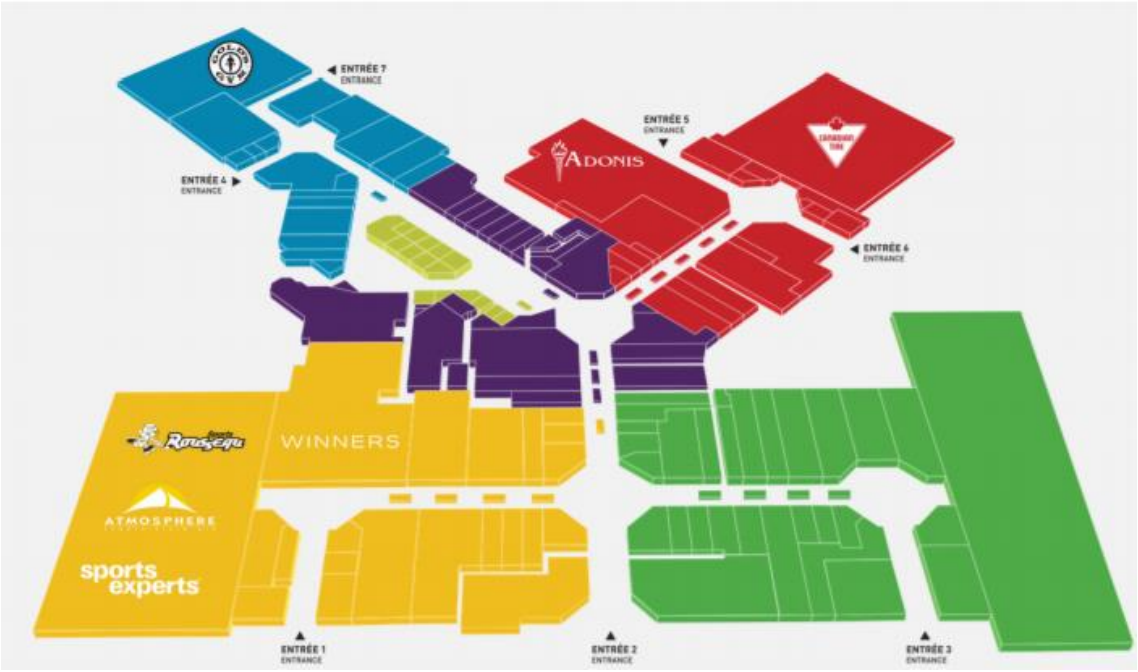
TECNOLOGÍAS DE DESARROLLO DE SISTEMAS UBICUOS

RABBITMQ & ANDROID APP & JavaFX APP



CONVOCATORIA: JUNIO

ALUMNO: JOSÉ MANUEL BERNABÉ MURCIA
CORREO: JOSEMANUEL.BERNABE@UM.ES

Índice

1 INTRODUCCIÓN

Hoy en día estamos rodeados de dispositivos: tablets, portátiles, teléfonos, proyectores, microcontroladores, con capacidad de computo notable y conectividad total e incluyendo gran cantidad de sensores en algunos casos. Con estos dispositivos podemos programar un comportamiento inteligente, si somos capaces de construir aplicaciones útiles y que entiendan un lenguaje en común. Es decir, volver inteligentes los espacios, con una mínima atención del usuario en un mundo de escasos recursos cognitivos, a lo que se refiere a la integración de la informática en el entorno de la persona, de forma que los computadores no se perciban como objetos diferenciados. A esto lo denominamos computación ubicua, desde hace unos años también se denomina inteligencia ambiental.

La computación ubicua o AmI (Inteligencia Ambiental) es un desarrollo tecnológico que actualmente está en proceso de mejoras para que los computadores no se perciban en un entorno. Con esto se busca que el usuario tenga la mayor comodidad y haga el menor esfuerzo posible al usar esta tecnología donde no se pretende tener alguna posición exacta para usarla, ya sea sentado, de pie o acostado, teniendo en cuenta que la interacción del usuario con la tecnología debe darse a través de eventos biológicos como la voz o los gestos, donde su tendencia es proyectar el software de una forma diferente en donde la pantalla pueda ser trasparente y los objetos conocidos a diario se convertirán en objetos de trabajo dirigidos a base del software.

Para la realización de esta práctica se simulará el movimiento de usuarios por un entorno, y dependiendo de la zona que nos encontremos recibiremos un mensaje u otro. El envío y recepción de mensajes se hará bajo el protocolo AMQP.

El protocolo AMQP es un protocolo de estándar abierto en la capa de aplicaciones de un sistema de comunicación. Las características que definen al protocolo AMQP son la orientación a mensajes, encolamiento ("queuing"), enrutamiento (tanto punto-a-punto como publicación-subscripción), exactitud y seguridad.

