profundidad_total) {
                        nfol = obtener_followers(id, lista_followers);
                        fol_sig_prof = fol_sig_prof + nfollowers;
                  } //end si
               } //end si no
            } //end si no
      } //end desde
      profundidad_actual++;
      nfol_añadidos = fol_sig_prof;
} mientras que (profundidad_actual <= profundidad_total);
```

Este módulo genera también el fichero "log.txt" que recoge información útil para ver cómo se ha ejecutado el programa y cuándo se producen fallos, por lo que es interesante en caso de analizar la traza de un programa o en tareas de depuración.

4.3.2 Módulo procesador

El pilar fundamental de este módulo consiste en desarrollar el patrón de comunicación entre el proceso maestro y los procesos esclavos. El esqueleto básico del programa principal (`processor.c`) sería:

```
si soy maestro {  //proceso maestro
     //en este punto, todos los procesos esclavos están libres
      para cada proceso esclavo {
            enviar(TRABAJAR, esclavo);
            trabajo = obtener_trabajo();
            enviar(trabajo, esclavo);
      }
      mientras haya trabajo {
            recibir(estoy_libre, esclavo);
            enviar(TRABAJAR, esclavo);
```

```
            trabajo = obtener_trabajo();

            enviar(trabajo, esclavo);

      }

      //no hay más trabajo, recolectar los datos de cada esclavo

      para cada proceso esclavo {
            enviar(ENVIAR_RESULTADOS, esclavo);
      }

      //los resultados parciales de cada esclavo van a totalresultado

      reducir(misresultados, totalresultado, maestro);

      //se finaliza a los procesos esclavos

      para cada proceso esclavo {
            enviar(FIN, esclavo);
      }

}
si no { //proceso esclavo

      mientras no fin {

            recibir(mensaje, maestro);

            si (mensaje == TRABAJAR) {

                  recibir(mitrabajo, esclavo);

                  misresultados = trabajar(mitrabajo);

                  enviar(esclavo_libre, maestro);

            }

            si (mensaje == ENVIAR_RESULTADOS) {

                  reducir(misresultados, totalresultado, maestro);
            }

            si (mensaje == FIN) {

                  fin = true;

            }
      }
}
```

Básicamente, el proceso maestro prepara una pieza de trabajo que envía a un proceso esclavo que está libre para trabajar, a través de un proceso de envío y recepción de mensajes. Cuando el maestro encuentra que no hay más trabajo, solicita a los esclavos que envíen los datos parciales que cada uno de ellos ha recolectado con reducir. Cuando esto ha terminado, se procede a finalizar a cada esclavo, usando de nuevo enviar y recibir.

El mecanismo básico de comunicación es la transmisión de datos entre un par de procesos: uno envía y el otro recibe. Esto se conoce como una comunicación punto a punto.

La operación usada para realizar los envíos se corresponde con:

```
MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm)
```

Esta función envía el mensaje cuya dirección inicial es `buf`, de tipo `datatype` y que tiene `count` elementos, al proceso `dest`.

La operación de envío debe utilizarse conjuntamente con la de recibir:

```
MPI_Recv (void *buf, int count, MPI_Datatype datatype, int tag,
          MPI_Comm comm, MPI_Status *status)
```

Recibe el mensaje en `buf`, que es del tipo `datatype` y tiene `count` elementos.

Ambas operaciones son de tipo bloqueante: el proceso que realiza el envío permanece bloqueado hasta que la comunicación ha terminado y el receptor tiene el mensaje.

Puede verse como ambas funciones incluyen el parámetro `tag`. Esta "coletilla" permite seleccionar los mensajes en el lado del receptor: este puede filtrar los mensajes para recibir solo aquellos que tengan un determinado valor para `tag`, o puede utilizar un comodín para recibir mensajes con cualquier `tag`.

Existe otro tipo de comunicaciones, las colectivas, donde los datos se transmiten a lo largo de todos los procesos en un grupo especificado. La operación reducir es de este tipo. Reducir efectúa una operación de reducción con los datos de cada procesador, dejando el resultado en uno de ellos.

