Tema 7. Servicios sobre móviles

Ingeniería de Servicios

2023-2024

¿Qué son los servicios sobre móviles?

Son servicios cuyo cliente (y raras veces servidor) se ejecuta en un dipositivo móvil (teléfono, PDA, tablet)

Gracias a la convergencia de las redes móviles e Internet, estos servicios usan los protocolos TCP/UDP/IP y muchos de los ya antes vistos (HTTP, SOAP, REST, SMTP, IMAP, XMPP, RTP, etc)

Es decir, **son los mismos que ya conocemos**.

Entonces ¿por qué un tema de servicios móviles?

¿Qué tiene de particular un dispositivo móvil?

Estas son algunas características que lo hacen único:

- Ancho de banda y consumo de datos limitados.
- Memoria y espacio de almacenamiento limitados.
- Autonomía limitada (minimizar consumo de batería)
- Conexión a la red es intermitente (y de calidad variable)
- A menudo el dispositivo se desconecta y reconecta a diferentes redes
- A menudo la IP del dispositivo es privada (NAT)
- La interfaz de usuario es diferente a la de sistemas de escritorio (GUI, táctil, menos espacio)
- Disponibilidad de múltiples sensores (GPS, acelerómetros, cámara, micrófono)
- Disponibilidad de múltiples conexiones (WiFi, 3G, Bluetooth, infrarrojos)
- Si es un teléfono, la recepción de llamadas tiene la máxima prioridad

¿Qué hay que saber? (1)

- Conceptos básicos de Operativos Móviles
  - ✔ Veremos sólo una breve introducción
- Acceso a sensores del dispositivo
  - ✘ *No se aborda en esta asignatura*
- Programación de GUIs para dispositivos móviles
  - ✘ *No se aborda en esta asignatura*
- Eficiencia de las comunicaciones
  - ✔ ¿Cómo influyen las comunicaciones en el consumo de batería?
  - ✔ ¿Qué estrategias seguir para minimizar este consumo?

¿Qué más habría que saber?

- Conexión punto-a-punto en la red local
  - El problema de conocer las IPs de otros dispositivos locales
  - Descubrimiento de otros dispositivos y servicios
  - Protocolos relacionados: Zeroconf (mDNS, DNS-SD), UPnP (SSDP)
- Conectividad
  - El problema de estar detrás de un NAT
  - Protocolos para resolver el problema (STUN, TURN, ICE)
  - El problema de TCP en redes móviles (y estrategias para resolverlo)
  - El problema de la IP cambiante (Mobile-IP)

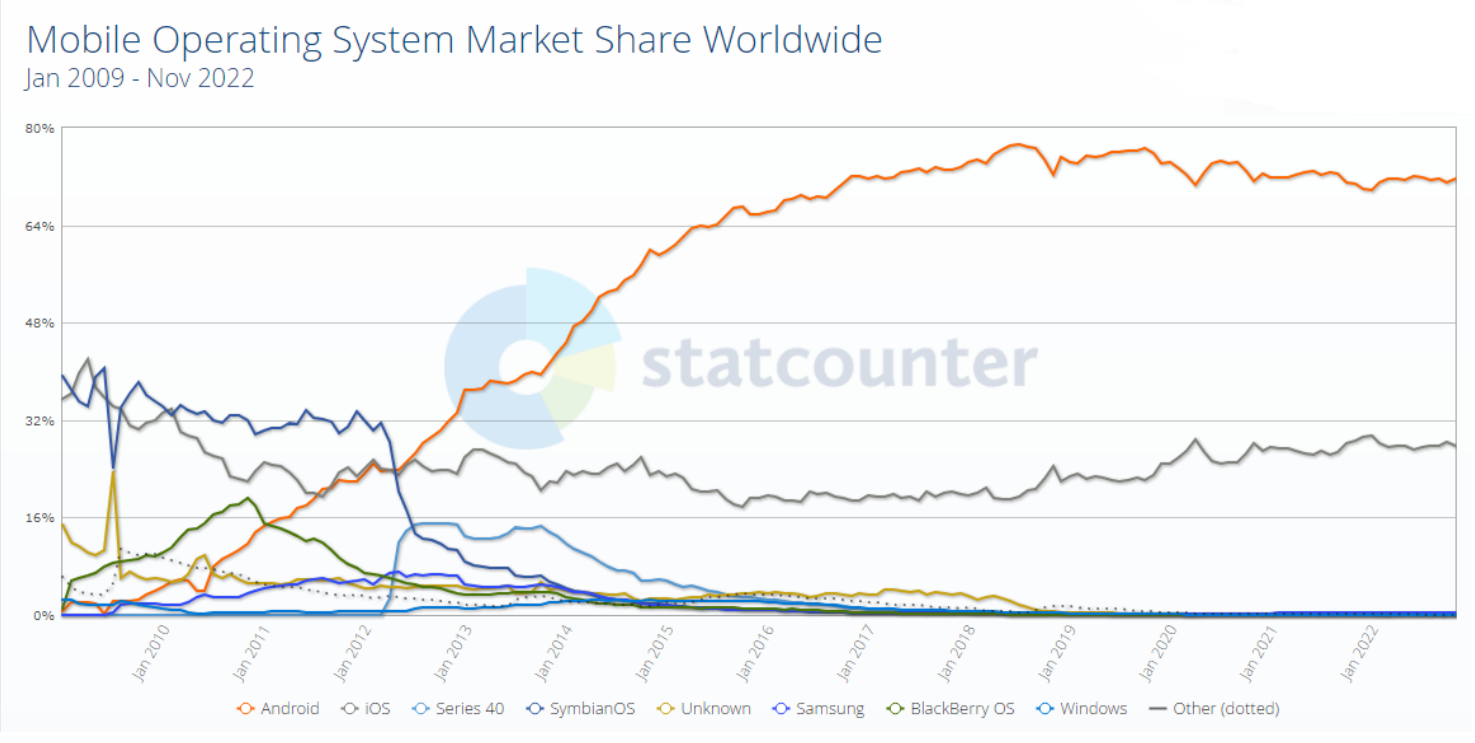
Los sistemas operativos para móviles

Hoy día los operativos más importantes para móviles son:

- **Android**. Basado en el *kernel* de Linux. Propiedad de Google. *Open Source* (aunque no las apps de Google, incluyendo partes de sus bibliotecas, ni ciertos *drivers* necesarios). Varios fabricantes usan Android en sus teléfonos (Samsung lidera el mercado). Programable en Java y Kotlin.
- **iOS**. Basado en OSX. Propiedad de Apple. Disponible sólo para dispositivos Apple (iPhone, iPad). Programable en Objective C y Swift.
- **iPadOS**. Basado en OSX. Propiedad de Apple. Disponible solo para dispositivos Apple iPad Pro. Programable en Objective C y Swift. Es muy similar a iOS pero con mayores capacidades para la multitarea.

Cuota de mercado

S.O.



¿En qué se diferencia un Operativo Móvil de uno convencional?

Hay más restricciones:

- *Sandboxing*: Cada aplicación se ejecuta en un "entorno aislado" (*sandbox*) de las demás aplicaciones
  - En Android cada aplicación es un "usuario" y tiene su propia VM java
  - En iOS el acceso al sistema de ficheros está muy restringido.
  - Compartir datos entre aplicaciones es más complejo que en un SO de escritorio

¿En qué se diferencia un Operativo Móvil de uno convencional?

- Acceso al hardware: Cada aplicación "declara" de antemano qué partes del *hardware* necesita. El usuario debe autorizarlo. Un intento de acceder a otro hardware es impedido por el SO.
- Planificación de tareas:
  - La multitarea está muy limitada
  - El SO puede eliminar tareas de la memoria si no están en primer plano
  - La ejecución de tareas en *background* es compleja y restrictiva

Ciclo de vida de una aplicación

Aunque los detalles varían en cada operativo, todos tienen en común lo siguiente:

- Una aplicación "en primer plano" es la que interactúa con el usuario, **solo hay una en cada momento**
- Cuando una aplicación pasa a segundo plano, se **detiene su ejecución**
  - El usuario no suele notarlo porque guardan/restauran su estado
- Una aplicación en segundo plano puede ser **finalizada** por el operativo si necesita memoria
- Si se necesita ejecutar código en segundo plano, debe programarse de forma específica para ello

Esto es muy importante en relación a los servicios de red, ya que la aplicación puede ser interrumpida e incluso eliminada mientras una transacción de red estaba "a medias"

Ejemplo: iOS

En iOS tenemos los siguientes estados para las aplicaciones:

- **Fuera de ejecución** Aún no ha sido cargada, o ha sido eliminada
- ***Inactive*** Está ejecutándose en primer plano, pero no interactúa con el usuario. Es un estado transitorio.
- ***Active*** En primer plano, visible, interactuando con el usuario
- ***Background*** No es visible ni interactúa con el usuario, pero aún ejecuta código. Es un breve estado intermedio hacia *suspendida*, a menos que solicite estar más tiempo en este estado.
- ***Suspended*** Está en *background* pero no ejecuta código.

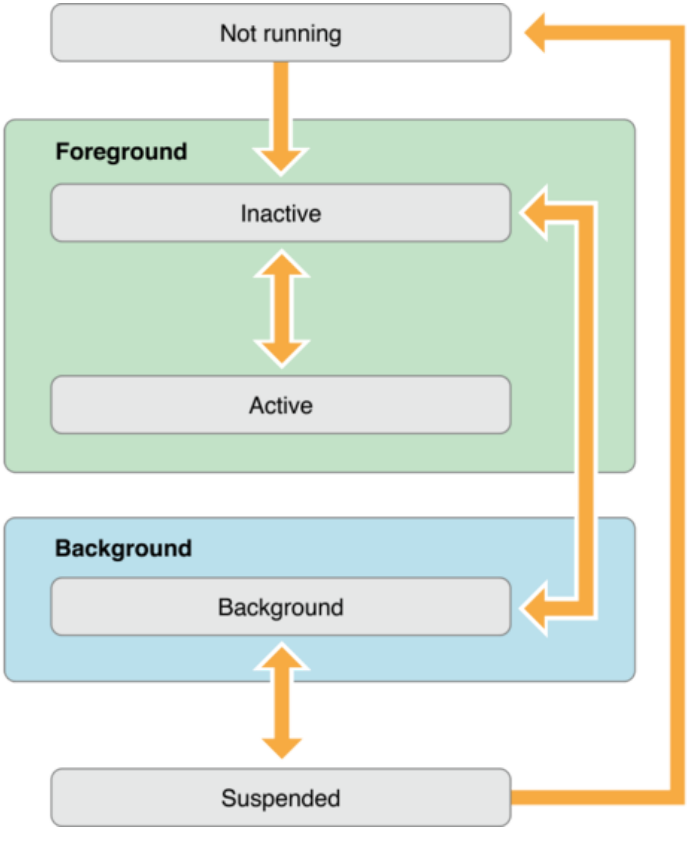
iOS puede cambiar una aplicación de *Background* a *Suspended*, o de *Suspended* a eliminada, sin notificación.

