

DISEÑO E IMPLEMENTACIÓN DE UN
SERVICIO DE ANUNCIOS DISTRIBUIDO
Reto 1

Presentado por:
JORGE GAVIRIA LÓPEZ
DANIEL MÚNERA SÁNCHEZ
JAVIER QUINTERO GONZÁLEZ

Presentado a:
EDWIN MONTOYA MÚNERA
Profesor

UNIVERSIDAD EAFIT
DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS
TÓPICOS ESPECIALES EN TELEMÁTICA
MEDELLÍN
2012-1

CONTENIDO

1.	PLANTEAMIENTO DEL PROBLEMA.....	3
2.	ANÁLISIS Y DISEÑO DE LA ARQUITECTURA DISTRIBUIDA.....	4
3.	CONSIDERACIONES ACERCA DEL SISTEMA DISTRIBUIDO.....	7
4.	ATRIBUTOS DE CALIDAD DE LA ARQUITECTURA.....	9
5.	DEFINICIÓN DEL PROTOCOLO.....	10
	Definición del Servicio.....	10
	Suposiciones del Entorno.....	10
	Vocabulario de Mensajes.....	10
	Codificación de Mensajes.....	11
	Métodos utilizados para la decodificación de Mensajes.....	12
	Explicación de los métodos para la decodificación de Mensajes.....	12
	Reglas del Protocolo.....	14
6.	DESPLIEGUE DE LA APLICACIÓN.....	17
7.	RECURSOS UTILIZADOS.....	21
8.	CONSIDERACIONES DEL DESARROLLO.....	21

1. PLANTEAMIENTO DEL PROBLEMA

Diseñe e Implemente un sistema de anuncios (advertisement) que permita a un conjunto de clientes obtener mensajes de anuncios de productos o servicios. El sistema tiene dos módulos principales:

- Proveedor de anuncios (AdFuente), es decir, es desde donde se generan los mensajes
- Cliente de anuncios (AdCliente), el cual recibe los mensajes enviados a un Canal por un AdFuente

El sistema debe tener un sistema de gestión que permita:

- 1) Canales y Mensajes: *Canales* a través de los cuales fluyen los *Mensajes* originados en un *AdFuente* (pueden haber varias fuentes en un mismo Canal) hacia uno o más AdCliente
- 2) Envíos: son los mensajes que son enviados por un AdFuente hacia un Canal y que le debe llegar a los AdCliente. Tenga en cuenta la situación cuando los clientes están o no en línea, que supuesto realizar al respecto.

Los Canales son temáticos (deportes, tecnología, noticias, culinaria, bolsa, etc) y deben ser gestionados en el sistema (crear, modificar, borrar, etc un canal). Los anuncios son recibidos por los clientes en modo PUSH y PULL. Por anuncios PUSH se entiende la característica de recibir en un cliente mensajes sin haber sido solicitada explícitamente por el cliente. Por anuncios PULL se entiende cuando el cliente explícitamente recupera mensajes de un canal específico. Debe tener en cuenta que criterio utilizará para borrar mensajes de una cola.