```
MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
           datatype, MPI_Op op, int root, MPI_Comm comm)
```

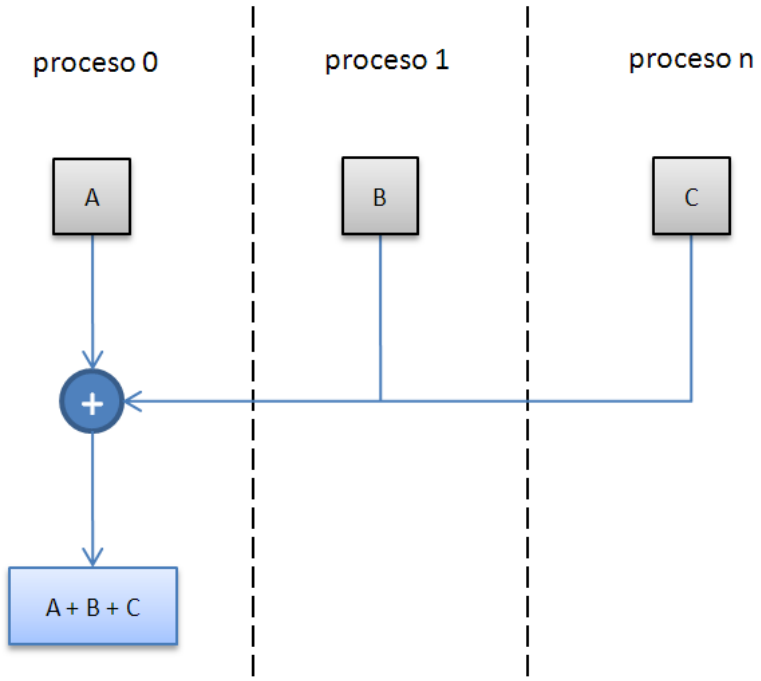


Ilustración 42: MPI_Reduce con operación suma y proceso 0 como root

`sendbuf` es el dato que se envía y `recvbuf` es el destino sobre el que se deposita el valor de la reducción. Ambos deben tener el mismo número de elementos `count` y ser del mismo tipo `datatype`. `root` es el proceso que obtiene el resultado y `op` la operación que se realiza sobre los datos (por ejemplo, suma, multiplicación, media, mayor...).

Los datos que se envían a través de cualquier tipo de comunicación consisten en un conjunto de entradas sucesivas del tipo indicado por `datatype`. Una comparación sería la de un *array*. Por tanto, se pueden enviar conjuntos de enteros y de caracteres, pero no estructuras.

Una vez conocido cómo hacer los envíos, la meta es realizar el menor número de ellos. El maestro conocerá el tamaño en pares del trabajo que debe asignar a cada esclavo, puesto que su valor será un parámetro de entrada de la subaplicación. Conocido este dato, accederá a cada fichero recogiendo tantos JSON como necesite de cada uno. Póngase un ejemplo. Si el valor de la cantidad de pares de JSON que debe recoger es 2, el proceso maestro deberá obtener 2 JSON del fichero de perfiles y otros 2 JSON del fichero de *tweets*. Todos ellos formarán una pieza de trabajo, definida en la librería `processinglib.h`, cuyas funciones están desarrolladas en `processinglib.c`.

El maestro calcula y almacena la longitud de cada JSON individual (4 longitudes en total) y concatena los JSON de forma ordenada en una única cadena. El conjunto de esos datos forma un trabajo que debe ser enviado a un esclavo.

Conocidos sus tamaños individuales, cualquier proceso podrá ahora extraer un JSON de la cadena teniendo en cuenta que los JSON del perfil se encuentran en la primera mitad de la cadena, y los JSON de los *tweets*, en la segunda.

Así, el proceso maestro podrá enviar un trabajo a un proceso hijo utilizando únicamente 2 envíos: uno para la cadena y otro para el *array* de tamaños.

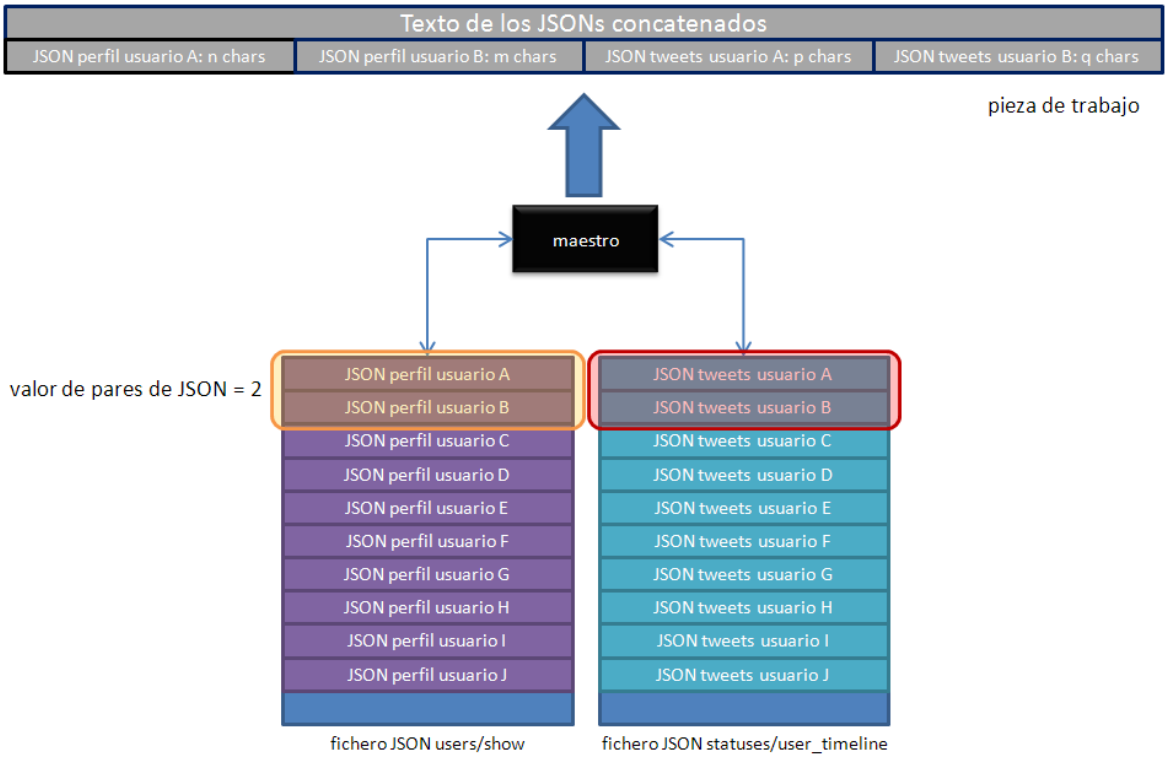


Ilustración 43: pieza de trabajo de tamaño 2

Utilizando este método, se consigue un equilibrio entre el número de comunicaciones que se establecen, que siempre debe ser el menor posible, y el volumen de la información transmitida en las mismas, de forma que se reduzca al máximo el tiempo perdido en sincronizaciones y se obtengan reducciones de tiempo significativas en la ejecución paralela. Además, este método permite también modificar el valor del grano (el tamaño del trabajo) al valor que deseé el usuario.

El trabajo de los procesos esclavos consiste en recoger ciertos datos sobre cada uno de los JSON para localizar al usuario en alguna ciudad. Se explican a continuación los diferentes procesamientos que se realizan y la información que se obtiene de cada uno de ellos.

Por cada JSON de perfil, el proceso esclavo accederá al objeto *location* (utilizando el *parser* cJSON) y realizará una búsqueda utilizando las ciudades incluidas en el fichero de entrada. Si se encuentra alguna coincidencia se toma constancia de ello, almacenando qué ciudad fue la encontrada. Si el contenido del objeto es NULL o si no encuentra ninguna ciudad (la localización es desconocida) son datos que también se

almacenarán. La suma de los tres datos proporcionará el número total de etiquetas *location* procesadas, que equivale al número de usuarios tratados. Todos los datos se recogen una estructura llamada `Statistics_location` y la función que codifica esta tarea es `process_location`.

Por cada JSON obtenido al solicitar la línea temporal de un usuario, se accede al texto de cada *tweet* (objeto *text)* y a las etiquetas de geolocalización que cada *tweet* puede tener (objeto *place).*

Sobre cada objeto *text* se buscan todas las ciudades que contiene, sin contabilizar varias veces aquellas que aparezcan múltiples veces en el mismo *tweet* . En concreto, las búsquedas se realizan utilizando patrones, que resultan de concatenar cada preposición con cada una de las ciudades contenidas en el fichero de entrada. Las preposiciones inglesas utilizadas son "at", "in" y "on". El número total de patrones viene determinado por el nº de preposiciones * el nº de ciudades a buscar. A través de este método de búsqueda pueden recolectarse datos como el número total de *tweets* procesados y el número de *tweets* nulos (que se almacenan en `Statistics_tweets)` y el número de coincidencias con patrones así como sobre qué ciudades se han dado (guardados en `Statistics_status)`. Esta búsqueda está programada en la función `process_status`.

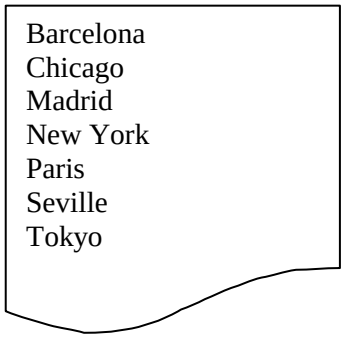
Cada *tweet* puede tener asociada una etiqueta geo. Este será otro objeto a partir del cual obtener datos susceptibles de contener una localización. De nuevo, a través del *parser* cJSON se accede al objeto place y se comprueba que no es nulo; si se cumple, se accede a *place_type* y se verifica si su valor es "city"; si lo es, se accede al objeto *name*, y se recoge el nombre de la ciudad que se cotejará con las ciudades a buscar. No se debe acceder al objeto *geo* , ya que este solo indica si la opción "twittea con tu ubicación" está habilitada o no. Los datos que se generan de este procesamiento son el número total de tweets que tenían etiqueta geo con valor "city", cuantas ciudades fueron encontradas, cúales fueron y cuales resultaron desconocidas por no aparecer en el fichero de ciudades. La función encargada de este procesamiento es `process_geo` que almacena en la estructura `Statistics_status` los datos.

Existe además una estructura (`Statistics_users_errors`) que permite recoger datos de tipo global y datos de error como son, el número total de usuarios procesados, el número de usuarios suspendidos, el de usuarios cuyas cuentas han sido borradas, usuarios que arrojan errores sin especificar en la información de sus perfiles, el número total de *tweets* recogidos, el número de usuarios que tienen protegidos sus *tweets*, el número de usuarios que no han publicado ningún *tweet*, el número de errores indefinidos encontrados en los *tweets*.

No solo es interesante conocer los resultados individuales de cada método de búsqueda, sino que realizar una comparación entre ellos por cada usuario procesado permitiría conocer qué métodos obtienen los mismos resultados. Para ello, se ha añadido una función encargada de comparar uno a uno los distintos resultados

generados por cada método (que son tres en total) llamada `compare_results` y que almacena sus resultados en `Statistics_comparision`. Esta función procesa ciertos datos generados durante el procesamiento por parte de los procesos esclavos (que principalmente, son referencias a los IDs asociados a cada ciudad) y utilizándolos como entradas calcula cuantos usuarios han sido procesados en total, de cuantos se han obtenido datos para cada uno de los tres métodos de recogida de información (*location*, *geo* y *status*), y para cuantos de ellos coincide la ciudad encontrada según diferentes métodos (*location vs geo*, *location vs status* y *geo vs status*).

Las ciudades que deben ser buscadas se encuentra en un fichero de texto que debe ser pasado como parámetro de entrada al módulo procesador. Este fichero de texto, cuya extensión será "txt", contendrá en cada línea una ciudad (cada ciudad separada por el carácter '\n'), estará escrita en inglés y no podrá superar los `MAXLENCITY` caracteres de longitud.



Cada ciudad será almacenada en una estructura `City_info` compuesta por dos campos: un entero sin signo que será el ID de la ciudad (de tipo `Location_id`) y una cadena que almacenará el nombre. El fichero será leído desde el principio asignando el ID 0 a la primera ciudad leída, el 1 a la segunda y así sucesivamente hasta alcanzar el fin del fichero.

Todas las ciudades estarán almacenadas en un *array* en memoria principal donde su posición en el mismo se corresponderá con el valor de su ID. Todos los procesos (incluido el maestro) deben tener acceso a estos datos, por lo que se tienen dos opciones: que cada uno de ellos lea el fichero y cree su propio *array* de ciudades, o que un único proceso lo lea y lo distribuya a los demás mediante una operación de comunicación colectiva de tipo *broadcast*. Ya que esta información debe estar disponible antes de poder hacer ninguna otra tarea (los procesos esclavos no pueden trabajar si no conocen qué ciudades deben buscar), se realizará la lectura por todos los procesos y cada uno de ellos creará su propio *array*, sin necesidad de utilizar ninguna comunicación *broadcast*.

Finalmente, será el proceso maestro el encargado de almacenar en el fichero cuyo nombre viene dado por uno de los parámetros de entrada los resultados obtenidos del procesamiento de la población de usuarios de Twitter.

Dentro del código MPI que contiene el fichero `processing.c` se ha embebido código MPE para generar un log que permitirá estudiar como se ha desarrollado la ejecución después de que esta haya finalizado. Además de poder ver las comunicaciones estándar (*send*, *recieve* y *reduce*) se registrarán los tiempos que tardan cada una de las tres búsquedas creando tres eventos personalizados con las funciones `MPE_Log_event`, `MPE_Log_get_event_number` y `MPE_Describe_state`, y el tiempo total de ejecución con `MPI_Wtime`.

5 EVALUACIÓN

Durante el siguiente capítulo se describe como se ha llevado a cabo la evaluación del sistema. Para este fin, se ha realizado un conjunto determinado de pruebas de rendimiento para evaluar el sistema propuesto sobre distintos escenarios.

5.1 Descripción del entorno de evaluación

Se han utilizado dos entornos diferentes para la ejecución y evaluación de cada uno de los módulos.

Para la obtención de resultados a través del módulo recolector, este fue lanzado en el *cluster* Tucán vía conexión SSH y bajo la aplicación Screen.

Screen es un multiplexor de terminales. Permite iniciar cualquier cantidad de aplicaciones de consola dentro de una única terminal y ejecutarlos de forma completamente independiente unas de otras. Los programas continúan ejecutándose cuando su ventana no se encuentra visible e incluso cuando la sesión de Screen es desacoplada de la terminal de usuario. Screen permite que un usuario pueda utilizar programas productivamente desde una única interfaz, mantiene los procesos activos aunque se salga de la sesión, ver procesos en tiempo real y mantener procesos remotos activos en conexiones vía SSH al salir de la sesión, que es la característica que se buscaba para poder lanzar el módulo recolector de forma remota.

En el caso del módulo procesador, es donde se utiliza todo el potencial que ofrece el hardware de Tucán y su característica de *cluster*, puesto que es este módulo el que explota el paralelismo. Este *cluster* está formando por 24 computadores con las siguientes características:

- Procesador: Intel(R) Xeon(R) CPU Quad-core E5405

- Velocidad del procesador: 2.00GHz
- Tamaño de la memoria RAM: 4 GB
- Tamaño del disco duro: 500 GB
- Sistema Operativo: Ubuntu Server 10.10
- Red local Gigabit Ethernet

El *cluster* está gestionado por PBS (*Portable Batch System*), un software que permite soportar la carga de múltiples usuarios, garantizando el uso exclusivo de los recursos del *cluster* a través de un sistema de colas. PBS permite a los usuarios enviar, monitorizar y eliminar trabajos. Al enviar el trabajo, este debe estar parametrizado indicando las necesidades del trabajo en cuestión a través de un *script*.

Cada nodo se conecta a través de una red Gigabit Ethernet que es capaz de alcanzar tasas de transferencia de hasta 942 MBits/seg.

Las ciudades almacenadas en el fichero de ciudades que utiliza el módulo procesador como entrada para realizar las búsquedas se han obtenido a partir de GeoWorldMap de Geobytes (56). GeoWorldMap contiene información que permite integrar localizaciones geográficas dentro de la base de datos de una organización.

El número de ciudades que contiene es de 25.100. Se han eliminado del fichero ciudades repetidas (existen ciudades con el mismo nombre en diferentes países) y se han colocado en orden alfabético. Se debe destacar también que los nombres de algunas ciudades son palabras con significado completo en inglés (por ejemplo, *may*, *winter*, *saint*, *man* o *fate)*, lo que hace que sea necesario tener en cuenta la existencia de falsos positivos.

Se recomienda encarecidamente que el fichero de ciudades no contenga acentos ni caracteres específicos de ciertos idiomas como pueden ser, ´, ¨, ^, `, ç, Õ, Å. Además, el *parser* cJSON no reconoce textos en japonés ni coreano puesto que trabaja con codificación UTF-8.

5.2 Descripción de los casos de evaluación

Con respecto a los parámetros utilizados en la ejecución, se han buscado usuarios de Twitter que cumplan ciertas características. Los candidatos a usuarios raíz que se perfilan como más idóneos cumplir los siguientes requisitos:

- Escribir los *tweets* en inglés. Las búsquedas del módulo procesador se realizan en este idioma por lo que es necesario que el usuario raíz escriba en inglés. Se sobreentiende que sus *followers* serán capaces de leer y entender sus *tweets* y por tanto, serán potenciales publicadores de *tweets* en inglés.
- Tener un número de *followers* adecuado al estudio que se desee realizar sobre el mismo. No olvidar que al incrementar la profundidad en el nivel de la recogida de datos, el número de usuarios de los cuales se almacenarán sus datos aumenta

de forma exponencial. Si no encaja, utilizar una ejecución de tipo parcial sería lo ideal para así reducirlo a un número que se ajuste a nuestras necesidades.

- Tener una actividad en Twitter notable, preferentemente publicando uno o más *tweets* diarios.
- Ser una personalidad destacada y conocida, a nivel local o mundial, puesto que de este modo se evita que sus *followers* lo sean solo por el mero hecho de corresponder a una amistad, sin que exista un interés real en el usuario al que se sigue.
- Tener una cuenta verificada por Twitter, para asegurar que se trata de la persona rea que se desea estudiar y no de una usurpación de identidad.

Considerando estas características, se han perfilado dos pruebas diferentes, utilizando diferentes usuarios raíz y diversos parámetros de entrada, que configuran la exploración de los nodos de una forma diferente y proporcionan la recogida de datos de una u otra forma.

5.2.1 Escenario 1

El primer escenario será una prueba básica en la que se utilizará un usuario raíz sobre el que recoger la información asociada a todos sus *followers*. De este modo se pretende estudiar cómo se distribuyen los *followers* directos de una persona. Entiéndase que si una persona sigue a otra se supone que es porque tiene algún interés en ella, sea del tipo que sea. Generalmente, este interés vendrá definido por la actividad que realice el usuario al que se sigue. En esta prueba podrá conocerse el área de influencia geográfica del usuario en cuestión, lo que equivaldría a conocer dónde se encuentran las personas que se interesan en su actividad.

En este escenario, el usuario elegido es aquel cuyo nombre de usuario es HIDEO_KOJIMA_EN. Esta cuenta pertenece a Hideo Kojima. Nacido el 24 de agosto de 1963 en Tokyo, Kojima es un diseñador de videojuegos que trabaja en la compañía japonesa Konami. Anteriormente fue el vicepresidente de Konami Computer Entertainment Japan, pero ahora lidera el equipo de desarrollo de videojuegos Kojima Productions. Hideo Kojima es el creador y director de varios videojuegos de éxito mundial, incluyendo la saga Metal Gear (57).

Kojima cumple los requisitos presentados para elegir usuarios potencialmente relevantes. Kojima es un usuario regular de la red social, llegando a escribir varios *tweets* al día.

5.2.2 Escenario 2

En este escenario, se han recopilado datos parciales, utilizando esta opción en el módulo recolector. Los parámetros utilizados han sido profundidad p = 8, recoger un máximo de n = 4 *followers* de cada usuario y un máximo de 100 *tweets* de cada uno de ellos. Con estos parámetros, en el mejor de los casos se tendría un resultado de

$$\sum_{x=1}^{p} n^x = \sum_{x=1}^{8} 4^x = 4^8 + 4^7 + \ ... \ + 4^1 = 87.380$$

perfiles.

El usuario raíz ha sido Lady Gaga (su *screen name* en Twitter es ladygaga). Lady Gaga es el nombre artístico de la cantante Stefani Joanne Angelina Germanotta, nacida el 28 de marzo de 1986. Lanzó su disco debut 'The Fame' en agosto del 2008, siendo un gran éxito y obteniendo reconocimiento internacional con los singles "Just Dance" y "Poker Face".

Inspirada por artistas *glam rock* como David Bowie, Elton John y Queen, y cantantes pop como Madonna, Michael Jackson y Amy Whinehouse, Lady Gaga es reconocida por su excentricidad y particular sentido de la moda (58).

Siguiendo la información proporcionada por Twitaholic.com, Lady Gaga es el usuario de Twitter que tiene el mayor número de seguidores, por encima de Justin Bieber, Britney Spears o Barrack Obama (59).

5.3 Análisis de los resultados

Se realizará un análisis de los datos recogidos por la aplicación para ambos escenarios. Nótese que para ambos escenarios la utilización de un grano u otro o de un número menor o mayor de procesos no influyen en los resultados finales (excepto en el valor del tiempo de ejecución), sino en su rendimiento. Este tema se tratará en un apartado propio.

5.3.1 Escenario 1

Los ficheros de entrada de este supuesto recogen la información de 37.217 usuarios de Twitter, los cuales comparten la característica de ser *followers* del usuario utilizado como raíz: Hideo Kojima.

5.3.1.1 Location

Los datos recogidos para el procesamiento del campo 'Localización' del perfil de cada usuario se resumen en los siguientes:

LOCATION		
Concepto	***Valor***	***Porcentaje (%)***
Total usuarios procesados	37.217	100
Suspendidos	1	0,0026
Eliminados	0	0
Errores indefinidos	18	0,049
Válidos	37.198	99,95
Perfiles útiles	37.198	100
Ciudades encontradas (diferentes)	19.187 1.732	51,58

Ciudades con valor *null*	11.560	31,76
Ciudades desconocidas	6.451	17,34

Tabla 1 .- Datos obtenidos para el campo 'location' del escenario 1

De 37.217 usuarios procesados, 1 perfil se corresponde con el de una cuenta suspendida por Twitter, 18 perfiles contienen errores indefinidos arrojados por Twitter, ninguno de ellos pertenece a una cuenta eliminada por el usuario y 37.198 fueron válidos para obtener el valor del campo *location* (prácticamente la totalidad de los JSONs recogidos, un 99,95%). De esos 37.198 usuarios, se recogieron 19.187 ciudades, (de las cuales 1.732 eran diferentes), pero 11.560 usuarios tenían el campo vacío o a *null* (representa más de un cuarto de los JSON válidos obtenidos con un 31,76 %) y 6.451 usuarios contenían texto sobre el que no se encontró ninguna ciudad conocida (un 17,34 %).

Para facilitar el comentario de los resultados, se ha elaborado una tabla resumen que contiene las 30 ciudades con resultados más altos.

Los resultados completos pueden encontrarse en los ficheros digitales adjuntos a esta memoria.

Cómo puede observarse, los puestos más altos de la tabla los copan palabras monosílabas o bisílabas, muchas de ellas con significado propio en inglés (*as, art, man*). Recuérdese que aunque se han elegido los escenarios para que arrojasen el mayor número de usuarios posibles que escriban en ingles, se debe asumir que es muy probable que se hayan recogido y estudiado numerosos datos de usuarios que, además de utilizar el inglés, escriben también en sus idiomas nativos. Si se tiene en cuenta que el campo *location* es un campo de texto de entrada libre, los resultados obtenidos no parecen extraños.

Con el fin de filtrar estos resultados que no parecen reflejar información útil de localización, se ha añadido un segundo bloque a la tabla que muestra el top 30, en la que se han eliminado todas aquellas palabras que tienen significado propio en inglés (podrían encontrarse en un diccionario) o que se corresponden con nombres propios de persona y hacen referencia a ciudades escasamente pobladas, así como otras palabras sospechosas de interferir en la veracidad de los datos.

TOP 30				
1	AS	1678	MEXICO	455
2	BO	565	LONDON	448
3	ADA	557	BRAZIL	361
4	MEXICO	455	ENGLAND	344
5	LONDON	448	CALI	281
6	MO	448	PARIS	236
7	BRAZIL	361	NEW YORK	231
8	ENGLAND	344	LOS ANGELES	176
9	CALI	281	CHESTER	170

10	ART	258	MADRID	155
11	PARIS	236	CHICAGO	151
12	NEW YORK	231	ITALY	124
13	MAN	191	FLORIDA	113
14	PAU	183	SANTIAGO	108
15	LOS ANGELES	176	TORONTO	97
16	CHESTER	170	COLO	95
17	EDE	161	FINLAND	77
18	ELAND	155	NORWAY	74
19	MADRID	155	RIO	73
20	CHICAGO	151	SEATTLE	68
21	KING	149	SAN DIEGO	63
22	BAR	147	DUBLIN	59
23	ARK	142	MIAMI	59
24	TOK	142	MICHIGAN	59
25	ITALY	124	SCOTLAND	58
26	AMA	119	ATLANTA	54
27	ETHER	119	CHINA	53
28	FLORIDA	113	KUWAIT	51
29	CISCO	110	ONTARIO	51
30	SANTIAGO	108	LIVERPOOL	50
	TOTAL	*8.477*	*TOTAL*	*4.394*

Tabla 2.- Top 30 de ciudades encontradas para 'location' en el escenario 1