AMQP define una serie de entidades, las más relevantes son:

- Broker
- Usuario
- Conexión
- Canal

2 OBJETIVO

El objetivo principal de esta práctica es simular un entorno "inteligente". En este caso un centro comercial oferta productos a usuarios que estén interesados en dichas secciones, así como mostrar información sobre nuevos productos, dependiendo de su localización.

Como objetivo pasivo para el vendedor y el centro obtener a dichos usuarios el máximo tiempo en un área determinada, para obtener mayores ganancias y cubrir áreas que estén menos pobladas de usuario.

El "usuario comprador" recibirá las ofertas de las que está interesado a través de una APP para Android y con la que simulará su entrada y movimiento en el centro comercial. Por otro lado, el "usuario vendedor" o "usuario centro comercial" tendrá disponible otra aplicación de escritorio realizada en Java con la que podrá visualizar a todos los "usuarios compradores", y obtener información y estadísticas sobre ellos y globales sobre el centro.

Por tanto, tenemos un entorno con un sistema híbrido entre clásico e inteligente que sería el que utiliza el "usuario centro comercial", esto es debido a que el "usuario centro comercial" puede interactuar con la aplicación y mandar mensajes privados o globales a los usuarios, además en segundo plano la aplicación estará ejecutándose de forma autónoma para conseguir mayores ventas. Por otro lado, tenemos un entorno inteligente que es para el "usuario comprador", ya que él no tendría que interactuar con la aplicación apenas, excepto para ver los mensajes y suscribirse a los productos que le interesen.

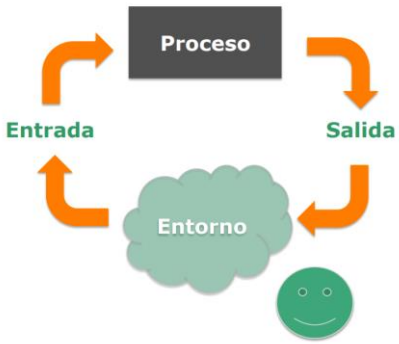


Figure 1 Entorno inteligente

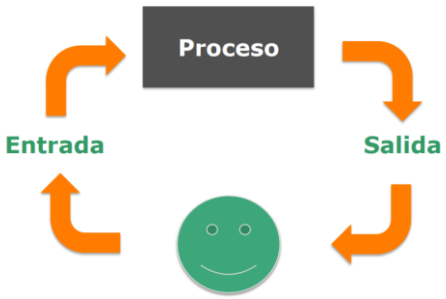


Figure 2 Entorno clásico

3 DISEÑO Y RESOLUCIÓN

Como hemos podido leer anteriormente, tenemos dos aplicaciones: una en Java y otra en Android. La cual las dividiremos en aplicación del usuario comprador y aplicación del usuario centro comercial.

Además, como el servidor utilizado como bróker en el que se realizaba esta práctica proporcionado por la Universidad ha estado caído, he tenido que montar mi propio bróker.

Antes de comenzar detallando las distintas aplicaciones y el bróker, vamos a detallar mi visión del escenario.

3.1 ESCENARIO

El escenario se trata de un centro comercial. El centro comercial tiene un supermercado con diferentes secciones como son:

1. Charcutería
2. Repostería
3. Frutería
4. Pescadería
5. Productos del hogar

El centro comercial también dispone de un cine:

1. Cartelera
2. Tienda

El usuario debe ser capaz de suscribirse a las distintas secciones propuestas, y recibir información de los distintos productos de dichas secciones.

3.2 BRÓKER

3.2.1 Instalación

El bróker elegido ha sido RabbitMQ tal y como especifica el guion de prácticas. RabbitMQ es capaz de interactuar con diferentes protocolos de mensajería como son MQTT, AMQP o STOMP.

La instalación ha sido bajo un entorno Debian en una Raspberry PI Model A+.

Acceso al bróker:

- **IP Broker:** 155.54.204.46
- **Puerto:** 5672
- **Username:** master
- **Password:** master

El acceso a RabbitMQ por el puerto 15672 no esta disponible fuera de mi red local.

La instalación ha sido muy simple, ejecutando **“apt-get install rabbitmq-server”** e inicializando el servicio **“service rabbitmq-server start”**.

No se garantiza el funcionamiento del bróker debido a que el servicio contratado en mi casa no es un servicio con una IP fija, por tanto, al ser dinámica la IP en cualquier momento la aplicación no podrá conectarse con el bróker a no ser que se cambie en el código la configuración.

