# AreyouIN? By Peeple

# Description

*AreYouIN* es una app móvil que simplifica la forma en la que quedas con tus amigos y te ayuda a estar al día y organizar las múltiples actividades que te proponen tus amigos para realizar durante la semana. Con *AreYouIN* podrás saber al instante qué van a hacer tus amigos esta noche y apuntarte al plan que más te guste. Además, podrás hacer planes con otras personas de tu ciudad de modo que puedas conocer a gente nueva haciendo aquello que más te gusta.

*AreYouIN* es la aplicación ideal para hacer cosas con tus amigos de siempre y hacer nuevas amistades en tu ciudad.

# Technical Description

El servicio que debe ofrecer *AreYouIN* es muy sencillo desde el punto de vista conceptual. Este consiste básicamente en una forma de comunicar eventos entre diferentes usuarios, de modo que un usuario pueda publicar un evento y el resto de usuarios pueda recibirlo y decidir entonces si participar en dicho evento.

Desde el punto de vista conceptual, la aplicación manejará principalmente eventos. Este será el tipo de datos que manejará toda la lógica de negocio. El papel de las personas en la aplicación es importante pero secundario conceptualmente. Por ejemplo, el rol de una persona en cada evento será como autor (creador del evento), destinatario (persona o grupos de personas a quien se dirige el evento) o participante (subconjunto de personas de entre los destinatarios que deciden participar en el evento). En ese sentido, la persona podría ser un mero identificador sino fuera porque es necesario que esta se autentifique, así como poder conocer su lista de amigos en base a sus contactos y otros medios, razón por la cual la persona como tipo de datos tendrá también entidad propia junto a los eventos. A los eventos y las personas se une un tercer elemento, los mensajes.

Los mensajes son empleados para comunicar eventos a otros usuarios, así como para manejar una serie de operaciones que pueden realizarse sobre dichos eventos, como por ejemplo añadir participantes, modificar eventos, etc. En definitiva, los mensajes sirven para comunicar distinta información a los usuarios. Los mensajes se enviarán empleando un protocolo binario que funcionará directamente sobre TCP/IP empleando sockets TLS (canales cifrados).

El servicio consiste en dos partes bien diferenciadas. Por un lado el cliente, que consistirá en una aplicación móvil sencilla que permitirá enviar y recibir eventos. Por otro lado, el servidor o backend encargado de proporcionar el servicio haciendo uso de la infraestructura necesaria. Por tanto, la arquitectura deberá derivar de la típica cliente-servidor, pudiendo haber idealmente más de un servidor (sistema distribuido). La comunicación entre el cliente y el servidor se llevará a cabo empleando paso de mensajes haciendo uso de un protocolo que se definirá en este documento.

De forma general, el servicio debería funcionar de la siguiente manera. El cliente, empleando el protocolo que se defina, envía un mensaje de creación de un nuevo evento al punto de entrada del servicio, el gestor de la sesión con el cliente. En este mensaje va incluida la lista de destinarios (solo en caso de eventos privados, en los públicos debería cambiar). El gestor de la sesión recibe dicho mensaje, crea el evento y lo guarda en la base de datos de eventos. Tras esto, el gestor de la sesión puede confirmar al cliente la recepción del evento. A partir de esta confirmación, el servicio debe garantizar que el mensaje llega a los destinarios dado que este todavía no se ha enviado. Dado que los destinatarios pueden no estar todos online, es necesario añadir un nivel de indirección de modo que el mensaje se conserve el tiempo necesario en el servidor hasta su entrega. Cuando el cliente está online, su gestor de sesión le hará llegar los mensajes pendientes. En este momento, el mensaje ya habrá llegado a su destino.

En cuanto a las garantías de entrega del mensaje, garantizar que una vez recibido el mensaje por parte del gestor de la sesión este llega a entregarse es más complicado de lo que parece. Antes de iniciarse el proceso de entrega del mensaje a los destinatarios, o durante el mismo, podrían suceder diversos errores: el proceso podría fallar, la base de datos puede tener un error, pueden surgir problemas en la red, etc. Todos estos problemas dificultan esta garantía además de aumentar la latencia desde que un cliente envía un mensaje (publicar evento) hasta que es recibido por otro cliente (notificar nuevo evento o cambios). Una alternativa sería que el gestor de la sesión no confirmase al cliente la recepción hasta que realmente se ha llevado a cabo la entrega del mensaje de forma síncrona a todos los destinatarios. Sin embargo, la espera no sería constante y dependería del número de destinarios que tuviera el mensaje. Por tanto, a priori, esta opción no es recomendable. Sería conveniente, pues, cambiar la recepción del mensaje de modo que este se distribuya de forma interna entre diferentes nodos y no se confirme al cliente la recepción hasta que por lo menos 3 nodos del servicio interno confirmen su recepción al gestor de la sesión con el cliente. Pero todo esto complica enormemente la gestión de los eventos, sobre todo en lo concerniente a futuras modificaciones.

Aunque todavía faltaría comentar el tema de las actualizaciones, con todo lo comentado hasta el momento, se puede intuir que el uso de un *Message Oriented Middleware* (MOM) puede gestionar muchos de los procesos relacionados con los mensajes. Idealmente, un *Distributed Message Queue* con persistencia de mensajes sería recomendable. Mientras que para distribuir la base de datos de eventos Cassandra sería la opción recomendada. NOTA: Argumentar mejor, ¿Por qué Cassandra? ¿Qué características tiene a parte de ser NoSQL y distribuída?. En cuanto al MOM, su uso tiene sentido ya que voy a emplear mensajes, pero no queda tan claro la necesidad de persistencia. Esto último dependerá de si existe un almacén de los eventos a partir del cual se pueda crear el estado de todos los usuarios en cuanto a los eventos pasados y futuros, o si este estado se crea a partir de los mensajes enviados/recibidos. Por otro lado, la persistencia sería solo necesaria para comunicar con usuarios offline, ¿pero sería también necesaria para mensajes ya entregados?.

# Requirements (Backend)

* El identificador público de un usuario es su e-mail. Por tanto, no pueden haber dos usuarios con el mismo e-mail. Igualmente, el identificador empleado internamente consistirá en un número único de 40 bits (5 bytes).
* La cuenta de usuario debe tener una clave fuerte.
* Un usuario solo puede tener un número de teléfono asociado a su cuenta y este no puede pertenecer a otro usuario, es decir, no se repite.
* La cuenta de usuario se vinculará a un único dispositivo por motivos de seguridad. Si se inicia sesión desde otro dispositivo habría que realizar dicho emparejamiento desactivando el dispositivo antiguo y volviendo a verificar el número.
* El emparejamiento se realiza entre la cuenta de usuario, el dispositivo y el número de teléfono. De hecho, hasta que no se tiene número de teléfono la cuenta de usuario no se empareja con el dispositivo.
* Solo puede haber una sesión activa del usuario a la vez. Esto supondrá un problema a la hora de sacar más interfaces de acceso como, por ejemplo, una versión basada en web.
* Permitir el envío de eventos a usuarios incluso cuando están offline.
* Una vez confirmada la recepción de un mensaje al cliente, en caso de que el mensaje necesite reenviarse a una lista de destinatarios, deberá garantizarse que el mensaje llega al destino.
* Soportar millones de usuarios. El servicio debe poder escalar linealmente respecto al número de nodos. Es decir, si para 1 nodo tengo 150 ops (operaciones por segundo), para 2 nodos debería tener 300 ops, y para 3 nodos 450 ops, y así sucesivamente.
* Por otro lado, las operaciones a realizar dependerán del número de usuarios, N, por lo que su complejidad debería ser del orden O(N) en el peor de los casos.

# Message Life Cycle

# Cuando un cliente envía un mensaje, este debe permanecer en el sistema hasta que los destinatarios lo reciben. Una vez recibido, el mensaje se borra completamente del servidor.

