UNIVERSIDAD DE CHILE

FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

APLICACIÓN MÓVIL GEORREFERENCIADA DE BÚSQUEDA DE INTERESES EN COMÚN POR MEDIO DE UN SISTEMA DISEÑADO PARA ALTA DEMANDA:

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS JOSÉ BENSAN ARAYA

PROFESOR GUÍA:

NELSON BALOIAN TATARYAN

MIEMBROS DE LA COMISIÓN:

NANCY HITSCHFELD KAHLER

MARIA CECILIA RIVARA ZUÑIGA

SANTIAGO DE CHILE

2014

Resumen:

APLICACIÓN MÓVIL GEORREFERENCIADA DE BÚSQUEDA DE INTERESES EN COMÚN POR MEDIO DE UN SISTEMA DISEÑADO PARA ALTA DEMANDA:

La motivación principal del presente proyecto es invertir la tendencia de aislamiento, egoísmo y soledad que provocan los adelantos tecnológicos, pues fomentan un tipo de relaciones personales más artificial, falso y lejano. El hecho de perder el ritual de tener que reunirse para compartir con los otros enfría las relaciones interpersonales en todos los niveles. El producto desarrollado (desde ahora Near-U) se plantea como una posible solución a dicho problema, teniendo como objetivo el desarrollo de un sistema informático para reunirse y conocer gente en base a nuestros propios intereses.

Para esto se busca diseñar y desarrollar un sistema que permita a sus usuarios contactar con otros que tengan intereses similares y estén a una distancia menor o igual a cierta distancia dada. Por lo que se desarrolló una plataforma cliente – servidor a través de la cual los usuarios anotan una lista de todos sus intereses que deseen compartir. El sistema a su vez sigue los desplazamientos de sus usuarios a través de sus coordenadas geográficas. Y cuando dos usuarios están a una distancia menor o igual a cierta distancia dada, se envía a cada usuario una notificación indicando dicho evento. De este modo el sistema le indica a cada usuario cuando está cerca de otro usuario con los mismos intereses.

Uno de los mayores desafíos fue la comparación de distancia entre cada par de usuarios en el sistema, tarea que requiere un gran poder de cómputo. Para solucionarlo se diseñó una arquitectura de software enfocada en bridar servicios confiables y robustos en escenarios de alta demanda. Dicha arquitectura se compone de una aplicación web siguiendo el modelo de diseño MVT (Modelo – Vista - Template) y un servicio externo para mantener la estructura de datos que permite calcular las distancias entre cada par de puntos de forma eficiente.

Al final del trabajo se puede observar como la estructura de datos espaciales ayuda enormemente a reducir los tiempos de respuesta en contraposición con una solución por fuerza bruta, hasta en un orden de magnitud. Sin embargo no resulta tan útil al momento de reducir la tasa de pérdida de requests al servidor.

Como potenciales mejoras al sistema de Neau-U, se destaca la posibilidad de mantener aquellos datos de uso más frecuente en una memoria de acceso rápido (caché) en el servidor. Otra potencial mejora es el escalamiento horizontal de los servidores de procesamiento (aquellos que calculan las distancias entre cada par de puntos).

Dedicatoria:

Dedico el presente trabajo a mi familia, sin cuyo apoyo y comprensión durante todos estos años no podría haber llegado a las últimas instancias de la carrera de ingeniería. Por esto es que les estoy profundamente agradecido con mi papá Carlos, mi mamá Alejandra, y a mis hermanos Kiki y Juan José. Nunca dejaron de ayudarme con todo lo que tenían con cada tropiezo que tuve.

Agradezco también el enorme apoyo que me brindaron mis amigos Fernando y Marcos. Y al grupo de ayuda de viejones de la Universidad, Raisa, Katherine, Marcela, Ignacio, Rhida, Sebastián, Daniela y Pamela.

Por otro lado también agradezco a las grandes personas que conocí en el departamento, que me enseñaron con hechos el valor del trabajo bien hecho, del sacrificio para superar las adversidades, del compañerismo, y de todos esos momentos infantiles que hacen a uno sentirle un nuevo sabor a la vida. Al AFA, al Jorge, al Rafa, a Pablo, a Felipe, a Diego, etcétera. Sin llegar a conocerlos tanto pude aprender de las grandes personas que son.

Finalmente agradezco a la Sandra por prestarme siempre la corchetera.

Tabla de Contenido:

Índice de Figuras:

1. Introducción

1.1. Motivación:

El trabajo realizado para el presente informe busca desarrollar una aplicación que comunique a sus usuarios, llevando la experiencia de compartir en un "espacio digital" a compartir en la realidad, en base a reunirse con aquellas personas que tengan intereses similares a uno. De este modo se intenta romper uno de las más lamentables consecuencias de las tecnologías modernas, el hecho de que cada vez más la gente se centra en su propio mundo sin compartir ni comunicarse con quien puede estar inmediatamente a lado de uno. De este modo se plantea una solución para comunicarse con quienes uno tenga cerca, mediante el uso de la tecnología.

Quizá uno de los aspectos más interesantes del proyecto asociado al trabajo que pretende ilustrar este informe es el hecho de que se "invierte la ecuación" tradicional que se asocia a las tecnologías móviles, como bien menciona Sherry Turkle en Abril de 2012 [1]. Ella manifiesta que ve a las Tecnologías de la Información como herramientas que han simplificado de enorme manera la forma en cómo las personas se comunican, y en alto acceso a éstas que existe hoy en día. Sin embargo, a pesar de que las Tecnologías de la Información permiten al ser humano comunicarse de manera más cómoda, masiva, económica, y fácil; también han provocado una pérdida de contacto más personal y cercano. Se advierte en las personas una tendencia a formas cada vez más superficiales, rápidas, frías, y altamente efectivistas de comunicarse, resultando así en una exacerbación del individualismo por sobre el sentir colectivo, y se ha dejado de valorar la riqueza del lenguaje corporal y ambiental relativa al mismo proceso de la comunicación.

Pero quizá para entender el cómo nos relacionamos con la tecnología, hay que dirigir la atención a libros como "Second Self" de la misma Sherry [2], que ya en 1984 plantea la tesis de que las computadoras modernas no son sólo herramientas que nos ayudan en nuestros trabajos, sino que son parte de nuestra vida psicológica, capaz de catalizar cambios en lo que hacemos y lo que pensamos. Hoy, treinta años después, se puede ver más claro que nunca el cómo la sucesión de nuevas generaciones de dispositivos electrónicos cada vez más avanzados determinan fuertemente el modo de actuar y el tipo de relaciones de tienen aquellos niños que crecen con estos dispositivos.

Hace treinta años muy poca gente se planteaba usar una máquina de escribir teniendo un computador, mucho más cómodo y rápido. Hace 20 años el uso del walkman permitió a los jóvenes escuchar su música favorita sin tener que "compartirla" con el resto de la gente; ya no era necesario acordar una música de gusto general. Hace 10 años el uso de los SMS y las aplicaciones de chat permitían a los jóvenes comunicarse y coordinarse, saludarse, intentar una conquista, etcétera, sin necesidad de estar físicamente en el mismo lugar. En otras palabras eliminó la necesidad de

reunir a la gente para otras actividades que antes si lo necesitaban. Hoy en día no son pocos los niños de 3 a 8 años que encuentran en la Tablet o el celular de su papá un mundo nuevo que explorar. Mundo que fue reemplazando al “mundo real” por la comodidad que les significa a los padres y la seguridad del pequeño. De este modo cada generación que adopta nuevas tecnologías se ha vuelto cada vez más una generación de hombres introspectivos, egoístas, autorreferentes, y con menos capacidad de crear lazos sociales profundos.

El punto anterior se ha acrecentado dramáticamente los últimos años con la llegada de los smartphones y la conectividad de banda ancha móvil al punto de que actualmente se acuñó el término *nomofobia* [3] para describir la ansiedad que experimentan los usuarios al dejar sus celulares (ya sea porque se acabó la batería, se agotó el saldo, etcétera). Paradójicamente el miedo a desconectarnos de nuestros contactos en las redes sociales no se condice con la calidad de las relaciones que tenemos con quienes compartimos el espacio físico, en donde la misma hiperconectividad nos vuelca a tener relaciones de bajísima calidad y de enlaces sociales de muy poca profundidad.

Es por esto que el proyecto asociado al presente Trabajo de Título se plantea como una forma de invertir esta percepción tan generalizada, usando de manera completamente diferente las Tecnologías de la Información, de manera de diseñar y ofrecer una herramienta que acerque a la gente y que la incentive a conocer y compartir con quienes estén cerca de ella y posean intereses similares. Como fin último el proyecto pretende usar las TIC como una herramienta para crear comunidades y relaciones interpersonales, ya no virtuales, sino que reales.

También cabe mencionar el potencial uso de la plataforma, relacionándola con los campos de la publicidad y el marketing directo. De este modo se podrían crear clientes web móviles que accedan al servidor en cuestión, y soliciten los servicios ofrecidos por las distintas compañías afiliadas en la ubicación particular de cada usuario. Lo anterior tomando en cuenta que existe un programa de marketing conducente a brindar servicios de publicidad al usuario en el contexto de su ubicación. De igual manera se puede tener una valiosa fuente de información para **investigación de mercado geolocalizado** (se sabe qué desea la gente, y en qué lugares surge esa necesidad).

Finalmente es de particular interés para el proyecto el desarrollo de un software el que implementa una solución al problema de **Fixed-Radius Near Neighbor** (los puntos a distancia menor o igual, dado una distancia euclidiana). De este modo, se desarrolló un sistema que permite comparar las distancias entre distintos puntos de forma eficiente, sin necesidad de acudir a la comparación manual de las distancias una a una. Como se menciona más adelante sin embargo, éste es un problema aún contingente, y sin una solución general, aplicable a diversos campos de la ingeniería. Pero de las soluciones seleccionadas para abordar este proyecto, se termina finalmente eligiendo una que ofrece los rendimientos esperados en términos de mejora en la calidad de servicio requerida (tiempos de respuesta del servidor).

1.2. Antecedentes:

1.2.1. Uso de Móviles:

Hoy en día el total de tráfico de internet está siendo generado y consumido cada vez más por dispositivos móviles como tablets y smartphones. Lo que hasta hace un par de años era una tendencia para ser tomada en cuenta a futuro, hoy es parte del presente; y sin duda el desarrollo de emprendimientos digital, tanto extranjeros como de incubación interna, deben tomar en cuenta este factor para adaptar sus productos y servicios a un nuevo tipo de consumidor. Este nuevo consumidor es un usuario conectado las 24 horas del día, siempre en contacto con sus cercanos, y fuertemente habituado a todo tipo de herramientas sociales y de trabajo colectivo (Google Docs, Dropbox, Evernote, etcétera).

A la oferta de smartphones en el mercado nacional, se le debe también adicionar el incremento exponencial de infraestructura por parte de las empresas de telecomunicaciones (desde ahora TELECOs) con el fin de ofrecer servicios de transmisión de datos mediante tecnologías como UTMS (3G), HSDPA y LTE (4G). No es de extrañar entonces el cambio en los hábitos de uso de las tecnologías móviles que han experimentado los chilenos durante los últimos 5 años, pasando desde usar el celular como una extensión del teléfono fijo con pequeñas funcionalidades para asistencia en caso de eventualidades, a usar el smartphone como un verdadero reemplazo del tradicional PC (o Notebook) en donde se tiene acceso a todos los servicios que requiere; sumado a esto la versatilidad y conveniencia que significa poseer todas estas funcionalidades en un dispositivo portable.

Para ilustrar lo anterior se presentan cifras que ayudarán a comprender mejor el panorama en Chile actualmente. Durante el año 2013 se vendieron más de 6.619.000 smartphones en Chile, además de 1.575.000 tablets y 1.792.000 computadores. Por lo que se puede observar que el grueso de los usuarios de internet a nivel nacional acceden a éste a través de un dispositivo móvil (ya sea un smartphone o una tablet); llegando al 79.5% [4]. A la fecha del 30 de Diciembre de 2013, del total de 23.659.441 abonados a telefonía móvil, 9.769.694 de ellos (un 41.29%) hace uso de banda ancha móvil (tecnología HSDPA o superior) [5]. Incluso la prestigiosa multinacional Cisco proyecta que para el año 2017 habrán aproximadamente 16 millones de smartphones en Chile, alcanzando un tráfico anual de 30 Petabytes (esto es 10 veces el tráfico total del año 2012).

Al analizar el tráfico en telecomunicaciones de los chilenos, se vislumbran datos aún más clarificadores de la tendencia al alza de los dispositivos móviles. Según indica el Informe Estadístico Anual 2013 de la Subtel [6], el 73.5% de los accesos a internet en chile se realizan a través de un smartphone.

En cuanto al tipo de usuario al cual apunta el software desarrollado en el presente Trabajo de Título, joven nativo digital entre 15 y 25 años, un 92.4% de ellos

actualmente posee conexión a internet. En este segmento también se puede observar una rápida adopción de las nuevas tendencias en el uso de aplicaciones móviles como son las aplicaciones de mensajería. De este modo se observa que desde el 2011 hasta el 2013 el número de mensajes de texto bajó desde 2012 mil millones hasta 1874 mil millones de mensajes.

Es en este contexto que resulta particularmente interesante el desarrollo de soluciones tecnológicas que se adapten a las condiciones provistas por los "teléfonos inteligentes" en cuanto a las capacidades limitadas de cómputo de dichos aparatos, las limitaciones en el acceso a datos, y las reducidas dimensiones de sus pantallas. Los productos y plataformas a desarrollar deben ser particularmente prolijos en manejar sus datos, el tráfico generado, y los tiempos de espera que ofrecen al usuario, logrando que en todo momento que la experiencia de usuario sea atractiva. Especialmente en las aplicaciones que requieren cálculos de alta complejidad, en las cuales los tiempos de espera pueden perjudicar gravemente su usabilidad.

1.2.2. Uso de Aplicaciones:

En cuanto al uso de las distintas aplicaciones móviles, este aumentó en un 113%, siendo las aplicaciones sociales las que más aumentaron (203%) [7]. A nivel mundial el número de descargas aumentó de 64 billones en 2012 a 102 billones en 2013, y se estima que se llegará a 260 billones para 2017. En cuanto al tráfico de terminales móviles, de los 18 exabytes de 2013 se prevé que se llegará a 190 exabytes para 2018. Por lo que la tendencia de los consumidores a migrar sus sistemas a plataformas móviles no es ni mucho menos un hecho aislado o un peak en cierta tendencia, sino más bien un nuevo direccionamiento de convergencia de los sistemas de comunicación para la próxima década.

1.3. Objetivos:

1.3.1. Objetivo Principal:

El objetivo general del presente Trabajo de Título es la creación de un sistema de procesamiento de cruce entre los intereses de los clientes atendidos, dadas las posiciones geográficas de cada uno de ellos. Dicho sistema debe ser integrado a través de librerías (unidades de software) y algoritmos que permitan brindar un servicio estable a una gran cantidad de usuarios.