3.2.2 Exchange, Canales y Routing Keys

Nuestro exchange recibirá el nombre de "Exchange_SuperMercado" y lo tendremos configurado como ***TOPIC.*** Este tipo de exchanges enrutan mensajes a colas basadas en coincidencias de comodines entre la routing key y el routing pattern especificado por el binding de la cola. Los mensajes se enrutan a una o varias colas en función de si hay coincidencia entre una routing key de mensaje y este patrón.

A la hora de diseñar las distintas colas tenemos que tener en cuenta lo que se quiere conseguir:

1. Un canal donde los usuarios compradores mande información que llegue a la aplicación del usuario centro comercial, dicha información será el lugar donde se encuentran y las suscripciones que le interesan.
2. Un canal donde recibamos información global sobre el centro comercial, el usuario comprador estará suscrito por defecto a este canal.
3. Un canal privado para cada usuario por cada sección a la se hayan suscrito, es decir, que puedan llegar ofertas personalizadas a un usuario determinado de un producto determinado.

Vamos a tener los siguientes routing keys:

- **Usuario comprador**
  - **Parte pública (Todos los usuarios recibirán estos mensajes)**
    - ofertas.supermarket.reposteria
    - ofertas.supermarket.pescaderia
    - ofertas.supermarket.hogar
    - ofertas.supermarket.charcuteria
    - ofertas.supermarket.fruteria
    - ofertas.cine.* (incluyendo la cartelera y la tienda)
    - informacion.centro
  - **Parte privada (Solo un usuario en específico recibirá el mensaje)**
    - cliente.idCliente.ofertas.respoteria/pescaderia/centro… . Si hubiera puesto un asterico estaría recibiendo mensajes de canales a los que no estoy suscrito y aunque los descartase a nivel de aplicación estaría leyéndolos.

- **Usuario centro comercial**
  - cliente.*.informacion.centro .Por aquí leerá todos los mensajes de cada cliente. Podríamos haber escogido un routing pattern "cliente.informacion.centro", pero la decisión de hacerlo hecho de tal forma ha sido en intentar sobrecarga lo mínimo posible un solo canal, por lo que la aplicación del usuario centro comercial escuchará todos los canales individuales de cada usuario.
  - La aplicación del usuario centro comercial enviará los mensajes por las routing key del usuario comprador.

3.3 MENSAJES

La estructura de los mensajes enviados por la parte del usuario comprador se ha definido de tal forma que el receptor (aplicación del usuario centro comercial) realice una serie de funciones dependiendo de la "cabecera".

cabecera	IdCliente	Intereses

Existen cuatro posibles mensajes que la aplicación del usuario centro comercial puede llegar a recibir, donde el valor de la cabecera cambia en función a lo que se quiere:

1. **Hall**. Cuando la cabecera tiene este valor significa que el usuario ha entrado al centro comercial.
2. **Disconnect**. Cuando la cabecera tiene este valor significa que el usuario ha salido del centro comercial.
3. **Intereses.** Cuando la cabecera tiene este valor significa que el usuario ha modificado sus intereses.
4. **Default**. Cuando la cabecera no vale ninguno de los otros valores significa que el usuario se ha movido de área dentro del centro comercial.

Un ejemplo de envío sería: "hall 2312312 ofertas.supermarket.reposteria,ofertas.cine.*".

Mientras que la aplicación del usuario centro comercial simplemente manda por un routing key un mensajes, y la aplicación del usuario comprador leerá el routing key por el que viene y la visualizará donde corresponda, más adelante en la parte de la aplicación de Android se mostrará como se visualizan los mensajes.

3.4 APLICACIÓN USUARIO CENTRO COMERCIAL

Como se mencionó anteriormente esta aplicación se ha realizado en Java, teniendo una interfaz hecha en JavaFX. He hecho uso de Maven para mayor comodidad en las dependencias de paquetes.