# Event Life Cycle

Cuando se crea un nuevo evento, este estará vigente el tiempo indicado hasta que, finalmente, el evento expire. En cualquier momento el evento puede ser eliminado por el creador, en ese caso se considera que el evento ha sido cancelado. Por tanto, el ciclo de vida de un mensaje es el siguiente:

1. Pendiente de publicar. El mensaje ha sido recibido por el servidor, confirmado al cliente y creado en la base de datos. Pero todavía no se ha iniciado el proceso de envío a los destinatarios. Hasta ese momento el mensaje estará pendiente de publicar. Una vez se haya iniciado el proceso y se haya enviado a, al menos, un usuario se considerará publicado.
2. Publicado. El mensaje ha sido enviado a, al menos, un usuario y todavía no ha caducado. Únicamente durante este estado se puede modificar la lista de participantes.
3. Expirado. El mensaje ha caducado debido a que la fecha de finalización del evento ya se ha superado. En este estado el evento se congela y no puede ser modificado ni cambiar de estado.
4. Cancelado. El autor del evento ha decido cancelarlo. El evento no se puede modificar ni cambiar de estado.

**NOTA:** Falta diagrama de estados indicando cómo cambiar entre los diferentes estados.

# Architecture

*AreYouIN* debe soportar un gran número de usuarios, del orden de millones, conectados al mismo tiempo al servicio. Igualmente debe estar operativo todo el tiempo y el tiempo de respuesta de los servicios ofrecidos debe ser el menor posible, del orden de los milisegundos. Además, debe poder crecer de forma lineal con respecto al número de nodos haciendo así posible crecer en número de usuarios sin tener que realizar grandes inversiones en nuevos equipos por lo que, principalmente, la arquitectura debe ser capaz de escalar de forma horizontal añadiendo nuevos equipos. Con estas características es necesario diseñar la arquitectura de un sistema distribuido.

Como se ha comentado, el problema a resolver consiste en permitir que millones de usuarios envíen mensajes a otros usuarios. Esos mensajes tendrán verbos bien definidos para crear, borrar y modificar eventos. Por lo que los mensajes darán lugar a la creación de objetos (eventos) y modificarán el estado de los mismos.

La arquitectura deberá soportar el envío de mensajes entre clientes, incluso cuando estos no están conectados al servicio. El envío de un mensaje al servicio debe tener muy poca sobrecarga (milisegundos). Una vez un mensaje ha sido recibido y confirmado al cliente, la arquitectura debe garantizar que el mensaje llega a su destino, incluso en situaciones de malfuncionamiento (fallo de la red, nodos caídos, etc.). Finalmente, la arquitectura debe permitir que, una vez publicado un evento (mensaje recibido y confirmado, evento creado y enviado a, al menos, un destinatario), nuevos amigos o contactos del autor del evento puedan recibir dicho evento. Igualmente, amigos existentes durante la publicación del evento que, habiendo recibido el evento, lo han perdido debido a cualquier error en el cliente, deben ser capaces de poder recuperar todos los eventos publicados y vigentes que se les ha enviado en algún momento.

La solución para el tema de los mensajes será construir, para cada cliente, dos colas de mensajes –una para los mensajes salientes (OUTBOX) y otra para los mensajes entrantes (INBOX)–. De este modo, cuando un cliente envía un mensaje, este será almacenado en su cola de mensajes salientes en el servidor hasta que se envíe a los destinatarios. Dado que cada cliente del servicio tendrá una cola de mensajes entrantes en el servidor, el envío a todos los destinatarios siempre podrá realizarse. Una vez todos los destinatarios tienen el mensaje en sus respectivas colas de mensajes entrantes, el mensaje se elimina de la cola de mensajes salientes inicial. Cuando los clientes están conectados, reciben todos los mensajes que hay en las respectivas colas de mensajes entrantes y se eliminan los mensajes conforme se confirme la recepción por parte del cliente. Es importante notar aquí que las colas pueden estar distribuidas o no, disminuyendo o aumentando la probabilidad de perder un mensaje, respectivamente. En cuanto al INBOX y OUTBOX, independientemente de que sea distribuido, debería ser persistente para poder recuperar mensajes que no han sido entregados en caso de fallo del nodo o falta de suministro de energía.

En cuanto a los eventos, una vez recibido en el servidor y confirmado al cliente, este se almacenará en la base de datos de eventos. Cualquier mensaje que modifique el estado del evento alterará también el correspondiente registro en la base de datos. El evento nunca se eliminará de la base de datos, salvo casos excepcionales. Igualmente, cuando un cliente recibe un mensaje de que se ha publicado un nuevo evento y confirma su recepción al servidor, el servidor creará dicho evento en la base de datos de eventos. El cómo gestionar esta duplicidad de datos tendrá que resolverse de algún modo. No obstante, el contar con todos los eventos publicados y recibidos, facilita la recreación de mensajes en caso de que el cliente pierda los eventos recibidos o necesite sincronizar su estado con el servidor.

Con todo esto, a continuación se presentan tres soluciones arquitecturales, cada una con sus ventajas y desventajas:

**Opción 1**

* INBOX distribuido y base de datos de eventos distribuida.
* Comunicación entre nodos a través del INBOX.

De las tres arquitecturas desarrolladas la más sencilla es esta primera opción. En la Figura 1 puede verse en que consiste. Aunque en la figura aparecen únicamente 2 nodos situados tras un balanceador de carga, el número de nodos puede ser mucho mayor. En esta arquitectura cada nodo soporta varios clientes simultáneamente, donde cada cliente cuenta con una serie de recursos: cola de mensajes entrantes (INBOX), base de datos de eventos y un gestor de la sesión que gestiona la comunicación con el cliente. Este último gestor estará presente únicamente cuando exista una conexión activa entre el servicio y el cliente. Por tanto, el sistema tendrá un gestor por cada cliente conectado.

En esta arquitectura, tanto el INBOX como la base de datos de eventos está distribuida. Debido a esto, el gestor de la sesión puede escribir en el INBOX de cada cliente independientemente del nodo en el que esté. Es decir, la comunicación entre nodos se realiza a través del INBOX. Sin embargo, únicamente puede leer y eliminar mensajes de su propio INBOX, así como añadir mensajes a su propia base de datos de eventos. Cuando un cliente envía un mensaje de publicación, su gestor de la sesión escribe directamente en el INBOX de cada usuario al que el mensaje va dirigido. Una vez se ha escrito el mensaje en todos los INBOX, así como en la base de datos de eventos del propio autor del mensaje, el gestor de la sesión confirma al cliente la recepción y publicación del mensaje. La recepción de los mensajes por parte de los destinatarios se realizará cuando estos estén conectados. En ese caso, únicamente tendrán que leer su INBOX. Como se ha dicho anteriormente, conforme se van recibiendo los mensajes se crean los eventos en la base de datos de eventos de cada cliente y se elimina el mensaje del INBOX.

La principal ventaja de esta arquitectura es su simplicidad. La idea es tener dos tipos de contenedores, distribuidos, uno para el intercambio de mensajes y otro para almacenar el estado de los eventos. Al ser distribuido la comunicación es sencilla y cada session handler se encarga solo de un cliente. Se puede separar nodos de datos de nodos que ejecuten session handlers de diferentes usuarios, si se desea. También hay que destacar el carácter síncrono durante la publicación del mensaje. La desventaja reside en el tiempo de publicación de un mensaje, dado que involucra demasiada comunicación entre diferentes nodos de la red. Igualmente, mientras se realiza la publicación el cliente no puede realizar ninguna otra comunicación con el servicio por lo que la comunicación se congela hasta que finalice. Otra desventaja es la duplicidad dado que el mismo evento se envía a diferentes INBOX dando lugar, posteriormente, a la creación del mismo evento en cada base de datos. Esto da lugar a que las modificaciones de los eventos deben también difundirse entre todos los nodos y cada uno debe aplicar esos cambios. No se controla el número de réplicas de un evento, correspondiéndose este con el número de destinatarios de dicho evento. Malgasta espacio.



Figura 1 Arquitectura del sistema 1.

**Opción 2 (mi preferida)**

* INBOX y OUTBOX distribuido.
* Base de datos de eventos distribuida.
* Comunicación entre nodos a través del INBOX y OUTBOX.