La ejecución prolongada en *Background* sólo se permite bajo ciertas condiciones.

iOS: Estados y transiciones

S.O.

-Ciclo de vida



Android: Componentes

En Android una aplicación consta de diferentes **componentes**, y cada componente tiene su ciclo de vida.

Los componentes son:

- **Actividades** Son las "pantallas" que ve el usuario. Sólo una actividad puede estar en ejecución en cada momento. Tienen un ciclo de vida complejo (ver después)
- **Servicios** Código al que se permite continuar su ejecución en segundo plano. No tiene interfaz de usuario. Puede haber varios en ejecución a la vez, y en paralelo con actividades
- **Content providers** Son una "interfaz" a los datos de la aplicación, que permite compartir estos datos con otras
- **Broadcast receivers** Son una especie de "manejadores de señales", que reciben notificaciones de sistema como "batería baja", etc.

Ciclo de vida de una *actividad*

La actividad pasa por varios estados. El sistema usa *callbacks* para notificar a la actividad cuándo entra en un nuevo estado.

- ***Created***: Acaba de ser creada. Estado transitorio
- ***Started***: Acaba de comenzar su ejecución. Estado transitorio
- ***Resumed***: Está en primer plano, en ejecución. Es el modo "normal" mientras el usuario la utiliza
- ***Paused***: Aún visible, pero no ejecuta código. Otra aplicación (no en pantalla completa) la tapa. Si vuelve a ejecución pasará a *Resumed*
- ***Stopped***: No es visible porque otra aplicación pasó a primer plano. No ejecuta código.
- ***Destroyed*** El sistema necesita memoria y ha eliminado la aplicación, o el usuario la ha terminado.

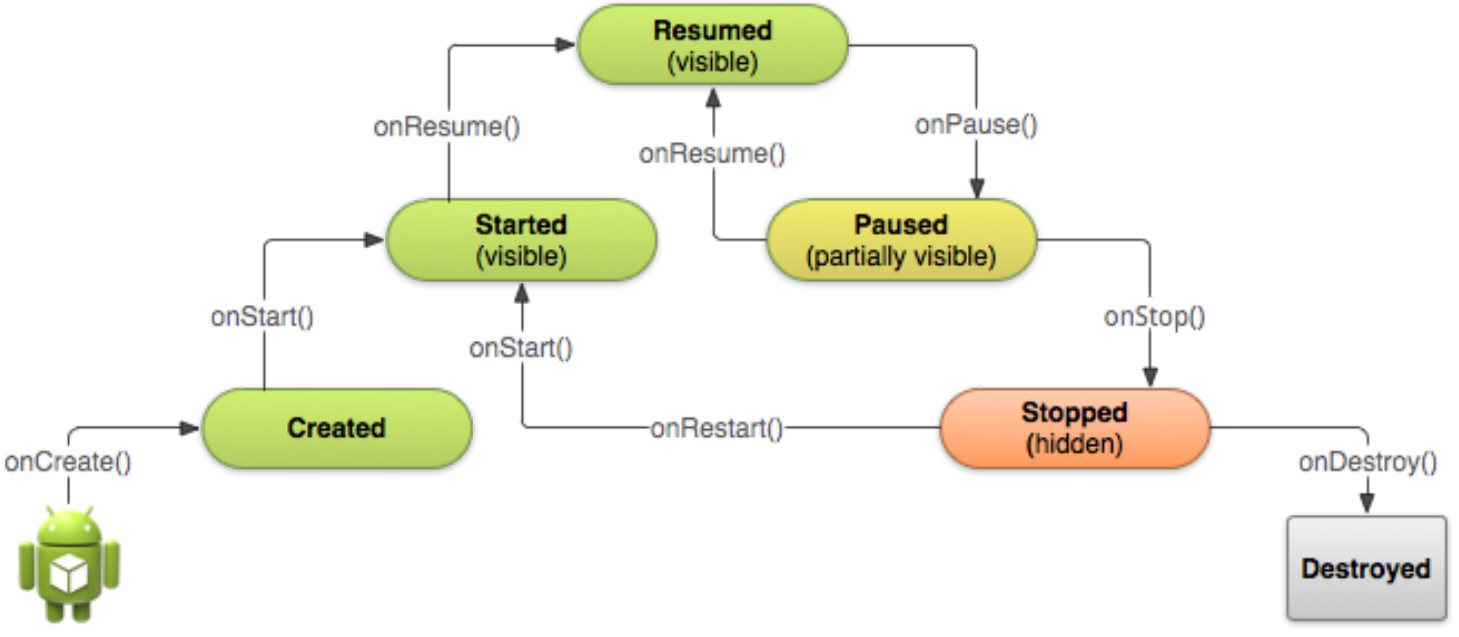
Cuando está ***Stopped***, el operativo puede *matarla* si está escaso de recursos, sin notificación vía *callback*.

S.O.