2. ANÁLISIS Y DISEÑO DE LA ARQUITECTURA DISTRIBUIDA

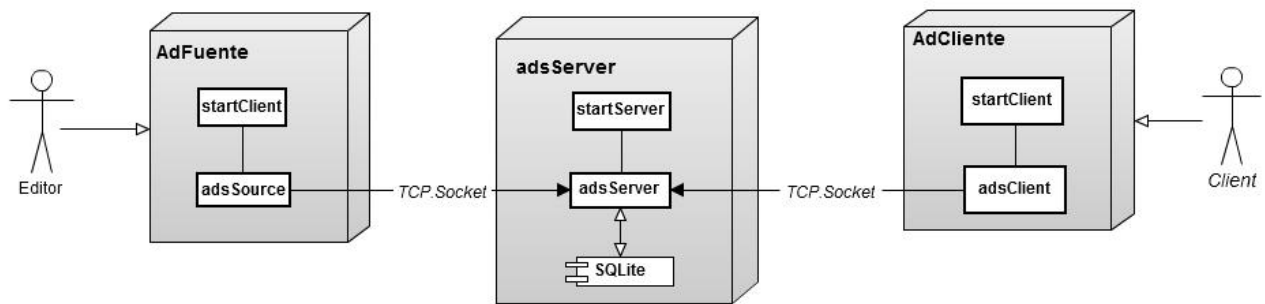
Para poder desarrollar esta aplicación procedimos identificando las interacciones que se deben llevar a cabo entre las entidades reconocidas (AdFuente y AdCliente). El primero se encarga de enviar información al segundo y para hacer esto posible se necesita conocer la ubicación de los clientes con el fin de enviarles la información que han solicitado. Así mismo puede darse el caso de que varios AdFuente quieran hacer llegar anuncios a los mismos clientes y en lugar de tener dicha información replicada en cada uno de ellos, se puede optar por establecer un modelo Cliente Servidor, en la cual la información que deben tener los AdFuente es la localización del Servidor y la especificación de la clase de mensaje que envía; el servidor se encarga de enviar los mensajes, incluyendo el manejo especial que debe dársele a los mensajes destinados a clientes que se encuentran offline.

Otro aspecto importante para decidirse por dicha arquitectura, es que disminuye la complejidad del problema ya que concentra en el servidor todo la lógica relacionada con la recepción y envío de mensajes, y gracias a las características de la arquitectura cliente servidor se contribuye a la escalabilidad de la aplicación y se desacoplan los emisores de mensajes de los clientes que los reciben.

Dentro del desarrollo de la aplicación se hace uso del protocolo TCP ya que este brinda una transmisión de mensajes segura y con ello se reduce la complejidad de la aplicación, no teniendo que preocuparse por el desarrollo de un protocolo que garantice el envío y recepción de mensajes.

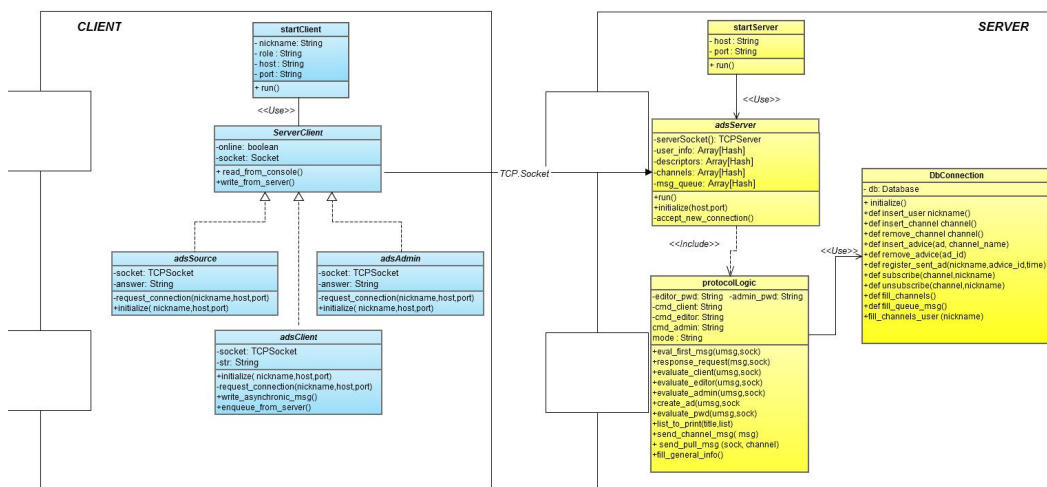
El siguiente diagrama muestra el despliegue de nuestro sistema. En él se pueden ver claramente las 3 entidades identificadas y la manera como entre ellas se establece el paso de mensajes. La entidad AdFuente a través de la clase startClient lanza la aplicación, la cual permite a este cliente establecer la conexión con el servidor, que es lanzado por la clase startServer; a su vez es la clase startClient la que permite el lanzamiento del cliente e invoca a la clase adsClient para poder realizar la conexión. En este diagrama también podemos como en el servidor se utiliza el módulo de SQLite para poder manejar la persistencia.