La principal diferencia con el caso anterior consiste en la introducción de otro elemento más que también está distribuido, el OUTBOX (ver Figura 2). Este añade un nivel de indirección de modo que, desde el punto de vista del cliente, la publicación sea más rápida. En este caso, cuando un cliente publica un mensaje, su gestor de la sesión escribirá dicho mensaje en el OUTBOX y confirmará al cliente la recepción del mismo (debería asegurarse antes que el mensaje está almacenado en al menos 3 nodos distintos). Este proceso es mucho más rápido y supone menos sobrecarga para el gestor de la sesión. Dado que el OUTBOX es distribuido, la entrega del mensaje a los destinatarios puede realizarse por cualquier nodo de la red. El *Message Dispatcher* será el encargado de realizar esta tarea. Para ello, leerá el OUTBOX del cliente y procesará sus mensajes escribiéndolos en el INBOX de los destinatarios. Una vez escritos borrará los mensajes del OUTBOX. En esta arquitectura el gestor de la sesión únicamente puede leer de su INBOX y escribir tanto en su propio OUTBOX como en el banco de datos de eventos. Al igual que antes, la comunicación entre nodos se lleva a cabo entre los INBOX de la mano del *Message Dispatcher*. Es importante mencionar que este elemento no pertenece a los recursos del cliente, sino que es un proceso a nivel de nodo. De este modo, se puede dedicar un número limitado de *Message Dispatcher* (por ejemplo 1, 2 o 3) para que gestionen los OUTBOX de miles de clientes.

La ventaja de esta arquitectura es que la publicación (realmente recepción del mensaje por parte del servicio) es más rápida, no dejando congelada la comunicación. Igualmente, la seguridad se puede controlar mejor al hacer que el gestor de la sesión no



Figura 2 Arquitectura del sistema 2.

sobrepase los límites de sus propios recursos, es decir, se puede limitar su funcionamiento para que solo pueda leer de sus recursos. Igualmente, se puede adaptar mejor la capacidad de publicación del servicio al estar limitado el número de procesos que pueden llevar a cabo dicha tarea. En la primera arquitectura se podía exceder fácilmente si, por ejemplo, todos los clientes publicaran un evento al mismo tiempo.

No le encuentro desventajas más allá de que puede aumentar la complejidad del sistema. Tanto en este caso como en el anterior dependo de que exista un *Distributed Message Queue* que me pueda servir, ya que no voy a implementar yo los INBOX y OUTBOX empleados para encolar mensajes.

**Opción 3**

* INBOX y OUTBOX locales.
* Eventos distribuidos.
* Comunicación entre nodos a través del *Message Dispatcher*.

La principal diferencia reside en la utilización de INBOX y OUTBOX locales. Al tener tiempos de vida de los mensajes tan cortos, el OUTBOX puede ser local. Tal vez, el tiempo de vida de los mensajes en el INBOX puede ser mayor en algunos casos debido a que depende de que el cliente esté conectado o no. En este caso, al no estar distribuidos ninguno de los dos cada nodo tendría los “buzones” de determinados clientes. Por tanto, los *Message Dispatcher* tendrían que comunicarse entre sí para

redirigir mensajes entre nodos (ver Figura 3). Así, cuando un cliente envía un mensaje, el gestor de la sesión lo almacena en el OUTBOX y lo confirma al cliente inmediatamente. Entonces, el *Message Dispatcher* procesa el mensaje, establece la lista de destinatarios y, escribe directamente en el INBOX de cada uno en caso de encontrarse estos en el propio nodo. En caso de estar en otro nodo, redirigirá el mensaje al *Message Dispatcher* del otro nodo para que sea este último el que escriba el mensaje en el INBOX del destinatario.

La ventaja es que los INBOX y OUTBOX pueden implementarse de forma muy sencilla sin depender de un servicio externo más (no hace falta un *Message Queue*). Los eventos ya publicados siguen almacenándose de forma segura y distribuida por lo que en caso de fallo, los eventos ya publicados no se pierden y se puede crear el estado del cliente. Sí que se pierden los eventos pendientes de publicar y cuya recepción se confirmó al usuario. Además, sería necesario programar toda la lógica necesaria para migrar a clientes de un nodo a otro y forzar la resincronización con los clientes cuando estos se conectan.

La desventaja es que, cualquier fallo en el nodo provocaría una degradación en el servicio que afectaría a los clientes de ese nodo (bien debido a la sobrecarga de la lógica para gestionar esas situaciones, o porque debe forzar resincronización) y posiblemente también a otros. Igualmente, el balanceo entre nodos no podría realizarse correctamente dado que cada cliente está atado a un nodo concreto. Además, dado que la entrega de los mensajes a los destinatarios involucra varios *Message Dispatcher* que escriben en los INBOX y leen y borran mensajes de los OUTBOX, se hace necesario realizar este proceso empleando transacciones, es decir, diferentes protocolos para saber cuando se puede borrar o no un mensaje en función de si se pudo entregar o no.

Una solución podría ser añadir también lógica de negocio para distribuir los mensajes pero, en ese caso, estaríamos reinventando la rueda al implementar un *Distributed Message Queue*, que al ser propio lo entendería mejor y podría modificarlo a mi gusto cuando fuera necesario. Pero seguramente cometeré muchos errores al desarrollarlo y me aparta de mi aplicación.

# Dudas sobre arquitectura

La arquitectura 2 es la que más me gusta y mejor se ajusta al tiempo disponible. No obstante, la arquitectura 1 podría ser la que desarrollara inicialmente debido a su simplicidad. No obstante, en cualquier caso, necesito un *Distributed Message Queue*. Las alternativa que considero son las siguientes:

* NATS (gnatsd, [www.nats.io](http://www.nats.io)). Cola de mensajes distribuida entre varios nodos. El clúster de nodos actúa como bróker. De entre las colas de mensajes de este tipo, esta es la que mejor rendimiento tiene, capaz de enviar y recibir 200 mil msg/seg. Soporta semántica pub-sub, es decir, crear un *topic* al que se suscriben varios consumidores y publicar mensajes dirigidos al mismo que son recibidos por todos los suscriptores. En mi caso, cada evento tendría un *topic* y los destinatarios se suscribirían al mismo automáticamente para recibir todos los cambios que sufra dicho evento. Es decir, el OUTBOX e INBOX se abstraería mediante NATS. Cada cliente del servicio tendría su equivalente en NATS en forma de productor/consumidor. El gestor de la sesión al recibir un mensaje para publicar un evento, crearía un nuevo *topic*, suscribiría a los destinatarios a ese *topic* y enviaría el mensaje a ese *topic*.

# Workflows

**Open application**

Cuando la aplicación se abre, se comprueba que no se haya hecho *login* anteriormente. En caso afirmativo, se muestra la pantalla para elegir si crear una nueva cuenta de usuario o si se entra en una cuenta de usuario ya existente. En caso de que se hubiera hecho *login* anteriormente, se presenta la pantalla de login y, en caso de que se disponga de un token de usuario, se iniciará la identificación automática.

**Sign up (registration)**

En la pantalla de registro se pueden elegir varias opciones: Facebook, Google o añadir los datos de forma manual. Los campos necesarios para el proceso de registro son:

* *Nombre para mostrar*. Este sería el nombre de la persona o un alias.
* *E-mail* . Se utiliza como nombre de usuario. Internamente se asocia a uno u otro un identificador numérico de 5 bytes.
* *Clave*. En caso manual, será necesaria también una clave para poder volver a utilizar la aplicación si se desinstala. La clave solicitada no se usará de manera muy común en la aplicación móvil (sí en cualquier versión de escritorio) por lo que es necesario informar al usuario de la importancia de la clave y de que la memorice para poder recuperar su cuenta en cualquier momento.
* *Device ID*. Identificador único del dispositivo.

Además de los datos anteriores, también se podrían solicitar en algún momento los siguientes datos opcionales:

* *Edad*. Puede ser útil para saber que tipos de eventos públicos mostrar.