-Ciclo de vida

-Android

Android: Estados y transciones de una *actividad*



Android: Procesos

- El concepto de proceso es el mismo que en Unix.
- En el dispositivo se están ejecutando siempre varios procesos.
- Una aplicación se ejecuta dentro de un proceso (raramente puede usar varios)
  - El proceso ejecuta una máquina virtual java (bajo diferente UID, de la app)
  - Todos los componentes de la aplicación se ejecutan en el mismo proceso
  - Pero puede tener varios hilos
- Si el OS detiene y elimina un proceso, todos los componentes de la correspondiente aplicación mueren.

Cada proceso puede contener varios hilos para ejecutar simultáneamente diferentes funciones de un mismo o diferentes componentes.

Android: Procesos y su eliminación

Android tiene un algoritmo sofisticado para ordenar los procesos por *importancia*, teniendo en cuenta qué componente de la aplicación está procesando y su relación con otras aplicaciones.

- Procesos de *foreground* (actividades visibles, servicios enlazados a ellas y receptores *broadcast* recibiendo)
- Procesos *visibles* (actividades en estado *paused* y servicios enlazados a ellas)
- Procesos de *servicio* (servicios no enlazados a actividades)
- Procesos de *background* (actividades en estado *stopped*)
- Procesos *vacíos* (no albergan componentes, existen para agilizar el arranque de procesos nuevos)

El OS comenzará eliminando procesos por abajo en esta lista.

Un proceso *foreground* no será eliminado, salvo por emergencias (llamada entrante, memoria llena, ...)

Android: Hilos y actividades

La actividad se ejecuta

- Un hilo atiende eventos de la **interfaz de usuario**
- Si son necesarias **comunicaciones de red** deben lanzarse en otro hilo (el hilo del GUI no debe bloquearse)
- Android proporciona la clase `AsyncTask` para implementar tareas que deban ser ejecutadas en otro hilo.
- El hilo del GUI **se detiene** si la actividad es *pausada* o *detenida*
- El hilo del GUI es *destruido* y *creado* de nuevo ante un cambio de configuración (ej: orientación del dispositivo)

Android: Hilos y actividades

Observa que la aplicación puede tener, además de actividades, componentes de tipo servicio, que no son detenidos cuando la actividad se pausa o detiene.

Cuestión En una aplicación en la que el usuario pulsa un botón para subir una foto, ¿Cómo implementarías la función que sube la foto?

Problemas del consumo de servicios desde una app móvil

Puesto que la aplicación puede ser terminada sin previo aviso, el uso de servicios web es conflictivo:

Escenario 1

- Una app usa REST para solicitar un GET (en un hilo de una actividad).
- El usuario abre otra app → La actividad pasa a segundo plano (el proceso pasa a *background*)
- El operativo está escaso de recursos → elimina el proceso en *background*
- *La respuesta al GET nunca es procesada*

> El problema no es muy grave. Cuando la aplicación vuelva a cargarse, hará de nuevo el `GET`

Escenario 2

- Una app usa REST para solicitar un GET (en un hilo de una actividad).
- La respuesta llega, es procesada, el resultado se guarda en memoria (y se usa para actualizar la interfaz)
- El usuario abre otra app → La actividad pasa a segundo plano (el proceso pasa a *background*)
- El operativo está escaso de recursos → elimina el proceso en *background*
- *La información recibida y procesada se ha perdido*

> El problema aquí es el desperdicio de CPU y de ancho de banda

La solución es almacenar el resultado de forma persistente (fichero en tarjeta SD, base de datos, o mejor aún, un `ContentProvider`). Cuando la app se cargue de nuevo puede usar los datos almacenados en lugar de hacer un nuevo `GET`

Escenario 3

- Una app usa REST para solicitar un **POST** (en un hilo de una actividad).
- El servidor procesa, quizás, la petición y envía una respuesta
- Llega una llamada telefónica → La actividad pasa a segundo plano (el proceso pasa a *background*)
- El operativo está escaso de recursos → elimina la actividad que estaba en segundo plano
- *La respuesta se ha perdido*

> Cuando la aplicación se cargue de nuevo ¿debe reintentar el `POST`?

Estrategias paraeste problema

Es un problema complejo. La solución depende de la plataforma.

Por ejemplo en Android se pueden usar:

- *Services* para implementar las comunicaciones y procesamiento en *background*
- *Content providers* para almacenar las respuestas ya procesadas de forma persistente, pero a la vez mantener una *cache* en memoria para acceso rápido y no penalizar la velocidad
- Mantener en el *Content provider* el estado de la transacción (mediante constantes que especifiquen qué tipo de operación se ha iniciado o completado)
- La actividad lanza el servicio. Si la actividad es eliminada, cuando vuelva puede consultar el estado de la transacción con el *Content provider*
- `SyncAdapter` es un componente de Android que facilita la sincronización de recursos locales y remotos y optimiza el uso de la red (más después).
- Soluciones "precocinadas" como Volley