Deployment Diagram



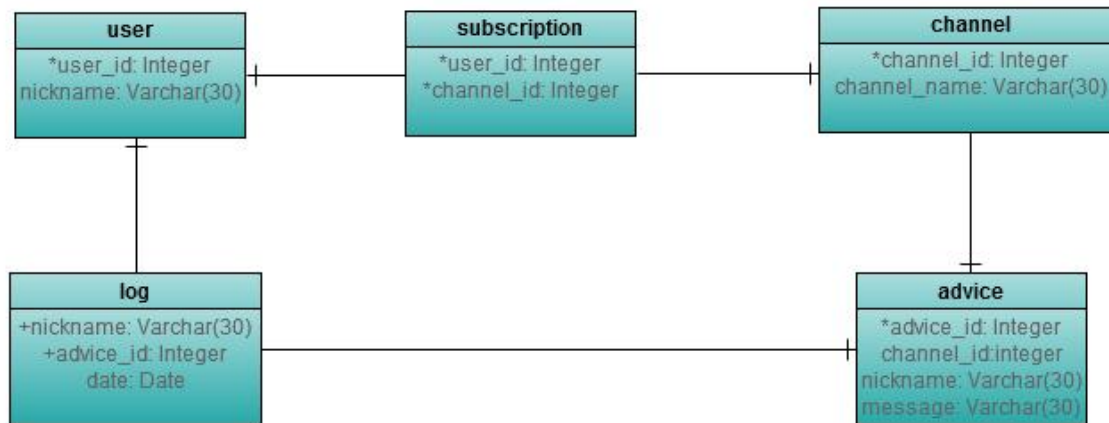
El siguiente diagrama de clases nos puede clarificar más la arquitectura del sistema de mensajería. En él vemos como a través de un patrón Factory se crean los diferente tipos de cliente (adsSource, adsClient, adsAdmin) que se encargan de heredar de la clase abstracta ServerClient los métodos initialize(), read_from_console(), request_connection(), write_from_server(). Una vez definido el tipo de cliente creado, se establece la conexión TCP Socket con el server, que tiene en la clase protocolLogic la implementación de la lógica necesaria para el manejo de los mensajes.