Una vez se introducen los datos o se conecta al servicio mediante Facebook/Google por primera vez, en el lado del servidor es necesario llevar a cabo la creación de la cuenta del usuario, empleando el e-mail como usuario y asociándole al mismo un UserID interno (5 bytes), además de almacenar el modo de acceso. En caso de introducir los datos de forma manual, será necesario también verificar el e-mail proporcionado.

A partir de este punto, el usuario ya está registrado e identificado en el sistema. Sin embargo, será necesario asociar un número de teléfono con la cuenta de usuario. Por tanto, al final del proceso de registro debe introducirse el número de teléfono que se asociará a la cuenta y verificarse. Al final del proceso, la cuenta quedará vinculada al número de teléfono y dispositivo.

La ventaja de tener un nombre de usuario que no sea el teléfono es que el usuario podrá acceder al servicio en el futuro también a través de la web facilitando su propio nombre de usuario y clave (o conectando a través de Google/Facebook). La ventaja de contar inicialmente con un número de teléfono es que la búsqueda de contactos que ya cuenten con la aplicación se facilita. **Por tanto, para no frenar la expansión de la aplicación es imprescindible contar con el teléfono desde el principio aunque se llegue a incurrir en un coste para el usuario y suponga algún problema de seguridad.**

Cuando se procesan los datos es importante comprobar que el ID generado no existe previamente y que tanto el e-mail como el teléfono no se han utilizado antes. En caso de que haya discrepancias no se creará la nueva cuenta de usuario. Por ejemplo, si el usuario intenta registrarse con un e-mail ya existente, debería avisarse y, tal vez, presentarse la pantalla de iniciar sesión (login). Si el teléfono asociado ya existe debería tratarse de la misma manera. Es importante que la cuenta de usuario se cree realmente cuando tanto el e-mail como el número de teléfono son nuevos.

Tabla 1. Resolución de conflictos

|  |  |  |
| --- | --- | --- |
| E-mail | Teléfono | Acción |
| 0 (usado o no válido) | 0 (usado o no válido) | Informar error de e-mail inválido. Nunca se llega a validar el teléfono. |
| 0 | 1 | Imposible. Si el e-mail es inválido no se puede ingresar el teléfono. En caso de solicitar primero el teléfono y es válido pero el e-mail proporcionado no coincide, debería permitirse recuperar la cuenta. |
| 1 | 0 | Se informa de que el número de teléfono es inválido y se permite corregirlo. |
| 1 | 1 | Se pasa a la pantalla principal de la aplicación. |

Puede darse el caso de que el usuario no haya cambiado de teléfono pero no recuerde el e-mail con el que se registró. En ese caso, sería necesario recuperar la cuenta. Del mismo modo, si el usuario pone un e-mail nuevo vinculado al teléfono que pertenece a otra cuenta, se debería poder recuperar la cuenta de algún modo.

Cuando el usuario se acaba de registrar, inmediatamente estará ya identificado (login) en el sistema (sesión creada y activa). El teléfono guardará un token de tamaño fijo (32, 64 o 128 caracteres) que lo identificará en posteriores sesiones. Por lo que, salvo la primera vez, el resto de veces no será necesario que introduzca datos de acceso. Además, la cuenta de usuario habrá quedado vinculada al dispositivo.

Por tanto, los pasos a seguir serán los siguientes:

1. El usuario se descarga la aplicación de Google Play o Apple Store.
2. La aplicación se instala en el dispositivo y el usuario la abre.
3. Si es la primera vez que se abre la aplicación y no tiene cuenta de usuario vinculada se procede con el registro.
   1. Se introducen los datos personales (e-mail, clave, nombre para mostrar, device id) o se accede a través de Google/Facebook que nos da acceso al e-mail, nombre para mostrar y lista de amigos.
   2. En caso de no usar ni Google ni Facebook, se procede a verificar el e-mail proporcionado.
   3. Se crea el usuario junto a un ID interno (5 bytes). Además, se almacena un token en el terminal para identificarse automáticamente en posteriores accesos al servicio.
   4. En este punto el usuario estaría registrado e identificado ante el servicio.
   5. Ir al paso 5.
4. Sin embargo, si es la primera vez pero sí se tiene cuenta de usuario se procede a identificarse ante el sistema, bien través de Facebook, Google o introduciendo e-mail y clave, dando lugar a la creación de un nuevo token que identifica al usuario ante el sistema de forma automática.
5. (OPCIONAL en este flujo) Se solicita el número de teléfono y este se verifica. Al finalizar, la cuenta de usuario se vincula con el número verificado y el device id proporcionado.

**Sign in (login)**

Cuando el usuario abre la aplicación en la pantalla de login, se comprueba primero si se ha iniciado sesión anteriormente y existe, por tanto, un token de identificación. En caso de existir, se lleva a cabo la identificación automática proporcionando dicho token (no caduca) y device id. Si no se dispone de dicho token por ser la primera vez que se inicia sesión en el servicio, se permitirá hacer login introduciendo <e-mail> y <clave> o mediante Facebook/Google. Junto a la misma petición de login se enviará también el <device id>.

El servidor procesa entonces los credenciales, ya sea token, usuario y clave o Facebook/Google. La correcta identificación dará lugar a la creación de un token que se almacenará en el terminal y que se usará para identificarse de forma automática en los accesos siguientes.

1. Se abre la pantalla de identificación.
2. Si no se dispone de un token para hacer login de forma automática, se presentará al usuario varias opciones: acceder mediante Facebook, Google o introducir su usuario y clave. En caso contrario, se utiliza el token como credencial.
3. Se envían los credenciales (datos o token) al servidor junto al device-id.
4. Posteriormente, se comprueban los credenciales en el servidor y se identifica al usuario.
5. Finalmente se crea, en caso de no existir o porque necesite actualizarse, un token en el terminal que servirá para identificarse de forma automática en posteriores accesos.

**Sign out (logout)**

**Retrieve account**

**Verify e-mail**

En el caso del e-mail, la validación típica consiste en enviar un e-mail con un código o enlace. El usuario ingresará el código recibido en la aplicación o, siguiendo el enlace, el código será capturado automáticamente por la aplicación verificando que el correo existe y se tiene acceso al mismo. **Estudiar problemas de seguridad.**

**Verify phone number**

Lo habitual es enviar un SMS con un código al número de teléfono facilitado. Cuando se recibe el mensaje, la aplicación puede leer el mensaje para comprobar el código de forma automática. En caso contrario, el usuario necesitará introducir dicho código de forma manual en la aplicación. Para poder verificar el teléfono sería necesario contar un proveedor *Bulk SMS* para enviar esos mensajes. **Una solución barata es enviar un SMS desde el propio móvil a sí mismo. El coste del SMS recae sobre el usuario, por eso sería importante informar antes y decir que si tiene un plan de tarifa plana de SMS este mensaje no tendrá coste. Lo malo es la seguridad al tener que confiar en el cliente, el cual puede no ser confiable. Aunque WhatsApp utiliza este método, es necesario estudiar problemas de seguridad.**

Otra alternativa es hacer la verificación siguiendo un enfoque P2P. Es decir, en lugar del usuario verificarse a sí mismo, el usuario envía un mensaje a otro usuario para verificarlo. Por ejemplo, dados dos usuarios que se acaban de registrar en el servicio y están en el proceso de verificar su número:

1. Usuario 1 envía petición de verificar <número 1> al servidor.
2. Usuario 2 envía petición de verificar <número 2> al servidor.
3. El servidor contesta al Usuario 2 con el <número 1> y <código 1>.
4. Usuario 2 envía SMS a <número 1> con <código 1>.
5. Usuario 1 recibe mensaje de <número 2> con <código 1>.
6. Usuario 1 envía petición al servidor con <código 1>.
7. Servidor comprueba que el <código 1> recibido es el mismo que el Usuario 2 envío a <número 1> por lo que el número de teléfono queda verificado.
8. Para verificar al Usuario 2 se llevaría a cabo el mismo procedimiento, bien empleando a Usuario 1 para enviar un SMS a usuario 2, o empleando cualquier otro usuario.