Eficiencia de las comunicaciones

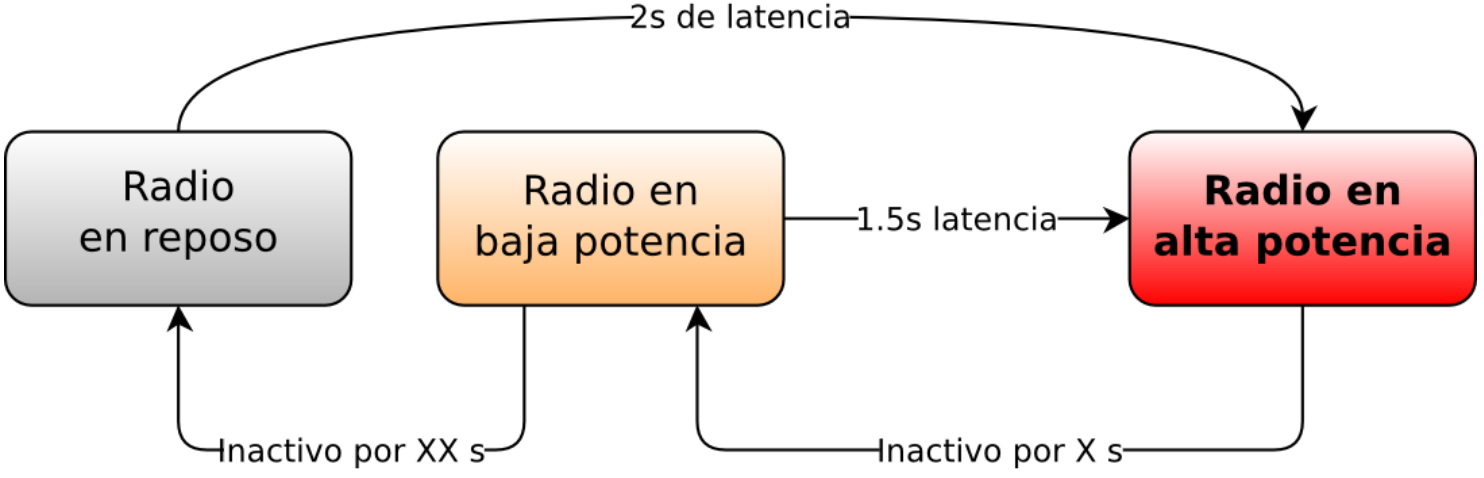
La antena (3G o WiFi) es una fuente de consumo muy importante. Debe reducirse su uso.

La antena no está permanentemente consumiendo energía. Pasa por tres estados:

- ***Standby*** No transmite ni recibe. Consumo mínimo. Pasar al modo *full* requiere tiempo y energía.
- ***Full power*** Está transmitiendo o recibiendo. Máximo consumo.
- ***Low power*** No transmite ni recibe. Consumo medio. Pasar a *full power* requiere menos tiempo y energía.

Para optimizar el consumo, tras una comunicación el *hardware* mantiene la antena en *full* un tiempo antes de pasar a *low* y finalmente a *standby*.

Diagrama de estados de la antena



Los valores de *X* y *XX* dependen de la operadora, localización, etc. Suelen ser *X*=5s, *XX*=12s

Esto implica que, tanto si transmitamos un solo byte como 1Mb, la antena está consumiendo energía por casi 20s

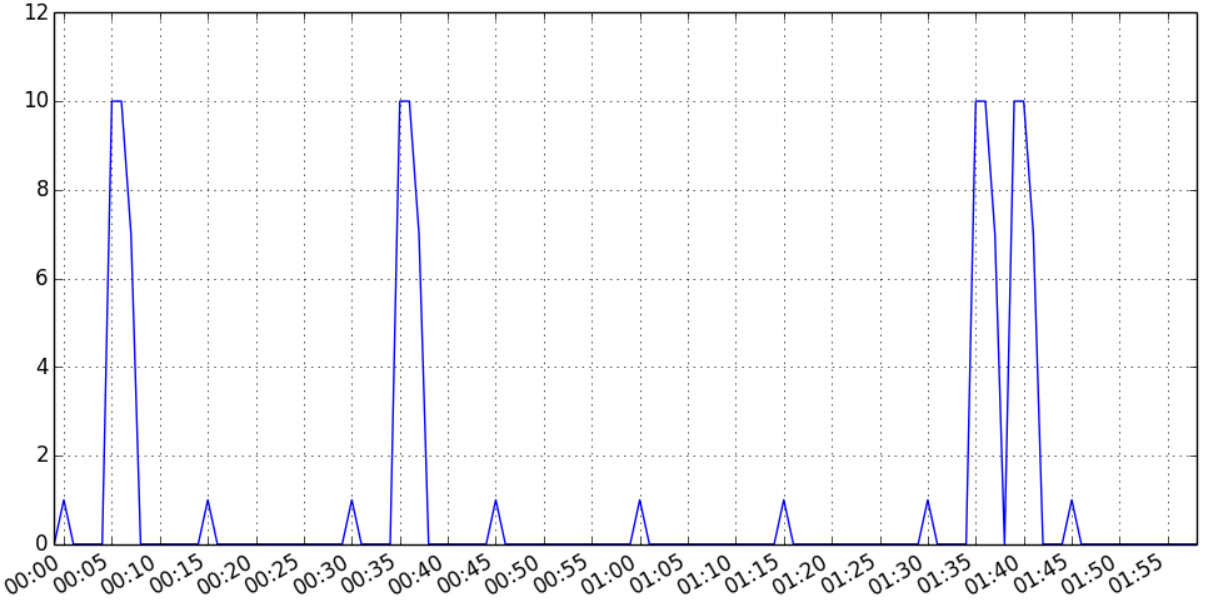
Dilema ¿agrupar y planificar acceso a la red?

¿Qué es preferible?