En la columna de los resultados filtrados algunas ciudades han sido marcadas con color rojo. Estas hacen referencia a ubicaciones que aunque son nombres de ciudad, son más destacadas como nombre de países o de estados. Sin duda se puede afirmar que las ciudades llamadas *Brazil*, *England*, *Italy*, *Finland*, *Norway*, *Scotland*, *China y Kuwait* no son grandes conocidas y que el grueso del conteo de las mismas se debe a que el usuario está haciendo referencia al país y no a la ciudad (no así *Mexico*, puesto que tanto la ciudad como el país son destacados). Sin embargo, como son ubicaciones geográficas válidas que se consideran relevantes, se han mantenido.

Las ciudades de *Cali y Colo* se han identificado con color verde puesto que este es un caso muy especial: Cali es frecuentemente utilizado para designar al estado de California de Estados Unidos en el argot del inglés americano, y lo mismo sucede con Colo y Colorado. La ciudad de nombre *Rio* también se incluye en este grupo, puesto que podría relacionarse con la ciudad de Rio de Janeiro en Brasil (aunque también existen otras ciudades con *Rio* en Argentina y España, pero de mucho menos calado).

En vista de los resultados se puede afirmar que la ciudad con un mayor número de seguidores de Hideo Kojima es México (aunque debería extenderse al país y no solo a la ciudad), seguido muy de cerca de Londres.

El país que cuenta con mayor representación en el *top* 30 es Estados Unidos: Nueva York, Los Ángeles, Chicago, Florida, Seattle, San Diego, Miami, Michigan y Atlanta (no olvide California y Colorado como estados).

Considerando a *Cali* como California, y a las ciudades de Los Ángeles y San Diego (que se encuentran en él), este estado tiene una presencia importante de seguimiento a Hideo Kojima.

Múltiples ciudades y países del sur del continente americano se encuentran en la lista: Méjico, Brasil, Santiago (de Chile), Río (de Janeiro).

También el Reino Unido tiene una fuerte presencia con ciudades como Londres, England (supóngase el nombre del propio país), Chester, Scotland (de nuevo, el país), y Liverpool, así como Canadá, con Toronto y Ontario (aunque esta última también existe en los Estados Unidos).

La única representación española está marcada por la ciudad de Madrid en el puesto número 10. Otras ciudades europeas son París (en Francia) y Dublín (en Irlanda).

La nota más interesante la marcan los países/ciudades de Kuwait (un país no tan desarrollado la mayoría de los presentes en la lista) y China (controlado por la censura), que ocupan los puestos 28º y 27º respectivamente.

Puede resultar llamativo el hecho de que no aparezca ninguna ciudad japonesa, país de origen de Hideo Kojima. Esto podría justificarse por el hecho de que Kojima tiene otra cuenta de Twitter donde escribe en japonés, por lo que sus fans japoneses probablemente sean seguidores de esta cuenta alternativa en vez de seguir la cuenta en la escribe en inglés, más orientada al público occidental. Además, recuérdese que la escritura japonesa no se realiza utilizando el alfabeto romano, sino que se utiliza escritura ideográfica (*kanjis*) y silabaria (*hiragana* y *katakana*).

Ilustración 44.- Cuenta de Hideo Kojima en japonés.

5.3.1.2 Geo

Para los resultados obtenidos a través de la evaluación de los datos de geolocalización, se han estudiado los *tweets* pertenecientes a 37.217 usuarios, de los cuales 3.437 usuarios los tenían protegidos (y por tanto, inaccesibles) y los *tweets* correspondientes a un usuario han arrojado un error indefinido. De esos usuarios accesibles, 2.103 usuarios no han escrito nunca un *tweet*.

Teniendo presente que el máximo de *tweets* por usuario que se solicitaron ha sido de 100, se han analizado 2.008.947 *tweets*, de los cuales 1.981.113 no poseían datos de geolocalización, un 98,61%. Recuérdese que el nivel de la información de geolocalización que se puede añadir a un *tweet* es personalizable en grano, desde ciudades hasta barrios, calles o países, por lo que hay que aclarar que esta recolección de datos considera que un *tweet* no contiene datos de geolocalización cuando no se ajusta a lo buscado (que en el caso que acontece, son ciudades). Considérese también que la funcionalidad de geolocalización de Twitter no está disponible en todos los países, y ello repercute en los resultados, aunque no parece descabellado afirmar que la funcionlidad "Twittea con tu localización" no está siendo tan utilizada como a priori podría parecer.

GEO		
Concepto	***Valor***	***Porcentaje (%)***
Total usuarios procesados	37.217	100
Usuarios con tweets protegidos	3.437	9,23
Errores indefinidos	1	0,0027
Usuarios sin tweets	2.103	5,65
Usuarios con tweets	31.676	85,11
Tweets procesados	2.008.947	100
Tweets sin datos geo	1.981.113	98,61
Tweets con datos geo	27.834	1,39
Tweets útiles	27.834	100
Ciudades encontradas **(diferentes)**	16.563 1.377	59,5
Ciudades desconocidas	11.271	40,49

Tabla 3.- Datos obtenidos para el campo 'geo' del escenario 1

Así pues, solo un pequeño conjunto de *tweets* (27.834) generan los resultados obtenidos: 16.563 ciudades encontradas de las cuales, 1.377 son diferentes, y 11.271 ciudades fueron desconocidas.

Este análisis no proporciona falsos positivos, puesto que estos datos de geolocalización los puebla Twitter. Por tanto, en este caso solo se presentan los 30 mayores resultados encontrados.

TOP 30		
1	RIO DE JANEIRO	306
2	SHEFFIELD	220

3	COVENTRY	148
4	QUINCY	147
5	SUNDERLAND	140
6	GUADALAJARA	138
7	CURITIBA	134
8	SALVADOR	133
9	MUNCIE	131
10	LOS ANGELES	122
11	READING	120
12	YORK	111
13	RECIFE	108
14	GALWAY	107
15	BELO HORIZONTE	102
16	FAIRFIELD	96
17	PORTO VELHO	94
18	CAPE TOWN	93
19	DERBY	92
20	BOURNEMOUTH	90
21	PORTO ALEGRE	90
22	WEST CHESTER	89
23	ATHENS	88
24	MEDFORD	88
25	TOLEDO	86
26	CAMBRIDGE	84
27	OXFORD	84
28	SELBY	82
29	SOLIHULL	80
30	MODESTO	78
	TOTAL	*3.481*

Tabla 4.- Top 30 de ciudades encontradas para 'geo' en el escenario 1

La ciudad con mayor número de resultados es Río de Janeiro (Brasil), que obtiene 86 resultados más que la segunda de la tabla, Sheffield (Inglaterra).

Lo más destacable del *top* 30 es que la mayoría de las ciudades se concentran en los siguientes países (entre paréntesis se indica el puesto que ocupa la ciudad dentro de la clasificación):

- Brasil: Río de Janeiro (1), Curitiba (7), Salvador (8), Recife (13), Belo Horizonte (15), Porto Velho (17), Porto Alegre (21).
- Inglaterra: Sheffield (2), Coventry (3), Sunderland (5), Reading (11), York (12), Derby (19), Bornemouth (20), Medford (24), Cambridge (26), Oxford (27), Solihull (28).
- Estados Unidos: Quincy (4), Muncie (9), Los Angeles (10), West Chester (22), Medford (24), y Modesto (30).

Varias ciudades del *top* existen en varios países:

- Guadalajara: México y España
- Fairfield: muy común en países de habla inglesa (Australia, Canadá, Nueva Zelanda, Reino Unido y Estados Unidos).
- Toledo: España, Brasil, Estados Unidos, Colombia y Uruguay.

Otras ciudades destacadas son Atenas (en Grecia), Galway (en Irlanda), Cape Town (en Sudáfrica).

Los resultados obtenidos analizando la etiqueta *geo* son parecidos a los obtenidos a través del campo *location*, donde Estados Unidos e Inglaterra pueblan las posiciones del *top*. También quedan patentes las presencias de Brasil y México (aunque este último, en bastante menor medida).

5.3.1.3 Tweets

En total, se procesaron 2.008.946 *tweets* y se encontraron 105.213 ciudades (recuérdese que en las búsquedas dentro de los tweets se utiliza el sistema de patronaje preposición + ciudad).

Concepto	***Valor***
Total tweets procesados	2.008.946
Tweets con ciudades	105.213
Ciudades diferentes encontradas	4.554

Tabla 5.- Datos obtenidos para los tweets en el escenario 1

Al igual que en el caso del campo *location*, se realizado un filtrado del top 30 para destacar los resultados de una forma realista.

TWEETS				
1	MO	6.562	LONDON	512
2	BO	3.812	BREA	409
3	TIM	3.120	TOKYO	343
4	AS	3.082	COLO	304
5	MAN	2.267	CANADA	289
6	HOME	1.924	ENGLAND	217
7	BLACK	1.782	BOSTON	215
8	REA	1.711	CHINA	200
9	SON	1.480	NEW YORK	197
10	BIEBER	1.249	CALI	195
11	DAY	1.205	SAN FRANCISCO	187
12	UNA	967	BRAZIL	186
13	STAR	953	CHICAGO	182
14	GUY	810	PARIS	170
15	MAY	771	MEXICO	167
16	SUN	714	TORONTO	163

17	SALE	707	EGYPT	159
18	AMA	692	RIO	145
19	CALL	666	CALIFORNIA	128
20	THE PAS	629	SEATTLE	126
21	WHITE	624	HONG KONG	122
22	ORD	613	SILVA	120
23	MEANS	600	CUA	119
24	BULLE	592	MANCHESTER	116
25	ART	582	LIVERPOOL	114
26	TEA	580	HOLLYWOOD	112
27	PARK	569	AUSTIN	107
28	SUR	530	ITALY	106
29	MARVEL	522	IRELAND	103
30	TRAIL	517	MIAMI	97
	TOTAL	*40.832*	*TOTAL*	*5.610*

Tabla 6.- Top 30 de ciudades encontradas en los tweets en el escenario 1

La ciudad que ocupa el puesto número uno es Londres, con 512 coincidencias. De nuevo, Estados Unidos es el país con más presencia: Brea, Colo (Colorado), Boston, Nueva York, San Francisco, Chicago, California, Seattle, Hollywood, Austin y Miami. Destánquese de nuevo los encuentros de Cali y California, estado que vuelve a tener una presencia destacable (San Francisco y Brea pertenecen a este estado).

Se puede ver también como Brasil, México, representaciones de China (como Hong Kong), Irlanda e Italia aparecen en la lista, como sucediera en casos anteriores.

Silva es un nombre de ciudad muy común en el idioma portugués.

En el estudio de los *tweets* aparecen ciudades de localizaciones que no se habían dado antes como Tokyo (en Japón). Egipto y Cúa (en Venezuela).

Nótese que múltiples palabras de las que aquí se encuentran en el top 30 también aparecen en el top 30 del campo *location (as, bo, mon...).*

5.3.1.4 Comparativa

Se ofrece a continuación una comparativa entre los 3 diferentes métodos de búsqueda de datos, donde puede verse las coincidencias que se han dado entre ellos.

Ténganse en cuenta que el número total de comparaciones solo recoge aquellas comparaciones que se dan cuando ambos métodos a comparar han obtenido algún resultado satisfactorio (esto es, el campo no tenía un valor *null*, vacío o una ciudad desconocida).

COMPARATIVA ENTRE LOS DIFERENTES MÉTODOS DE BÚSQUEDA	
Número total de usuarios válidos	37.198
Número total de comparaciones	15.725

Comparaciones location y geo	1.066
Coincidencias entre location y geo	196 (18,39 %)
Comparaciones entre location y búsqueda en tweets	13.326
Coincidencias entre location y búsqueda en tweets	1.286 (9,65 %)
Comparaciones geo y búsqueda en tweets	1.333
Coincidencias entre geo y búsqueda en tweets	199 (14,93 %)

Tabla 7.- Comparativa entre los métodos de búsqueda del escenario 1

De 37.198 usuarios (y recuérdese que cada uno de ellos, hasta con un máximo de 100 *tweets*) en 15.725 casos sucedió que los dos datos a comparar estuviesen poblados (con información útil).

Las búsquedas que más han coincidido han sido las realizadas a través del campo *location* y *geo*, los cuales contenían simultáneamente datos relevantes en 1.066 ocasiones y obtuvieron el mismo resultado en 196 de ellas. Se considera razonable que los campos realmente destinados al efecto de contener información geolocalizable sean los más propicios para obtener este tipo de información. Al fin y al cabo, un *tweet* puede contener información extremadamente diversa, cuando un campo *location* no debería (y mucho menos un campo *geo*). Sin embargo, mientras el campo *location* sea una entrada de texto libre, los resultados a obtener no serán tan interesantes como si fueran parte de un conjunto limitado de valores predefinidos de entrada.

5.3.2 Escenario 2

Los ficheros de entrada de este ejercicio contienen los datos obtenidos de 65.950 usuarios de Twitter. La característica común de todos ellos es que se puede llegar a cualquiera navegando a través de relaciones de *following* partiendo desde el usuario Lady Gaga y en un máximo de 8 saltos.

5.3.2.1 Location

De 65.950 JSONs obtenidos, 997 de ellos contienen información de usuarios con cuentas suspendidas, 136 contienen errores indefinidos (asumibles) y ninguno pertenece a una cuenta eliminada.

Por tanto, se mantienen 64.817 JSONs útiles. De la mitad de ellos se han obtenido ciudades (32.178) y del resto, la mitad contenían texto sobre el que no se ha identificado ninguna ciudad y la otra mitad tenían el campo vacío o *null*.

LOCATION		
Concepto	*Valor*	*Porcentaje (%)*
Total usuarios procesados	65.950	100
Suspendidos	997	1,51
Eliminados	0	0
Errores indefinidos	136	0,21
Válidos	64.817	98,29
Perfiles útiles	64.817	100
Ciudades encontradas	32.178	49,64

(diferentes)	2.028	
Ciudades con valor *null*	16.769	25,88
Ciudades desconocidas	15.870	24,49

Tabla 8.- Datos obtenidos para el campo 'location' en el escenario 2

El *top* 30 está encabezado por la ciudad de Nueva York, con 697 apariciones. Estados Unidos coloca a 14 de sus ciudades (y estados) en la lista: Nueva York, California, Los Ángeles, Chicago, Florida, Atlanta, Colorado, Detroit, Miami, San Diego, Houston, Seattle, Michigan y Phoenix.

El segundo puesto es para Londres, que sigue muy de cerca a Nueva York. Destáquese también la novena posición de Inglaterra dentro de la lista.

TOP 30				
1	AS	3099	NEW YORK	697
2	ART	1468	LONDON	678
3	BO	940	CALI	663
4	ADA	909	LOS ANGELES	541
5	NEW YORK	697	CHICAGO	398
6	LONDON	678	FLORIDA	352
7	CALI	663	ATLANTA	350
8	REA	556	BANDUNG	277
9	MO	543	ENGLAND	266
10	LOS ANGELES	541	COLO	222
11	MAN	420	CHESTER	166
12	CHICAGO	398	TORONTO	157
13	FLORIDA	352	BRAZIL	151
14	ATLANTA	350	DETROIT	144
15	ERA	321	MIAMI	144
16	TATE	297	SAN DIEGO	143
17	BANDUNG	277	MEDAN	140
18	ENGLAND	266	MEXICO	140
19	AMA	234	HOUSTON	136
20	BAR	224	SEATTLE	135
21	COLO	222	MICHIGAN	119
22	ARK	215	BERLIN	111
23	BAY	211	ILA	110
24	TOK	204	PHOENIX	104
25	KING	194	SEOUL	98
26	BIEBER	193	AMSTERDAM	88
27	ACE	178	INA	86
28	BEACH	176	ITALY	86
29	CHESTER	166	PARIS	85
30	CISCO	166	BALI	80
	TOTAL	*15.158*	*TOTAL*	*6.867*

Tabla 9.- Top 30 de ciudades encontradas para el campo 'location' en el escenario 2

Indonesia toma una importancia interesante, colocando a 3 ciudades en la lista: Bandung (8), Medan (17) y Bali (30). Otras ciudades de Asia son Seúl (Corea del Sur, puesto 25) e Ina (nombre de varias ciudades de Japón, 27).

En representación de Europa puede verse a Berlín (Alemania, 22), Ámsterdam (Holanda, 26), Italia (28) y Paris (Francia, 29).

Al igual que ocurriera en el escenario 1, México y Brasil aparecen en el *top* 30.

5.3.2.2 Geo

Lo más destacable es la pequeña cantidad de usuarios que utilizan la geolocalización en sus *tweets*: de 4.341.605 *tweets* tan solo el 0,53% contenían información de geolocalización de una ciudad. También destaca que 10.190 ciudades hayan sido desconocidas (casi la mitad de las obtenidas), por lo que tan solo 13.065 han sido útiles.

GEO		
Concepto	*Valor*	*Porcentaje (%)*
Total usuarios procesados	65.950	100
Usuarios con tweets protegidos	4.665	7,07
Errores indefinidos	0	0
Usuarios sin tweets	1.560	2,36
Usuarios con tweets	59.725	90,56
Tweets procesados	4.341.605	100
Tweets sin datos geo	4.318.350	99,46
Tweets con datos geo	23.255	0,53
Tweets útiles	23.255	100
Ciudades encontradas (diferentes)	13.065 1.011	56,18
Ciudades desconocidas	10.190	43,82

Tabla 10.- Datos obtenidos para el campo 'geo' del escenario 2

TOP 30		
1	LOS ANGELES	278
2	HOUSTON	244
3	DEPOK	213
4	CHICAGO	193
5	BLACKPOOL	163
6	RIO DE JANEIRO	154
7	NEW YORK	144
8	LAS VEGAS	126
9	SPRINGFIELD	123
10	ATLANTA	115
11	COLLEGE PARK	105

12	BELO HORIZONTE	102
13	PEEL	102
14	FARMINGTON	99
15	DURHAM	98
16	HONOLULU	98
17	MIAMI	97
18	BATTLE CREEK	93
19	MCKENZIE	92
20	PONTIAC	92
21	NORTH LAS VEGAS	90
22	NEW ORLEANS	89
23	PLANO	89
24	ESTERO	88
25	FREDERICKSBURG	86
26	PORTLAND	86
27	SAYREVILLE	84
28	PATERSON	81
29	FAIRFIELD	79
30	FOUR OAKS	79

Tabla 11.- Top 30 de ciudades encontradas para 'geo' en el escenario 2

La ciudad que ocupa el puesto número 1 es Los Ángeles, con 278 apariciones. De nuevo, numerosas ciudades americanas se sitúan en el *top*, exactamente 23: Los Ángeles, Houston, Chicago, New York, Las Vegas, Springfield, Atlanta, College Park, Farmington, Honolulu, Miami, Battle Creek, McKenzie, Pontiac, North Las Vegas, New Orleans, Plano, Estero, Fredericksburg, Portland, Sayreville, Paterson y Fairfield.

La ciudad Springfield no solo existe en los EE.UU sino también en Australia, Canadá, Reino Unido e incluso Sudáfrica y Nueva Zelanda. Lo mismo sucede con College Park en Australia, Canadá y Reino Unido, Peel en Australia y Canadá, Farmington en EE.UU, Canadá y el Reino Unido, Portland en Australia, Canadá, Irlanda, Nueva Zelanda, Reino Unido y EE.UU, Paterson en Australia, EE.UU y Sudáfrica, Fairfield en Australia, Canadá, Nueva Zelanda, Reino Unido y EE.UU.

De las pocas ciudades restantes destacan dos ciudades brasileñas, Río de Janeiro (en el puesto 6) y Belo Horizonte (en el 12), y dos ciudades indonesias, Depok (en el puesto 3) y Durham (15).

5.3.2.3 Tweets

Un total de 4.341.604 *tweets* fueron procesados, y fueron encontradas 237.056 ciudades.

Concepto	***Valor***
Total tweets procesados	4.341.604
Ciudades encontradas	237.056

Ciudades diferentes encontradas	6.789

Tabla 12.- Datos obtenidos para los tweets en el escenario 2

Londres se coloca en la primera posición de las ciudades más repetidas, con 1.295 apariciones.

Estados Unidos recoge el mayor número de ciudades (y estados) presentes, con Nueva York, Colorado, Brea, Hollywood, California, Miami, Austin, Florida, Chicago, Las Vegas, Los Ángeles, Dallas, Atlanta, San Francisco, Phoenix y San Diego.

Otras ciudades y países a destacar resultan Canadá, en la 3ª posición, y Toronto,China, París (Francia), Egipto, Inglaterra (recuérdese que Londres ocupa la primera posición), Dublín (Irlanda), México, Italia, Tokyo (Japón), Berlín (Alemania) y Brasil

TWEETS				
1	MO	10.297	LONDON	1.295
2	BIEBER	9.697	NEW YORK	860
3	BO	7.778	COLORADO	784
4	TIM	4.980	CANADA	729
5	AS	4.688	BREA	719
6	HOME	4.351	CHINA	668
7	DAY	3.995	HOLLYWOOD	638
8	MAN	3.772	CALIFORNIA	622
9	REA	3.621	MIAMI	585
10	SALE	2.308	AUSTIN	572
11	DEAL	1.960	FLORIDA	566
12	SON	1.848	CHICAGO	536
13	SUN	1.784	PARIS	518
14	ART	1.772	LAS VEGAS	501
15	ORD	1.658	LOS ANGELES	441
16	AMA	1.650	EGYPT	423
17	BAR	1.537	CALIFORNIA	419
18	STAR	1.485	DALLAS	407
19	MAY	1.442	TORONTO	388
20	MARK	1.382	ATLANTA	365
21	DOW	1.301	ENGLAND	348
22	LONDON	1.295	SAN FRANCISCO	323
23	AUTO	1.294	DUBLIN	317
24	POST	1.290	MEXICO	312
25	PRICE	1.078	ITALY	301
26	EARTH	1.027	TOKYO	298
27	TEA	979	BERLIN	295
28	START	927	BRAZIL	290
29	ONLY	872	PHOENIX	268

30	NEW YORK	860	SAN DIEGO	265
	TOTAL	*82.928*	*TOTAL*	*15.053*

Tabla 13.- Top 30 de ciudades encontradas en los tweets en el escenario 2

A diferencia con los resultados obtenidos en el estudio del campo *location*, no aparecen ciudades indonesas, pero sí existe gran similitud en el resto de resultados obtenidos, especialmente en la parte alta de la lista, coronadas por Londres y Nueva York.

5.3.2.4 Comparativa

El número total de comparaciones entre diferentes métodos de búsqueda asciende a 20.583. Para los métodos de *location* y *geo* se han efectuado 810 comparaciones, de cuales más de un 20% resultaron coincidentes.

Entre *location* y la búsqueda en los *tweets* se realizaron 2.641 comparaciones, de las que sólo casi un 12% obtuvieron el mismo resultado.

Entre *geo* y búsqueda, de 959 comparaciones, casi el 22% recogieron la misma ubicación, marcando un gran resultado.

En este caso, los métodos *geo vs búsqueda* resultan los más coincidentes, seguidos muy de cerca por las comparaciones entre el campo *location* y *geo.* Con este resultado se demuestra que la búsqueda de ciudades y lugares dentro de los *tweets* para geolocalizar a un usuario es efectiva.

Por otro lado, vemos como los resultados entre los campos de entrada libre, *location y tweets*, coinciden con una tasa menor. Resulta lógico puesto que estos campos pueden, al fin y al cabo, contener cualquier dato: aunque *location* haya sido ideado para portar información de ubicación, esta no tiene porque ser geográfica (en casa, en la playa, en la Universidad...).

COMPARATIVA ENTRE LOS DIFERENTES MÉTODOS DE BÚSQUEDA	
Número total de usuarios válidos	65.950
Número total de comparaciones	24.435
Comparaciones location y geo	810
Coincidencias entre location y geo	167 (20,62 %)
Comparaciones entre location y búsqueda en tweets	22.666
Coincidencias entre location y búsqueda en tweets	2.641 (11,65 %)
Comparaciones geo y búsqueda en tweets	959
Coincidencias entre geo y búsqueda en tweets	207 (21,59 %)

Tabla 14.- Comparativa entre los métodos de búsqueda del escenario 2

5.4 Análisis del rendimiento

Para cada uno de los dos escenarios se han lanzado 18 ejecuciones diferentes:

- Escenario 1: se utilizarán 2, 4, 8, 16, 32 y 64 procesos y con unos valores de grano (que es el que define la carga de trabajo de cada proceso esclavo) de

tamaño pequeño, medio y grande, como son 32, (es decir, cada esclavo recibirá del maestro la información de 32 usuarios respectivamente), 290 y 581, utilizando para las comunicaciones la red Gigabit Ethernet.

- Escenario 2: se utilizarán 2, 4, 8, 16, 32 y 64 procesos con un grano de 32, 509 y 1019, bajo la red Gigabit Ethernet.

Los valores de los procesos no han sido elegidos arbitrariamente: cada nodo del clúster posee un procesador Quad-Core, por lo que existen 4 núcleos independientes sobre los que operar dentro de un mismo computador. Así pues, para la fase de mapeo se forzará a que cada nodo del clúster asigne un proceso a uno de sus *cores* antes de utilizar otro nodo del clúster, o lo que es lo mismo, se intentará poblar al máximo los *cores* (que hacen las veces de procesadores físicos reales) de cada nodo que se emplee. Para ello, se utilizan los siguientes comandos y parámetros:

```
/home/software/mpich2-1.2.1/bin/mpdboot -v -n 16 -f nodes --
ifhn=$MPDHOST

/home/software/mpich2-1.2.1/bin/mpiexec -n 64 ./processor.64 32
../ladygaga/ladygaga1.txt ../ladygaga/ladygaga2.txt
../ladygaga/cities_uniq_order.txt ladygagaresults.64.txt

/home/software/mpich2-1.2.1/bin/mpdallexit
```

Para lanzar 64 procesos, se necesitan únicamente 64/4 = 16 nodos del clúster para asegurar que los 4 cores de cada uno de ellos estará poblado.

Los valores de los granos también se han elegido conforme al siguiente cálculo: el grano más grande se corresponde con el resultado de dividir el número de registros (o usuarios, o JSON) de los ficheros utilizados como entrada entre 64, que es la cantidad máxima de *cores* que se van a utilizar simultáneamente del clúster. De este modo, para el escenario 1 se tiene que 37.217 / 64 = 581 (tras un redondeo hacia el entero menor más cercano) y 65.250 / 64 = 509. El grano medio es la mitad del valor de grano mayor y el grano menor es de 32 en ambos casos.

Para cada una de las ejecuciones se calculará el tiempo de ejecución, el *speedup* y la eficiencia.

Los tiempos de ejecución de cada una de las pruebas realizadas pueden obtenerse de dos fuentes diferentes: del log generado a partir de las llamadas MPE embebidas en el código, o del fichero que se crea tras la ejecución y que contiene los datos obtenidos.

Los *logs* generados tienen un formato clog2, que debe ser transformado a slog2 para su visualización con la herramienta Jumpshot. El tiempo total de ejecución se corresponde con el valor 'Global Max Time'.

Ilustración 45.- Valor 'Global Max Time' en Jumpshot

El tiempo de ejecución mostrado en el fichero generado que contiene los resultados obtenidos, está calculado haciendo uso de la función `MPI_Wtime` y `MPI_Barrier` de forma auxiliar.

5.4.1 Rendimiento del Escenario 1

Se puede comprobar que, para una misma ejecución, ambos resultados no son idénticos pero sí muy cercanos. Para los cálculos realizados en el presente documento, se ha optado por utilizar los valores que se obtienen de los *logs*.

En la Tabla 15 puede verse de forma muy clara como los tiempos de ejecución se reducen de forma drástica al aumentar el número de procesos esclavos. Se observa también como el tamaño del grano no tiene un impacto excesivamente significativo en el tiempo de ejecución. Ambos resultados son totalmente lógicos y concuerdan con los esperados: un mayor número de procesos hace que exista más paralelismo en el desempeño del trabajo por lo que, necesariamente, este concluye antes. El *overhead* que causan las comunicaciones es muy pequeño y no afecta a estos resultados, principalmente porque el grueso del tiempo de la ejecución se corresponde con el procesamiento que realizan los esclavos de los datos. Respecto al valor del grano, debido de nuevo a que la mayor parte del tiempo los procesos se encuentran procesando los datos, no resulta demasiado importante encontrar un equilibrio entre los tamaños de los mensajes y la frecuencia de los mismos con el fin de reducir el *overhead* de las comunicaciones. Además, en esto también influye el hecho de que dado un determinado valor de grano, cada uno de los diferentes procesos tarda un tiempo muy similar en tratar el mismo número de usuarios.

Escenario 1 (37.217 registros)				
Nº PROC	**GRANO**	**TIEMPO EJECUCIÓN (s)**	**SPEEDUP**	**SPEEDUP IDEAL**
2 (secuencial)	581	23.020,930943	1,000000	1
4	581	7.890,303060	2,917623	3
8	581	3.601,374633	6,392262	7
16	581	1.873,399267	12,288321	15
32	581	1.124,980323	20,463408	31
64	581	736,250134	31,267812	63
2 (secuencial)	290	23.041,125881	1,000000	1
4	290	7.761,609630	2,968601	3
8	290	3.451,400723	6,675877	7
16	290	1.702,016825	13,537543	15
32	290	922,684743	24,971829	31

64	290	555,154385	41,503997	63
2 (secuencial)	32	23.010,844281	1,000000	1
4	32	7.718,998698	2,981066	3
8	32	3.321,932064	6,926946	7
16	32	1.562,439137	14,727514	15
32	32	775,785283	29,661357	31
64	32	398,284675	57,774867	63

Tabla 15.- Tiempos de ejecución, speedup y eficiencia para el escenario 1

A continuación se presenta un gráfico que permite comparar de una forma rápida y cómoda los tiempos de ejecución de los diversos granos y números de procesos.

Así, se puede afirmar que la ejecución más rápida se produce con un número muy alto de procesos (64) y con un grano de trabajo pequeño (32): un total de 398,28 segundos o 6,65 minutos.

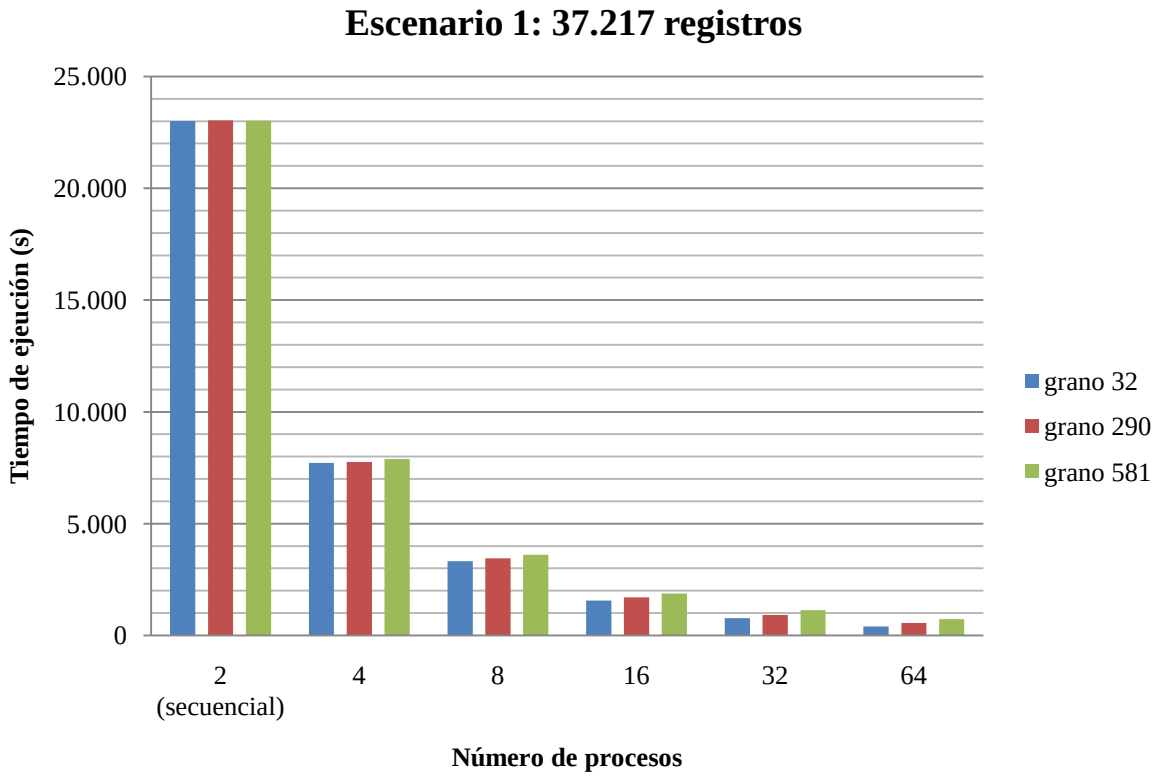


Ilustración 46.- Gráfica para los tiempos de ejecución del escenario 1

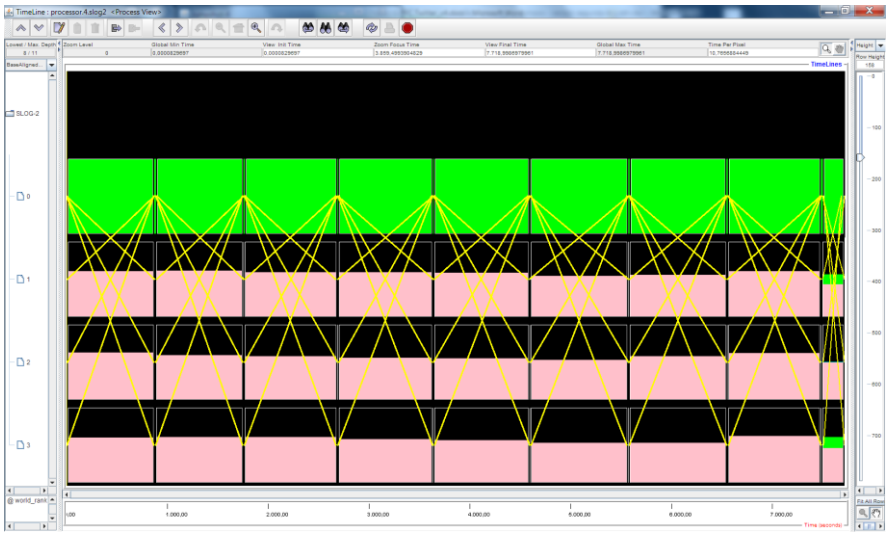


Ilustración 47.- 'Process View' de Jumpshot para la ejecución del escenario 1 con 4 procesos y grano 32

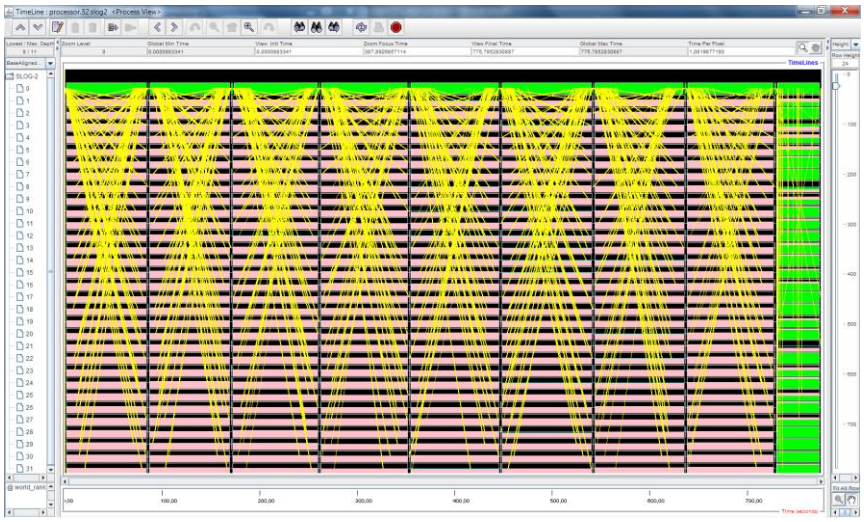


Ilustración 48.- 'Process View' de Jumpshot para la ejecución del escenario 1 con 16 procesos y grano 8

Para cualquiera de los *logs* generados por una ejecución del escenario 1 (ilustraciones 47 y 48), puede verse que en los procesos esclavos (estos son aquellos cuyo número de identificación es distintos de 0) domina el color rosa, que pertenece a un evento llamado 'Status'. Este es un evento personalizado que marca el tiempo que un proceso dedica dentro de la rutina `process_status` (que se encarga de realizar las búsquedas de las ciudades dentro del texto de un *tweet*). Del mismo modo, el proceso maestro se mantiene principalmente bloqueado en `MPI_Recv`, ya que espera a que algún proceso esclavo termine para enviarle una nueva pieza de trabajo.

El *speedup* o la aceleración (S) es una medida que captura el beneficio relativo de resolver un problema en paralelo. Se calcula dividiendo el tiempo de ejecución secuencial (que es el obtenido por 2 procesos, ya que no hay ningún paralelismo en esta ejecución) entre el tiempo de ejecución paralelo utilizando *p* procesadores.

$$S = \frac{T_s}{T_p}$$

El *speedup* ideal es aquel cuyo valor tiende al número de procesos con los que se ha ejecutado. Sin embargo, solo un sistema ideal de *p* procesadores puede alcanzar una aceleración de p. Esto se debe a los costes asociados a las sincronizaciones, comunicaciones, etc.

Viendo las gráficas a continuación se puede de nuevo asegurar que en este tipo de procesamiento es absolutamente conveniente utilizar paralelización, puesto que el tiempo de ejecución se reduce cuantiosamente. Nótese además que se obtienen unos resultados muy buenos para la aceleración (las gráficas adquieren una forma más o menos lineal). Sin embargo, no puede afirmarse que un aumento en el número de procesos producirá siempre una mejora: alcanzado un determinado número de procesos, el coste de comunicar datos entre ellos a través de la red será más costoso que tener un número más reducido de procesos lo que repercutiría en los tiempos de ejecución, que resultarían mayores.

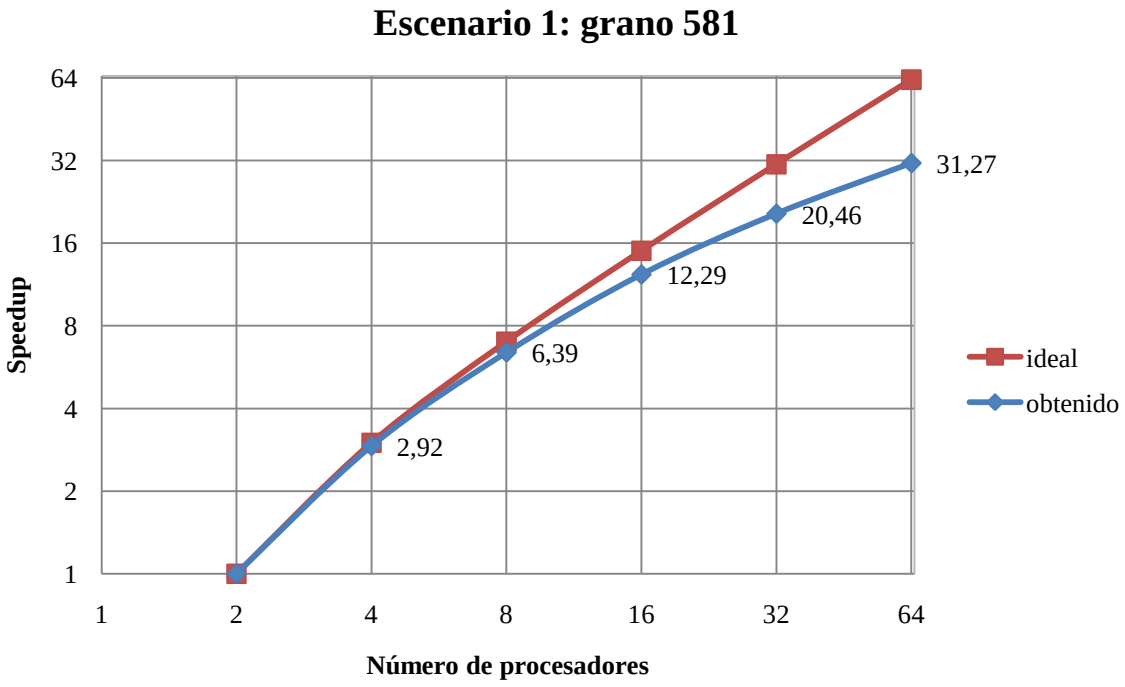


Ilustración 49.- Speedup para un grano 581 en el escenario 1

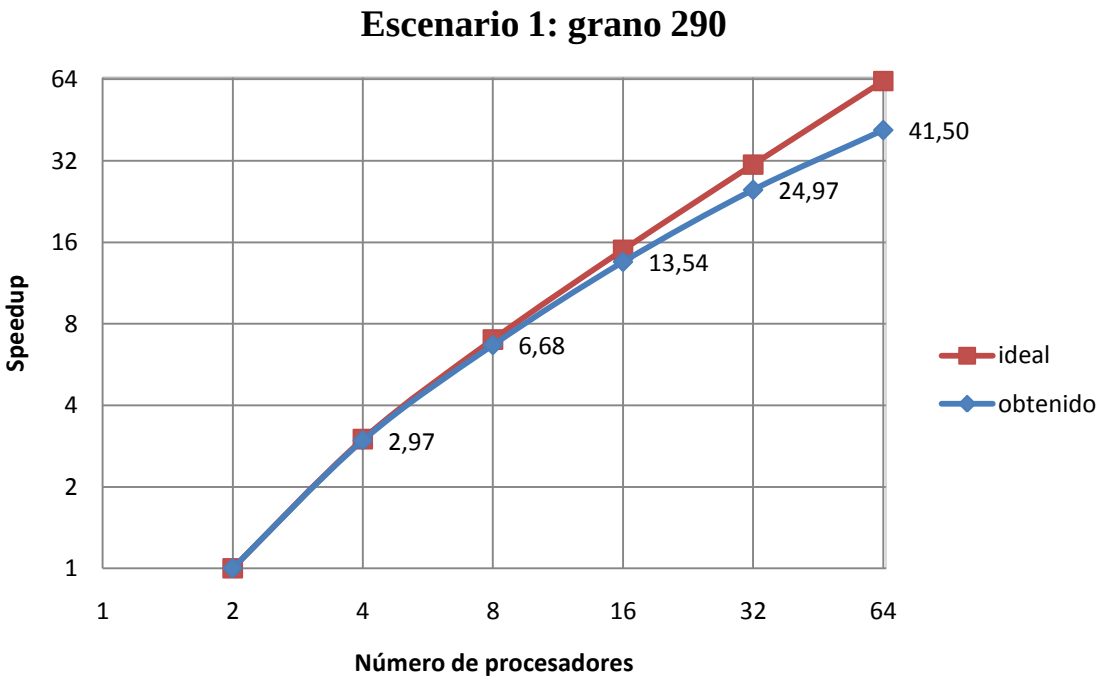


Ilustración 50.- Speedup para un grano 290 en el escenario 1

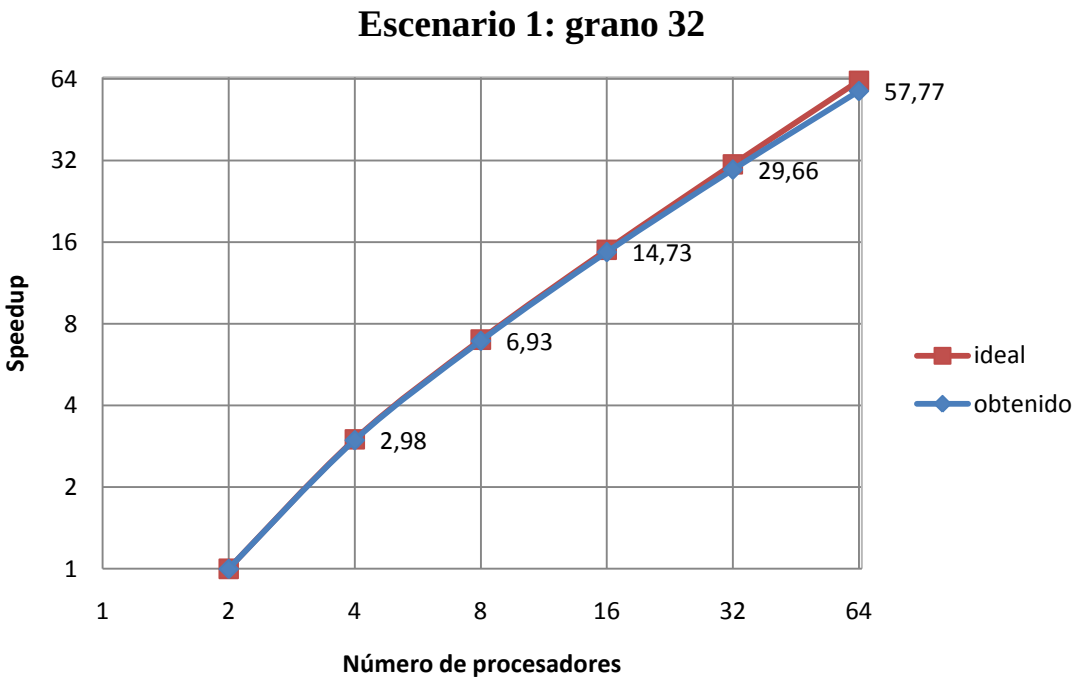


Ilustración 51.- Speedup para un grano 32 en el escenario 1

Nótese que para las ejecuciones con un grano mayor la aceleración es menor que para el caso del grano más pequeño. Por ejemplo, en una ejecución con 64 procesos, un grano de valor 581 arroja un *speedup* de 31,27 mientras que para un grano 32, este toma un valor de 57,77, mucho más interesante.

5.4.2 Rendimiento del escenario 2

Las conclusiones del análisis del rendimiento en el escenario 2 son una copia de las obtenidas para el escenario 1, por lo que se presentarán los tiempos de ejecución (que obviamente, son mayores que los del escenario 1 puesto que la cantidad de datos a procesar es el doble de grande), las gráficas y unos pequeños comentarios al respecto de las mismas.

Los mejores tiempos los registra la ejecución con 64 procesos y un grano de 32, que ha tardado 1.064, 25 segundos o 17,74 minutos en terminar.

Escenario 2 (65.250 registros)				
Nº PROC	**GRANO**	**TIEMPO EJECUCIÓN (s)**	**SPEEDUP**	**SPEEDUP IDEAL**
2 (secuencial)	1019	62.395,513470	1,000000	1
4	1019	21.160,654278	2,948657	3
8	1019	9.332,835411	6,685590	7
16	1019	4.635,615577	13,460028	15
32	1019	2.555,342529	24,417671	31
64	1019	1.606,227880	38,845991	63
2 (secuencial)	509	62.254,861190	1,000000	1
4	509	20.987,407417	2,966296	3
8	509	9.230,345635	6,744586	7
16	509	4.457,332535	13,966842	15
32	509	2.331,186289	26,705228	31
64	509	1.404,911345	44,312306	63
2 (secuencial)	32	62.255,862718	1,000000	1
4	32	20.898,950519	2,978899	3
8	32	8.988,829117	6,925915	7
16	32	4.244,947727	14,665873	15
32	32	2.090,799527	29,776103	31
64	32	1.064,349689	58,491926	63

Tabla 16.- Tiempos de ejecución, speedup y eficiencia para el escenario2

Del mismo modo, el mejor registro de la aceleración lo marcan las ejecuciones realizadas con un tamaño de grano pequeño (32), y las peores, el grano más grande utilizado (1019).

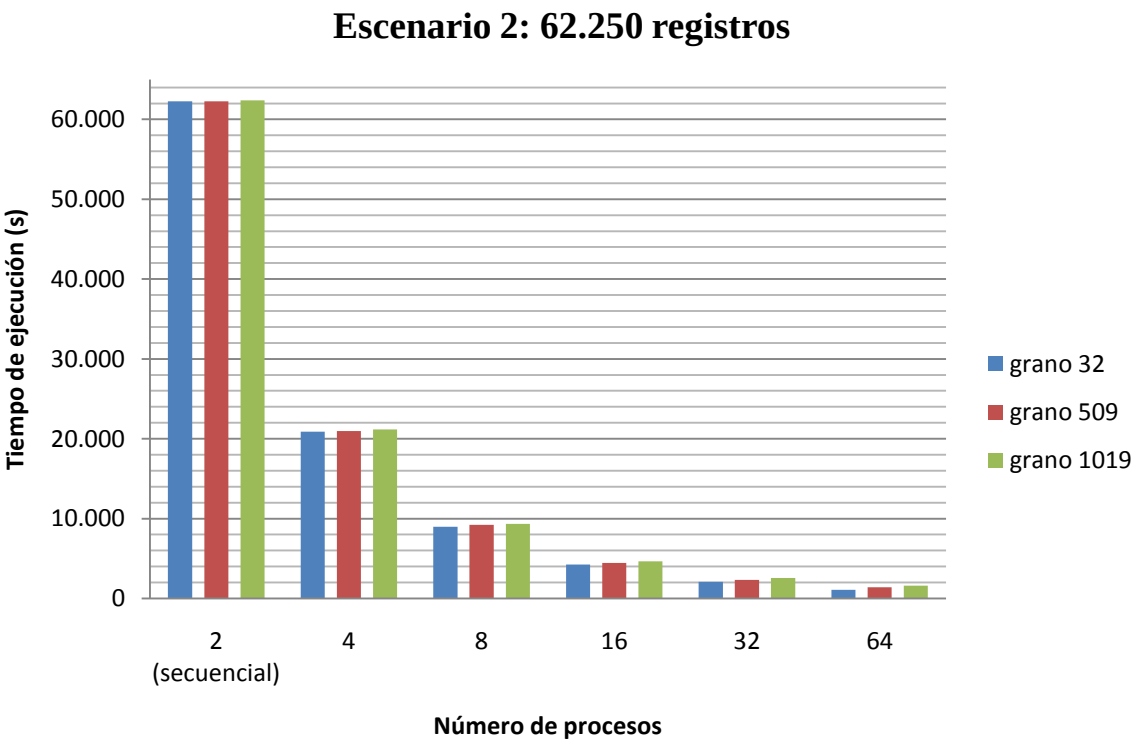


Ilustración 52.- Gráfica para los tiempos de ejecución del escenario2

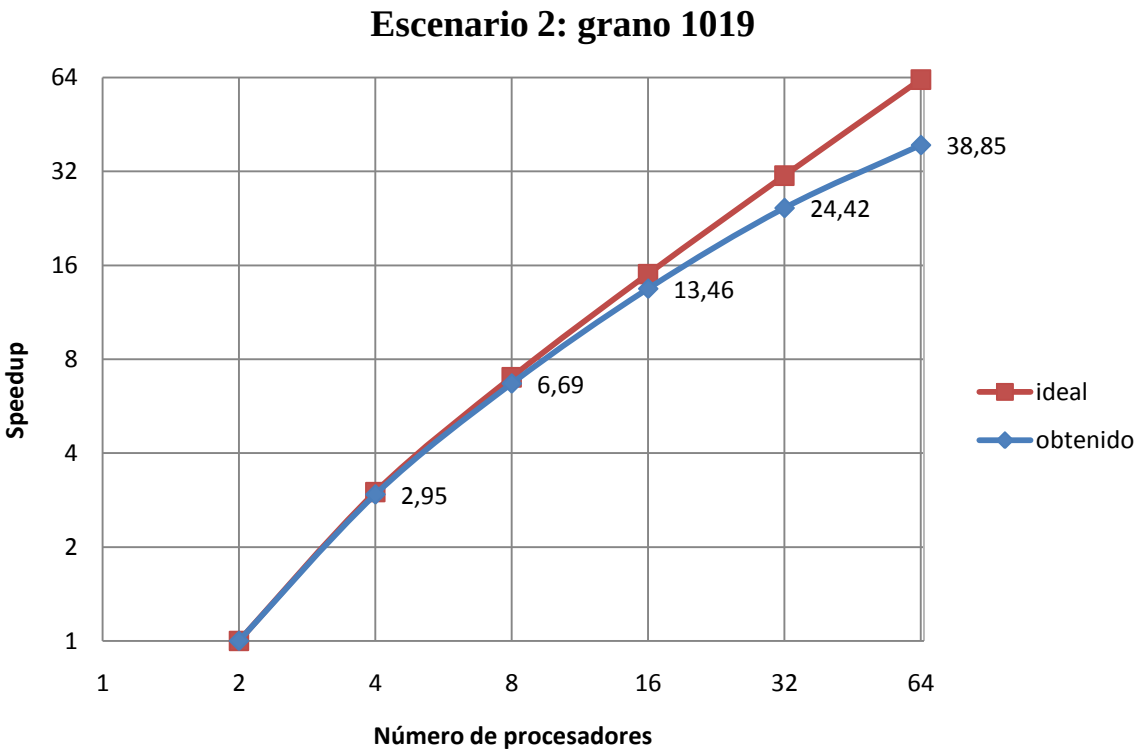


Ilustración 53.- Speedup para un grano 1019 en el escenario2

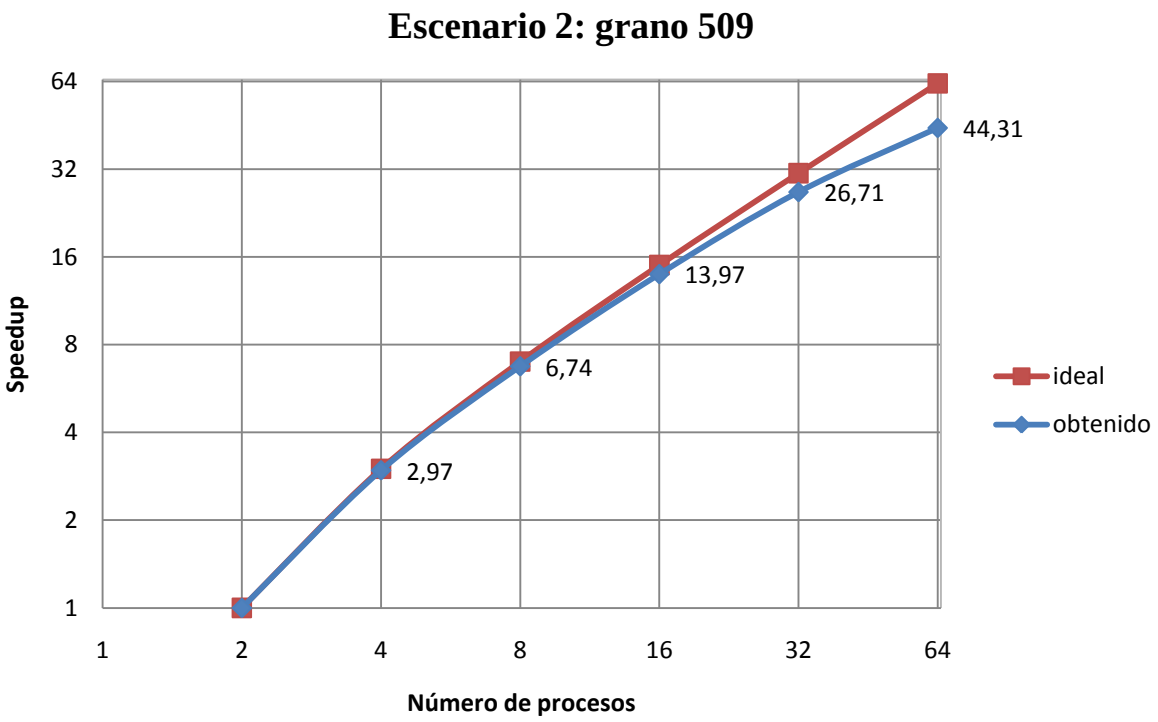