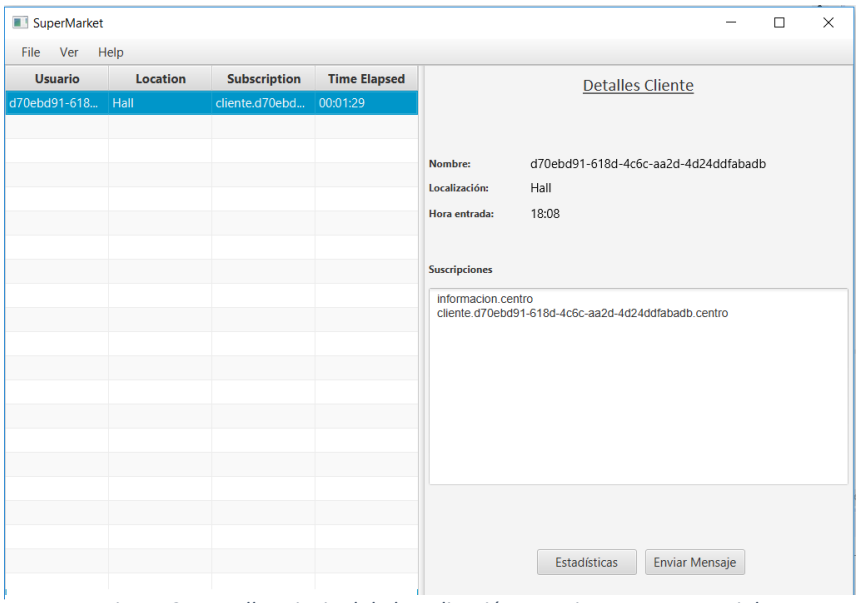


Figure 3 Pantalla principal de la aplicación usuario centro comercial

El objetivo de esta aplicación es conocer el número de usuarios que tenemos en el centro comercial, su localización, obtener los tiempos que pasa cada usuario cada lugar, y obtener estadísticas del tiempo promedio que pasa un usuario en X área.

En la figura 3 podemos ver la pantalla principal de la aplicación, tenemos un Split en el centro donde a la izquierda tenemos una tabla con el id del usuario, localización, suscripciones, tiempo desde que ingreso (se actualiza cada segundo). Esta tabla nos permite ordenar por cada atributo. Si clickeamos en alguno de los clientes de la tabla a la derecha podremos ver más detallada sus suscripciones, hora de entrada al centro comercial, pulsando el botón de estadísticas veremos el tiempo que ha pasado en cada uno de las zonas, y en el botón de enviar mensaje podremos enviarle un mensaje privada por los canales que tenga disponible. Para más información sobre la interfaz vaya a la sección de manual de usuario.

Una vez creada una idea sobre como es y se comporta la interfaz, pasaré a hablar del código de la aplicación. El código lo tenemos separado en tres paquetes.

1. com.javafx.RabbitMQ.entities
2. com.javafx.RabbitMQ.interfaz
3. com.javafx.RabbitMQ.utilidades

También disponemos de una carpeta test donde se han hecho pruebas antes de realizar la aplicación del usuario comprador en Android.

3.4.1 com.javafx.RabbitMQ.entities

Las clases que podemos encontrar en este paquete son clases son:

Cliente

En esta clase almacenaremos toda la información referida al cliente:

- Id del cliente.
- Intereses del cliente
- Hora de entrada al centro comercial.
- Localización actual.
- Historial de localizaciones anteriores y el tiempo que ha pasado en cada una de ellas (en minutos).

Consumidor

Esta clase que extiende de Thread, y es la encargada de estar leyendo los mensajes que nos llegan, anteriormente mencionamos los distintos mensajes que pueden llegar. Cuando se recibe uno de ellos llama al controlador ya sea para añadir un nuevo usuario, eliminarlo, cambiar sus intereses o su localización. El controlador será el encargado de crear, eliminar o modificar el objeto Cliente y de ponerse en contacto con la vista para que aparezca en la tabla de la vista principal (patrón MVC).

Controlador

El controlador es una clase con un patrón singleton, y es el encargado de la comunicación de la vista con las clases y viceversa. El controlador contiene un HasMap de clientes que almacena usuarios cuya key es el id del usuario (HashMap<String, Cliente>), esto es para facilitar procesos

como es la eliminación de usuarios, ejemplo, llega un mensaje con la cabecera a disconnect, por tanto, obtenemos el Id del usuario con el accedemos al hashmap y obtenemos el objeto cliente el cual eliminamos y le pasamos a la vista para que lo elimine de la tabla.

HiloPublicadorPrivado

Esta clase que extiende de Thread, y es la encargada de enviar los mensajes privados a las routing keys privadas de los usuarios dependiendo de ciertos requisitos, por ejemplo, permanecer más de X tiempo en un sitio. En el apartado de autonomía de la aplicación se describe más en profundidad su funcionamiento.

HiloPublicadorPublico

Esta clase que extiende de Thread, y es la encargada de enviar mensajes por las routing keys públicas para todo aquel usuario que este suscrito pueda leerlo. En el apartado de autonomía de la aplicación se describe más en profundidad su funcionamiento.

HiloRefresh

Esta clase que extiende de Thread, es la encarga de mantener refrescada la vista de la aplicación, por ejemplo, el tiempo transcurrido de un usuario en el centro se va refrescando (última columna) y si estamos viendo los detalles de un cliente y este actualizará sus suscripciones o su localización nosotros lo veríamos actualiza en tiempo real también sin tener que refrescar a mano.