1. Transmitir datos sólo cuando se necesiten. Paquetes pequeños, frecuentes. Baja latencia para el usuario.
2. Agrupar transferencias y planificar cuándo hacerlas. Paquetes grandes, poco frecuentes. Uso de *prefetching* (descarga anticipada) para disiminuir la latencia para el usuario.

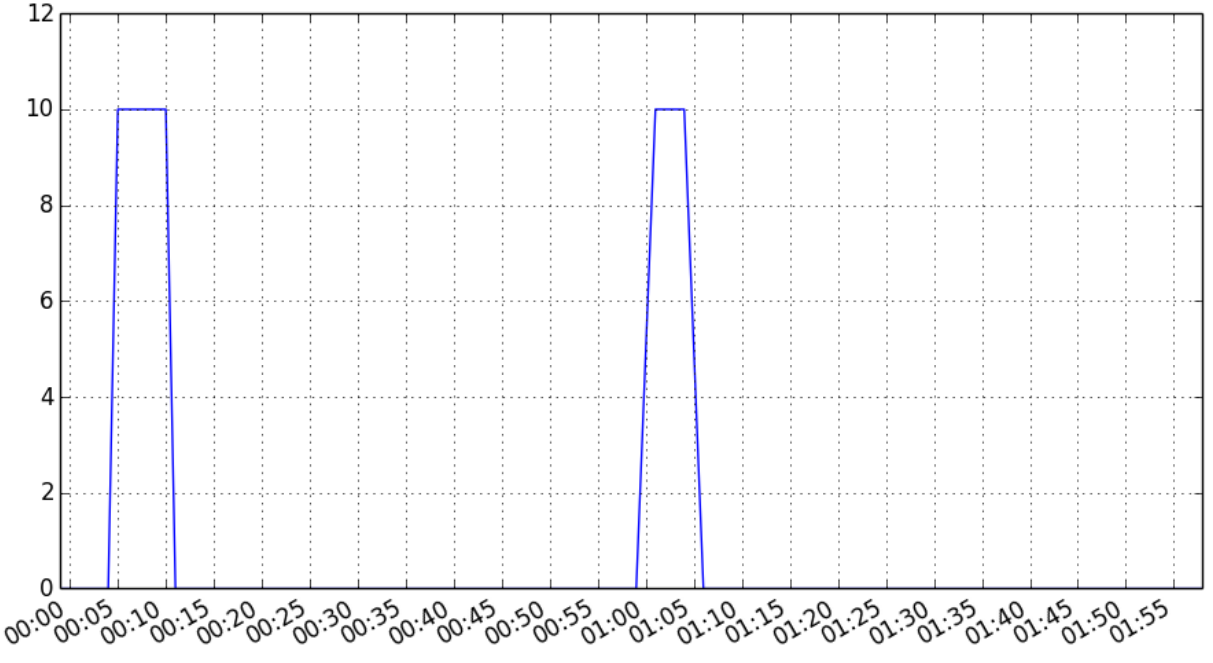
¿Qué opinas?

Uso de la antena con la primera estrategia



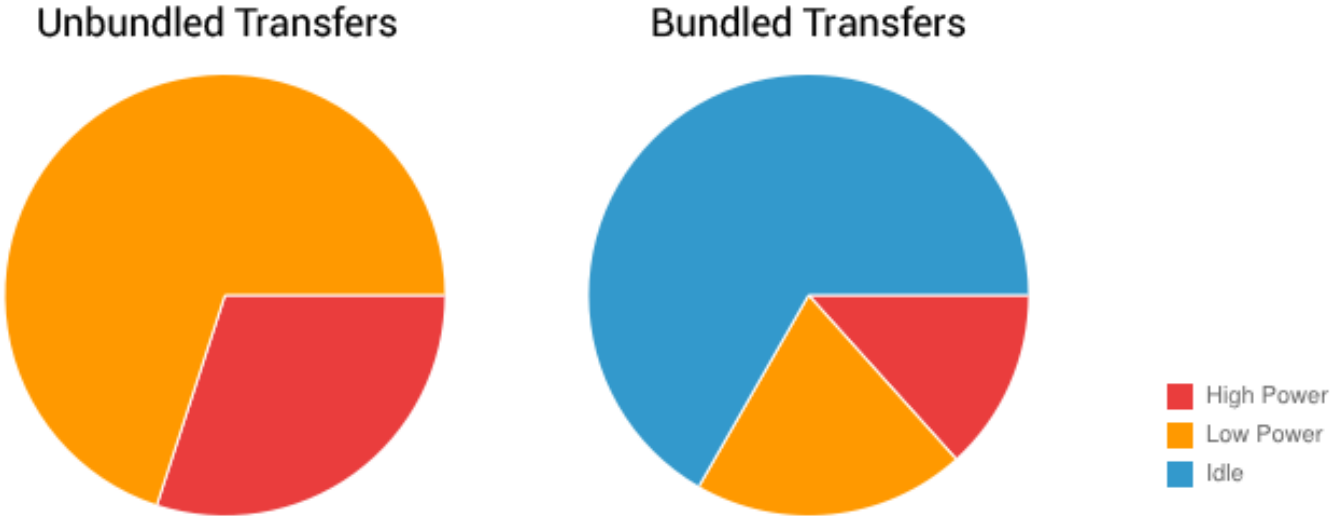
Hay una pequeña descarga periódica (un *polling* quizás) cada 15s, y otras mayores en momentos aleatorios ¡La antena nunca pasa a *standby*!