Ilustración 54.- Speedup para un grano 509 en el escenario2

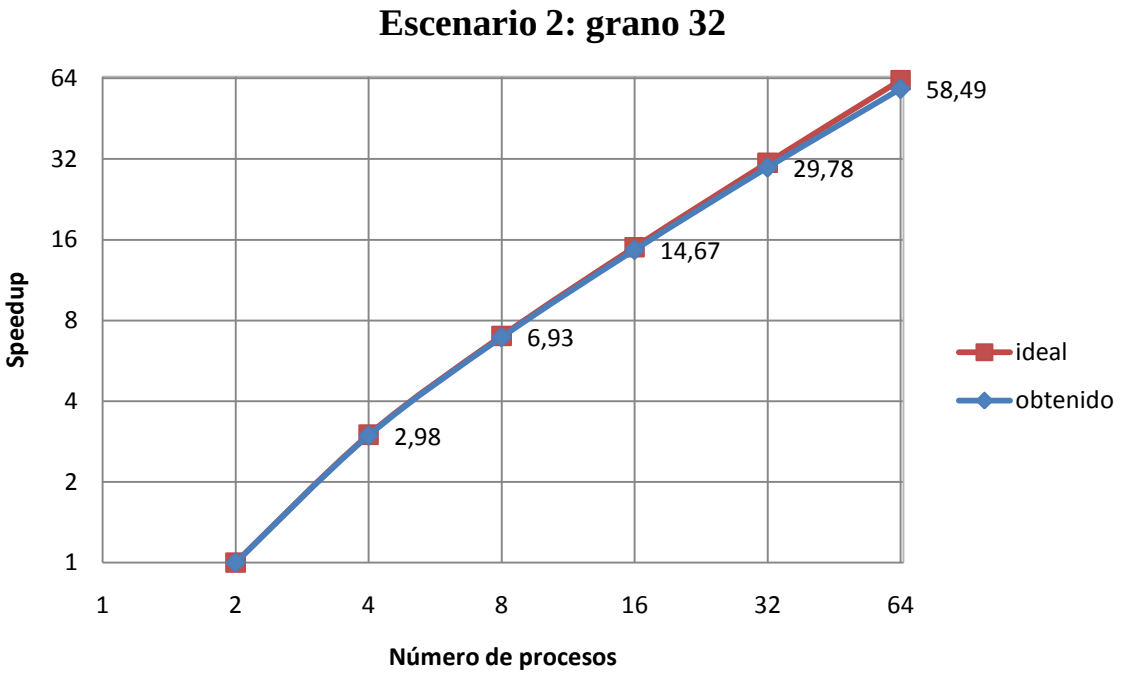


Ilustración 55.- Speedup para un grano 32 en el escenario2

6 CONCLUSIONES Y PRESUPUESTO

En este capítulo se revisan los objetivos planteados al inicio del proyecto, así como del grado de cumplimiento de los mismos. Por último se proponen futuras mejoras y líneas de trabajo.

Además se comentará la metodología seguida durante la realización del proyecto, así como la planificación y el presupuesto final para la realización del mismo.

6.1 Conclusiones

El trabajo presentado ha cumplido con todos los objetivos y expectativas previstas inicialmente. Se ha llegado a analizar y estudiar la geolocalización de la red social Twitter, haciendo uso de programación paralela para reducir tiempo de procesamiento.

Los objetivos cumplidos son los siguientes:

- Se ha implementado un sistema capaz de capturar y analizar información de la red social Twitter de forma masiva.
- Se ha usado el API de Twitter para capturar distintos focos de información.
- Se ha procesado una gran cantidad de datos haciendo uso de MPI como paradigma de paso de mensajes.
- Se han podido sacar conclusiones de los resultados obtenidos.
- Se ha demostrado que la solución paralela obtiene un rendimiento muy similar al rendimiento ideal.

6.2 Líneas futuras de trabajo

Este proyecto puede ser ampliamente mejorado y ser fuente de futuros trabajos de investigación. Como trabajo previo se considera necesario tanto buscar una mejora en el rendimiento de los módulos recolector y procesador, como en la eficacia para encontrar las localizaciones a través de la creación de nuevos métodos de búsqueda más complejos y completos.

Se proponen las siguientes líneas de trabajo:

- Aumentar la precisión de las búsquedas para evitar el problema que se tiene cuando ciudades diferentes comparten el mismo nombre. La búsqueda debería tratar de ubicar a la ciudad en el país al que pertenece, obteniendo así resultados más precisos y fiables.
- Mejorar las búsquedas de ciudades dentro de los *tweets* con el fin de evitar los falsos positivos que arrojan aquellas ciudades cuyos nombres tienen significado propio en la lengua inglesa. La técnica de añadir una preposición delante de la ciudad a buscar trata de evitar este problema, pero podrían idearse patrones más complejos, como por ejemplo, patrones.
- En principio, los módulos están diseñados para buscar y procesar únicamente ciudades, pero sientan las bases para que sea sencillo añadir búsquedas de otros conjuntos geográficos, como barrios o países. Se podrían por tanto realizar búsquedas de diferentes ámbitos geográficos (con granos más finos o más gruesos), con el propósito de obtener más y mejor información partiendo de un mismo conjunto de datos.
- Los datos que genera el módulo procesador se escriben como texto plano, que hace que los datos sean 'poco visuales'. Tampoco existe un agrupamiento de las ciudades encontradas, por ejemplo, por países o continentes. Una ampliación de este desarrollo podría dibujar sobre un mapamundi los datos recogidos, haciendo uso de, por ejemplo, el API de Google Maps o el proyecto 'The WebGL Globe'.

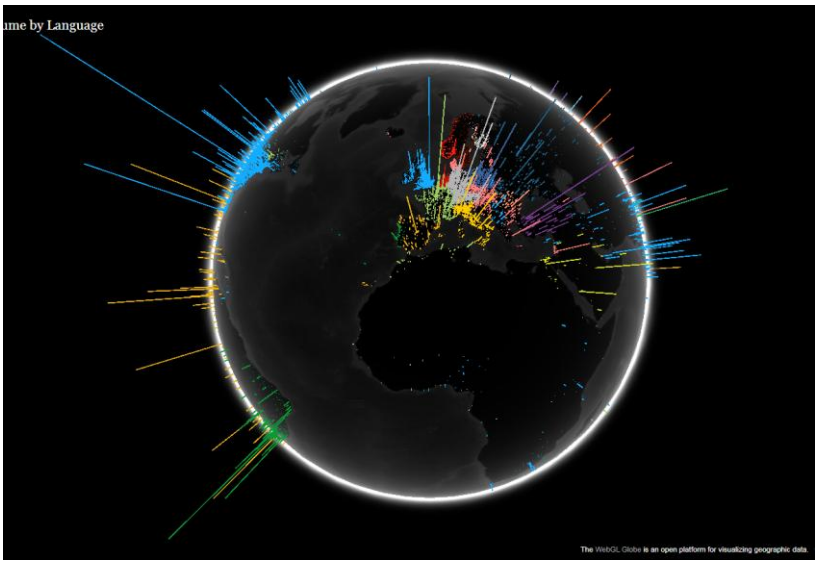


Ilustración 56.- Ejemplo del proyecto 'The WebGL Globe'

- Mejorar el significado de los resultados obtenidos utilizando para ello otros datos auxiliares, como por ejemplo, conocer cómo se distribuye la población mundial, cuáles son las cifras de los habitantes que pueblan los diferentes países y ciudades o conocer la cantidad de usuarios de Twitter en los diferentes países.
- Finalmente se propone el uso de otras herramientas de paralelización como Map-Reduce y Hadoop, las cuales encajan perfectamente con el problema resuelto en este Proyecto Fin de Carrera.

6.3 Método de trabajo

Para la realización del presente Proyecto Fin de Carrera se ha utilizado un ciclo de vida incremental. La elección de este ciclo de vida ha resultado inherente al proceso surgido durante el desarrollo del mismo, especialmente durante el desarrollo del módulo recolector. Puesto que la documentación de Twitter no era demasiado completa, hubo que realizar pruebas de 'ensayo y error', que daban a lugar a numerosas iteraciones con productos que, progresivamente, iban teniendo mayores funcionalidades y que refinaban y mejoraban los resultados obtenidos en la iteración inmediatamente anterior.

Con este método, cada versión o fase es un sistema funcional capaz de realizar progresivamente la función deseada, y permite validar el sistema a medida que se construye.

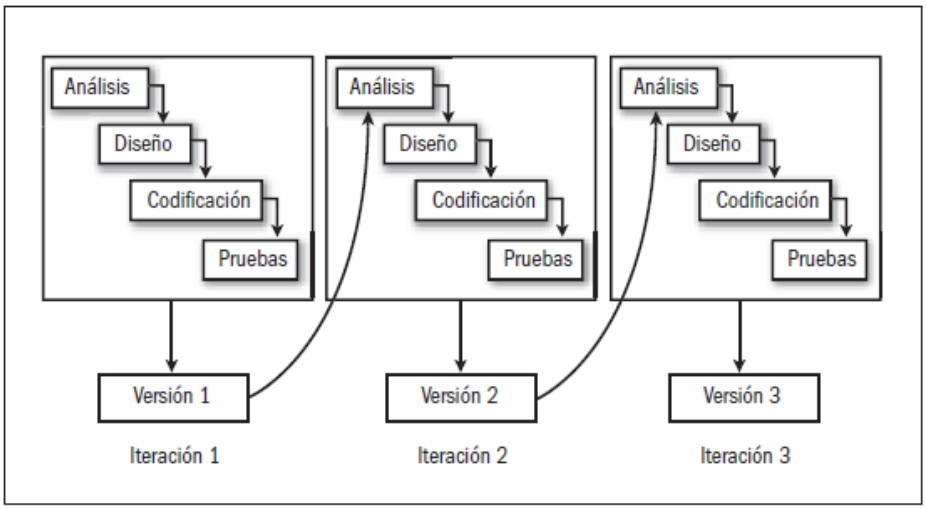


Ilustración 57.- Ciclo de vida incremental

6.4 Presupuesto

Para los cálculos de los costes se han tenido en cuenta las siguientes consideraciones:

- La fecha de inicio del proyecto se establece el 5 julio del año 2010 y la fecha de finalización prevista es del 17 de junio del 2011.
- Existe un periodo vacacional desde el 18 de septiembre al 11 de octubre, ambos inclusive, lo que hacen un total de 235 días de trabajo.
- La jornada laboral comprende de lunes a viernes, 5 horas diarias.
- El total de horas trabajadas asciende por tanto a 1.175.

6.4.1 Cálculo de personal

En el desarrollo de este Proyecto Fin de Carrera ha estado involucrada una única persona: la autora del mismo. Será por tanto la que asuma todos los roles dentro del desarrollo. La siguiente tabla resume el coste asociado al personal:

Fase	Rol	Coste por hora (€ / h)	Total horas	Total coste (€)
Estudio previo	Analista	60 € / hora	200	12.000
Análisis	Analista	60 € / hora	200	12.000
Diseño	Analista	60 € / hora	200	12.000
Codificación	Programador	30 € / hora	400	12.000
Pruebas	Programador	30 € / hora	100	3.000
Documentación	Analista	60 € / hora	75	4.500
TOTAL			**1.175**	**55.500**

Tabla 17.- Costes de personal

6.4.2 Costes de equipo y tecnología

Ha sido necesaria la utilización de un ordenador a lo largo de todo el desarrollo, así como 8 equipos de gama alta que conforman el *cluster* para la realización de las pruebas. El coste se calcula mediante la amortización de todos los *items* en el plazo de

uso de los mismos (que es idéntico a la duración del proyecto). La fórmula utilizada puede verse en el documento de rúbrica del apartado 6.4.4 .

Concepto	Coste (€)	Unidades
Ordenador personal	700	1
Equipos gama alta (*cluster*)	300	8
TOTAL	**404,55**	

Tabla 18.- Costes de equipo y tecnología

6.4.3 Costes de software

En el ordenador personal utilizado para el desarrollo el Sistema Operativo utilizado ha sido Windows 7, instalado bajo una licencia gratuita para estudiante. Los nodos del clúster operan bajo una distribución gratuita de Linux: Ubuntu Server. El código se ha desarrollado en un editor de texto de libre distribución y su compilación, a través de gcc, siendo ambos gratuitos. El presente documento se ha elaborado con la Suite Office en su versión 2007.

Concepto	Coste (€)	Unidades
S.O nodos del *cluster* (Ubuntu Server 10.10)	0	8
S.O ordenador personal (Win7 licencia estudiante)	0	1
Suite ofimática Office 2007	100	1
TOTAL	**100**	

Tabla 19.- Costes de software

6.4.4 Coste total del proyecto

Atendiendo a los supuestos anteriores, el presupuesto del proyecto será el siguiente:

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:

Beatriz Rodríguez Ruiz

2.- Departamento:

Arquitectura y Tecnología de Computadores

3.- Descripción del Proyecto:

- Titulo: **Adquisición y análisis de información geolocalizable de una red social, apoyada en procesamiento paralelo**
- Duración (meses): **7,83**

Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

67.214 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Rodríguez Ruiz, Beatriz	DNI1	Analista	5,14	7.874,40	40.500,00	
Rodríguez Ruiz, Beatriz	DNI1	Programador	3,81	3.937,00	15.000,00	
					0,00	
					0,00	
					0,00	
		Hombres mes	**8,95**	**Total**	55.500,00	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador personal	700,00	100	8	60	91,35
Clúster	2.400,00	100	8	60	320,00
		100		60	0,00
		100		60	0,00
		100		60	0,00
					0,00
				Total	411,35

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Licencia Office 2007	Microsoft	100,00
	Total	100,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas,

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	55.500
Amortización	411
Subcontratación de tareas	0
Costes de funcionamiento	100
Costes Indirectos	11.202
Total	67.214

Tabla 20.- Coste total del proyecto en la plantilla de rúbrica

El coste total del proyecto asciende a **SESENTA Y SIETE MIL DOSCIENTOS CINCO CON CUARENTA Y SEIS EUROS (67.205,46 €).**

Concepto	Total coste (€)
Coste de personal	55.500
Coste de equipo y tecnología	404,55
Costes de software	100
Costes indirectos	11.200,91
TOTAL	**67.205,46**

Tabla 21.- Desglose y resumen del coste total del proyecto

Leganés a 17 de junio de 2011

El ingeniero proyectista

Fdo. Beatriz Rodríguez Ruiz

7 BIBLIOGRAFÍA

1. *Social Networks, Gender and Friending: An Analysis of MySpace Member Profiles.* **Thelwall, Mike.** 2008, Journal of the American Society for Information Science and Technology.