1.3.2. Objetivos Específicos:

1. Definición de una arquitectura de Hardware lo suficientemente robusta y estable para atender sobre 10 000 de usuarios, a una tasa de un requerimiento

por usuario de uno cada cinco segundos (2000 de requerimientos por segundo).

2. Implementación y uso de un tipo de datos abstracto (desde ahora TDA) que permita ordenar los clientes de acuerdo a sus locaciones, brindando una solución al problema: Fixed-Radius Nearest Neighbor Search sobre un software de cruce de locaciones entre sus usuarios

3. Definir una métrica y evaluar el rendimiento en tiempo de ejecución y número de fallas de los algoritmos y estructuras de datos utilizadas.

1.4. Boceto de la Solución:

En este proyecto se desarrolló una aplicación para smartphone que permite a sus usuarios la siguiente funcionalidad: mantener en todo momento listas con sus intereses. Al aproximarse a otro usuario con los mismos intereses (similitud lexicográfica entre los intereses respectivos), cada usuario es notificado de dicha similitud con el otro, y tiene la posibilidad de contactarse con éste.

De este modo usuarios con intereses similares, y que estén en cierto radio de cercanía uno de otro, pueden reunirse en un espacio físico, y no únicamente en ambientes virtuales que facilitan la interacción entre las personas dentro de un espacio virtual.

Para hacer una analogía algo burda pero bastante explicativa al punto tratado anteriormente, la solución implementada se asemeja a tener un cartel en todo momento con la lista de los intereses que cada uno desea compartir. Y un mapa con otras personas que tengan un interés similar.

1.5. Alternativas en el Mercado:

Actualmente existen varias alternativas al sistema a desarrollar, pero todas con un foco distinto al de Near-U. Todas las aplicaciones disponibles se enfocan más en los lugares específicos de cada usuario (como Foursquare) o en un segmento particular de usuarios (Tinder en el caso de citas virtuales). Sin embargo, ninguna de las posibles alternativas analizadas da indicio de requerir un poder de procesamiento de las posiciones de cada usuario como Near-U, en donde las coordenadas geográficas de cada usuario son variables en el tiempo.

Sin embargo, conviene realizar una reseña de cada una de las alternativas analizadas de modo de entender mejor el contexto en que se integra Near-U como una nueva forma de recuperar la interacción social a través de la tecnología, y no de un reemplazo de relaciones por encuentros en espacios virtuales.

1.5.1. Foursquare [8]:

Red social orientada al compartir reseña de lugares con tu red de contactos. Acá se pueden dejar comentarios en el lugar en el que uno está, y ver los comentarios de otros. Además de una gran guía de turismo y de salidas, es también una poderosa plataforma de marketing localizado. Miles de marcas, locales de comidas, tiendas, y otras empresas han elegido a Foursquare para desplegar campañas de atracción y fidelización en donde se pide a los usuarios que dejen cierto tipo de comentarios, o realicen ciertas acciones, a cambio de obtener descuentos y/o beneficios.

Al ser fijos los lugares, no tiene sentido que se invierta esfuerzo en calcular con mayor frecuencia las distancias entre los usuarios, por lo que no existe una similitud en cuanto a complejidad de la aplicación, con Near-U.

1.5.2. Gowalla [9]:

Del mismo modo que Foursquare, Gowalla permite a los usuarios de la red realizar Check-In en ciertos lugares, ya sean lugares privados, públicos, o hitos en la ciudad. Mientras más Check-In haga uno, más puntaje acumula. Los puntajes desbloquean ciertas funcionalidades bloqueadas para usuarios más nuevos. La mayor distinción de esta aplicación es la importancia de la jerarquía entre los usuarios con respecto a su escala de puntaje.

1.5.3. Facebook Places [10]:

Más que una nueva aplicación es una nueva funcionalidad añadida a Facebook, en donde se puede adjuntar un lugar a cada publicación. De manera de ver a otros amigos cercanos a dicha ubicación, o incluso los comentarios dejados con respecto a ésta.

Pero no realiza calce de intereses con personas desconocidas ni actualiza las ubicaciones frecuentemente, por lo que no cumple con los objetivos de Near-U.

1.5.4. Tinder [11]:

Red social enfocada en la búsqueda y “vitrineo” de personas cerca de uno, pero mayormente enfocada en establecer relaciones personales entre personas con intereses comunes, en base a sus hábitos en redes como Facebook y el perfil que los mismos usuarios dispongan en la aplicación. Acá el sistema se encarga de recomendar personas cercanas en base el propio perfil del usuario, y si esa persona

también aprueba al usuario, ambos se ponen en contacto a través de la misma aplicación.

Al estar tan enfocada en las relaciones sentimentales, Tinder pierde el foco de que quizás el interés puede ser sólo un item de la lista de intereses de un usuario. Incluso puede tratarse de un interés pasajero (por ejemplo, la compra de una entrada a algún evento) o un interés comercial (comprar o vender un producto). Además se debe notar que la ubicación de los otros usuarios nunca es gravitante ni actualizado con mucha frecuencia, por lo que la gran mayoría del tiempo se estará cotejando con las ubicaciones más frecuentes de cada usuario. Sin embargo, es la alternativa que más se aproxima a la los objetivos de Near-U.

1.5.5. Snoox [12]:

La plataforma de recomendaciones Snoox permite dar consejos y reseñas de prácticamente lo que sea, datos que serán enviados luego cuando un amigo consulte por el tópico el cual mencionaste en alguno de tus consejos. Lo anterior lo ha transformado en una de las mayores plataformas de consejos del mundo.

En este caso los consejos si pueden corresponderse a lugares. Pero al igual que Foursquare, los lugares son estáticos. Por lo que no actualizan su ubicación dinámicamente.

1.6. Contenido

El presente informe está organizado por secciones con tres niveles de anidamiento, siendo la presente la primera sección.

En la segunda sección se detallan los aspectos teóricos y de alto nivel del trabajo desarrollado, y la correspondiente solución de software implementada. Primero se da la descripción del software desarrollado para el servidor. Luego se describe el estudio realizado en las soluciones del problema de Búsqueda de Vecinos Cercano, profundizando los aspectos algorítmicos de la solución escogida por el autor.

En la tercera sección se explica la implementación del software desarrollado, detallando cada uno de sus componentes. Desde la arquitectura general del sistema hasta los algoritmos utilizados en cada etapa del procesamiento, esta sección provee la información necesaria para la total comprensión del sistema antes mencionado.

En la cuarta sección se exponen los resultados obtenidos luego de realizada la simulación de varios usuarios haciendo uso de la plataforma.

Finalmente la quinta sección contiene las conclusiones luego de realizado el análisis de los datos obtenidos.

2. Marco Teórico:

Para el desarrollo del sistema se debieron enfrentar varios desafíos, principalmente en el área del desarrollo de software. Primero diseñar un sistema computacional que diera abasto al gran número de requerimientos que contempla el sistema, de forma robusta, escalable y correcta; y el encontrar un algoritmo que permitiera comparar las distancias entre los puntos de forma rápida y eficiente, de modo de poder satisfacer la capacidad de cómputo requerido por los clientes concurrentes.

A continuación se describen las opciones escogidas para cada desafío.

2.1. Software Servidor:

Para el desarrollo del software del servidor se analizaron diversas opciones de lenguajes y frameworks especializados en este tipo de soluciones. Para no afectar la productividad y la calidad del diseño se optó por soluciones que tuvieran las siguientes características:

- La plataforma deben ser predecible. Deben poseer una correcta interpretación humana de lo que un computador ejecutará.
- La plataforma debe ser consistente. Abstracciones que se ven igual deben hacer lo mismo, abstracciones que se ven diferentes deben hacer cosas distintas.
- La plataforma debe ser concisa en introducir nuevos modelos estándares propios
- Todo el lenguaje en su conjunto (tanto el lenguaje de programación como las construcciones propias del framework usado) debe ser confiable y minimizar el número de problemas que se introducen producto de las abstracciones implementadas.

Luego de la correspondiente reflexión acerca de las diversas alternativas disponibles, se analizaron las siguientes alternativas:

Por una parte se consideró la opción de desarrollar con una herramienta basada en la pila LAMPP (Linux, Apache, Pearl y PHP). Las alternativas consideradas fueron Codeigniter, Laravel, y Symfony 2, principalmente por la cantidad de documentación disponible y por la experiencia que el encargado tenía trabajando en éstas. Pero fueron rápidamente descartadas por problemas intrínsecos al lenguaje

PHP, como lo son su falta de predictibilidad y consistencia. Ambas cualidades son de suma importancia para la productividad, y PHP las viola desenfrenadamente.

Por otro lado se tenían opciones basadas en el lenguaje Java como Spring y JSF. Lamentablemente ambas opciones son bastante extensas en el número de construcciones y configuraciones iniciales que se deben manejar antes de efectivamente puedan ser herramientas productivas. El número de conceptos ajenos al dominio del problema y el poco valor que aportan dichos conceptos provocaron un rápido descarte de las opciones antes mencionadas.

Se analizó brevemente el concepto de servidores asíncronos como NodeJS y Tornado Web Framework. Pero por el interés del presente proyecto de mantener múltiples hilos de ejecución (para atender múltiples clientes al mismo tiempo) mientras se toman métricas del tiempo de cálculo de las distancias entre los usuarios, se descartaron dichas opciones. Aun así no deja de ser interesante su potencial ventaja frente a otras opciones si se decide extender la aplicación con funcionalidades que tengan uso intensivo de operaciones I/O (operaciones de lectura y escritura de medios externos).

Finalmente se optó por el framework Django sobre el lenguaje python. La elección del lenguaje se fundamenta en la elegancia y expresividad de las expresiones desarrolladas, la cantidad de documentación disponible, el excelente manejo de excepciones y la consistencia implementada en la gran mayoría de librerías disponibles. En cuanto al framework se eligió a Django frente a alternativas como “Web.py” o “Flask” pues es el que mejor combina lo extensa de sus librerías incorporadas y lo simple que resulta implementar nuevas funcionalidades (llamadas Ajax, uso de distintos motores de base de datos, gestión de eventos, etcétera). Además de esto se destacan las siguientes características deseables al momento de emprender un desarrollo de mayor complejidad:

- Su probada robustez para el desarrollo de aplicaciones web, ampliamente usada en importantes sitios de alto tráfico (www.disqus.com , www.pinterest.com , www.instagram.com)

- Su interfaz de administrador que facilitó el desarrollo de casos de prueba, y sus correspondientes verificaciones.

- Su batería de tests que permiten un desarrollo más seguro.

El sistema está desplegado sobre un servidor Gunicorn v 16.0 especial para operar sobre el protocolo WSGI en ambientes de producción. Se eligió por su rapidez ante requerimientos concurrentes, la simpleza de su configuración, su baja tasa de fallos, y su bajo consumo de memoria. Otras alternativas analizadas fueron:

- Servidor Apache 2 + módulo “mod_wsgi”: Descartado por su alto consumo de memoria en cada proceso levantado. Además de la dificultad de significa cambiar los parámetros de configuración del servidor, el servidor mantiene un gran stack de tecnología soportadas, lo que lo vuelve lento y aparatoso.

- uWSGI: Servidor con menor soporte que Gunicorn, y algo más difícil de configurar. Pero tiene a su favor ser de los más rápidos que hay disponibles.

- Aspen: Demasiado simplista para los objetivos del presente proyecto. Además la integración con Django requerían tiempo que no compensaba el resultado esperado.

Como enlace de entrada al servidor se eligió el proxy inverso NGinX v1.6, alojando los archivos estáticos en un fichero aparte.

Todo el sistema fue montado en un Sistema Ubuntu Server 14.04, elegido por su disponibilidad de extensiones, fácil instalación de complementos y amplio soporte en la comunidad.

2.2. Algoritmo de Búsqueda de Vecinos Cercanos:

2.2.1. Problema de Fixed Radius Nearest Neighbor Search:

El problema de buscar, dado un usuario, aquellos usuarios que tengan intereses comunes dentro de un radio fijo, es uno de los problemas clásicos de optimización computacional llamado **“Fixed Radius Nearest Neighbor Search” (FR-NNS)**. Dicho problema puede ser enunciado de la siguiente manera:

Dado un conjunto de puntos **S** en un espacio métrico **M** (frecuentemente euclidiano), una distancia **D**, un radio **R** y un punto **q en M**, encontrar el conjunto **S’** ⊆ **S** tal que:

$$S' = \{ p \in S : D(q , p) <= R \}$$

En este caso M se define como un espacio **bidimensional euclidiano**, y la distancia se asume como **distancia euclidiana tradicional**.

Este problema puede generalizarse a un problema muy usado actualmente diversos campos de la ciencia y la ingeniería, el problema de **Nearest Neighbor Search (NNS)**:

Problema de NNS:

Dado un conjunto de puntos **S** en un espacio métrico **M**, y un punto **q en M**, encontrar los **K** puntos más cercanos en **S** a **q**.

Variantes de Solución a la Familia de Problemas NNS:

Siguiendo las metodologías de resolución de problemas de optimización basadas en **Branch and Bound**, se han aplicados diversas estrategias de partición del espacio para solucionar el problema de NNS. Uno de los más conocidos, mediante el uso de "índices espaciales", es el uso de estructuras de datos del tipo **k-d-tree**, que iterativamente va dividiendo en dos el espacio a analizar. Luego, la búsqueda de los vecinos más cercanos se realiza desde la raíz hasta una de las hojas, evaluando el punto a buscar en cada una de las divisiones de un nodo [13]. La especialización de este tipo de estructuras se denomina **quadtree**, que almacena en cada nodo rectángulos bidimensionales, y permite operaciones de inserción y búsqueda de forma rápida y eficiente.

Alternativamente la estructura de datos **R-Tree** [14] permite realizar NSS con argumentos dinámicos como la cantidad de vecinos a buscar, y la métrica a usar. Este tipo de estructuras goza de gran popularidad para modelar transposición de rectángulos en espacios métricos por realizar optimización heurística sobre el área que cubre determinado rectángulo en cada nodo interno.

Una variante que resulta útil tomar en cuenta es el **R*-Tree** [15], que permite representar diferentes tipos de unidades geométricas dentro de un mismo espacio euclidiano. Esto a cambio de aumentar ligeramente los costos de búsqueda en comparación con el R-Tree.

Otra metodología muy usada es el "hashing por localidad" que consiste en agrupar los puntos cercanos en "**buckets**" de acuerdo a cierta métrica usada. Luego, los puntos cercanos entre ellos bajo la métrica a usar se ubicarán en el mismo bucket con mayor probabilidad.

Problema k NNS:

El problema de **k-Nearest Neighbor** corresponde al más básico de los algoritmos de "aprendizaje de máquinas", en donde se tiene un conjunto de puntos **S**, y se clasifica a cada punto de acuerdo a la clasificación que tengan sus **k** vecinos más importantes. Así, cada punto es clasificado en un conjunto **C** de clases.