Uso de la antena con la segunda estrategia



Se ha eliminado el *polling*. Las transferencias se han agrupado y planificado para hacerse momentos separados.

Estados por los que pasa la antena en cada estrategia



Claramente la segunda estrategia (*bundled*) es mejor en términos de consumo de la batería.

Estrategias a seguir

- Usar *prefetching* (difícil equilibrio, pues también malgasta espacio y ancho de banda)
- Eliminar el *polling* (pasar a protocolos *push*, tipo XMPP)
- Agrupar transferencias que no sean críticas (*bundle*)
- Usar los momentos en que se deben hacer transferencias críticas para hacer también las acumuladas

-Prefetching

Se trata de aprovechar cuando la antena está activa, para precargar datos que previsiblemente el usuario solicitará después.

¿Cuántos datos precargar? Depende de la tecnología inalámbrica del dispositivo:

- En 3G un valor por defecto que depende de la aplicación (no es lo mismo descargar mensajes, que noticias, que música)
- En GPRS usaríamos la mitad de ese valor
- En 4G podemos cuadruplicar el valor por defecto
- En WiFi podemos ir al máximo posible que permita la memoria y espacio en dispositivo (ej: precargar todos los mensajes nuevos, o todas las noticias)

Qué precargar

En general, se recomienda los que el usuario vaya a requerir en los próximos 2 a 5 minutos.

Es impredecible, pero hay ciertos heurísticos

- Música: la canción completa, y la siguiente
- Noticias: todos los titulares, y los cuerpos de las primeras
- Analizar hábitos del usuario
- ¡Analizar incluso su movimiento! (caminando se lee menos que quieto)

Agrupando transferencias (*bundling*)

Típicamente hay dos tipos de transferencias de datos:

- **Dependientes del tiempo**: han sido iniciadas por el usuario tocando la interfaz. Espera respuesta inmediata. Si los datos no están disponibles (por *prefetching*) hay que descargarlos en ese momento.
- **Independientes del tiempo**: típicamente iniciados por la aplicación para consultar actualizaciones, mostrar publicidad, enviar estadísticas y analíticas de uso de la aplicación, etc.

La idea es retrasar y acumular las independientes, y realizarlas cuando se inicie una dependiente del tiempo.

Implementación del *bundling* (1)

Idea simplista:

- Los métodos que implementen las transferencias **independientes** del tiempo, en lugar de hacer la transferencia, guardarán en una **cola** los datos relativos a la misma.
- Los métodos que implementen las transferencias **dependientes** del tiempo, incorporarán llamadas a métodos que **transmiten** también la cola de pendientes.

Problema: Si la aplicación es eliminada por falta de memoria, se perderán los datos no transmitidos de la cola.

Implementación del *bundling* (2)

Soluciones (Android):

- Usar un *Content provider* para guardar la cola (y éste usará almacenamiento persistente para los datos). Cuando hayan sido enviados, se usa el *Content provider* para eliminarlos del almacén.
- Usar un `SyncAdapter` como solución "precocinada"
  - Es una tecnología de Google, inicialmente diseñada para Gmail y otros servicios
  - Puede ser usado por varias aplicaciones que necesiten sincronización periódica
  - Es un servicio funcionando en el dispositivo que se ocupa de guardar la cola de forma persistente, y enviarla de forma eficiente y amigable con el uso de antena.
  - Si no hay red disponible, se ocupa de reintentar cuando la haya