En este caso, se optará por la solución más sencilla y económica. Por tanto, la verificación del número de teléfono se realizará enviando un mensaje desde el propio terminal en lugar de utilizar un proveedor de *Bulk SMS*. El proceso sería el siguiente:

Siempre que el usuario accede al sistema tras haberse identificado correctamente, se lleva a cabo la comprobación del número de teléfono y posterior verificación, en caso de ser necesario. En todo inicio de sesión el servidor comprobará que la cuenta del usuario tiene un número de teléfono asociado (y por tanto un device-id, o viceversa) dado que, durante el registro, el último paso (paso 5) pudo no haberse completado.

Si la cuenta no tiene un número asociado, se solicitará el número de teléfono al usuario y este se verificará. Al finalizar, la cuenta de usuario se vincula con el número verificado y el device id.

Si la cuenta dispone ya de un número de teléfono, se comprobará que el terminal desde el que se inició sesión corresponde con el mismo terminal usado para verificar el número. Para ello se comparará el device id almacenado en el servidor y el device id proporcionado durante el login. Si ambos coinciden, se da por hecho que el terminal continua con el mismo número. Este debería ser el caso general. Sin embargo, puede provocar que si instalo la aplicación en un terminal, verifico número y desinstalo; la siguiente instalación no solicitará verificación del número.

En caso de no coincidir significará que se ha iniciado sesión desde otro terminal y, por tanto, el número de teléfono puede ser otro. En este caso, es necesario llevar a cabo una verificación del número de teléfono. Tras el proceso, si el número es el mismo se asignará el nuevo device id y se desactivará el acceso desde el device id anterior. En caso de que el teléfono sea diferente, se realizará el cambio y se enviará un SMS al teléfono antiguo con información para deshacer dicho cambio en las 72 h siguientes. Esto es necesario para impedir, en cierta medida, el robo de una cuenta de usuario permitiendo además realizar un cambio de número. **NOTA:** Podría estar bien introducir algún tipo de seguridad en plan, bloquear transferencia de número o cambio de número, etc.

Al finalizar el proceso, la cuenta de usuario estará vinculada al nuevo dispositivo y número de teléfono verificado(desactivándose la anterior en caso de existir).

1. El usuario se identifica ante el sistema.
2. Se comprueba si la cuenta del usuario tiene un número asociado.
3. Si NO hay número asociado:
   1. Se solicita al usuario que escriba su número de teléfono, indicando el código del país (+34, +31, etc.).
   2. Se envía una petición al servidor indicando que se va a proceder a verificar el número <número>.
   3. Si el número no está en la base de datos, el servidor contesta con un <código> aleatorio de 6 dígitos. En caso contrario, se cancela la verificación.
   4. El terminal envía un mensaje a <número> con <código>.
   5. El terminal recibe el mensaje y comprueba que el origen y código recibidos corresponde con <número> y <código>.
   6. En caso de éxito, el terminal contacta al servidor informando que <número> es válido.
4. Si hay número asociado:
   1. Se comprueba el *device id* del terminal con el almacenado para el número de teléfono asociado.
   2. (caso general) Si ambos coinciden se da por hecho que el terminal no ha cambiado de número y se acaba la verificación (ya había sido verificado).
   3. (caso especial) Si no coinciden es necesario llevar a cabo la verificación del número de teléfono del mismo que se realiza en 3.
   4. Si tras la verificación el número es el mismo, se asignará el nuevo *device-id* y se desactivará el acceso desde el *device-id* antiguo.
   5. Si tras la verificación el número es diferente, se realiza el cambio y se envía un SMS al teléfono antiguo con información para deshacer dicho cambio en las 72 h siguientes.
5. Al finalizar el proceso, la cuenta de usuario estará vinculada al nuevo dispositivo y número de teléfono verificado(desactivándose la anterior en caso de existir).

**Find friends automatically**

No contar con el número de teléfono verificado desde un principio supone que este no puede ser empleado en el proceso de encontrar amigos ya que, si se utiliza un número no verificado, cualquiera puede mentir y por tanto se estarían descubriendo personas que no deberían. El registro debería prevenir esto en la medida de lo posible y poner algún tipo de restricción de seguridad a la hora de qué amigos pueden encontrarse. Esto sería un fallo enorme para la privacidad del resto de usuarios que esperan que sus amigos de verdad los puedan encontrar. A continuación se muestran los mecanismos que pueden explotarse para encontrar amigos que utilicen el servicio:

* *Buscar amigos a través de Google Plus*. Independientemente de cómo se registrara el usuario, se pueden buscar amigos de Google Plus que hagan uso de la aplicación y añadirlos automáticamente.
* *Buscar amigos a través de Facebook*. Al igual que con Google Plus, también se pueden descubrir amigos empleando Facebook. De hecho, esta opción debería desarrollarse antes por tener más usuarios.
* *Buscar amigos a través de los contactos y el e-mail*. Del mismo que se hace con el número de teléfono, se podría utilizar el propio correo electrónico que tenemos de cada contacto para saber si es usuario de la aplicación o no. No obstante, al contrario que con el número de teléfono, poca gente guarda realmente el correo electrónico de sus amigos.
* *Buscar amigos a través de los contactos y el número de teléfono*. Igual que la anterior pero con el número de teléfono en caso de ser un número verificado (requisito).
* *Añadir contactos de manera individual*. Como se suele hacer en las aplicaciones IM, una forma sería añadir a los contactos de forma manual. El inconveniente es que, una vez instalada la aplicación, esta no es funcional hasta que se llegan a tener un buen puñado de contactos.

El inconveniente de estas formas es que tendríamos por un lado los contactos del móvil y por otro lado los de la aplicación. Aunque puede tener sus ventajas, su sencillez complica el proceso y no la hace atractiva. **Lo ideal es añadir un contacto al móvil y que, automáticamente, si ambos contactos se tienen en sus respectivas agendas, estos aparezcan ya disponibles en la aplicación. En otras palabras, la gestión de contactos se debe realizar de forma transparente.**

**Publish an event**

Los eventos, junto a los usuarios, son los dos tipos de datos esenciales sobre los que se construye *AreYouIn*. A la hora de publicar un evento es muy importante hacerse la pregunta ¿Quién puede verlo? y, por tanto, es muy importante tener siempre presente la visibilidad del mismo. Básicamente un evento puede ser público, lo puede ver cualquier usuario de la aplicación, o privado. La visibilidad de los eventos privados puede controlarse de modo que solo determinados usuarios, grupos, etc. puedan verlos. En otras palabras, a la hora de publicar un evento se debe poder establecer los destinarios del evento de modo que se pueda controlar la difusión del mismo. En cuanto a la difusión, para algunos usuarios puede ser interesante que los eventos que ellos crean puedan difundirse más allá de sus propios amigos, pero sin perder la naturaleza privada del evento. Por tanto, los destinatarios de los eventos podrán compartirlos si el creador del mismo así lo permite. Los saltos que puede dar un evento se controlan en la creación del mismo.

# Data types

**Event**

* Message (140 chars.)
* Author ID (5 bytes)
* Author Name // Required when showing events
* Creation date
* Event date
* Duration of the event
* Visibility or distribution list
  + Public
  + Private –persons, groups, friends, etc.–
* Sharing level
* Geolocalización

**User**

* User ID (5 bytes) *# UNIQUE*
* Name
* E-mail *# UNIQUE*
* Phone number *# UNIQUE*
* Age
* Friends
  + User ID
  + Name

# Operations

# Message Protocol

## Requisitos

* El protocolo debe ir versionado para poder hacer mejoras al mismo y permitir la convivencia de clientes que hablen diferentes versiones del protocolo en un momento dado.
* El protocolo será binario para reducir sobrecarga.
* Debe permitir “parsear” los mensajes de forma sencilla y rápida.
* Los mensajes se deben clasificar en Modificadores (cambian estado de objetos en el servidor), Notificaciones (informan del cambio de estado de un objeto), Peticiones (requieren una respuesta síncrona) y Respuestas (respuestas a peticiones).
* Todos los mensajes Modificadores deben tener una respuesta inmediata (síncrono) para reconocer la recepción del mensaje (OK) o error (ERROR). Posteriormente, en algún momento, se recibirá la Notificación correspondiente (asíncrono).
* El protocolo es síncrono, es decir, hasta que no he obtenido respuesta del mensaje modificador o petición, no puedo enviar otros mensajes al servidor. Sin embargo, las notificaciones se pueden recibir en cualquier momento.
* Los mensajes recibidos del servidor serán completos, no pudiendo intercalarse mensajes dentro de otros mensajes.
* Algunos mensajes pueden generar bastante resultados, por lo que será necesario establecer mecanismos para iterar sobre los mismos.