2.2.2. Antecedentes Técnicos del problema FR-NNS:

El problema de los algoritmos necesarios para encontrar a los **N** vecinos más cercanos, dado un conjunto finito de vecinos ha sido abordado en muchas aplicaciones y temas de investigación académica:

En [16], se describen y analizan los algoritmos para resolver el siguiente problema: Dado un conjunto de puntos S conformado por S1 y S2, los puntos en S se clasifican todo nuevo punto de acuerdo a los puntos ya clasificados, viendo si el punto más cercano al nuevo punto está en S1 o en S2, y clasificándolo como corresponda. Los autores concluyen que la cota natural para el problema es **O(n log k)**, siendo **n**: número de puntos en el espacio métrico por clasificar y **k** el número de puntos que contribuyen a la clasificación de los **n** puntos antes mencionados.

Sin embargo para referirnos al caso particular de Near-U, y el problema subyacente del FR-NNS, conviene centrarse en el estudio descrito en [17], que presenta diversas técnicas para resolver el problema de **FR-NNS**. El autor hace hincapié en que en el paper no se resuelve nada nuevo, sino que sólo pretende recopilar distintas soluciones propuestas al problema del NNS con Radio Fijo.

A continuación se enumeran las distintas estrategias analizadas. Para todos los ejemplos se tendrá en consideración el siguiente modelo: Un conjunto **S** de **n** puntos, en un espacio euclidiano **M** (de **m** dimensiones), con una distancia **D** (norma euclidiana) y un radio **R**.

1 Fuerza Bruta: Se almacena cada punto de **S** en una lista u otra estructura simple. Cada vez que llega una consulta, se actualiza la ubicación del punto consultado y luego se compara contra todos los otros puntos de la estructura. Esta estrategia tiene costo de:

Para n puntos: $\boldsymbol{n^2}$ comparaciones.

2 Proyección: (Friedman, Baskett y Shustek) Estrategia que permite mantener los elementos en S ordenados por sus correspondientes coordenadas, y realizar la búsqueda de los vecinos cercanos a un punto de forma iterativa dimensión por dimensión. De esta forma para cada dimensión **i** se ordenan todos los puntos en S por su coordenada i-ésima en una lista ordenada correspondiente a dicha dimensión. De esta forma se tendrán **m** listas ordenadas para las m dimensiones consideradas.

El costo de agregar un elemento (o cambiar su ubicación) es el de agregarlo a cada una de las listas anteriores. Siendo m el número de dimensiones el costo de agregación es: **m * log2(n)**

Para la búsqueda de los vecinos más cercanos a **X** basta con tomar una dimensión y recorrerla desde la posición de **X** hasta encontrar un elemento **X'** tal que **D(X,X') > R**. Entonces se toman todos los puntos entre **X** y **X'** como parte de la selección. Y se repite este procedimiento para las otras dimensiones.

La estrategia de proyección resulta particularmente eficiente en caso donde los puntos no son uniformemente distribuidos, sino que formar "clusters" en las vecindades de un conjunto discreto de puntos.

3 Técnicas Celdares: Son útiles cuando un gran porcentaje de los puntos en **S** están posicionados en un subespacio del espacio euclidiano **M**, y distribuidos de forma uniforme dentro de este. La técnica celdar genérica divide cada dimensión en segmentos de igual longitud, por lo que se tiene finalmente una grilla **D** dimensional. A cada celda se le asocia una estructura simple para almacenar puntos (por ejemplo, una lista) y se marca aquellas celdas de la grilla que contengan puntos en ella. Entonces cada punto es almacenado en la estructura de la grilla que le corresponda de acuerdo a las coordenadas del primero (por ejemplo, si son coordenadas geodésicas; el punto (-70.322442, -33.533342) será almacenado en la estructura de la grilla que tiene como segmentos asociados: [-70.3225, -70.3224] x [-33.5334 , -33.5333]). En cierto sentido, esta técnica corresponde a un Hash con cubetas cúbicas de **m** dimensiones.

El espacio requerido para aplicar la técnica celdar genérica es igual al número de celdas totales = refinamiento de 1 dimensión * **m**. Más el número total de puntos.

4 Kd-Trees: Es la generalización de un árbol binario en **k** dimensiones. Pero a diferencia de un árbol binario, en un **Kd-Tree** además de presentar un valor a comparar cada nodo también debe especificar la dimensión sobre la cual comparar. La variante "**Kd+Tree**" establece que los puntos sólo pueden ir en los nodos exteriores del árbol.

El almacenamiento de un **Kd-Tree** es proporcional a **n**, y su coste de construcción es: **O(k*n*log(n))** . A pesar de que originalmente **el Kd-Tree** está pensado para consultas del tipo NNS, se puede fácilmente adaptar para resolver problemas del tipo FR-NNS. Más aún, este tipo de estructura es una estructura "asintóticamente óptima" al problema FR-NNS. Aunque para un número pequeño de puntos en **S**, los costos de construcción y actualización del **Kd-Tree** pueden no compensar el ahorro en la selección de los vecinos suficientemente cercanos.

En [14] se propone la estructura **R-Tree** como una forma de indexar objetos no puntuales (con dimensiones distintas de cero) en un espacio euclidiano, de forma de

mejorar el rendimiento de las operaciones de inserción y búsqueda por rangos. Acá el autor plantea que las bases de datos con índices unidimensionales no son capaces de optimizar las búsquedas cuando estas son por rangos multidimensionales (en este caso, número de puntos en una esfera de radio **R**).

En un R-Tree cada nodo hoja contiene una referencia a un cuerpo dentro de **M**, y permite realizar búsquedas de objetos con un número menor de visitas a nodos. Más específicamente un R-Tree se comporta de la siguiente manera:

- R-Tree es un árbol balanceado (todas sus hojas en el mismo nivel).

- Cada objeto espacial se representa por la tupla **(Q , id),** en donde **id** es el identificador del objeto y **Q** el menor cubo R-dimensional que lo contenga. En el caso de Near-U, **id** es el identificador de cada usuario en el sistema, y **Q** un cuadrado de 200 metros x 200 metros que representa su “área de visualización” de otros usuarios.

- Cada nodo interno contiene entre **m** y **M** tuplas de la forma **(Q’, <hijo>)**, en donde **<hijo>** es un puntero a un nodo inferior y **Q’** un cubo que contiene a todos los cubos en las tuplas de dicho hijo.

- Cada hoja contiene entre **m** y **M** tuplas de objetos espaciales, al menos que sea la raíz.

- Con esta estructura de datos se pueden realizar la operación de búsqueda, la cual será esbozada en python. Para esto se supondrá que todos los **n** usuarios del sistema están en el R-Tree **T**, en tuplas del tipo **(Q , id_usuario)**:

Se desea buscar los usuarios a 100 metros o menos de **X**.

```
def buscar(X, T):
        if T is raiz:
                cercanos = []
                for tupla in T.tuplas:
                        if seInterceptan(X.Q , tupla.Q):
                                cercanos.append(tupla.id_usuario)
        else:
                for tupla in T.tuplas:
                        if seInterceptan(X.Q , tupla.Q):
                                cercanos = Concatenar(cercanos , buscar(X , tupla.hijo))
```

Experimentalmente se puede notar que el algoritmo de búsqueda tiene un costo de **O(n*logm(n))** cuando se elige M para que el tamaño de los nodos sea aproximadamente el tamaño del caché del CPU (y pueda descargarse varias veces sin acceso a memoria). Esto es verdadero para cerca del 93% de los casos. Una de las bondades del R-Tree es que permite encontrar los menores cubos R-dimensionales que cubran los hijos de cada nodo por sucesivas aproximaciones heurísticas, logrando de esta manera los resultados antes indicados.

Las propiedades de cómo se construye un **R*-Tree** permiten obtener tiempos de inserción aproximadamente un 20% más rápidas que con un R-Tree y hasta un 50% más rápida para operaciones con entidades espaciales y entidades puntuales. Lamentablemente en el caso de Near-U sólo se trata con entidades espaciales (los usuarios del sistema y sus respectivas áreas de observación), caso en que un R-Tree mantiene una pequeña ventaja asociada al mecanismo de construcción del mismo.

En el estudio [18] el análisis que realizan los autores sobre las Bases de Datos Espaciales (Bases de Datos que permiten almacenar puntos y espacios geométricos de forma eficiente) menciona que las principales consultas a este tipo de Base de Datos es encontrar a los k puntos más cercanos a un punto dado. Este tipo de consultas se conoce como **kNNS** (por las siglas k-Nearest Neighbor Search). Este artículo propone el uso de **Diagramas de Voronoi** de primer orden para resolver el problema de kNNS en este tipo de bases de datos. El principal problema a resolver para responder consultas como la consulta kNNS es el costo (en términos de cálculos necesarios) de cálculo de las distancias entre los puntos.

El algoritmo de Voronoi se basa en dividir el espacio euclidiano en pequeñas "**regiones de Voronoi"**, y luego calcular las distancias tanto entre los puntos dentro de una misma región, como entre ellas. De esta manera se ahorra memoria al no tener

que almacenar todos los nodos a computar, mientras que se ahorra también en costos de cómputo al no tener que calcular la distancia de cada par **<nodo, nodo>**. Finalmente se verifica empíricamente que el rendimiento del algoritmo mejora hasta en un orden de magnitud.

En el [19] el autor realiza un estudio general de los algoritmos conocidos para resolver el problema de **NNS** (Nearest Neighbor Search). Se analizan diferentes distancias como la distancia euclidiana tradicional, la distancia rotatoria terrestre, y la distancia de edición de Strings (cadenas de texto). Primero se propone un algoritmo casi óptimo (de la clase de los algoritmos de hashing) con el uso de la distancia euclidiana. Luego se concluye que la cota inferior para ciertos tipos de distancia siempre será **lineal-logarítmico**. Más tarde se propone una nueva aproximación que, usando la **"distancia de Ulam"**, puede resolver el problema en tiempo logarítmico. Finalmente se hace un repaso por los usos y aplicaciones que tendrá este nuevo algoritmo en variados campos de las ciencias y la industria. Sin embargo, más allá de un análisis de sensibilidad, de aspecto muy teórico, no llega a ninguna implementación factible que solucione el problema de NNS en tiempo logarítmico.

3. Descripción de la Solución:

3.1. Introducción:

Para resolver el problema expuesto en los objetivos se diseñó y desarrolló un sistema de georreferencia para ser desplegado en servidores preparados para sostener una alta demanda de usuarios concurrentes. El sistema sirve como sustento para una plataforma que permita mantener las locaciones de muchos usuarios, una lista de intereses por cada uno de los usuarios, y que permita un cálculo eficiente de cuando dos usuarios con algún interés similar se encuentren en las proximidades, a cierta distancia mínima. Más formalmente, se implementó un sistema que cruza listas de intereses entre sus usuarios, y les avise de otros usuarios con al menos una coincidencia entre sus intereses y que estén dentro de cierto radio de cercanía, definido por el usuario.

Para el desarrollo del sistema se priorizó principalmente el tiempo de respuesta que él mismo debía brindar a los distintos usuarios, de forma de lograr una experiencia de usuario que permitiera usar el software con comodidad.

Por un lado, se desarrolló la aplicación de un servidor central capaz de soportar el manejo de las listas de cada usuario, así como encontrar los usuarios más cercanos dado un radio de cercanía. Dicho sistema es capaz de realizar estas comparaciones entre las posiciones de los usuarios de forma eficiente y robusta.

Por otro lado, se diseñó una aplicación cliente a instalar en los dispositivos móviles de los usuarios del sistema, que permita la conexión asíncrona con el servidor central, y procure que cada una de las funcionalidades descritas sea ofrecida por una interfaz adaptada para dispositivos móviles. Dicha aplicación, además de ser liviana y sencilla de ocupar, busca ofrecer la funcionalidad de buscar otros usuarios con los mismos intereses de forma rápida.

Desde el punto de vista de la implementación del sistema, éste se compone de dos partes independientes (aplicación cliente móvil diseñada para smartphones, y aplicación de servidor) que interactúan de manera de ofrecer al usuario las funcionalidades indicadas a continuación.

1) Un usuario puede registrarse en la aplicación móvil y el sistema crea internamente su perfil, retornando la validación de la operación.

2) Un usuario puede iniciar sesión en la aplicación cliente con su identificador y su password. El servidor responde con sus datos personales, su lista de “***intereses***”, y los “***matches***” (calces entre dos intereses) para cada interés en particular, con intereses que otro usuario pudiese tener. Para esto, cada

"interés" de la "lista de intereses" de un usuario tiene almacenados en su interior una lista de todos los calces con intereses de otros usuarios.

3) Un usuario puede agregar un "*interés*" al sistema. El sistema lo agrega en la Base de Datos, y retorna la confirmación.

4) Un usuario puede modificar el radio de búsqueda de "***matches***", modificando una variable interna de la aplicación cliente. Acá no hay interacción con el servidor.

5) A intervalos regulares la aplicación cliente envía al servidor un "***requerimiento de matches***". En este requerimiento se envían como parámetros el "***radio de búsqueda***", la "***locación del usuario***", su "***identificador***", y su "***prioridad***". El servidor retorna una lista con las "***locaciones***" de los usuarios con los que se tiene al menos un match y estén dentro del radio de búsqueda, y otra lista con los "***nuevos matches***" detectados en esta iteración. La lista de "**locaciones**" se usa para actualizar el mapa de la aplicación cliente, permitiendo ubicar a los usuarios en éste. La lista de "**nuevos matches**" se usa para actualizar los matches de la lista de intereses de la aplicación cliente.

El procedimiento antes mencionado es ejecutado por cada uno de los clientes a intervalos regulares de tiempo de modo que garanticen una "buena" experiencia de usuario en términos del tiempo de respuesta que brinda la aplicación, así como su correspondiente interactividad asociada. Por esto es que se asume que los intervalos de solicitudes al servidor (y sus correspondientes respuestas asociadas) sean una cada cinco segundos. De esta manera se brinda al usuario la sensación de interactividad que demanda el desarrollo de aplicaciones móviles actualmente en el mercado.

3.2. Relevancia de la Solución:

La importancia de contar con una forma eficiente de comparar distancias entre los usuarios permite ofrecer tiempos de respuesta a los usuarios de modo que perciban un uso fluido de la aplicación Near-U. De otro modo, con tiempos de respuesta más largos que lo normal, se ve afectado de forma negativa el uso de la aplicación.

Por el lado del diseño de una plataforma de testing de esfuerzo, el diseño y desarrollo de esta permite tomar métricas, de tiempo de respuesta y de correctitud, del comportamiento de una aplicación para casos de uso reales de una api asíncrona (como en este proyecto se propone). De este modo se pueden realizar diversas mejoras sobre cómo reacciona la aplicación ante distinto escenarios de demanda, al mismo tiempo que se encuentran aquellos puntos del software en que se pueden perder y/o alterar los resultados esperados.

Finalmente es importante señalar la importancia de lograr que la aplicación cliente contenga una interfaz de usuario amigable, correcta, simple, y estéticamente agradable. En otras palabras: La aplicación debe ser entendible y fácil de usar. No debe llevar al usuario a cometer errores. No debe sobrecargar la mente del usuario con opciones que no son útiles ni aporten valor. Y debe tener una línea gráfica que invite a ser usada.