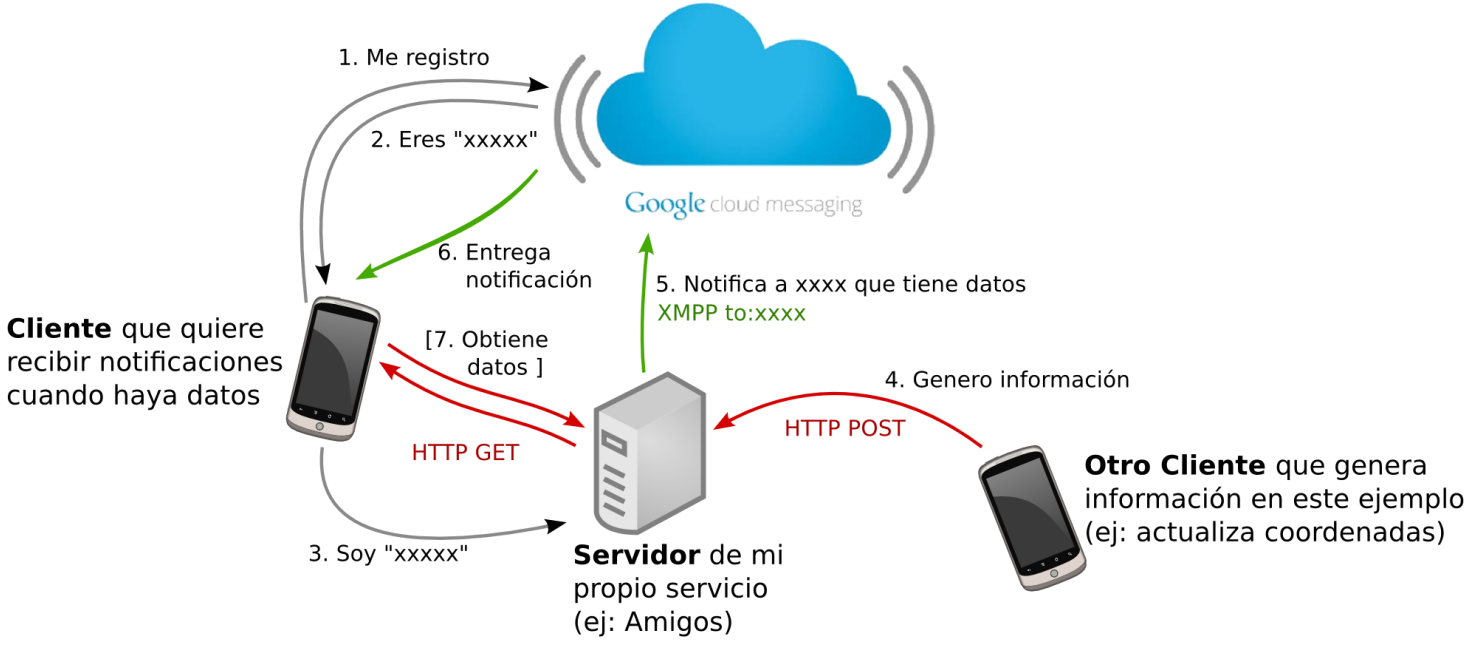
Eliminación del *polling*

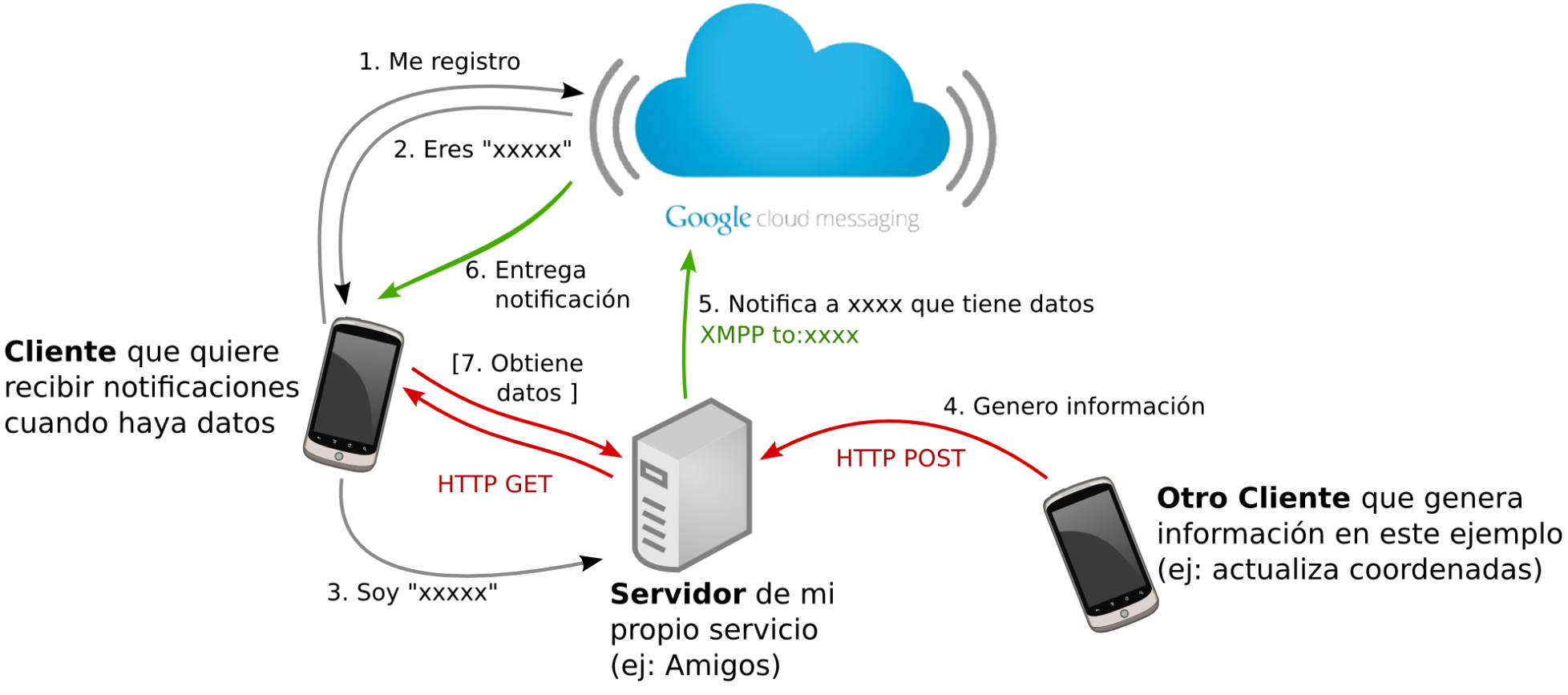
Para eliminar el *polling*, es necesario un protocolo que permita una conexión TCP permanente entre el dispositivo y el servidor, y por la cual el servidor pueda enviar datos nuevos cuando estos estén presentes.

Alternativas:

- Implementar protocolo propio
- Usar XMPP
- Usar tecnologías propietarias:
  - Google: ~~*Google Cloud Messaging* (GCM)~~ (*deprecated*)
  - Google: Firebase Cloud Messaging.
  - Apple: *Apple Push Notification Service* (APNs). Parte de iOS y OSX.

Ejemplo con Google Cloud Messaging





(Imagen más grande)