Class Diagram

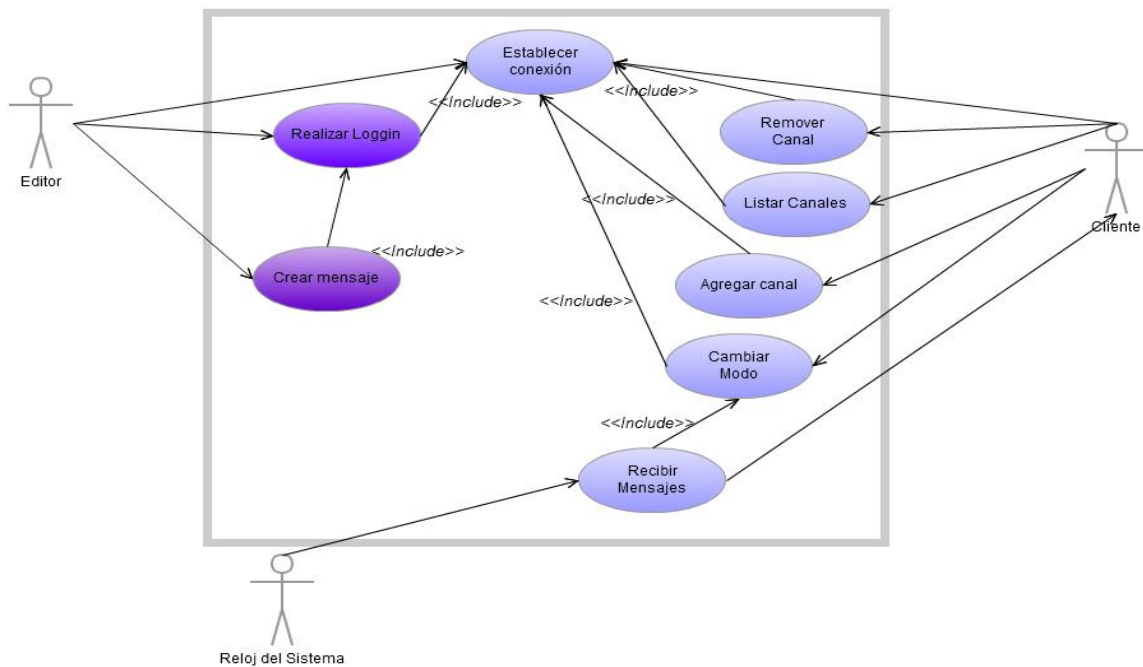


Como pudimos ver en el diagrama de despliegue existe un módulo para realizar el manejo de persistencia. En un sencillo conjunto de tablas se guardará la información correspondiente a los canales que actualmente existen así como los clientes que están suscritos a ellos y los mensajes que son enviados a los diferentes canales para ser distribuidos, también se guardarán los mensajes que ya han sido consumidos por el cliente en la tabla log.

Entity Relationship Diagram (ERD)



El siguiente diagrama de casos de uso permite entender un poco más la aplicación y ver de manera resumida las principales acciones que pueden ser llevada a cabo por dos clases de clientes especiales: el editor y el cliente. Es importante resaltar que para realizar las diversas acciones es necesario en primer lugar establecer la conexión. Una vez se establezca el modo de recepción en push el reloj del sistema se encargará de enviar los mensajes al usuario.



Finalmente podemos señalar que en el desarrollo de esta aplicación emulamos un protocolo de aplicación que permite el envío y recepción de mensajes. Así mismo no existen parámetros como direcciones IP o ports quemados en el código ya que al momento de establecer la conexión es el usuario que está ejecutando dicho caso de uso el que tiene que ingresar vía consola estos datos.

3. CONSIDERACIONES ACERCA DEL SISTEMA DISTRIBUIDO

- **Interacción Sincrónica y Asincrónica:** la relación entre el AdFuente y el AdCliente es asincrónica porque la interacción entre estas dos entidades finales no es bloqueante, ni siquiera es necesario que el destinatario final de los mensajes se encuentre online en el momento en que estos son enviados, ya que llegan al servidor y este encargará de hacerlos llegar a su destino final. Otra relación asincrónica presente en la aplicación se da entre AdCliente en modo PUSH y el servidor, debido a que el cliente puede seguir interactuando con el servidor mientras que está recibiendo los nuevos mensajes de su canal.

La relación entre AdFuente con el servidor y AdCliente en modo PULL con el servidor es sincrónica, ya que una vez que el usuario solicita los mensajes de un canal específico, se bloque hasta que recibe los mensajes seleccionados. De igual manera la relación existente entre AdFuente y el servidor es sincrónica ya que una vez que la fuente envía un mensaje al servidor, se bloque hasta espera la confirmación de la recepción exitosa de su envío.

- **Interacción Simétrica:** teniendo en cuenta el modelo Cliente Servidor y las 3 entidades que participan en dicho sistema (AdFuente, Servidor, AdCliente) podemos considerar las siguientes interacciones:
 - Interacción Asimétrica entre AdFuente y Servidor: la función del servidor será la de recibir los mensajes que envíe AdFuente, con el correspondiente formato establecido.
 - Interacción Simétrica entre Servidor y AdCliente: debido a que existen 2 maneras de recuperar los mensajes Pull y Push, es necesario establecer una comunicación en doble sentido. En el modo Push es el servidor el que se encarga de enviar mensajes a los correspondientes clientes y en el modo Pull es el cliente el que interactúa con el servidor pidiéndole información sobre un tema en particular.

- **Manejo de Sesión:** consideramos que es necesario un manejo de Sesión, ya que mientras un cliente se encuentre conectado es indispensable conocer su identidad para poder hacerle llegar los mensajes correspondientes a los canales a los cuales está suscrito. El manejo de estado es valioso, ya que a través de este se puede garantizar el envío seguro de mensajes, lo que librerá al sistema de encargarse de tener que brindar dicha funcionalidad y es por ello que se realiza la aplicación sobre el protocolo TCP.