3.3. Especificación del Problema:

El problema a resolver del presente Trabajo de Título es la creación de un sistema de procesamiento de cruce entre interés de los clientes atendidos, dadas las posiciones geográficas de cada uno de ellos. Dicho sistema fue diseñado para atender concurrentemente a todos los usuarios de forma concurrente, cada uno actualizando su ubicación cada 5 segundos. Para esto se contempló el desarrollo de un servidor que atienda las consultas, una aplicación cliente a ser instalada en celulares para testing, una aplicación de carga masiva de datos para simulación de usos reales, y una plataforma de clientes y servidores que realicen test de esfuerzo sobre éste último.

Para esto se debieron seguir una serie de pasos que se describen a continuación:

1) Definición de un API estándar que sea respetada tanto por el cliente como por el servidor.

2) Desarrollo de aplicación de testing sobre una página web adaptable a múltiples tamaños de pantallas, para la prueba de la interfaz comprometida en el punto anterior.

3) Desarrollo de aplicación servidor, que resuelve el problema de búsqueda de vecinos cercanos por “fuerza bruta”.

4) Diseño y desarrollo de plataforma de testing para ser montada sobre distintas máquinas, y que cada máquina envíe concurrentemente varias peticiones. Esto de manera de tomar métricas de cuánto tiempo demora una respuesta promedio.

5) Mejora en aplicación del servidor, que resuelve el problema de los vecinos cercanos de forma más eficiente.

6) Desarrollo de una aplicación cliente que permita, usando la API antes desarrollada, terminar la plataforma completa de Near-U lista para ser usada por usuarios.

La solución planteada para el problema original se descompone en un trabajo de:

1. Aplicación de servidor preparado para la alta concurrencia de usuarios.

2. Aplicación cliente desarrollada atendiendo a los principios de diseño de interfaces.

3. Aplicación de Testing de Esfuerzo.

A continuación se enumeraran los detalles de cada uno de estos tres ejes, con los detalles que permitan comprender mejor su composición.

3.4. Aplicación del Servidor:

La aplicación web desarrollada para la parte del servidor tiene como principal misión el recibir todos los requests HTTP de los distintos usuarios (a través de las múltiples instancias de la aplicación cliente) y procesar sus respuestas, derivándolos previamente a otro proceso especializado en el caso de los requerimientos de mayor complejidad. En la práctica el único request derivado a otro proceso externo a la aplicación es el que calcula los usuarios a cierta distancia a cada usuario particular.

Se desarrolló en python 2.7 con el uso del framework Django y las librerías WTForms y Gunicorn. Como capa de persistencia se optó inicialmente por una Base de Datos en MySQL. El trabajo se organizó en los siguientes pasos:

1. Definir API a desarrollar **[Anexo 1]**

2. Implementar Modelos de la BBDD

3. Implementar Página de Testing del API

4. Desarrollar API de funcionalidades básicas (login, registro, agregar need)

5. Desarrollar funcionalidad avanzada (búsqueda de matches).

Los pasos cuatro y cinco fueron optimizados iterativamente de modo de obtener mejores resultados a través de mejoras en el algoritmo de comparación de distancias y en la forma de manejar la persistencia de los datos.

A continuación se describe la primera versión plenamente funcional de la aplicación servidor. Luego, en las subsecciones 3.5 y 3.6 se detallan las sucesivas optimizaciones realizadas sobre el software servidor.

3.4.1. Especificación de Aplicación del Servidor

Para el desarrollo del servidor se utilizó el modelo MVT (Model View Template) incorporado en Django, efectuando una serie de modificaciones que permiten una mejor extensibilidad y mantenibilidad del software.

Se separaron las vistas en clases diferentes, agrupando a las primeras por su contenido semántico. Las vistas encargada de los requests de los usuarios en la clase **UsuarioView**, la vista encargada de los requests de administración de los needs de cada usuario en la clase **NeedView**, y la vista encargada del procesamiento de las locaciones de cada usuarios se implementó en la clase **MatchView**.

A su vez se crearon cinco modelos para almacenar y procesar la información del sistema. Dichos modelos se implementan como clases que actúan como enlace directo con sus correspondientes tablas en una Base de Datos Relacional.

Usuario: Mantiene información de los datos y la actividad reciente de cada usuario.

Need: Representa un interés que un usuario desea compartir. Puede ser de 3 tipos posibles:

- **Wish**: Un Need que refleja un deseo de un usuario de determinado bien o servicio. Sólo realizará un Match cuando se encuentre con otro Need del tipo **Do**.
- **Do**: Un Need que refleja la oferta de un producto que un usuario esté dispuesto a realizar, o un servicio que desea recibir. Sólo realizará Match cuando se encuentre con otro Need del tipo **Wish**.
- **Share**: Un tipo de Need que refleja el deseo de compartir cierto bien o servicio o experiencia de un usuario con otros usuarios cercanos. Sólo realizará Match con otros Needs del tipo **Share**.

Match: Una tabla cruzada entre dos needs. Representa que el dueño de un Need visualizó un calce con otro Need de otro usuario. Almacena una referencia a ambos Needs (el Need del usuario "descubridor" del calce, y el Need del usuario "descubierto" en el calce).

Locacion: Representa una ubicación de un usuario en cierto instante de tiempo. Por lo tanto, esta entidad almacena el mail del usuario (llave de la entidad Usuario), su latitud, su longitud, y el instante de tiempo en que ocurrió el registro. Su utilidad es el facilitar la consulta de otros usuarios que estén en las cercanías cuando un usuario consulte al sistema. Entonces el método de encontrar a los usuarios cercanos comenzará siempre por extraer de Location los registros que contengan en su tiempo de última actualización un instante no más distante a 10 segundos al tiempo de la consulta (ambos tiempos se extraen del sistema).

UsuarioConsultado: Entidad que sirve como forma de almacenar cuando un usuario **U1** ya consultó por Matches sobre otro usuario **U2**. Si ya lo realizó, y ninguno de los dos ha agregado algún Need, este registro lo indicará. Esto evita tener que recorrer ambas listas de Needs buscando Matches cuando se sabe a priori que todos los Matches posibles ya han sido creados.

Por otro lado se crean dos funciones auxiliares que se agrupan en una librería externa. Se toma esta decisión para respetar el principio de Responsabilidad Única de las vistas (toda clase debe tener una única razón para cambiar), ya que semánticamente las funciones no corresponden a ninguna vista.

- La primera función permite obtener la distancia geodésica (en metros) entre dos puntos cuyas coordenadas están expresadas como racionales. Se debe considerar que el radio de la tierra asumido de 111302[m] es aproximadamente el radio real a la altura de los trópicos.

```
def distanciaGeodesica(latitude1, longotude1, latitude2, longitude2):
    degtorad = 0.01745329
    radtodeg = 57.29577951
    dlong = (longitude1 – longitude2)

    dvalue = ( sin(latitude1 * degtorad ) * sin(latitude2 * degtorad) )
    dvalue += ( cos(latitude1 * degtorad) * cos(latitude2 * degtorad) * cos(dlong * degtorad) )

    distance = arccos(dvalue) * radtodeg
    metros = distance * 111302
    return metros
```

- Para obtener la diferencia lexicográfica entre dos Strings:

```
def getStringDiff(str1, str2):
    return SequenceMatcher(None, str1, str2, ratio)
```

3.4.2. Modelo de Datos:

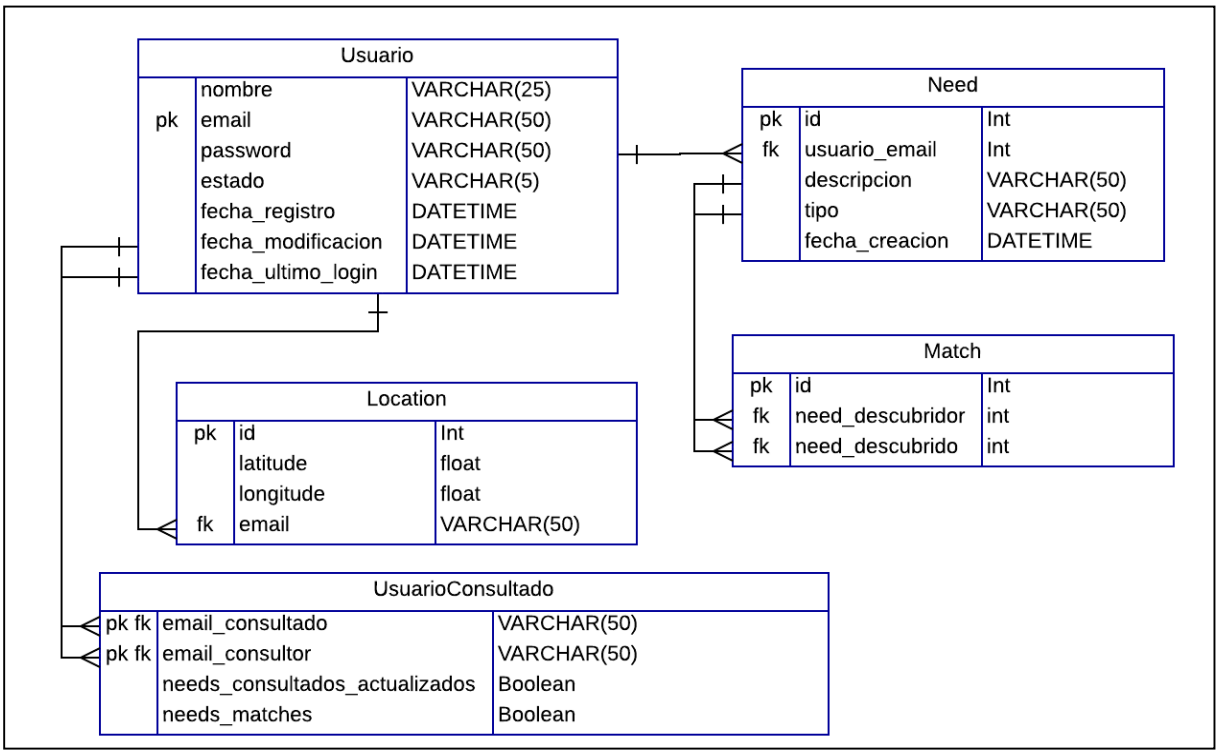
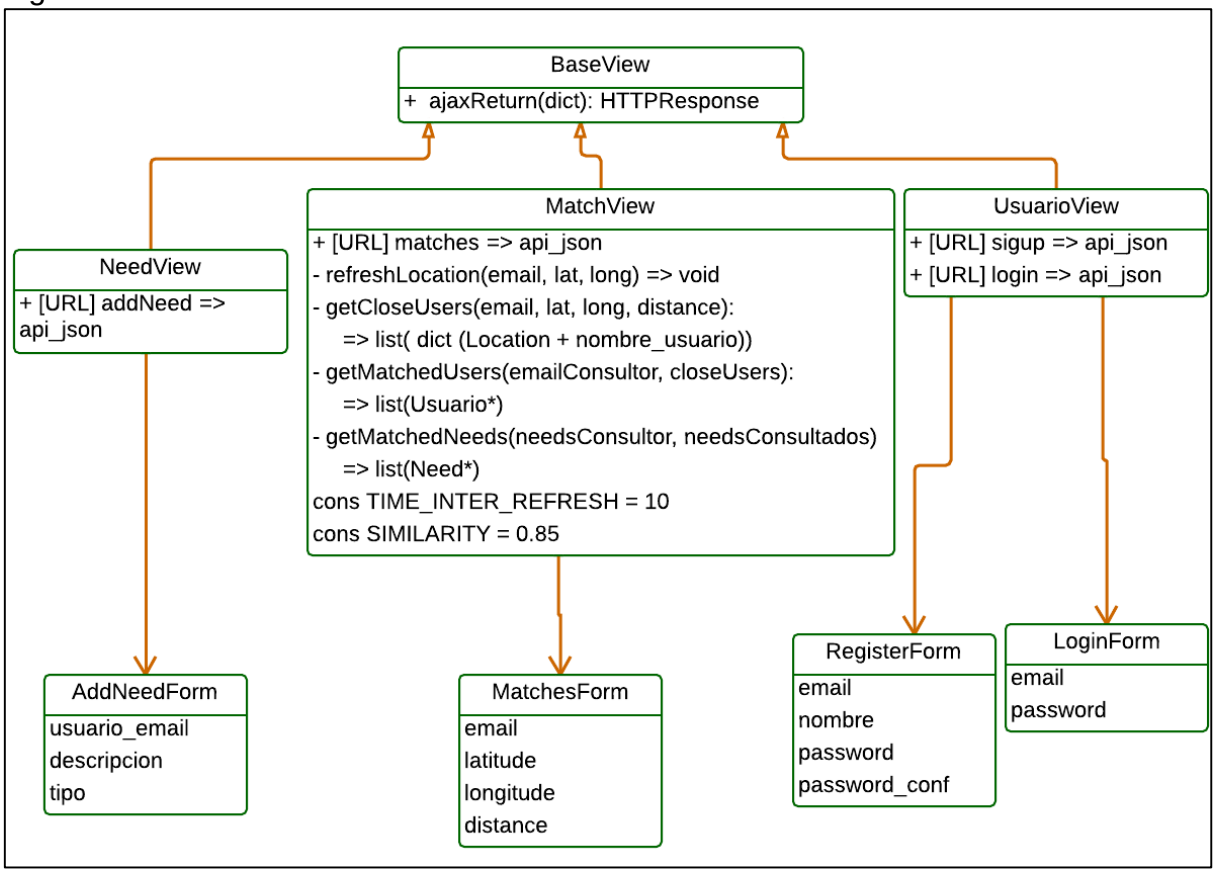


Figura 1

En la siguiente figura se pueden ver las tablas que componen el modelo relacional de la aplicación. Por un lado se tiene las tablas **Usuario** y **Need** para almacenar los usuarios y sus correspondientes Needs. La tabla **Match** almacena un calce entre los Needs, la tabla **Location** almacena la ubicación de cada usuario, y la tabla **UsuarioConsultado** almacena una consulta de un usuario sobre los calces con otro usuario. Esta última tabla funciona a manera de evitar iterar sobre los Needs de dos usuarios que ya se han visto antes (han consultado acerca de los Needs del otro) y ninguno de los dos ha agregado un nuevo Need.

3.4.3. Diagrama de Clases:

Figura 2



En la figura anterior se pueden observar las tres vistas que componen la aplicación. La vista **UsuarioView** que implementa los métodos para registrarse y loguearse en el sistema. La vista **NeedView** que permite agregar nuevas vistas, y la vista **MatchView** que implementa la principal funcionalidad de la aplicación.

A su vez también se puede observar que las clases para la validación de los parámetros HTTP que recibe la aplicación: **AddNeedForm**, **MatchesForm**, **RegisterForm**, y **LoginForm**.