Productor

Esta clase que también extiende de Thread se encarga de enviar el mensaje, su construcción es a partir de un Array de String dado, que sería el routing key y mensaje ({routingKey, mensaje}). Siempre que se quiera enviar un mensaje se crea un objeto de la clase Productor.

3.4.2 com.javafx.RabbitMQ.interfaz

Dentro de este paquete almacenamos todas las interfaces con sus respectivos controladores. Omito el desarrollo de cada una de las clases porque se trata simplemente de vistas y controladores de dichas vistas. En la figura 4 podemos ver todas las clases que componen al paquete interfaz.

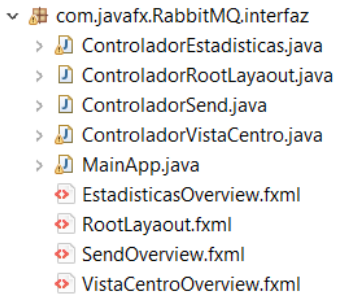


Figure 4 Clases de la interfaz

3.4.3 com.javafx.RabbitMQ.utilidades

Dentro de este paquete solo tenemos una clase.

Utils

En esta clase tenemos almacenadas las colas y localizaciones para poder acceder rápidamente a todas ellas.

3.4.4 Autonomía de la aplicación

Anteriormente hemos visto que teníamos dos clases llamadas "HiloPublicadorPrivado" e "HiloPublicadorPublico", estas dos clases son las encargadas de darle autonomía a la aplicación. Vamos a verlas un poco más en profundidad.

HiloPublicadorPrivado

Esta clase que es un hilo se encarga de dar recompensas a los clientes como incentivo para que sigan comprando o para que vuelvan al área de donde estaban:

- Si el usuario permanece un minuto o más en el cine le regalarán una entrada gratis.
- Si el usuario permanece un minuto o más en la charcutería le regalarán una hamburguesa de pollo.
- Si el usuario permanece un minuto o más en la frutería tendrá un 1kg de naranjas a mitad de precio.
- Si el usuario se va de la repostería después de haber permanecido un minuto o más le regalarán un pan pizza.
- Si el usuario se va de la pescadería después de haber permanecido un minuto o más tendrá 2x1 en lubina.
- Si el usuario se va de la sección de productos del hogar después de un minuto o más tendrá pañales a mitad de precio.

(He puesto 1 minuto para probarlo rápidamente).

Este hilo también se encarga de mandar un mensaje en caso de que no hayas seleccionado ningún interés, lo enviará por la routing key "cliente.id.centro".

Es un mecanismo sencillo aprovechando que tenemos canales privados con los usuarios y que almacenamos el historial de localizaciones y el tiempo en minutos que ha pasado en ellos.

Cabe destacar que este hilo se ejecuta cada segundo.

HiloPublicadorPublico

Esta clase que también es un hilo se encarga de enviar los mensajes por las routing keys de los usuarios compradores públicas. El objetivo es ver las áreas del centro comercial con promedio de tiempo menos pobladas y mandar mensajes sobre esas áreas para captar más usuarios.

Para conseguirlo sumamos el tiempo total de cada localización (el tiempo de cada usuario en esa localización) y lo dividimos entre el número de clientes. De esta forma por cada localización tenemos la media que pasa cada cliente en dicha área, ahora para saber sobre que área debemos mandar un mensaje para captar mayores usuarios sumamos todos los tiempos de cada

localización y lo dividimos por el número de localizaciones (todo esto sin tener en cuenta el hall) de esta forma obtenemos la media global, la localización que este por debajo de dicha media será elegida para enviar un mensaje sobre esa localización.

Esto es otro ejemplo de como aprovechamos los tiempos de las localizaciones obtenidos para tener un balance en el centro comercial.

Cabe destacar que este hilo se ejecuta cada 10 segundos. Podemos ver un ejemplo en la figura 13.

3.5 APLICACIÓN DEL USUARIO COMPRADOR

Esta aplicación como hemos mencionado anteriormente se ha realizado en Android, la he hecho lo más intuitiva posible y con un código bastante limpio.

La aplicación consta de:

Cuatro clases (entidades).

1. Mensaje
2. CanalVisual
3. Controlador
4. Usuario

Cuatro Activitys.

1. MainActivity
2. SettingActivity
3. CanalesActivity
4. ChatActivity

Dos servicios.

1. PubishService
2. SuscriptionService

Dos adaptadores.

1. CanalAdapter
2. MensajeAdapter

3.5.1 Entidades

CanalVisual

Esta clase representa los canales de las diferentes secciones a las que nos podemos suscribir. Un canal contiene ningún o muchos mensajes.