2. *Data Mining Emotion in Social Network Communication: Gender differences in MySpace.* **Thelwall, Mike, Wilkinson, David y Uppal, Sukhvinder.** 2009, Journal of the American Society for Information Science and Technology.

3. **Strano, Michele M.** User Descriptions and Interpretations of Self-Presentation through Facebook Profile Images. *Cyberpsychology.* [En línea] 2008. [Citado el: 23 de 10 de 2010.] http://www.cyberpsychology.eu/view.php?cisloclanku=2008110402&article=%28search%20in%20Issues%29.

4. *Measuring User Influence in Twitter: The Million Follower Fallacy.* **Cha, Meeyoung, y otros.** 2010.

5. **Clark, Sarah.** Police use facebook to identify weapon carriers. *The Journal Online.* [En línea] 11 de 2 de 2009. [Citado el: 23 de 10 de 2010.] http://www.journal-online.co.uk/article/5410-police-use-facebook-to-identify-weapon-carriers.

6. **Carra, Nicholas.** The Daily Campus. [En línea] 16 de 02 de 2007. [Citado el: 05 de 12 de 2010.] http://www.dailycampus.com/2.7440/two-arrests-made-1.1058550?pagereq=1.

7. **Wikipedia.** Geolocalización. *Wikipedia.* [En línea] [Citado el: 2011 de junio de 12.] http://es.wikipedia.org/wiki/Geolocalizaci%C3%B3n.

8. **Seguel, Diego Cerda.** El mundo según Google. Google Earth y la creación del dispositivo geosemántico global. [En línea] 9 de 2005. [Citado el: 12 de 6 de 2011.] http://sites.google.com/site/geosemanticagearth/.

9. **Bracero, Francesc.** La Vanguardia.com. [En línea] 11 de 3 de 2011. [Citado el: 12 de 6 de 2011.] http://www.lavanguardia.com/internet/20110311/54124889951/la-geolocalizacion-se-implanta-en-todos-los-ambitos.html.

10. *Social Network Sites: Definition, History, and Scholarship.* **Boyd, Danah M. y Ellison, Nicolle B.** 2008, Journal of Computer-Mediated Communication, Vol. 13, págs. 210 - 230. http://onlinelibrary.wiley.com/doi/10.1111/j.1083-6101.2007.00393.x/full.

11. **Twitter Support.** Cómo seguir a alguien. *¿Qué es Twitter?* [En línea] [Citado el: 13 de 09 de 2010.] http://support.twitter.com/groups/31-twitter-basics/topics/108-finding-following-people/articles/108082-c-xf3-mo-seguir-a-alguien.

12. **Twitter.** Sobre nosotros. [En línea] [Citado el: 11 de 11 de 2010.] http://twitter.com/about.

13. **Twitter Support.** ¿Cuales son los números de teléfono de Twitter? *Enviar actualizaciones de Twitter a través de mensajes de texto.* [En línea] [Citado el: 11 de 11 de 2010.] http://support.twitter.com/articles/76621-xbf-cuales-son-los-n-xfa-meros-de-tel-xe9-fono-de-twitter.

14. **Twitter Help Center.** ¿Qué significa seguir a alguien? [En línea] [Citado el: 12 de 11 de 2010.] http://support.twitter.com/groups/31-twitter-basics/topics/146-new-twitter/articles/248737-xbf-qu-xe9-significa-seguir-a-alguien-nuevotwitter.

15. **Twitter Bussiness.** Learn the lingo. [En línea] [Citado el: 11 de 11 de 2010.] http://business.twitter.com/twitter101/learning.

16. **Twitter Support.** Help Resources/Getting Started. *About the Tweet With Your Location Feature.* [En línea] [Citado el: 15 de 09 de 2010.] http://twitter.zendesk.com/entries/78525-about-the-tweet-with-your-location-feature.

17. —. Help Resources / Twitter Support- ¡en español! *Sobre los Tweets con Ubicación.* [En línea] [Citado el: 15 de 01 de 2011.] http://twitter.zendesk.com/entries/340682-sobre-los-tweets-con-ubicacion.

18. **Twitter Developers.** API Overview. *The Twitter API.* [En línea] [Citado el: 13 de 09 de 2010.] http://dev.twitter.com/pages/api_overview.

20. —. @raffi's intro to developing with @twitterapi. *Intro to developing for @twitterapi.* [En línea] [Citado el: 15 de 09 de 2010.] http://dev.twitter.com/pages/intro-to-twitterapi.

21. —. GET statuses/show/:id. [En línea] [Citado el: 15 de 09 de 2010.] http://dev.twitter.com/doc/get/statuses/show/:id.

22. —. Rate Limiting. [En línea] [Citado el: 18 de 10 de 2010.] http://dev.twitter.com/pages/rate-limiting.

23. **Twitter Support.** Política de usurpación de identidad. [En línea] [Citado el: 29 de 05 de 2011.] http://support.twitter.com/articles/72692-pol-xed-tica-de-usurpaci-xf3-n-de-identidad.

24. **OAuth.** OAuth.net. [En línea] [Citado el: 14 de 09 de 2012.] http://oauth.net/.

26. **OAuth Core 1.0.** [En línea] [Citado el: 13 de 11 de 2010.] http://oauth.net/core/1.0/.

27. **Hueniverse.** The Authoritative Guide to OAuth 1.0. *Terminology.* [En línea] [Citado el: 14 de 09 de 2010.] http://hueniverse.com/oauth/guide/terminology/.

28. **Hammer-Lahav, Eran.** The OAuth 1.0 Protocol. *IETF Documents.* [En línea] [Citado el: 14 de 09 de 2010.] http://tools.ietf.org/html/rfc5849.

29. **Internet Engineering Task Force (IETF).** Request for Comments: 5849. *The OAuth 1.0 Protocol.* [En línea] [Citado el: 13 de 11 de 2010.] http://tools.ietf.org/html/rfc5849.

30. **Hammer-Lahav, Eran.** Introducing OAuth 2.0. [En línea] [Citado el: 13 de 11 de 2010.] http://hueniverse.com/2010/05/introducing-oauth-2-0/.

31. **Reddy, Rama y Ziegler, Carol.** *C PROGRAMMING for Scientists and Engineers WITH APPLICATIONS.* s.l. : Jones & Bartlett Learning, 2009.

32. **JSON.** JSON.org. *Introducing JSON.* [En línea] [Citado el: 13 de 10 de 2010.] http://www.json.org/.

33. **Quinn, Michael J.** *Parallel programming in C with MPI and OpenMP.* s.l. : Mc Graw Hill, 2004. ISBN 007-123265-6.

34. **Almasi, G. S. y Gottlieb, A.** *Highly Parallel Computing.* 1989.

35. **Culler, David E., Singh, Jaswinder Pal y Gupta, Anoop.** *Parallel Computer Architecture. A Hardware/Software Approach.* San Francisco, CA : Morgan Kaufmann, 1999.

36. **Chevance, René J.** *Server Architectures: Multiprocessors, Clusters, Parallel Systems, Web Servers, Storage Solutions.* s.l. : Digital Press, 2004.

37. **MPI forum.** MPI: A Message-Passing Interface Standard. Version 2.2. *Chapter 1. Introudction to MPI. 1.1 Overview and Goals.* [En línea] [Citado el: 13 de 10 de 2010.] http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

39. **cJSON.** cJSON. *Sourceforge.* [En línea] [Citado el: 18 de 10 de 2010.] http://sourceforge.net/projects/cjson/.

40. **liboauth.** liboauth. *Introduction.* [En línea] [Citado el: 11 de 11 de 2010.] http://liboauth.sourceforge.net/index.html.

41. **libcurl - the multiprotocol file transfer library.** libcurl - the multiprotocol file transfer library. [En línea] [Citado el: 26 de 10 de 2010.] http://curl.haxx.se/libcurl/.

42. **cURL and libcurl.** Companies Using curl in Commercial Environments. [En línea] [Citado el: 10 de 26 de 2010.] http://curl.haxx.se/docs/companies.html.

43. —. libcurl Bindings. [En línea] [Citado el: 26 de 10 de 2010.] http://curl.haxx.se/libcurl/bindings.html.

44. **MPICH2.** About MPICH2. [En línea] [Citado el: 13 de 10 de 2010.] http://www.mcs.anl.gov/research/projects/mpich2/about/index.php?s=about.

45. **Argonne National Laboratory.** Performance Visualization for Parallel Programs. *Download. MPE (MPI Parallel Environment).* [En línea] [Citado el: 13 de 04 de 2011.] http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm.

46. —. MPE. [En línea] [Citado el: 13 de 14 de 2011.] http://www.mcs.anl.gov/research/projects/mpi/www/www4/MPE.html.

47. **Zaki, Omer, y otros.** Toward Scalable Performance Visualization with Jumpshot. *Introduction.* [En línea] [Citado el: 13 de 04 de 2011.] http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/jumpshot-2/node1.html#Node1.

48. **Argonne National Laboratory.** Performance Visualization for parallel programs. *Viewers. Jumpshot. Jumpshot-4.* [En línea] [Citado el: 13 de 04 de 2011.] http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm.

49. **Twitter Developers.** OAuth FAQ. *Technical.* [En línea] [Citado el: 18 de 10 de 2010.] http://dev.twitter.com/pages/oauth_faq.

50. —. Authenticating Requests with OAuth. *Introduction to OAuth.* [En línea] [Citado el: 14 de 09 de 2010.] http://dev.twitter.com/pages/auth.

51. —. Using one access token with OAuth. *Have a single-user use case?* [En línea] [Citado el: 18 de 10 de 2010.] http://dev.twitter.com/pages/oauth_single_token.

55. —. Documentation. *Account Resources: GET account/rate_limit_status.* [En línea] [Citado el: 16 de 11 de 2010.] http://dev.twitter.com/doc/get/account/rate_limit_status.

56. **Geobytes.** Geobytes. *Geobytes.* [En línea] [Citado el: 28 de 01 de 2011.] http://www.geobytes.com/.

57. **Wikipedia.** Wikipedia. *Wikipedia*. [En línea] [Citado el: 28 de 01 de 2011.] http://es.wikipedia.org/wiki/Hideo_Kojima.

58. —. Lady Gaga. *Wikipedia*. [En línea] [Citado el: 13 de 04 de 2011.] http://es.wikipedia.org/wiki/Lady_Gaga.

59. **Twitaholic.** http://twitaholic.com/. [En línea] [Citado el: 13 de 04 de 2011.] http://twitaholic.com/.

ANEXO I: MANUAL DEL USUARIO

Este pequeño manual contiene la información necesaria para ejecutar la aplicación presentada, incluyendo los requisitos software y hardware, y una descripción de los principales elementos.

La aplicación se divide en dos módulos claramente diferenciados: el módulo recolector (*gather*) y el módulo procesador (*processor*). Cada uno de ellos es ejecutado de forma independiente y tiene sus propios requisitos.

El módulo recolector podrá ejecutarse bajo un Sistema Operativo Linux, introduciendo en la consola la línea:

```
$> ./recolector <seed_screenname> <profiles_filename>
<tweets_filename> <max_tweets> <relationship_deep> <complete>
```

O bien:

```
$> ./recolector <seed_screenname> <profiles_filename>
<tweets_filename> <max_tweets> <relationship_deep> <partial>
<max_followers_on_each_level>
```

Los parámetros a introducir dependen de si se desea llevar a cabo una ejecución completa (explorar todos los nodos de cada nivel) o parcial (explorar n nodos de cada nivel). A continuación se ofrece una descripción de los parámetros:

`<seed_screenname>`: nombre de usuario en Twitter a usar como raíz.

`<profiles_filename>`: nombre del fichero de texto que contendrá la información de los perfiles de los usuarios procesados. Su formato debe ser .txt, como por ejemplo, "perfiles.txt". Si no existe, se creará. Si existe, se sobrescribirá.

<tweets_filename>: nombre del fichero de texto que contendrá la información de los tweets de los usuarios procesados. Su formato debe ser .txt, como por ejemplo, "tweets.txt". Si no existe, se creará. Si existe, se sobrescribirá.

<num_tweets>: número de *tweets* que se recogerán por cada usuario procesado. Su valor debe estar entre 1 y 200, ambos inclusive.

<relationship_deep>: profundidad a avanzar en las relaciones de *following* del usuario raíz. Una profundidad de 1 indicará que la población a estudiar será aquella formada por los followers del usuario raíz; una profundidad de 2, formará una población compuesta por los *followers* del usuario raíz y los *followers* de los *followers* del usuario raíz. Así sucesivamente.

<complete> / <partial>: indica el tipo de ejecución a realizar. Una ejecución completa obtiene los datos de todos los usuarios de cada nivel (tantos como se haya especificado en el parámetro <relationship_deep>). Una ejecución parcial explorará en cada nivel un número determinado de usuarios. Este número se define en <max_followers_on_each_level>.

<max_followers_on_each_level>: sólo se emplea en una ejecución de tipo parcial. Permite indicar el número de usuarios de los que se deben recoger los datos en cada nivel. Debe ser un número mayor que 0.

Si prefiere compilar el código deberá disponer de un compilador de C, por ejemplo, gcc, e instalar la librería liboauth. El *parser* cJSON se incluye como parte del código fuente. Deberá introducir por consola la siguiente línea:

```
$> gcc -Wall -Werror gather.o gatherlib.o cJSON/cJSON.c -o
recolector -lssl -loauth -lcurl -lm
```

Si la librería liboauth no se encuentra en el directorio por defecto (/usr/lib), utilice el *flag* -I para indicar su ruta.

Se aconseja que la máquina sea lo más potente posible y con una máxima disponibilidad de memoria principal y secundaria: cuanto mayor sea la población que desee estudiar, mayores exigencias existirán a nivel de consumo memoria.

No olvide que debido al límite de 350 peticiones por hora a la API de Twitter, el módulo recolector debe mantenerse en ejecución durante un tiempo prologado que depende principalmente del tamaño de la población a recoger. A modo de pauta, considere que por cada usuario que forma un nodo hoja (es decir, un usuario sobre el que no hay que buscar sus followers) se realizan 2 peticiones a la API; por cada usuario no hoja se realizan, en el **mejor** caso, 3 peticiones a la API.

Debido a esta limitación se recomienda que controle los valores de entrada puesto que el crecimiento de la población al aumentar la profundidad de exploración (el parámetro <relationship_deep>) es exponencial. Controle y conozca el número

de followers que tiene el usuario raíz <seed_screnname>, puesto que también es un factor que puede acotar (o ampliar) el tamaño de la población a recoger.

Utilice también las pautas dadas en el apartado 4.2.1 sobre cómo conocer el tamaño de las poblaciones a recoger.

El módulo procesador debe ejecutarse bajo un Sistema Operativo Linux. Para su ejecución, no es necesaria la utilización de un clúster, sino que basta con que la librería MPICH esté disponible. Sin embargo, para obtener el máximo potencial de la ejecución lo ideal es hacerlo en una arquitectura de memoria distribuida con un sistema PBS.

Para ejecutar la aplicación en un PBS, se incluye un *script* en el que se pueden configurar los parámetros:

```
#PBS -N myJob
```

- Asigna un nombre al trabajo. El nombre por defecto es el nombre del fichero *script*.

```
#PBS -l nodes=4:ppn=2
```

- Configura el número de nodos a usar del *cluster* (nodes) y el número de procesadores por nodo (ppn). ppn no es de uso obligatorio.

```
#PBS -l walltime=01:00:00
```

- Tiempo máximo de ejecución que requiere el trabajo. Si este tiempo se supera, el trabajo es eliminado automáticamente de la cola.

```
#PBS -o mypath/my.out
```

- Ruta del fichero que contendrá la salida estándar (STDOUT).

```
#PBS -e mypath/my.err
```

- Ruta del fichero que contendrá la salida estándar de error (STDERR).

El *script* también incluye los comandos que lanzan la aplicación. Sus parámetros pueden ser también modificados:

```
mpdboot -v -n <nnodes> -f $PBS_NODEFILE
```

- Inicializa MPICH en el clúster especificado.

`<nnodes>` es la cantidad de nodos del clúster a utilizar. Este valor hace referencia a nodos, independientemente del número de procesadores físicos o *cores* que cada uno de ellos tenga. Por tanto, debe ajustar este parámetro así como el `-l nodes` dependiendo de cómo desee realizar la ejecución (por ejemplo, si prefiere utilizar todos los *cores* de los que dispone un nodo, o prefiere utilizar un solo *core* por nodo).

```
mpiexec -n <nproc> ./processor <piece_of_work_cuantity>
<profiles_filename> <tweets_filename> <cities_filename>
<results_filename>
```

- Inicializa un trabajo paralelo dentro del PBS.

`<nproc>`: número de procesos que se crearán. Este valor debe ser igual o mayor que 2 para que la aplicación funcione correctamente (se necesita como mínimo, un maestro y un esclavo).

`<piece_of_work_cuantity>`: número que especifica el tamaño (en pares de JSON) de la unidad de trabajo de los procesos.

`<profiles_filename>`: fichero de entrada que contiene la información de los perfiles de ciertos usuarios de Twitter en forma de JSON. El fichero debe tener la extensión ".txt". Este fichero es una de las salidas generadas por el módulo recolector.

`<tweets_filename>`: fichero de entrada que contiene la información de los tweets de ciertos usuarios de Twitter en forma de JSON. El fichero debe tener la extensión ".txt". Este fichero es una de las salidas generadas por el módulo recolector.

`<cities_filename>`: fichero de entrada que contiene las ciudades sobre las que se podrán distribuir los usuarios estudiados. Este fichero debe contener una ciudad por cada línea, además de un retorno de carro en la última línea, que estará vacía. Su extensión debe de ser ".txt".

`<results_filename>`: fichero de salida que recogerá los resultados obtenidos por el módulo recolector. Su extensión debe ser ".txt".

```
mpdallexit
```

- Finaliza el entorno MPICH.