Observación:

En la clase **MatchView** se pueden distinguir las constantes **TIME_INNER_REFRESH** y **SIMILARITY**. Dichas constantes representan parámetros de configuración del sistema, por lo que no cambiaran durante la ejecución del mismo. La primera se refiere al tiempo que un objeto Location tiene validez antes de asumir que el usuario ya se desconectó. De esta manera cuando un cliente llama a la función “matches”, sólo compara contra los usuarios cercanos que tengan un Location actualizado en los últimos 10 segundos. La segunda constante corresponde al mínimo

de similitud lexicográfica que tienen que tener las descripciones de dos Needs para que se produzca un Match.

A continuación se describe el algoritmo para obtener los matches:

Pseudo Código Función matches:

```
def matches(request):
    form = MatchesForm(request.POST)
    if request.method == 'POST' and form.validate():

        email = request.POST['email']
        latitude = float(request.POST['latitude'])
        longitude = float(request.POST['longitude'])
        distance = int(request.POST['distance'])

        # se actuliza la posicion del usuario.
        affectedRows = Location.filter(email=email).update(latitude=latitude,
                                        longitude=longitude,
                                        last_update=int(time.time()))
        if affectedRows == 0:
            l = Location(email=email,
                    latitude=latitude,
                    longitude=longitude,
                    last_update= int(time.time()))
            l.save()

        # Obtiene lista de usuarios cercanos
        closeUsers = getCloseUsers(email, latitude, longitude, distance)
        # Obtiene lista de usuarios con los que comparte Needs
        matched_users = getMatchedUsers(email, closeUsers)

        return ajaxReturn({'status':'OK', 'matched_users':matched_users})

    else:
        return ajaxReturn({'status':'NO', 'forms_errors':form.errors})
```

Se puede observar que lo primero que se realiza es actualizar la posición del usuario. En caso de no existir aún dicha posición, se crea una posición nueva. Luego llama a la función **getCloseUsers** para obtener los usuarios más cercanos a cierto Radio, y finalmente pasa esta lista de usuarios cercanos a la función

getMatchedUsers para obtener los usuarios con los que tenga al menos algún calce, y dentro de cada uno de estos los calces que contenga.

Pseudo Código Función getCloseUsers:

```
def getCloseUsers(email, latitude, longitude, distance):
   # todas las locaciones, excepto las del usuario
   allLocations = Location.exclude(email = email)
   closeUsers = []
   now = int(time.time())

   for location in allLocations:
      if now - location.last_update < TIME_INTER_REFRESH:
         # caso en que locacion esté actualizada
         if distance == 0:
            closeUsers.append(location))

         else:
            distanceFromLocation = harvesianDistance(latitude,
                                    longitude,
                                    location.latitude,
                                    location.longitude)
            if distanceFromLocation <= distance:
               closeUsers.append(location)

   return closeUsers
```

En este caso la función itera sobre todos los usuarios del sistema seleccionando únicamente aquellos que están a una distancia menor o igual a cierta distancia dada.

Pseudo Código Función getMatchedUsers:

```
def getMatchedUsers(emailConsultor, closeUsers):
    myNeeds = None
    matchedUsers = list()

    for closeUser in closeUsers:
        try:
            # Si usuarios ya se han visto antes
            usuarioConsultado = UsuarioConsultado.get(email_consultor=emailConsultor,
                                        email_consultado=closeUser['email'])
            if usuarioConsultado.needs_consultados_actualizados:
                # Si usuario consultado ha agregado needs ultimamente
                needsConsultor = Needs.filter(usuario__email=emailConsultor)
                needsConsultado = Needs.filter(usuario__email=closeUser['email'])

                # Obtiene lista de Needs que tienen calce con dicho usuario
                newMatchedNeeds = getMatchedNeeds(needConsultor, needConsultado)

                if len(newMatchedNeeds) >= 1:
                    # Si hay un Need con calce, se almacena
                    closeUser['new_matched_needs'] = newMatchedNeeds
                    matchedUsers.append(closeUser)

                # Se actualizan datos de consulta entre ambos usuarios
                usuarioConsultado.needs_consultados_actualizados = False
                usuarioConsultado.save()

            elif usuarioConsultado.needs_matched:
                matchedUsers.append(closeUser)

        except UsuarioConsultado.DoesNotExist:
            # Si es primera vez que usuarios se ven
            needsConsultor = Need.filter(email=emailConsultor)
            needsConsultado = Need.filter(email=closeUser['email'])

            # Se crea registro de consulta entre ambos usuarios
            usuarioConsultado = UsuarioConsultado(consultor=emailConsultor,
                                    consultado=closeUser['email'],
                                    needs_consultados_actualizados=False,
                                    needs_matched=False)
            newMatchedNeeds = getMatchedNeeds(needsConsultor, needsConsultado)
            if len(newMatchedNeeds) >= 1:
                # Si hay un Need con calce, se almacena
                closeUser['new_matched_needs'] = newMatchedNeeds
                matchedUsers.append(closeUser)
            usuarioConsultado.save()
    return matchedUsers
```

Esta función recibe una lista de usuarios cercanos y retorna una lista del subconjunto de los primeros usuarios con los que el cliente tiene al menos un Match. Para esto se itera sobre la lista de usuarios cercanos preguntando a cada uno si se han vistos (si sus Needs han sido comparados) antes con el cliente y desde entonces ningún usuario ha agregado un nuevo Need. De ser así se concluye que no existen Matches posibles. En cualquier otro caso se pasan los Needs de cada usuario a la función **getMatchedNeeds** para retornar los posibles nuevos Matches entre ambos usuarios.

Pseudo Código Función getMatchedNeeds:

```
def getMatchedNeeds(needsConsultor, needsConsultado):
   matchedNeeds = list()
   for needConsultor in needsConsultor:
      for needConsultado in needsConsultado:
         # Se itera por ambas listas de Needs

         matches = Match.filter(descripcion_creador=needConsultor.descripcion,
                                descripcion_foraneo=needConsultado.descripcion)
         if len(matches) == 0:
            if getStringDiff(needConsultor.descripcion,
                        needConsultado.descripcion) > SIMILARITY:
               # Si existe similitud entre los Needs

               matchedNeeds.append(needConsultado)
               newMatch = Match(need_creador = needConsultor,
                           need_foraneo = needConsultado)
               newMatch.save()
         else:
            matchedNeeds.append(needsConsultado)

   return matchedNeeds
```

Esta función se itera sobre los Needs de dos usuarios particulares. Al momento de encontrar un Match, lo almacena en la Base de Datos y lo almacena en la lista a retornar.

3.4.4. Plataforma de Producción:

El servidor fue montado sobre una máquina con la que se estimó que las mediciones del rendimiento del algoritmo fuesen de la suficiente confiabilidad, y que poseía las siguientes características:

- Procesador 2Ghz x 4 núcleos.
- 8Gb Memorial Ram.
- Disco SSD de 80Gb.
- SO Ubuntu Server 14.04

Librerías Usadas en Producción:

- Django 1.6
- Gunicorn 19.0.0
- NGinX 1.6
- MySQL Server 14.14
- MySQL-python 1.2.5
- WTForms 1.0.5

3.5. Primera Optimización (Servidor v2):

3.5.1. Descripción:

En un primer intento de optimizar el cálculo de la función **"matches"** se cambia la forma de modelar los datos de un modelo relacional a un modelo de documentos mediante la Base de Datos **MongoDB**. De este modo se evitan los dos **join** necesarios para obtener los Needs y Matches de un usuario particular, además de disminuir las consultas a la Base de Datos de 4 a 1.

Con esta nueva forma de almacenar los datos se miden como mejoran los tiempos por **request**, y cuantas conexiones soporta el servidor sin caerse.

El modelo de datos pasa a ser un modelo de documentos con la siguiente estructura:

- Una colección de documentos llamada **Usuario** en donde se almacenan internamente los Needs de cada usuario, y dentro de cada need los matches con needs de otros usuarios del sistema. Cada documento posee la siguiente estructura:

```
{
   _id: ObjectId,
   nombre: String,
   email: String,
   password: String,
   needs : {
      descripción: String,
      tipo: String,
      matches: {
         descripción_foranea: String,
         owner_email: String,
         owner_nombre: String
      } *
   } *
}
```

- Una colección llamada **Location**, que cumple el mismo rol de la versión anterior. En este caso se determina no implementar campos referenciales a objetos BSON (análogo a las restricciones de llaves foráneas en Bases de Datos Relacionales), por lo que la integridad referencial del campo **user_email** se valida por software. Esta decisión se toma como forma de disminuir el tiempo de ejecución no creando referencias externas que involucran un deterioro en el indexamiento de los documentos.

- Una colección llamada **UsuarioConsultado**, que cumple la misma función anterior. Nuevamente se determina no usar campos referenciales a objetos BSON.

3.5.2 Descripción de MatchView de Primera Optimización:

Como la optimización se realiza sobre los tiempos de ejecución de la función “matches” se describe a continuación los cambios realizados en ella y las funciones auxiliares llamadas en su ejecución, en pseudo-código.

Pseudo Código Función matches:

```
def matches(request):
        form = MatchesForm(request.POST)
        if request.method == 'POST' and form.validate():
                # Obtener variables
                email = request.POST['email']
                latitude = float(request.POST['latitude'])
                longitude = float(request.POST['longitude'])
                distance = int(request.POST['distance'])

                # Actualiza la locacion
                db.location.update({'email': email},
                                                {'latitude': latitude,
                                                'longitude': longitude,
                                                'last_update': int(time())},
                                                {'upsert': True})

                closeUsers = getCloseUsers(email, latitude, longitude, distance)
                matched_users = getMatchedUsers(email, closeUsers)

                return ajaxReturn({'status':'OK', 'matched_users':matched_users})
        else:
                return ajaxReturn({'status':'NO', 'forms_errors':form.errors})
```

Se observa que la función matches actúa de forma similar que en la primera versión del software servidor. Primero actualiza la ubicación del usuario atendido, luego obtiene aquellos usuarios más cercanos que el radio de cercanía dado con la función **getCloseUsers**, finalmente llama **getMatchedUsers** para obtener aquellos usuarios con los que tiene al menos una coincidencia entre sus respectivos Needs. Cabe notar que en este caso la función **getMatchedUsers** retorna una lista de usuarios representados por un diccionario en el cual están insertos los Needs que coinciden con el usuario llamador, por lo que no es necesario una función auxiliar que cruce los Needs de cada usuario buscando coincidencias como en el caso de la primera versión del servidor y la función **getMatchedNeeds**.

Pseudo Código Función getCloseUsers:

```
def getCloseUsers(email, latitude, longitude, distance):
    # Obtiene todas las Locaciones excepto la del usuario
    allLocations = db.location.find({'email': notEqual(email)})
    closeUsers = []
    closeLocations = []
    now = int(time())

    for location in allLocations:
        # Si locacion está actualizada
        if now - location['last_update'] < TIME_INTER_REFRESH:
            if distance == 0:
                closeLocations.append(location)
            else:
                distanceFromLocation = harvesianDistance(latitude,
                                        longitude,
                                        location['latitude'],
                                        location['longitude'])
                if distanceFromLocation <= distance:
                    closeLocations.append(location)

    # Ahora se obtiene el usuario de cada Location
    for closeLocation in closeLocations:
        user = db.usuario.find_one({'email': closeLocation['email']})
        closeUsers.append(user)

    return closeUsers
```

Esta función itera sobre las locaciones de todos los usuarios del sistema. Para cada locación primero chequea que esta esté actualizada (pues una locación actualizada hace mucho tiempo no será confiable para saber la ubicación geográfica de un usuario). En caso de estar actualizada se compara la distancia con el usuario atendido, y si la distancia es menor al radio dado, se agrega como locación cercana. Finalmente se obtiene el usuario de cada locación para luego retornar la lista de todas las locaciones cercanas con sus respectivos usuarios.

Pseudo Código Función getMatchedUsers:

```
def getMatchedUsers(emailConsultor, closeUsers):
    consultor = db.usuario.find_one({'email': emailConsultor})
    matchedUsers = list()

    for closeUser in closeUsers:
        usuarioConsultado = db.usuario_consultado.find_one({'consultor': emailConsultor,
                                        'consultado': closeUser['email']})
        if usuarioConsultado == None:
            # Si usuarios no se han visto aún
            consultado = db.usuario.find_one({'email': emailConsultor})
            if hasMatches(consultor, consultado):
                # Si tiene al menos 1 match

                matchedUsers.append(closeUser)
                db.usuario_consultado.insert({'consultor': emailConsultor,
                                'consultado': closeUser['email'],
                                'needs_matched': True,
                                'needs_updated': False })
            else:
                # Si no hay matches
                db.usuario_consultado.insert({'consultor': emailConsultor,
                                'consultado': closeUser['email'],
                                'needs_matched': False,
                                'needs_updated': False })
        else:
            # Usuarios ya se han visto antes
            if usuarioConsultado['needs_matched']:
                # Si usuarios ya tienen un match previo
                matchedUsers.append(closeUser)
            elif usuarioConsultado['needs_updated']:
                # No existe match previo, pero "consultado" actualizo sus needs
                consultado = db.usuario.find_one({'email': closeUser['email']})

                if hasMatches(consultor, consultado):
                    matchedUsers.append(closeUser)
                    usuarioConsultado['needs_matched'] = True
    return matchedUsers
```

En esta función se obtiene aquellos usuarios con los que se tiene al menos un calce entre sus Needs y los Needs del usuario atendido. Primero se revisa que no se hayan visto antes (sus Needs no hayan sido comparados), de ser así se ve si existe un calce entre sus respectivos Needs mediante la función **hasMatches**, y entonces se agrega a la lista de usuarios a retornar. En caso de que si se hayan visto antes se debe revisar que exista una coincidencia previa, en cuyo caso también se agrega a la lista de usuarios a retornar, pues ya se sabe del calce entre al menos una par de los Needs respectivos. Por otro lado si no existe un match previo pero el usuario actualizó sus Needs desde la última vez que fueron comparados, entonces corresponde una nueva comparación para buscar calces y agregar al usuario a la lista a retornar en caso de que exista alguno.

Cabe observar que esta función se diferencia de la función **getMatchedUsers** de la primera versión del software del servidor en que no recurre a otra función auxiliar para obtener los Needs coincidentes, sino que retorna la lista de usuarios coincidentes con sus Needs dentro de cada usuario. Por lo que la comprobación de si existe un calce entre los Needs de dos usuarios se realiza por medio de la función **hasMatches**.

Pseudo Código Función hasMatches:

```
def hasMatches(consultor, consultado):
for myNeed in consultor['needs']:
   for otherNeed in consultado['needs']:
      if stringDiff(myNeed['descripcion'],otherNeed['descripcion']) > SIMILARITY:
         return True
return False
```

Esta simple función itera sobre los Needs de dos usuarios retornando un booleano si existe al menos una coincidencia entre ambas listas.

3.5.3. Plataforma de Producción de Servidor v2:

El servidor fue montado sobre una máquina con las siguientes características:

- Procesador 2Ghz x 4 núcleos.
- 8Gb Memorial Ram.
- Disco SSD de 80Gb.
- SO Ubuntu Server 14.04.