- **Modelo de Manejo de Fallos:** teniendo en cuenta que nuestro servidor puede atender peticiones concurrentes es importante que el fallo provocado en uno de los sockets no vaya a bloquear al servidor por completo, de manera que cuando un cliente introduzca un comando que solicite un servicio al que él no tengo acceso o que la sintaxis sea incorrecta, el servidor enviará un mensaje en el que notificará el error y continuará prestando servicio a los demás usuarios. Así mismo cuando por algún motivo el servidor deja de funcionar, se enviará este mensaje a todos los clientes, terminando la comunicación que hasta el momento venían desarrollando con él.

- **Niveles de Transparencia:** los niveles de transparencia los podemos observar desde los siguientes puntos de vista:
 - Transparencia de Localización: los encargados de emitir los mensajes no deben preocuparse por la localización física de los receptores de los mismos
 - Transparencia de Migración: los recursos de información relacionados con los canales de distribución o sobre los clientes puede ser movida sin que los AdFuente o los AdCliente se tengan que preocupar de ello.
 - Transparencia de Concurrencia: los fuentes no tienen por qué preocuparse de detalles acerca de problemas relacionados con el hecho de que varios AdFuente quieran enviar información a los mismos cliente a la misma vez, así mismo los AdCliente no tienen por qué preocuparse de los detalles relacionados con el hecho de que varios de ellos quieran recuperar información de los mismos canales al mismo tiempo.

Así mismo los AdFuente no tienen que preocuparse por los cambios que puedan realizar los AdCliente para la recepción de mensajes, para ellos es transparente que los usuarios estén recibiendo los mensajes en modo Pull o Push.

- **Multiusuario:** la aplicación está desarrollada de manera que se soporten múltiples usuarios que quieran establecer una conexión vía TCPSocket con el server ya que gracias al empleo de la función select proveía por Ruby se pueden conocer dentro de los sockets de conexión en cuáles se han producido un evento y atender dicho evento.

4. ATRIBUTOS DE CALIDAD DE LA ARQUITECTURA

- **HETEROGENEIDAD**

Este problema se ve en que las aplicaciones del cliente pueden ejecutarse sobre diversos sistemas operativos en los cuales sea soportado el lenguaje de programación ruby.

- **EXTENSIBILIDAD**

La extensibilidad del sistema está soportada por el desarrollo de un modelo cliente en el servidor en el cual en este último podrían desarrollarse nuevos módulos a través de los cuales se de soporte a nuevas funcionalidades. Al tenerse concentrada las funcionalidades principales de la aplicación en el servidor el trabajo de codificación tendría que realizarse allí e informar a los clientes de las nuevas funcionalidades incluidas.

- **SEGURIDAD**

La seguridad de este sistema es bastante básica y para el caso del adsSource y del adsAdmin, basta una contraseña que será validada por el servidor para poder tener acceso a la aplicación. Así mismo se verificará que los comando ingresados por el

usuario corresponden a funciones permitidas según su rol (administrador, cliente o editor).

- **ESCALABILIDAD**

De momento el sistema está diseñado para soportar una carga pequeña de clientes y gracias a la utilización de la función select brindada por ruby, se pueden atender los sockets en los cuales ha ocurrido un evento, de esta manera el servidor sólo atenderá aquello que necesiten ser atendidos.

- **TOLERANCIA A FALLOS**

Al trabajar en una aplicación que depende de la disponibilidad de conexiones, la tolerancia a fallos enfatiza en poder mantener disponible el servicio aunque los dispositivos o las conexiones fallen. Esto se hace diseñando la aplicación de manera que las interacciones con el servidor puedan ser manejadas a través del manejo de excepciones, de manera que cuando un usuario ingrese un comando inválido o sintácticamente incorrecto, el server capture esta excepción y lo notifique sin que por esto la conexión se caiga.

- **TRANSPARENCIA**

Fue explicada en la sección de consideraciones acerca del sistema distribuido.

- **CONCURRENCIA**

Cada vez que ocurre una solicitud de un cliente se establece un nuevo socket ligado con el servidor. Una vez más es importante la utilización de la función select que permite atender solo a los sockets en los cuales han ocurrido eventos.