# Requests

**LIST AUTHORED EVENTS [cursor]**

Obtiene una lista de todos los eventos creados por el propio usuario y que están activos. Opcionalmente se puede utilizar un cursor para poder recuperar la siguiente parte de la lista en caso de que esta sea demasiado grande.

**LIST PRIVATE EVENTS [cursor]**

Los eventos privados son aquellos que se dirigen a un grupo de usuarios donde cada uno de los miembros está indicado de forma individual. En otras palabras, son los eventos que incluyen invitación explícita a usuarios concretos. En este caso, se listan los eventos privados activos a los que el usuario ha sido invitado, es decir, los eventos compartidos con el usuario y que le han sido notificados previamente. Opcionalmente se puede utilizar un cursor para poder recuperar la siguiente parte de la lista en caso de que esta sea demasiado grande.

**LIST PUBLIC EVENTS user\_coordinates range\_in\_meters [cursor]**

Lista eventos públicos activos dentro del radio *range\_in\_meters* con centro la posición del usuario*.* La posición del usuario se establece con USER POSITION de forma periódica. El parámetro *range\_in\_meters* puede ser como máximo 100 Km. Este mensaje sobrescribe en las preferencias del usuario el valor de la variable *range\_in\_meters*.

**HISTORY AUTHORED EVENTS [cursor]**

Obtiene una lista de todos los eventos creados por el propio usuario y que están expirados. Opcionalmente se puede utilizar un cursor para poder recuperar la siguiente parte de la lista en caso de que esta sea demasiado grande.

**HISTORY PRIVATE EVENTS [cursor]**

Los eventos privados expirados de los que el usuario recibió invitación pueden clasificarse en: “Me uno”, “No puedo”, “No voy” y sin ninguna confirmación. Aunque los eventos importantes para el usuario deben ser los de asistencia confirmada, conservamos también en el historial aquellos a los que confirmó no asistencia (“No puedo”). En otras palabras, nos quedamos con los eventos de interés para el usuario. Por tanto, con este mensaje se listan todos los eventos privados expirados de los que el usuario recibió invitación y confirmó asistencia o no asistencia (“No puedo”). Opcionalmente se puede utilizar un cursor para poder recuperar la siguiente parte de la lista en caso de que esta sea demasiado grande.

**HISTORY PUBLIC EVENTS [cursor]**

Lista los eventos públicos expirados en los que el usuario confirmó asistencia o no asistencia (“No puedo”). Opcionalmente se puede utilizar un cursor para poder recuperar la siguiente parte de la lista en caso de que esta sea demasiado grande.

**CREATE EVENT size\_message message start\_date end\_date size\_participants participants**

Crea un nuevo evento indicando un mensaje de invitación que contenga información descriptiva sobre el evento en cuestión. Este mensaje se mostrará al resto de usuarios y está limitado a 140 caracteres (como twitter). También debe incluirse la fecha inicial y final del evento. Además, el evento debe contener una lista de invitados para poder llevar a cabo la publicación. Al recibirse este mensaje en el servidor, se añadirá automáticamente a la lista de participantes el autor del evento con asistencia confirmada.

CREATE EVENT tiene las siguientes limitaciones. 1) Un usuario no puede tener más de 10 eventos activos de los que es autor. 2) start\_date no puede comenzar una semana después de la creación del evento, ni ser posterior a la fecha de creación (no es un calendario sino cosas que quieres hacer en el muy corto plazo), 3) end\_date no puede ser menor que start\_date y como máximo será 24 h posterior a start\_date. 4) Los eventos privados podrán tener como máximo 100 participantes (como whatsapp) y los públicos ¿100000?. 5) Los participantes deben existir y ser contactos del usuario. Si no se cumple alguna de estas condiciones o se produce un error la operación fallará y se devolverá un ERROR.

Posteriormente a la creación del evento, se recibirá un EVENT CREATED.

**CANCEL EVENT event\_id size\_reason reason**

Cancela un evento existente del cual el usuario es autor. Obligatoriamente debe indicarse un mensaje que explique por qué el evento se ha cancelado (limitado a 140 caracteres). Si la cancelación falla se envía un ERROR al autor. Si es exitosa, se envía un EVENT CANCELLED a todos los participantes del evento, incluido el autor.

Aunque la cancelación del evento es una decisión exclusiva del autor del evento y, por tanto, puede molestar a muchos participantes que hayan confirmado asistencia al mismo, será el propio autor el que decida la forma de proceder. Por ejemplo, el autor del evento deberá considerar si es oportuno cancelar un evento o simplemente marcar su no asistencia y, tal vez, delegar la gestión del evento a un participante del mismo.

**INVITE USERS event\_id size\_participants participants**

Invitar a más participantes a un evento existente. Está sujeto a las mismas restricciones que CREATE EVENT. Si alguno de los participantes no existe, o no es un contacto del usuario se recibirá un ERROR.

**CANCEL USER INVITATION event\_id size\_participants participants**

Retira la invitación a participantes del evento invitados previamente. Al cancelar la invitación el usuario se elimina de la lista de participantes del evento una vez se completa la operación. Igualmente, el evento se elimina de la lista de eventos del propio usuario. Esta operación solo se puede llevar a cabo mientras el usuario todavía no haya confirmado. La notificación a todos los usuarios, incluido el usuario afectado, se lleva a cabo mediante un mensaje INVITATION CANCELLED. Si ocurre un error el autor recibirá el mensaje ERROR.

**CONFIRM ATTENDANCE event\_id action\_code**

Confirmar 1) *Asistencia (Me uno)*, 2) *No asistencia (No puedo)* o 3) *No asistencia (No quiero ir)* a un evento existente. En el caso de un evento privado debe haberse recibido invitación previa o ser autor del evento. En caso de un evento público, en el momento de confirmar el usuario debe encontrarse en la región a la que pertenece el evento, ser autor, o haberle sido notificado previamente. El resto de cambios podrán realizarse aunque no se esté ya en la región. Además, en los eventos públicos no se puede indicar “No quiero ir”, dado que no tiene sentido (no hubo invitación y el NO solo interesa a usuarios amigos). Si no se lleva a cabo ninguna acción se considera que se ignora la invitación. Tras recibir el evento y confirmar, el servidor debe generar un INVITATION STATUS para notificar al resto de participantes la decisión del usuario. Al igual que cualquier participante, el autor del evento puede cambiar su estado de asistencia. Si ocurre un error se recibirá ERROR.

**MODIFY EVENT DATE event\_id start\_date end\_date**

**MODIFY EVENT MESSAGE event\_id size\_message message**

**MODIFY EVENT event\_id size\_message message start\_date end\_date**

Modificar un evento existente. La modificación de un evento no podrá realizarse cuando la fecha del comienzo del mismo se encuentre próxima. En concreto, la ventana de tiempo durante la cual se podrán realizar cambios se encuentra en el rango [*creation\_date*, *No\_changes\_point*). Donde *No\_changes\_point* es min(*start\_date* – 0.20\**size*, *start\_date* – 15 min) y *size* es *start\_date* – *creation\_date*. Es decir, la ventana depende del tiempo de espera transcurrido desde que se crea el evento hasta que se celebra. Como mínimo, 15 min antes del evento no podrá cambiarse. Como máximo, 24 h antes del evento no podrá cambiarse. Igualmente, a partir del punto *No\_changes\_point*, no se podrán cambiar fechas ni la descripción del evento (sí se aceptan cambios en la asistencia y nuevas invitaciones).