Mensaje

Esta clase representará el mensaje que recibe el usuario de un routing key y se le asigna al canal correspondiente.

Usuario

Esta clase representa al propio usuario y almacenará su localización, lista de intereses, canales a los que esta suscrito y su Id.

Controlador

La clase controlador es un singleton, que almacenará a nuestro usuario y otras variables que necesitaremos para cuando cambiemos entre Activitys para no perderlo cuando se destruyan, por tanto, hará la comunicación entre las entidades y la vista (MVC).

3.5.2 Activitys

En las figuras 5, 6, 7, 8 y 9 podemos ver las interfaces de las activitys. Las activitys CanalActivity y ChatActivty contienen dos adaptares, esto es para que el ListView que tenemos en ambas Activitys podamos personalizarlo insertando objetos como ítems dentro. Los objetos que se insertan son VisualizarCanal y VisualizarMensaje, se construyen con la clase CanalVisual y Mensaje.

Figure 6 Pantalla principal

Figure 5 Navegación dentro del centro comercial

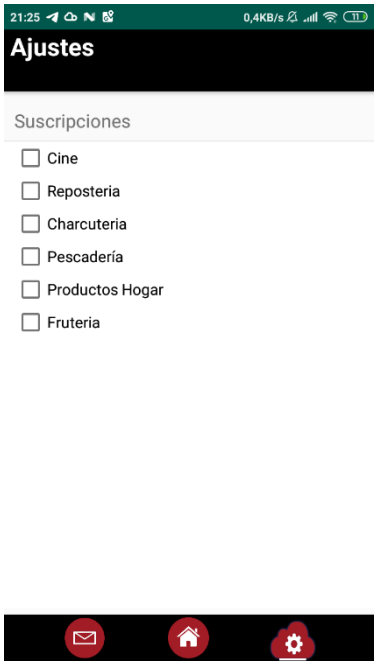


Figure 8 Pantalla de ajustes

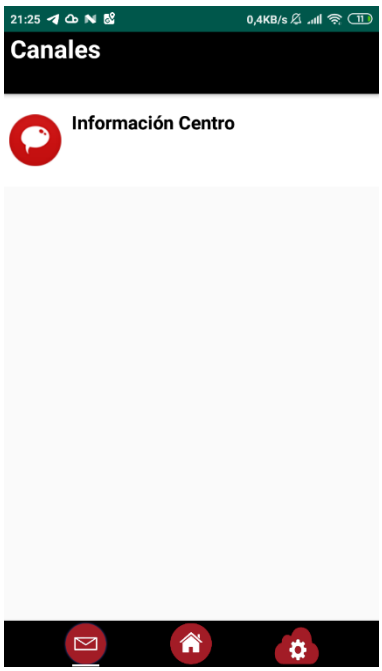


Figure 7 Pantalla de canales. Con el canal por defecto

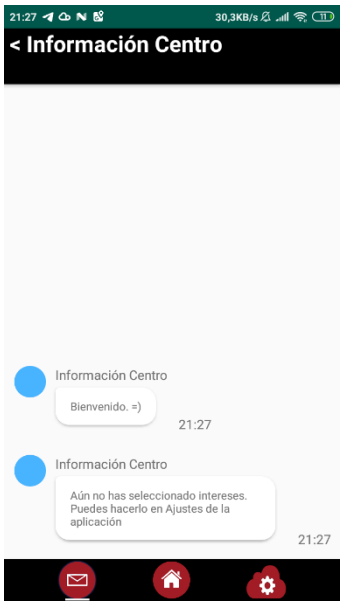


Figure 9 Chat. Canal información centro

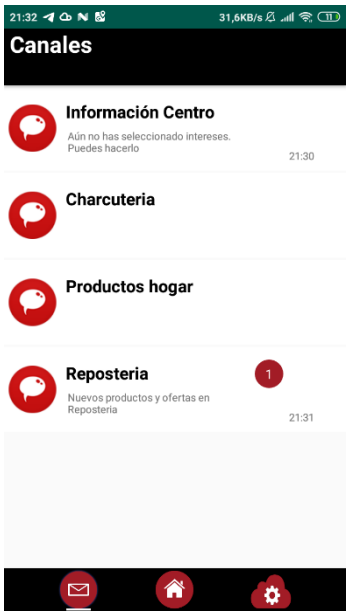


Figure 10 Pantalla de canales con suscripciones seleccionadas

3.5.3 Servicios

Tenemos dos clases (PublishService y SuscriptionService) que hemos declarado como servicios extendiéndolas de clase Service (extends Service), un servicio es un componente de una aplicación el cual se ejecuta en segundo plano (background), este puede realizar las mismas acciones que un Activity con la única diferencia que este no proporciona ningún tipo de interfaz de usuario.