Librerías Usadas en Producción:

- Django 1.6
- Gunicorn 19.0.0
- NGinX 1.6
- MongoDB 2.6.3
- djangotoolbox 1.6.2
- WTForms 1.0.5

Se puede observar la incorporación del motor de Bases de Datos MongoDB de modo de reemplazar el funcionamiento del ORM de Django, basado en bases de datos relacionales, por una basada en documentos.

3.6. Segunda Optimización (Servidor v3):

3.6.1. Descripción:

En este caso se opta por eliminar la colección **Location**, y se guardan las locaciones de cada usuario en una estructura de datos dinámica. Esto además de aumentar la velocidad de la función (por evitar un acceso a disco para traer a todos los usuarios), permite almacenar las locaciones en un **R-Tree** para evitar tener que consultar las posiciones de cada usuario en el sistema. De este modo se pretende disminuír considerablemente el tiempo de la función más compleja y exigente del algoritmo para obtener las coincidencias con otros usuarios, que es la función **getCloseUsers**. Cabe hacer notar que este es uno de los principales postulados del presente Proyecto de Título.

La implementación **del R-Tree** debe ser compartida entre los distintos procesos simultáneos que despliega el servidor. Por lo que optó por implementar en una aplicación aparte usando el stack **NodeJS** + **ExpressJS**. Con estas herramientas se creó un proceso que, mediante el uso de la librería **RTree** descrita en [20].

3.6.2. Descripción de MatchView de Segunda Optimización:

En este caso se mantienen las funciones **getMatchedUsers** y **hasMatches** de la clase **MatchView** implementada en la primera optimización del servidor, cambiando su modo de ejecución únicamente la función **getCloseUsers**.

Pseudo Código Función getCloseUsers:

```
def getCloseUsers(*kwargs):
    # se realiza llamado a servidor NodeJS
    closeLocations = json.loads(requests.post('localhost:5000/close_users', kwargs))

    # Ahora se obtiene el usuario de cada Location
    for closeLocation in closeLocations:
        user = db.usuario.find_one({'email': closeLocation['email']})
        closeUsers.append(user)

    return closeUsers
```

En este caso se puede observar que el algoritmo se simplifica bastante al reducir su ejecución a llamar al servicio externo que implementa el **R-Tree** y luego obtener el usuario de cada locación retornada.

La comunicación entre el servidor y el **R-Tree** se realiza por medio de llamadas HTTP entre máquinas dentro de la misma infraestructura **[Anexo 2]**.

3.6.3. Descripción del R-Tree de la Segunda Optimización:

La implantación del servicio de búsqueda de los vecinos más cercanos a cierto usuario se describe a continuación.

Pseudo Código Función getCloseUsers:

```
var express = require('express');
var app = express();
var RTree = require('../lib');

var difference = function(a,b){
   return Math.abs(a-b);
}

var rtree = new RTree();

app.post('/close_users', function(req, res){
   var email = req.param('email'),
      latitude = req.param('latitude'),
      longitude = req.param('longitude'),
      distance = req.param('distance');

   var lastLocation = rtree.get(email);
   if(difference(latitude, lastLocation.latitude) < 1 &&
         difference(longitude, lastLocation.longitude)){
      /* En caso de que ubicación haya cambiado, se reemplaza en el RTree*/
      rtree.remove(email);
      rtree.insert(email,{ 'x': latitude,
                           'y': longitude,
                           'w': distance,
                           'h': distance
      });
   }
   var closeUsers = rtree.search({'x': latitude,
                        'y': longitude,
                        'w': distance,
                        'h': distance
   });
  res.send(json.stringify(closeUsers));
});
```

En el pseudo-código anterior puede observarse como se declara la estructura afuera de la función que recibe las llamadas desde la aplicación del servidor de forma de garantizar que todos los usuarios compartan la misma estructura de datos mientras

el proceso esté en ejecución (lo que ocurrirá mientras la aplicación esté en producción).

3.6.4. Plataforma de Producción (Servidor v3):

El servidor, al igual que en las versiones anteriores, fue montado sobre una máquina con las siguientes características:

- Procesador 2Ghz x 4 núcleos.
- 8Gb Memorial Ram.
- Disco SSD de 80Gb.
- SO Ubuntu Server 14.04.

El R-Tree fue montado sobre una máquina con las siguientes características:

- Procesador 2Ghz x 2 núcleos.
- 2Gb Memorial Ram.
- Disco SSD de 40Gb.
- SO Ubuntu Server 14.04.

Librerías Usadas en Producción del Servidor:

- Django 1.6
- Gunicorn 19.0.0
- NGinX 1.6
- MongoDB 2.6.3
- djangotoolbox 1.6.2
- WTForms 1.0.5

Librerías Usadas en Producción del R-Tree

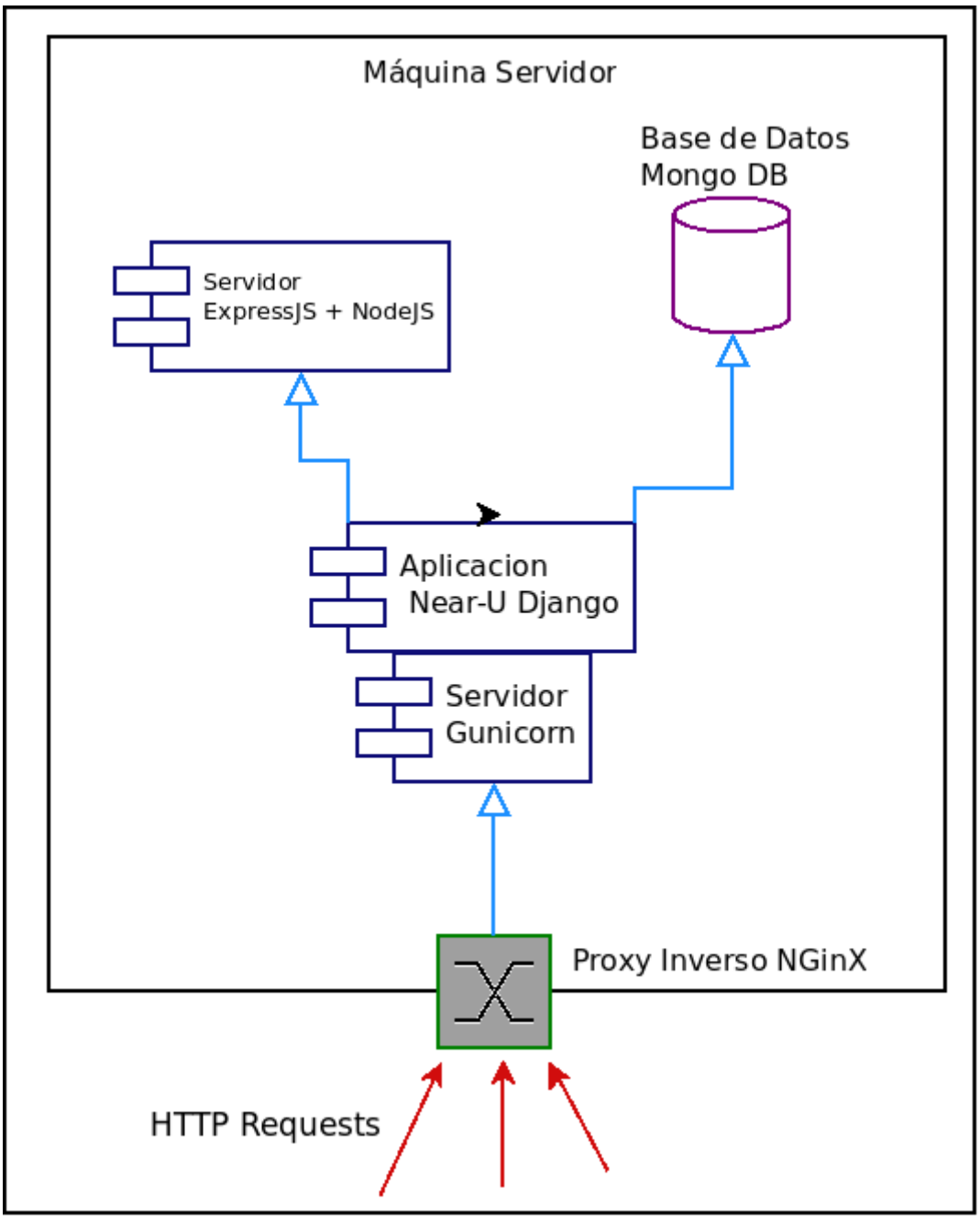
- NodeJS 0.10.18
- Express 4.5.0
- imbcmdth/RTree

Se puede notar que uno de los desafíos encontrados en el desarrollo de una aplicación de servidor concurrente es que la segunda optimización requiere que los distintos usuarios accedan todos a la misma estructura de datos desde diferentes procesos. Al ser una aplicación concurrente el software Gunicorn levanta un proceso por cada llamada que llega a la máquina donde se instala el software.

La solución propuesta en el presente trabajo de título es el uso de un proceso independiente que siempre esté en memoria a través de la plataforma NodeJS, parte de la arquitectura de desarrollo “MEAN Stack”. Entre las ventajas analizadas se cuenta una documentación extensa y de gran calidad, un paradigma de desarrollo asíncrono basado en eventos y de un único thread (hilo), y la estabilidad y fiabilidad para manejar grandes cargas de requerimientos.

3.6.5. Diagrama Arquitectónico Servidor v3:

Figura 3:



En el presente diagrama se puede observar cómo se configura el sistema para el desarrollo del software servidor de la segunda optimización. Cada llamada que llega al host (desde los múltiples clientes conectados al sistema en determinado momento) es recibida por el proxy inverso NGinX que se encarga de redirigirla al servidor o servir directamente el archivo en caso de que la llamada sea para cargar un archivo estático. A continuación el servidor Gunicorn se encarga de dos funciones: Ejecuta la aplicación del servidor **Near-U Django** y realiza las traducciones correspondientes entre la llamada HTTP y el formato WSGI que es consumido por dicha aplicación. La base de datos es una instancia maestra de MongoDB. Finalmente se ejecuta el proceso de

NodeJS para recibir las llamadas desde la aplicación Near-U Django con el uso del framework Express.js.

3.7. Aplicación Cliente:

3.7.1. Descripción:

A modo de implementar una solución real de los desarrollos llevados a cabo en el experimento del presente trabajo, se implementó una aplicación cliente para ser usada en teléfonos celulares. El principal objetivo de esto se fundamenta en la motivación principal del presente proyecto, unir a las personas a través de soluciones tecnológicas enfocadas en el compartir intereses comunes.

La aplicación cliente fue implementada como una **SPA** (Simple Page Application) desarrollada con HTML5 y Javascript. Para la realización de la interfaz de usuario se optó por el framework **IonicJS** por su sencillez y la limpieza de sus widgets.

La elección de crear una SPA se fundamenta en la universalidad del estándar HTML5, soportado por la gran mayoría de los smartphones vigentes hoy en día. En contraposición con el desarrollo para las plataformas nativas de AppStore y Google Play.

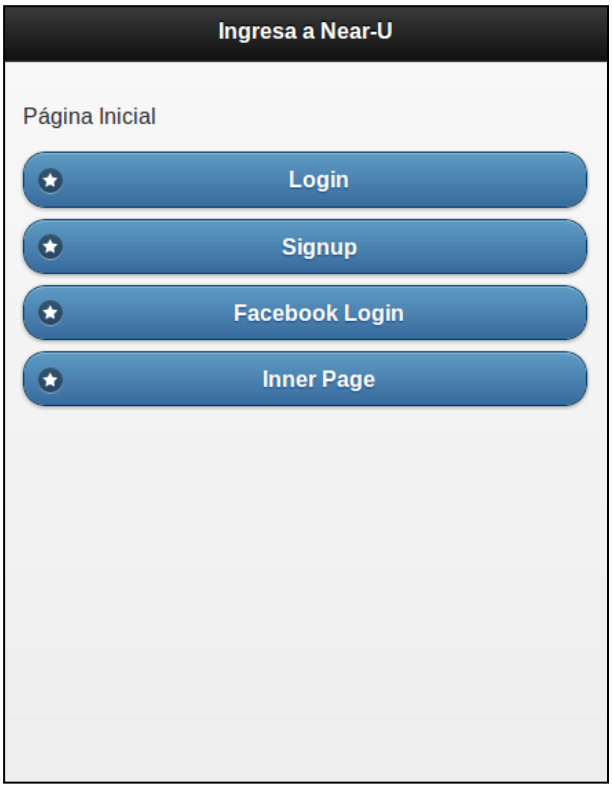
Observación:

Aunque se procuró un diseño de interfaz siguiendo los principales Guidelines de diseño y usabilidad, no se realizó ningún testing sobre la usabilidad de la aplicación cliente. Esto pues dicho punto escapa al alcance del proyecto.

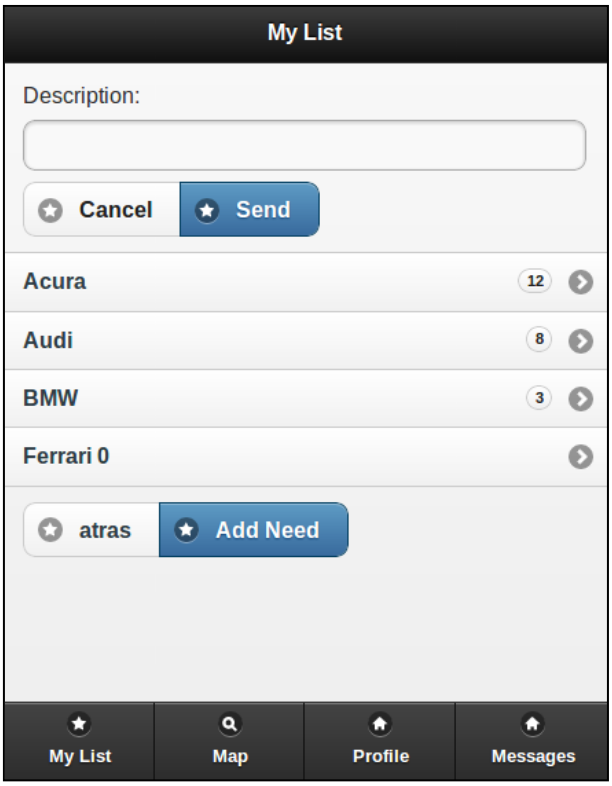
3.7.2 Capturas de aplicación cliente:

Figura 4:

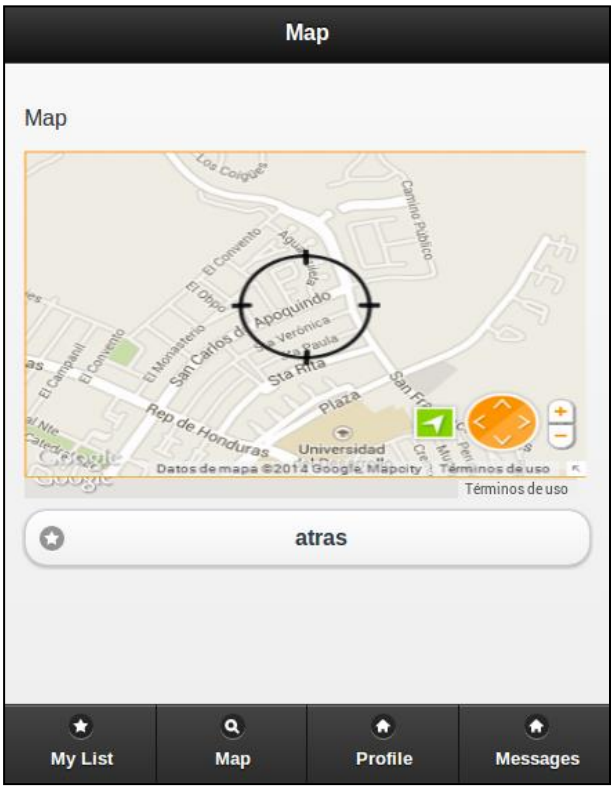
Pantalla de Inicio:



Pantalla Interna Principal:



Pantalla de vista de Mapa:



3.8. Desarrollo de Plataforma de Testing:

3.8.1. Descripción General:

Para validar las optimizaciones que se realizaron sobre el Servidor, bajo el concepto de tests de stress e integridad, se optó por desarrollar una aplicación en Python **TestCliente** que simula un número determinado de usuarios enviando requests a la API de ruta **"/matches"** cada 5 segundos. La simulación de muchos usuarios se realiza por medio de la creación de **threads** (procesos ligeros en UNIX), uno por cada usuario.