Cuando ningún participante ha confirmado todavía, el evento puede modificarse inmediatamente recibiéndose un EVENT MODIFIED \*. En caso contrario, se recibirá un NOTICE MODIFY EVENT \*, dando lugar al comienzo de la votación. Además, está sujeto a las mismas restricciones que CREATE EVENT. Si no se cumplen restricciones la operación fallará y se devolverá un mensaje ERROR.

**VOTE CHANGE event\_id change\_id YES/NO**

La modificación de un evento puede generar problemas cuando la gente ya ha confirmado porque podría cambiar el evento en sí mismo o la fecha. Cuando dos o más usuarios confirman (asistencia o no asistencia porque no se puede), es necesaria una votación para aceptar o no dicho cambio. En ese sentido, solo tienen derecho a votación aquellos invitados que han confirmado la asistencia o no asistencia al evento (porque no pueden), de modo que la decisión sea tomada solo por aquellos que tienen interés en dicho evento. Quedan fuera de la votación los que confirman no asistencia (porque no quieren) y los que han ignorado la invitación al evento. Si el voto se recibe con éxito se devuelve VOTE RECEIVED, mientras que si ocurre un error se devuelve ERROR.

La ventana de votación se abre con el mensaje MODIFY EVENT \*, y se cierra transcurrido el tiempo disponible para realizar la votación, una vez todos los invitados con derecho a votación hayan emitido su voto o alcanzado el punto *No\_changes\_point*. El tiempo disponible para la votación se encuentra en el rango [modify\_request, 1/3\*tiempo restante hasta *No\_changes\_point*). El tiempo máximo disponible para votar es de 48 h, el tiempo mínimo es de 5 min. Es decir, si 20 minutos antes del evento se sugiera una modificación, el tiempo disponible de votación será de 5 min porque los 6,6 min reales sobrepasan el *No\_changes\_point*.

Una vez finalizada la votación, se hará recuento ( Yes = Votos Yes / Derecho voto, No = Votos No / Derecho voto, Abstenidos = Sin respuesta / Derecho voto, Participación = Votos Yes & No / Derecho voto). Evaluando los votos emitidos uno por uno se estudiará la asistencia resultante, si esta es mayor que la actual, se acepta el cambio, entonces se emite CHANGE ACCEPTED, seguido de EVENT MODIFIED. En cambio, si es igual o inferior, se emite CHANGE DISCARDED.

**READ EVENT event\_id**

Lee completamente toda la información de un evento existente. En algún momento se obtiene un EVENT INFO como respuesta.

**USER POSITION global\_coordinates estimation\_err**

Establece la posición global del usuario. El cliente deberá enviar este mensaje cada 15 minutos. Si inicia la aplicación y han pasado más de 15 minutos desde la última vez, en ese caso también debe enviarse.

NOTA: Por ahora no es necesario.

**USER POSITION RANGE range\_in\_meters**

Establece el rango o tamaño de la región en que el usuario está interesado, considerando siempre como centro de la misma la posición del propio usuario. Este parámetro se guarda en el servidor para “saber” que eventos incluir en el mensaje NOTICE PUBLIC EVENTS. Igualmente, range\_in\_meters puede establecerse mediante el mensaje LIST PUBLIC EVENTS.

NOTA: Por ahora no es necesario.

**PING token current\_time**

Test para comprobar el estado del canal. El servidor debería contestar con un PONG incluyendo su current\_time. De este modo, será posible medir latencias y otras medidas de interés.

**USER AUTHENTICATION e-mail password device\_id**

**USER AUTHENTICATION token\_id device\_id**

Autentifica al usuario empleando su e-mail y password.

# Replies

**EVENT CREATED event\_id**

Notifica que el evento con *event\_id* ha sido creado satisfactoriamente.

**EVENT CANCELLED event\_id size\_reason reason**

Notifica que el evento con *event\_id* ha sido cancelado por el autor del evento con el mensaje *reason*.

**EVENT EXPIRED event\_id**

Notifique que el evento con *event\_id* ha expirado.

**EVENT MODIFIED event\_id start\_date end\_date**

**EVENT MODIFIED event\_id size\_message message**

**EVENT MODIFIED event\_id size\_message message start\_date end\_date**

Notifica de la modificación de los datos de un evento e incluye los cambios en la propia notificación. Esta notificación la reciben todos los invitados del evento, hayan confirmado o no.

**INVITATION RECEIVED event\_id, more info**

Notificación de nueva invitación al evento privado event\_id. De los eventos públicos no se reciben invitaciones por lo que deben recuperarse empleando LIST PUBLIC EVENTS.

**INVITATION CANCELLED event\_id**

La invitación al evento privado *event\_id* ha sido cancelada por el autor del evento.

**ATTENDANCE STATUS event\_id size\_participants (participants, status, reception\_code)**

Recibe el estado de la asistencia correspondiente con el evento *event\_id*, indicando para cada uno de los usuarios su estado de asistencia. Dependiendo de si el mensaje lo recibe el autor del evento u otros usuarios variarán los participantes incluidos. El autor recibirá el estado de todos y cada uno de los participantes invitados originalmente. Mientras que el resto de invitados recibirá únicamente los usuarios que hayan confirmado asistencia o no asistencia (porque no pueden pero tienen interés). Por otro lado, los usuarios que confirmen no asistencia (porque no quieren ir) solo se mostrarán a otros usuarios que sean ya amigos, no al resto de usuarios para así proteger la privacidad de estos usuarios.

*Size\_participants* debe ser, como máximo, el total de invitados a dicho evento. Pero habitualmente contendrá menos participantes. Status puede ser 0) Sin respuesta, 1) Asistencia, 2) No puede ir, 3) No quiere ir. *Reception\_code* puede ser: 0) Invitación Recibida en el servidor o 1) Invitación Entregada al usuario.

Este mensaje solo incluye el estado de participantes existentes que, en el momento de la invitación, son contacto del autor del evento. Además puede incluir otros participantes invitados por otros usuarios.

**EVENT CHANGE PROPOSED event\_id change\_id start\_date end\_date**

**EVENT CHANGE PROPOSED event\_id change\_id size\_message message**

**EVENT CHANGE PROPOSED event\_id change\_id size\_message message start\_date end\_date**

Notificación de propuesta para modificar un evento existente. Esta notificación solo la reciben los participantes que han confirmado (asistencia o no asistencia). De este modo, podrán votar si aceptan o no el cambio propuesto.

**VOTING STATUS event\_id change\_id status\_id**

**VOTING STATUS event\_id change\_id status\_id [start\_date end\_date elapsed\_time votes\_received votes\_total]**

Notifica el estado de la votación del cambio change\_id del evento event\_id. Este mensaje debe enviarse cada 30 segundos o 1 minuto a los participantes con derecho a voto. Status\_id es 1 si la votación está en curso o 0 si la votación ha finalizado.

**CHANGE ACCEPTED event\_id change\_id**

La votación sobre la modificación change\_id ha dado como resultado la aceptación del cambio. En la UI a los usuarios que votaron NO a dicho cambio se les debería sugerir que revisaran su asistencia. Se asume que los que votaron SÍ a dicho cambio es porque pueden asistir y, por tanto, no se les debe sugerir que revisen asistencia.

**CHANGE DISCARDED event\_id change\_id**

La votación sobre la modificación change\_id ha dado como resultado la no aceptación del cambio. En la UI a los usuarios que votaron SI a dicho cambio se les debería sugerir que revisaran su asistencia. Se asume que los que votaron NO a dicho cambio es porque pueden asistir y, por tanto, no se les debe sugerir que revisan asistencia.