La idea es que dada una acción que hagamos en la interfaz se ejecute por debajo el servicio de PublishService, en el caso de que tengamos que enviar un mensaje lo realice sin que afecte a ningún funcionamiento de la aplicación y cuando termine de realizarlo se "muere". Con el servicio SuscriptionService que se encarga de recibir los mensajes de la aplicación se inicia en cuanto entramos al centro comercial, y se para cuando salimos del centro comercial.

El servicio que se encarga de la lectura de los mensajes tiene incorporado un sistema de notificaciones básico, cuando llega un mensaje en la barra de notificaciones de Android recibiremos una notificación, la cual nos indicará que hemos recibido mensajes y la cantidad de mensajes que tenemos sin leer.

3.5.4 Detalles

He mencionado anteriormente que tenemos un sistema de notificaciones básico, pero también existen otros detalles en la aplicación, uno de ellos se puede apreciar en la figura 10, que nos esta indicando que tenemos un mensaje sin leer dentro de la aplicación. Otros detalles, que se pueden apreciar es que podemos ver el último mensaje que nos ha llegado y la hora dentro de la Activity de canales, sin tener que acceder a el chat.

3.6 PROBLEMAS

A la hora de la prueba y depuración de las aplicaciones he descubierto problemas que surgen en determinadas situaciones por culpa de los hilos publicadores que tenemos. Los problemas son debidos a el acceso múltiple de recursos compartidos como es la lista de clientes mientras se esta modificando. La solución esta en aplicar algún mecanismo como monitores o semáforos a los recursos compartidos. Por falta de tiempo no he podido resolver el problema.

4 MANUAL DE USUARIO

Para ejecutar las aplicaciones debemos hacerlo en el orden correcto. Primero ejecutaremos la aplicación de Java (la del usuario centro comercial) y posteriormente la de Android. Esto es debido a que es necesario de esta forma para que la aplicación Java recoja a los usuarios de la aplicación en Android.

4.1 APLICACIÓN USUARIO CENTRO COMERCIAL

4.1.1 Enviando un mensaje privado

Para enviar un mensaje privado por uno de los canales, que tiene disponible el usuario es tan sencillo como hacer doble click en la tabla de los usuarios que están disponibles o clickear en la ventana de los detalles del cliente, en el botón “Enviar Mensaje”. Una vez hecho eso nos saldrá una ventana como podemos ver en la figura 11. Seleccionaremos un canal y le daremos a enviar, una vez hayamos terminado de enviar mensajes privados a dicho usuario cerraremos la ventana.

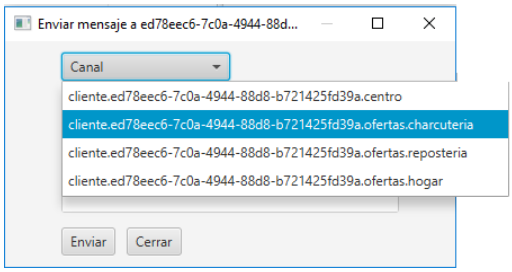


Figure 11 Envío privado

4.1.2 Enviando un mensaje global

Para enviar un mensaje para todos los que estén suscritos al canal por donde se envíe, debemos ir al menú desplegable en la parte superior de la pantalla con el nombre de “File” -> “Send All”. Nos saldrá una ventana como la de la figura 11, pero con los canales públicos.

4.1.3 Viendo estadísticas de un cliente

Para ver las estadísticas de un usuario tendremos que pulsar en el botón de estadísticas, lo podemos encontrar en la ventana de detalles del cliente. En la figura 12 podemos ver que es lo

que nos muestran estas estadísticas, ellas nos están indicando el tiempo que ha pasado en cada uno de los lugares sin tener en cuenta en lugar actual.

4.1.4 Viendo estadísticas globales del centro

Para ver el promedio que pasa cada usuario en cada lugar debemos ir al menú superior y en "Ver" -> "Estadísticas". Veremos una pantalla muy parecida a la de la figura 13.