De esta manera, cuando el **TestCliente** es ejecutado, crea a los usuarios. Al crear un usuario, TestCliente le pasa como parámetros los datos del usuario: {'email', 'latitud-inicial', 'longitud_inicial'}.

Luego de creado, cada usuario (thread) manda un request, espera la respuesta, y anota los resultados del tiempo de respuesta en una estructura compartida por todos los threads del **TestCliente**. Posterior a esto verifica que el tiempo de espera sea inferior a 5 segundos, y si es así "duerme" el tiempo restante para completar los 5 segundos. Este procedimiento lo repite hasta completar los 5 requests.

Es importante notar que la ubicación que envía cada usuario no siempre será la misma, pues para efectos de simular un caso de uso "real" se asume que un tercio de los usuarios están movilizándose al momento de enviar los request (variando sus coordenadas geográficas con un vector de velocidad aleatorio y módulo igual a 5 Km/h).

Para incrementar el porcentaje de usuarios cercanos, la simulación crea usuarios virtuales inicialmente ubicados en el espacio geográfico correspondiente al **Barrio República**: Coordenadas dentro del espacio: **[-33.435, -33.476] x [-70.623 , -70.68]**.

Los test fueron ejecutados desde cinco máquinas distintas, dentro de la misma infraestructura del Servidor. Para sincronizar que las máquinas se usó la librería **NTP**, que sincroniza el equipo con servidores de hora internacionales (con precisión atómica). Luego se creó en cada máquina un cronjob (tarea automatizada que el sistema ejecuta en cierto instante) para la ejecución del **TestCliente**. De este modo se aseguró que todos los programas sean ejecutados al mismo tiempo.

Este procedimiento se repitió con los siguientes números de usuarios concurrentes: [100 - 500 - 1000 - 1500 - 2000 - 5000 – 10000]

3.8.2. Implementación de TestCliente:

```
def main():
   threads = list()
   # el numero de usuarios se obtiene como parámetro del script
   num_usuarios = get_parametro('num_usuarios')
   # archivo lleno de nombres de usuarios
   fileUsuarios = open('fileUsuarios', 'read')
   # estructura que mantiene los usuarios simulados
   resumen = list()

   for n in xrange(num_usuarios):
      mail_usuario = fileUsuarios.readline()
      resumen.append({
         'latitude': str(random.uniform(-33.476 , -33.435)),
         'longitude': str(random.uniform(-70.68, -70.623))
      })

   for mail_usuario in resumen.keys():
      mi_thread = threading.Thread(target=client, args=(mail_usuario))
      mi_thread.start()
      threads.append(mi_thread)

   for mi_thread in threads:
       mi_thread.join()

   resumen.witeToFile('resumen.json', 'w')
```

En el pseudo-código acá expuesto se describe de manera general la manera de realizar los tests de alta demanda a las tres versiones del software de servidor antes expuestas. Primero se declara una lista en donde se almacenarán los tiempos de respuestas que registrará cada usuario simulado a lo largo de 5 solicitudes al servidor de forma sucesiva. Luego lee emails de ejemplo desde un archivo de emails y crea con cada email un nuevo usuario simulado al cual también se le agregan los campos de latitud y longitud generados al azar. Posteriormente por cada usuario se lanza un hilo de ejecución (desde ahora “thread”) en donde se usa el mail de dicho usuario para enviar 5 solicitudes sucesivas al servidor. Dichas solicitudes gatillan la funcionalidad de matches antes descrita. Luego se espera a que todos los threads terminen de ejecutar y se agregan los tiempos de demora de cada ejecución a su usuario respectivo, almacenado en la lista de usuarios llamada “resumen”. Finalmente se exportan los datos obtenidos en el experimento a un archivo externo en un formato de mayor compatibilidad general con los programas para graficar, el formato “json”.

Como observación cabe notar que la ubicación de cada usuario se fija en base a un área de 7.5 Km cuadrados, por lo que se garantiza al menos un match entre los usuarios simulados de cada iteración.

Otra observación importante es que este test es que fue diseñado para ser ejecutado paralelamente desde 5 computadoras al mismo tiempo, por lo que el número de threads que cada test “levanta” debe ser multiplicado por 5 para obtener el número real de threads que accederán al servidor en cada experimento. Para la correcta sincronización de las máquinas de tests se usa el software de sincronización de reloj “NTP” para que las máquinas ajusten sus relojes internos de acuerdo a distintos relojes atómicos de referencia mundial. De este modo se garantiza que todas las máquinas de testing tengan exactamente la misma hora. De este modo la ejecución del experimento se realiza creando un **cronjob** (tarea administrativa automática programable para sistemas UNIX) programado para ejecutar el test a determinada hora en cada una de las máquinas de test.

3.8.3. Pseudo Código de Threads de TestCliente:

A continuación se ilustra de forma clara y concisa el modo de operar de cada thread levantado por el proceso principal.

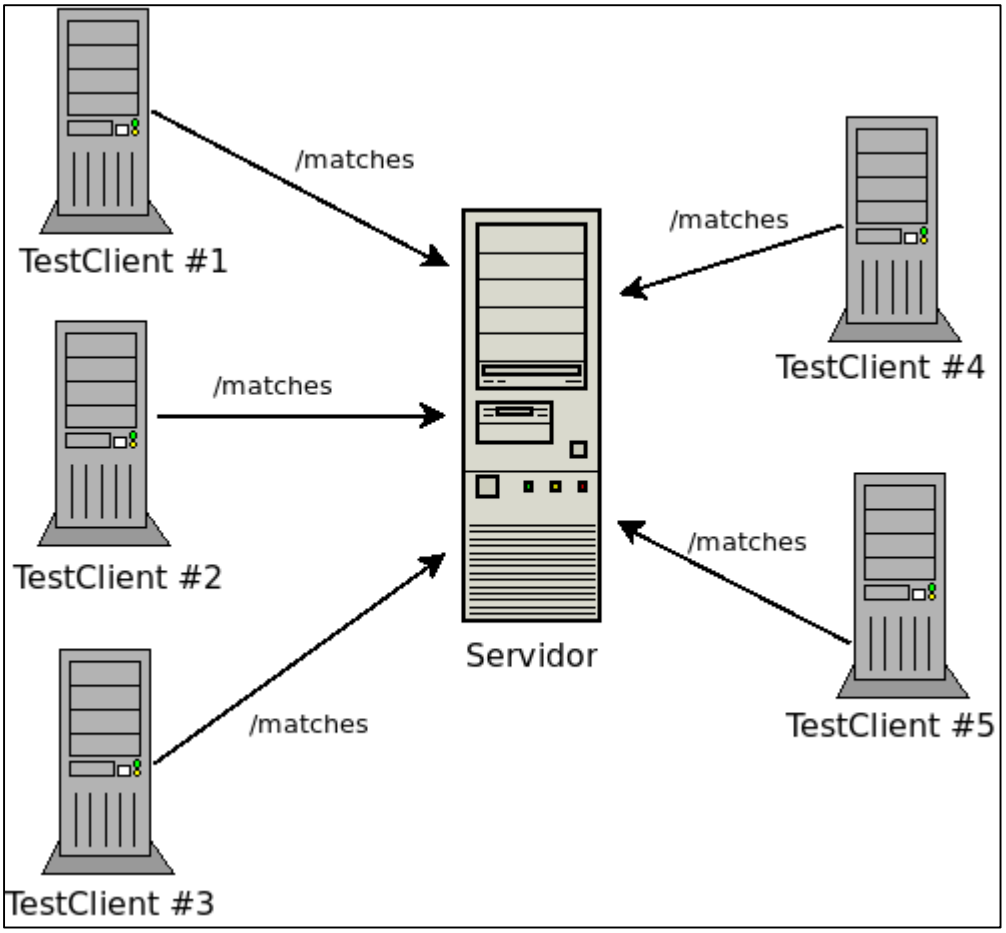
```
def client(mail_usuario):
    for j in range(5):
        inicio = time.time()
        parametros = {'email': mail_usuario,
                      'latitude': resumen['mail_usuario']['latitude'],
                      'longitude': resumen['mail_usuario']['longitude'],
                      'distance': resumen['mail_usuario']['distance']
                     }

        response = requests.post(matches_url, data=parametros)
        duracion = time.time() - inicio
        resumen[mail_usuario]['medida_'+i] = duracion

        if duracion < 5.0:
            time.sleep(5.0 - duracion)
```

En el código antes expuesto es fácil apreciar que a partir del correo de un usuario simulado se obtienen datos como su latitud, su longitud y la distancia de la solicitud. Luego se envía una solicitud al servidor, se espera por la respuesta, se toma el tiempo de respuesta del servidor y se anota el resultado en la misma estructura de cada usuario simulado. Finalmente se ve si la solicitud tomó más de 5 segundos. En caso de que tome menos, el thread queda inactivo hasta completar los 5 segundos. Esto se repite por 5 veces consecutivas hasta tener 5 mediciones de tiempo de respuesta para el usuario simulado pasado como argumento del thread,

La razón de la espera de al menos cinco segundos es que se busca simular a un cliente de la aplicación que actualiza su estado y busca matches con dicha frecuencia de tiempo.

3.8.4. Diagrama Arquitectónico de la Plataforma de Testing:

Figura 5:



En la figura anterior se observa la disposición general de la plataforma usada para realizar pruebas de esfuerzo y de integridad sobre el software detallado a lo largo de la presente sección. De modo de simular un caso de uso lo más cercano a la realidad posible se instala un cierto número de clientes de prueba automática en cada una de las máquinas **TestClient**, y la el software desarrollado en la máquina indicada como **Servidor**. Al momento de ejecutarse las pruebas cada máquina **TestClient** crean un número determinado de hilos, y cada hilo comienza a mandar solicitudes (requests) a la máquina **Servidor**. Esto se realiza de forma automática y dentro de la misma infraestructura TI de forma de garantizar que el tiempo de ruteo de los paquetes IP no sea determinante en la toma de resultados.

4. Validación de la Solución:

A continuación se muestran los resultados obtenidos para los tiempos de respuesta de las tres versiones de servidores desarrollados, para muestras de 100, 500, 1000, 1500, 2000, 5000 y 10000 usuarios concurrentes. Para cada muestra de usuarios se grafica el tiempo mínimo, el tiempo máximo y el tiempo promedio de una solicitud HTTP a la funcionalidad de encontrar matches de la aplicación de servidor (desde ahora un **request**), y el número de consultas perdidas.

Primero se grafica el tiempo medio de un request, que es a priori el resultado más relevante para conocer el comportamiento de las distintas soluciones planteadas.

DEFINICIONES:

Server MySQL: Servidor con Modelos mapeados a una Base de Datos Relacional MySQL.

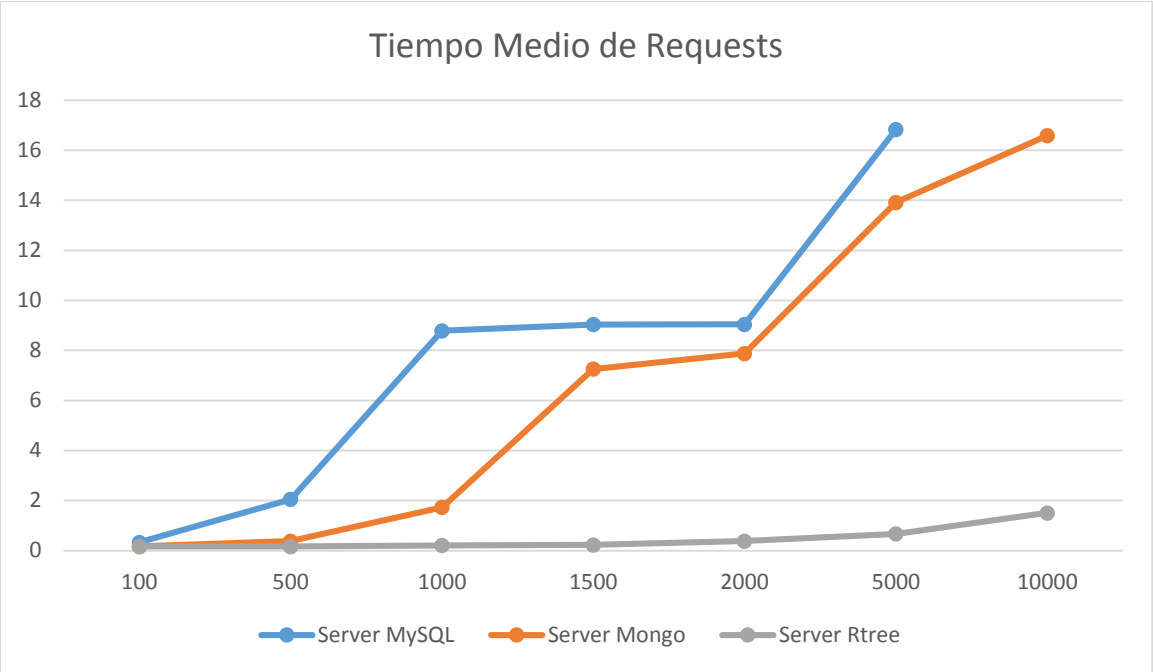
Server Mongo: Servidor que usa Base de Datos orientada a documentos MongoDB.