**EVENT INFO event\_id author\_id creation\_date size\_message message start\_date end\_date size\_participants (participants, status)**

Información sobre un evento: autor, mensaje, fecha del evento, fecha de creación y lista de invitados con su estado. *Size\_participants* debe ser, como máximo, el total de participantes que recibieron invitación. Este mensaje se obtiene como respuesta a un READ EVENT. La lista de participantes junto a sus estados debe estar sujeta a las mismas restricciones que INVITATION STATUS que considera qué contiene la lista cuando recibe el mensaje el autor del evento u otros usuarios.

**EVENTS LIST size\_events (events\_id, more info) [cursor]**

Lista de eventos que se obtiene en respuesta a un mensaje anterior LIST \* o HISTORY \*. El mensaje contiene una lista de eventos donde, para cada evento, se incluye el event\_id e información relacionada (autor, fecha, geolocalización, número participantes, …). La lista de eventos está ordenada por fecha descendente, es decir, los elementos más antiguos están al final de la lista y los más nuevos al principio. Si la lista contiene más de 50 eventos, se devolverá un cursor para poder obtener partes restantes de la lista.

**ERROR status\_code**

Hay operaciones como las de INVITE USERS, CANCEL INVITATION y otras cuya respuesta puede no recibirse pasado cierto tiempo. Por tanto, si sucede un error con la propia recepción del mensaje hay que notificarlo cuanto antes. Con este mensaje, se indica si hubo un error en la recepción del mensaje. Si no hubo error, en algún momento debería recibirse información sobre el estado relacionado con el mensaje.

**OK**

El mensaje se recibió y procesó correctamente.

**PONG token current\_time**

Contestación a un mensaje PING recibido anteriormente.

**Relación de Peticiones y Respuestas**

|  |  |
| --- | --- |
| **Message Request** | **Message Reply** |
| LIST AUTHORED EVENTS | EVENTS LIST, ERROR |
| LIST PRIVATE EVENTS | EVENTS LIST, ERROR |
| LIST PUBLIC EVENTS | EVENTS LIST, ERROR |
| HISTORY AUTHORED EVENTS | EVENTS LIST, ERROR |
| HISTORY PRIVATE EVENTS | EVENTS LIST, ERROR |
| HISTORY PUBLIC EVENTS | EVENTS LIST, ERROR |
| CREATE EVENT | EVENT CREATED, INVITATION RECEIVED, OK, ERROR |
| CANCEL EVENT | EVENT CANCELLED, OK, ERROR |
| INVITE USERS | OK, ERROR, INVITATION RECEIVED |
| CANCEL USER INVITATION | INVITATION CANCELLED, OK, ERROR |
| CONFIRM EVENT | ATTENDANCE STATUS, OK, ERROR |
| MODIFY EVENT | EVENT CHANGE PROPOSED\*, EVENT MODIFIED, OK, ERROR |
| VOTE CHANGE | OK, ERROR |
| READ EVENT | EVENT INFO, ERROR |
| USER POSITION | ERROR |
| USER POSITION RANGE | OK, ERROR |
| PING | PONG |

NOTA: La recepción de las peticiones en el servidor da lugar a eventos que, en última instancia, provocan la creación de las respuestas. En otras palabras, las respuestas del cuadro anterior se generan por eventos disparados por el usuario. Por ejemplo, la petición CREATE EVENT, generará el evento “event created” provocando los mensajes EVENT CREATED, EVENT STATUS y EVENTS LIST. De hecho, todos los requests son asíncronos en el sentido de que el proceso que recibe la petición no se queda esperando hasta que se resuelve la misma.

En la siguiente tabla se muestran otros eventos que no dependen directamente del usuario, sino de otros procesos del sistema.

|  |  |
| --- | --- |
| **Evento en servidor** | **Message Reply** |
| Servidor entrega invitación al usuario | ATTENDANCE STATUS |
| Fecha actual > fecha final evento | EVENT EXPIRED |
| Votación en curso | VOTING STATUS |
| Votación finalizada | CHANGE ACCEPTED, CHANGE DISCARDED |
| El evento se modifica | EVENT MODIFIED |

¿Quién recibe las respuestas?

* El usuario que crea el mensaje.
* Usuarios a los que afecta la acción.
* Todos (autor evento + resto participantes).

\*) Usuarios con derecho a voto

|  |  |
| --- | --- |
| **Modificadores**  **00XXXXXX** | CREATE EVENT  CANCEL EVENT  INVITE USERS  CANCEL USER INVITATION  CONFIRM EVENT  MODIFY EVENT  VOTE CHANGE  USER POSITION  USER POSITION RANGE |
| **Notificaciones**  **01XXXXXX** | OK  ERROR  EVENT CREATED  EVENT CANCELLED  EVENT MODIFIED  EVENT EXPIRED  INVITATION RECEIVED  INVITATION CANCELLED  ATTENDANCE STATUS  EVENT CHANGE PROPOSED  VOTING STATUS  CHANGE ACCEPTED  CHANGE DISCARDED |
| **Peticiones**  **10XXXXXX** | LIST AUTHORED EVENTS  LIST PRIVATE EVENTS  LIST PUBLIC EVENTS  HISTORY AUTHORED EVENTS  HISTORY PRIVATE EVENTS  HISTORY PUBLIC EVENTS  READ EVENT  PING |
| **Respuestas**  **11XXXXXX** | EVENTS LIST  EVENT INFO  PONG |

**Observaciones**

* No existe mensaje para cancelar un cambio propuesto, es decir, cancelar la votación.
* No existe un mecanismo para delegar/ceder la gestión de un evento a otro participante.
* No existe mensaje para recuperar las votaciones en curso o las propuestas de cambios. Si hubiera votación en curso recibiría mensaje VOTING STATUS pero no sabría a que change\_id se refiere.
* Definir códigos de error para los diferentes ERROR.
* Es necesario en todos los casos en los que se emplean event\_id, change\_id, etc. controlar en el lado del servidor que solo se pueda acceder a aquellos objetos a los que se tiene permiso. Por ejemplo, en el caso de eventos, solo se tiene acceso a aquellos eventos que sean públicos o cuya invitación se haya recibido anteriormente.
* Controlar flooding de mensajes.
* Falta establecer comandos para el registro del usuario.
* Falta establecer comandos para que el usuario modifique su perfil.
* Falta establecer comandos para añadir amigos, crear grupos.

**Tipos de mensajes**

Relacionados con eventos

Relacionados con movilidad de los usuarios a través de las regiones

Relacionados con las personas

# User Interface

**Lista de participantes**

La duda es si incluir a los usuarios que confirman *No asistencia (porque no quieren)*. Por un lado da una imagen negativa al evento. Por otro lado, de algún modo incurro en un pequeño problema de privacidad porque al final estaría exponiendo toda mi lista de contactos (resuelto a nivel protocolo) y el usuario que confirma No asistencia está exponiéndose a los demás. Esto puede ser un motivo para que los usuarios decidan ignorar eventos cuando saben de entrada que no van a asistir. Mientras que incluyendo solo a los que participan o están interesados, minimizo esto en parte. El único motivo para querer incluirlo es para que aquellos usuarios cuya asistencia depende de la asistencia de otros puedan decidir más rápido. A nivel de protocolo ya se contempla esto, faltaría por ver si esta lista se refleja tal cual sin cambios en la UI o se adopta otro mecanismo para indicar qué amigos tuyos invitados al evento han decidido no asistir.

**Crear evento**

Si no se indica una fecha, se asume que el momento es AHORA y, por tanto se considera una ventana de 1h y media desde la hora de creación. Por defecto, la lista de participantes serán los contactos del usuario. Por tanto, es necesario crear un grupo inicial que contenga los contactos del usuario que tienen la aplicación.

# SMS Providers (Bulk SMS)

<https://www.clickatell.com/get-creative-with-sms/bulk-sms-2/>

<http://www.textanywhere.net/>

<http://www.bulksms.com/>

# Android Example for sending messages

<http://codetheory.in/android-sms/>

<http://codetheory.in/android-broadcast-receivers/>

<http://android-developers.blogspot.in/2013/10/getting-your-sms-apps-ready-for-kitkat.html>