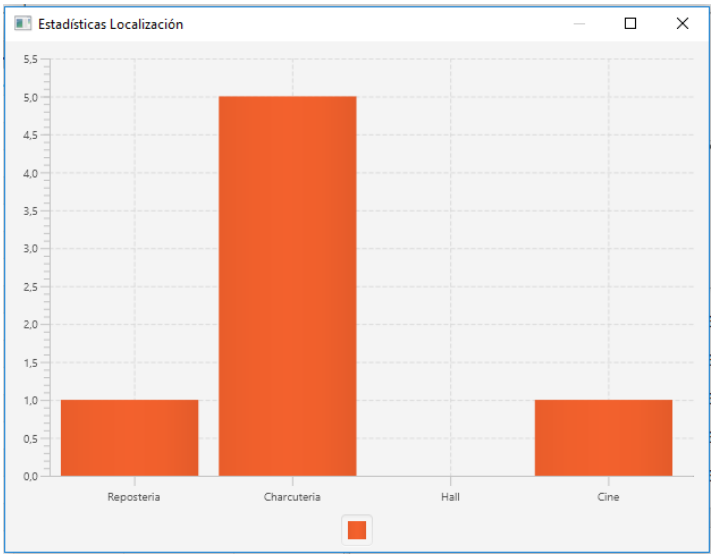


Figure 12 Estadísticas personales

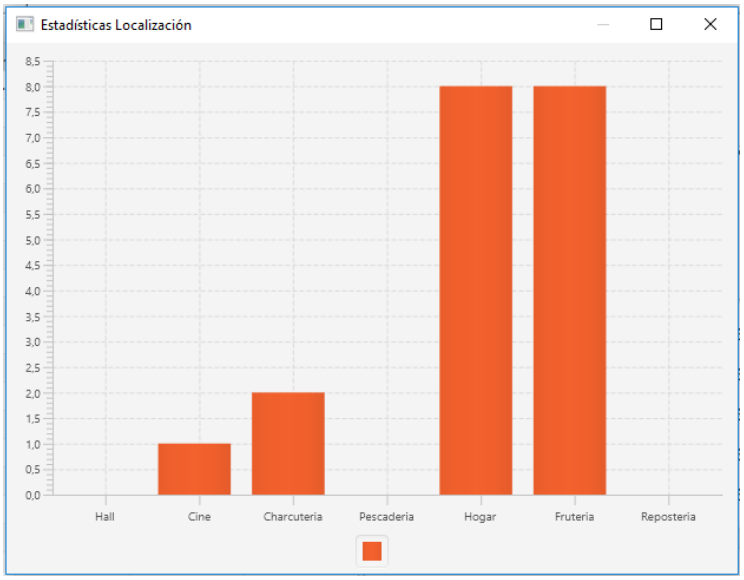


Figure 13 Estadísticas globales

4.2 APLICACIÓN USUARIO COMPRADOR

La aplicación en Android es bastante sencilla e intuitiva.

4.2.1 Menú principal

Como podemos ver el cualquiera de las figuras 5, 6, 7, 8 y 9. En la barra inferior tenemos el menú principal por el nos moveremos a los canales con sus respectivos chats (izquierda), a la navegación del centro comercial (centro), y a los ajustes (derecha).

4.2.2 Estableciendo suscripciones

En la figura 8, podemos ver en que en los ajustes tenemos los distintos canales para suscribirnos, cuando seleccionamos uno automáticamente se creará un canal de chat que podremos ver en la pestaña de canales.

4.2.3 Navegando por el centro comercial

En el botón del centro del menú principal (la barra inferior), cuando pulsemos el botón de ingresar en el centro comercial, entonces aparecerá los distintos lugares con los que podremos simular el movimiento.

5 CONCLUSIONES

He disfrutado mucho haciendo esta práctica que al final la he hecho más larga de lo que parece, ya que la curva de aprendizaje tanto de JavaFX como de Android ha sido pronunciada porque he usado funcionalidad “avanzada”, además hay funciones que no se utilizan, pero sí que están programadas como es un método que te devuelva el usuario con más tiempo.

En general pienso que es una práctica muy completa y en la que he usado diferentes tecnologías como: JavaFX, Maven, AMQP, Android.

En la parte de Android, nunca había programado nada, pienso que le he sacado bastante partido y la aplicación ha quedado bastante bien, ya que no me he limitado a tener solo una Activity, he creado adaptadores y servicios.

JavaFX tampoco había visto nunca y tenia ganas de usarlo, vi la oportunidad en esta práctica y la he aprovechado. El reto ha sido la actualización constante en las diferentes vistas.

6 FUTURAS AMPLIACIONES

Debería corregirse los problemas de concurrencia, la forma más eficaz sería creando una aplicación aparte que consultará los datos ya sea a través de una API REST o desde base de datos. Es decir, no tener hilos corriendo por debajo desde la misma aplicación del usuario centro comercial, sino que estuviera fuera.

Una ampliación sería añadir persistencia a posibles caídas del servidor y poder recuperarse con los datos.

Mejorar partes del código que debido a la entrega y otros trabajos ha sido bastante chapucero y poco fino.