Server RTree: Servidor que usa Base de Datos orientada a documentos MongoDB y maneja las locaciones de cada usuario con un proceso aparte con un R-Tree

4.1. Resultados Obtenidos:

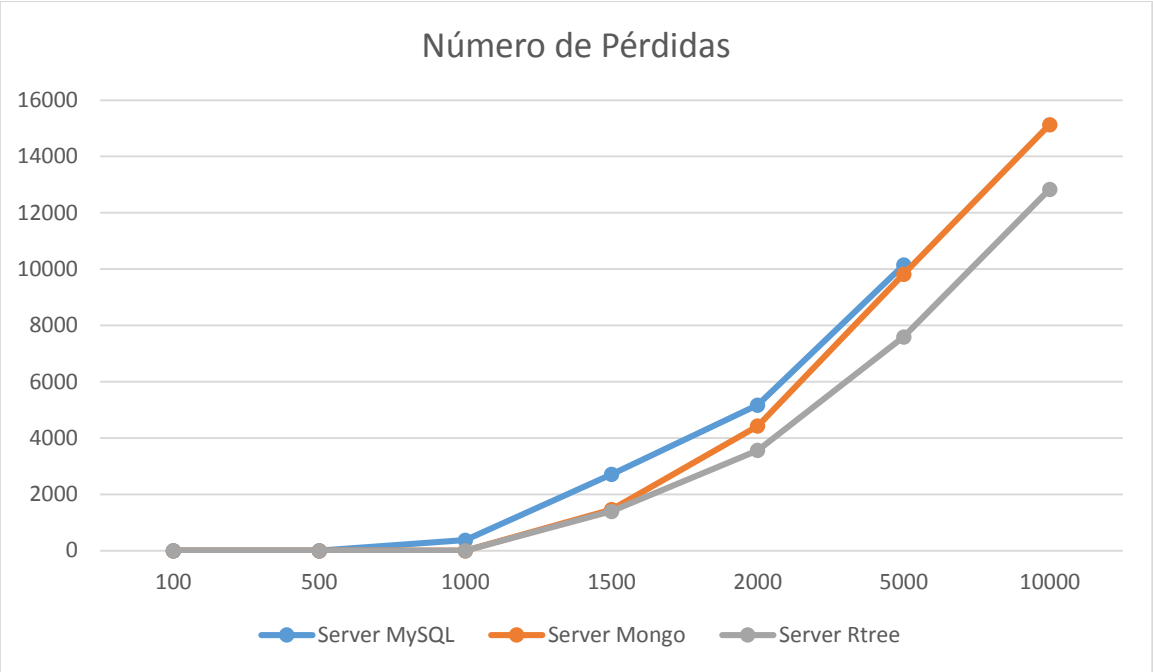
4.1.1. Tiempo Medio de un request [s] para múltiples usuarios concurrentes:

Num Usuarios / Servidores	100	500	1000	1500	2000	5000	10000
Server MySQL	0.33	2.05	8.788	9.033	9.04	16.835	<FALLO>
Server Mongo	0.167	0.379	1.728	7.255	7.876	13.919	16.583
Server RTree	0.169	0.165	0.21	0.23	0.383	0.671	1.51



4.1.2. Número de Pérdidas:

Num Usuarios / Servidores	100	500	1000	1500	2000	5000	10000
Server MySQL	0	0	372	2712	5172	10150	<FALLO>
Server Mongo	0	0	0	1459	4427	9825	15132
Server RTree	0	0	0	1396	3561	7592	12836


Número de Pérdidas
16000
14000
12000
10000
8000
6000
4000
2000
0
100
500
1000
1500
2000
5000
10000
Server MySQL
Server Mongo
Server Rtree

5. Conclusiones:

Luego de realizados los experimentos para determinar el tiempo promedio de cada implementación del servidor de Near-U, se pueden observar ciertos patrones que no dejan de ser llamativos, y que se describen a continuación.

5.1. Factor de Contaminación, la Gran Pérdida de Datos:

Quizá uno de los aspectos más relevantes es la gran cantidad de pérdida que arrojaron todos los experimentos (incluyendo los experimentos sobre el servidor R-Tree). Se puede observar que para los tres casos el nivel de pérdida comienza a ser significativo a partir de los 1500 usuarios concurrentes (300 requests por segundo).

Por lo tanto se infiere que a para el desarrollo de software que requiera gran poder de cómputo (uso extensivo de la CPU) por usuario se debe procurar siempre tener Hardware de las dimensiones adecuadas. Al contrario que otros sistemas de alta concurrencia en que la demanda de recursos viene de los accesos a disco (portales web, sitios de e-commerce, servidores de correo, etcétera), en este caso no sirven los mecanismos clásicos de escalabilidad horizontal como el pasar datos y recursos al caché del sistema. Y por la naturaleza de la aplicación Near-U, resulta lógico pensar en que los requerimientos de CPU fueron absolutamente subestimados para atender a 10000 usuarios concurrentes.

La gran pérdida de requests que se observa tiene como consecuencia negativa también una contaminación en la toma de datos del tiempo medio de respuesta del servidor. Sin embargo dicha contaminación es pareja en las tres implementaciones del servidor, por lo que se puede asumir como “ruido blanco” y obviarla al momento de concluir.

Por otro lado, al analizar los tiempos medio de demora se puede apreciar claramente el efecto de tener una estructura adecuada para calcular distancias entre puntos de un espacio euclidiano. Lo importante es que el resultado, sobre todo con números altos de usuarios, sobrepasa por amplio margen a la ventaja teórica que brinda el uso de un R-Tree. Pero este fenómeno se explica por la descongestión en los pipes de entrada y de salida de los procesos que levanta el servidor. Al delegar la tarea más pesada del servidor a un servicio externo se evita la congestión de las pilas, lo que finalmente permite tener los rendimientos observados.

Cabe notar que por el efecto de la descongestión de los pipes de los procesos del servidor, es imposible tomar el rendimiento puro del R-Tree para el cálculo de distancias euclidianas. Pero ese tampoco nunca fue el propósito del presente proyecto. Desde un principio se estableció que el proyecto estaba orientado a la construcción de un sistema robusto y eficiente, lo que cumple con creces.

Una posible mejora a futuro es el replicar el R-Tree en más de un proceso de modo de que la estructura tampoco tenga un cuello de botella por los requerimientos de los procesos del servidor. Para esto se debe asegurar de que todas las réplicas estén sincronizadas para mantener la integridad de los datos, y realizar las operaciones de sincronización de la forma más eficiente posible.

5.2. Usos en la Actualidad:

Se han desarrollado diversas soluciones y librerías informáticas que resuelven el problema. Entre las más reconocidas:

- Módulo Average Nearest Neibor de ArcGIS [7], una de las plataformas de mapas más importantes a nivel mundial.
- Extensión **PostGIS** de suite **PostGreSQL** [8], que permite construir y administrar bases de datos orientadas a objetos y espaciales.

El principal problema con los recursos antes mencionados es que no optimizan la obtención de los cruces entre las locaciones de cada cliente, sino que se limitan a crear una adaptación (variante) el problema inicial de NNS.

Áreas de uso de algoritmos de NNS:

1. Reconocimiento de Patrones (Visión Computacional).
2. Clasificación Estadística.
3. Bases de Datos Multimedia (Consultas por contenido).
4. Teoría de la Codificación y Criptografía.
5. Compresión de Datos.
6. Sistemas de Recomendaciones basados en Perfiles Personales.
7. Marketing por Internet.
8. Investigación de secuencias de ADN (biología molecular).
9. Corrección gramatical y asistencia semántica.
10. Aprendizaje de máquinas.
11. Detección de Plagio.

12. Probabilidades y estadísticas deportivas.
13. Análisis de Clusters y Data Warehouse.
14. Inteligencia Artificial.

6. Bibliografía

[1] S. Turkle, «Alone Together: Why We Expect More from Technology and Less from Each Other» ISBN 987-0-465-03146-7, New York: Basic Books, Perseus Books Group, 2011.

[2] S. Turkle, Second Half: Computers and the Human Spirit, New York: Simon & Schuster, 1984.

[3] L. E. Standard, «www.standard.co.uk,» 31 03 2008. [En línea]. Available: http://www.standard.co.uk/news/nomophobia-is-the-fear-of-being-out-of-mobile-phone-contact--and-its-the-plague-of-our-247-age-6634478.html. [Último acceso: 28 09 2013].

[4] Cooperativa, «www.cooperativa.cl,» 04 12 2013. [En línea]. Available: http://www.cooperativa.cl/noticias/tecnologia/industria/telefonia-movil/uso-de-smartphones-aumento-casi-un-50-por-ciento-en-chile/2013-12-04/084629.html. [Último acceso: 17 07 2014].

[5] S. d. Telecomunicaciones, «Subtel,» 01 03 2014. [En línea]. Available: http://www.subtel.gob.cl/informacion-estadistica-actualizada-e-historica4/informacion-estadistica4/. [Último acceso: 20 03 2014].

[6] S. d. Telecomunicaciones, «Subtel,» 28 12 2013. [En línea]. Available: http://www.subtel.gob.cl/images/stories/apoyo_articulos/notas_prensa/06032014/Informe_Estadistico_SUBTEL_2013.pdf. [Último acceso: 20 03 204].

[7] F. Richter, «Statista,» 14 01 2014. [En línea]. Available: http://www.statista.com/chart/1778/app-use-in-2013/. [Último acceso: 20 03 2014].

[8] N. S. Dennis Crowley, «Foursquare © 2014,» 11 03 2009. [En línea]. Available: https://es.foursquare.com/about/. [Último acceso: 03 21 2014].

[9] S. R. Josh Williams, «Gowalla,» 28 08 2007. [En línea]. Available: blog.gowala.com. [Último acceso: 21 03 2014].

[10] E. Migicovsky, «Facebook Places,» 18 08 2010. [En línea]. Available: /product/facebook-places. [Último acceso: 21 03 2014].

[11] W. W. J. B. C. G. J. M. a. J. M. Sean Rad, «Tinder,» 30 09 2012. [En línea]. Available: http://www.gotinder.com. [Último acceso: 21 03 2014].

[12] G. Poreh, «Snoox,» 30 11 2012. [En línea]. Available: www.snoox.com. [Último acceso: 21 03 2014].

[13] S. M. R. S. A. N. Jan Elseberg, «Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration,» *Journal of Software Engineering for Robotics, Jacobs University of Bremen,* vol. 1, nº 12, p. 12, 2012.

[14] A. Guttman, «R-TREES. A DYNAMIC INDEX STRUCTURE,» *University of California, Berkeley,* p. 11, 1984.

[15] H.-P. K. R. S. B. S. Norbert Beckmann, «The R*-tree: An Efficient and Robust Access Method for Points and Rectangles+,» *Praktuche Informatlk, Umversltaet Bremen,* p. 10, 1990.

[16] E. D. J. E. J. I. S. L. P. M. G. T. David Bremner, «Output-Sensitive Algorithms for Computing,» *Faculty of Computer Science, University of New Brunswick; MIT Laboratory for Computer Science; Computer Science Department, University of Illinois; Charge de recherches du FNRS, Universite Libre de Bruxelles; School Computer Science, Carleton University,* vol. 2748, pp. 451 - 461, 2003.

[17] J. L. Bentley, «A Survey of Techniques for Fixed Radius Near Neighbor Searching,» *US Energy Research and Development Administration,* nº 94305, p. 37, 1975.

[18] C. S. Mohammad Kolahdouzan, «Voronoi Based K Nearest Neighbor Search for Spartial Network Databases:,» *Department of Computer Science, University of Southern California, Los Angeles, CA,* vol. 30, pp. 840-851, 2004.

[19] A. Andoni, «Nearest Neighbor Search: The Old, The New, and The Impossible,» *Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology,* p. 178, 2009.

[20] J. C. Rivera, «RTree,» 23 05 2010. [En línea]. Available: https://github.com/imbcmdth/RTree. [Último acceso: 25 03 2014].

7. Anexos:

7.1. Especificación del API del servidor Near-U

A continuación se describe la API especificada por el servidor de Near-U. Para esto se usa una tabla para cada una de las funcionalidades ofrecidas por el servidor, con el siguiente formato:

HTTP + URL:	Acción HTTP: /url
Función:	Descripción de la funcionalidad ofrecida
Parámetros:	Lista de parámetros HTTP recibidos por el servidor. Cada parámetro se acompaña de los filtros que se le aplican para validarlos. Por ejemplo, una fecha debe pasar el filtro “DateTime”. Una dirección de correo electrónico debe pasar el filtro “email”, etcétera.
Respuestas:	La respuesta del servidor en formato JSON. Esta respuesta será ocupada por la aplicación cliente para actualizar su estado (información almacenada, comportamiento, etcétera).

Funcionalidad de Login:

HTTP + URL:	POST: /login
Función:	Autentificarse en el sistema. Obtener los datos de needs propios y matches logrados.
Parámetros:	<email: String | email> <password: String>

Respuestas:	{
	status: 'OK',
	usuario: {
	nombre: <String>,
	estado: “activo”,
	fecha_registro: <DateTime>,
	fecha_modificacion: <DateTime>,
	fecha_ultimo_login: <DateTime>,
	needs: [
	{
	descripcion: <String>,
	tipo: <String>,
	fecha_creacion: <DateTime>,
	matches: [
	{
	descripcion: <String>,
	fecha: <DateTime>,
	owner: <String | email>,
	}*
	]
	}*
	]
	}
	}

Funcionalidad de Registrarse en la Aplicación:

HTTP + URL:	POST: /signup
Función:	Registrarse en el sistema como un usuario nuevo
Parámetros:	<nombre: String> <email: String | email> <password: String> <password_conf: String | ==’password’>

Respuestas:	{
	status: ‘OK’
	}

Funcionalidad de Añadir un Nuevo Need:

HTTP + URL:	POST: /add_need
Función:	Añade nuevo need al usuario
Parámetros:	<email: String | email> <descripcion: String> <tipo: String>

Respuestas:	{
	status: ‘OK’
	}

Funcionalidad de Obtener todos los Matches con Needs de otros usuarios:

HTTP + URL:	POST: /matches
Función:	Actualiza la ubicación de un usuario. Obtiene como respuesta lista de los usuarios cercanos con los que se tenga al menos algún match, y lista de sus needs que contengan un nuevo match.
Parámetros:	<email: String | email> <latitude: float> <longitude: float> <distance: int>

Respuestas:	{
	status: 'OK',
	matched_users: [
	{
	nombre: String,
	email: String,
	latitude: float,
	longitude: float,
	new_matched_needs: [
	{
	id: int,
	descripcion: String,
	tipo; String,
	my_matched_needs:[
	Descripcion<String>*
	]
	}
	]
	}*
	]
	}

7.2. Descripción de API de RTree

A continuación se describe el API utilizado sobre la estructura RTree sobre un servidor con NodeJS:

HTTP + URL:	POST: /close_users
Función:	Obtener los usuarios cercanos, dentro de un área determinada.
Parámetros:	<email: String> <latitude: float> <longitude: float> <distance: int>

Respuestas:	{
	closeUsers: [
	{
	email: String,
	latitude: float,
	longitude: float
	}*
	]
	}