5. DEFINICIÓN DEL PROTOCOLO

- **DEFINICIÓN DEL SERVICIO**

El protocolo permitirá la comunicación entre dos entidades finales por medio del intercambio de mensajes y gracias a la mediación de un servidor que proveerá un esquema de persistencia en base de datos y permitirá la distribución de mensajes para los usuarios finales, ya sea que estos se encuentren actualmente conectados o no.

Este es un Protocolo que permite a un conjunto de entidades conocidas como AdFuente enviar mensajes los cuales se clasifican en categorías, especificando un canal (Un canal es un concepto utilizado para agrupar los mensajes por temáticas). Estos mensajes son recuperados por unos destinatarios o AdClientes, quienes podrán recibir los mensajes en 2 modos, Pull, cuando el cliente recupera mensajes de un canal específico y Push, cuando el cliente recibe mensajes sin haberlos solicitado explícitamente.

Características

- Manejo de Sesión y Estado

- Confiable (Protocolo TCP)
- Asimétrico/Simétrico
- Protocolo sincrónico

- **SUPOSICIONES DEL ENTORNO**

El protocolo utilizará un protocolo confiable, sobre el protocolo TCP a través de la API de Sockets, haciendo control de flujo y control de errores. Está desarrollado en modo consola

- **VOCABULARIO DE MENSAJES**

La aplicación maneja 3 tipos de usuarios, cada uno tiene un vocabulario de mensajes válidos para comunicarse con el servidor y ejecutar las funciones que le son permitidas en el sistema.

- **CLIENTE(adsClient):** {set_mode: , get_ads: , channel_list , rm_channel: add_channel: , my_channels }
- **EDITOR(adsFuente):** {create_ad: , rm_ad: , pwd: , ads_list}
- **ADMIN :** {add_channel: , rm_channel: , channel_list , change_pwd: change_editor_pwd: , pwd:}

- **CODIFICACIÓN DE MENSAJES**

-Estructura general de un mensaje:

La estructura general de un mensaje enviado por cualquiera de los tipos de usuario es simple, la primera palabra ingresada será el comando que el usuario quiere ejecutar, luego lo sigue, dependiendo del tipo de comando, desde 0 a n conjuntos de **espacio** seguido de un **parametro**.

<CMD> *{<ESPACIO><PARAM>}

Codificacion mensajes cliente:

- **user_info:** <nickname>
- **set_mode:** <espacio>[pull | push]
- **get_ads:**<espacio><channel> , donde channel debe ser un canal registrado en el sistema y presente en la lista de canales del usuario.
- channel_list

- **rm_channel:**<espacio><channel> , donde channel debe ser un canal registrado en el sistema y presente en la lista de canales del usuario.
- **add_channel:**<espacio><channel> , donde channel debe ser un canal registrado en el sistema.
- **my_channels**

Codificacion mensajes editor(fuente):

- **source_info:** <nickname>
- **create_ad:** <espacio><channel><espacio><,><espacio><advice>
- **rm_ad:**<espacio><ad_id>
- **pwd:**<espacio><entered_password>
- **ads_list**

Codificacion mensajes admin:

- **admin_info:** <nickname>
- **add_channel:**<espacio><channel>
- **rm_channel:**<espacio><channel>
- **channel_list**
- **change_pwd:**<espacio><new_pwd>
- **change_editor_pwd:** <espacio><new_pwd>
- **pwd:**<espacio><entered_password>

• **MÉTODOS UTILIZADOS PARA LA DECODIFICACIÓN Y CODIFICACIÓN DE MENSAJES**

adsSource- adsServer

- request_connection()
- accept_new_connection()
- entered_password()
- response_password()
- create_adv()
- response_adv()
- end_connection()

adsAdmin-adsServer

- request_connection()
- accept_new_connection()
- entered_password()
- response_password()
- send_command()

- response_command()
- end_connection()

adsClient-adsServer

- request_connection()
- accept_new_connection()
- send_command ()
- response_command()
- send_channel_msg()
- end_connection()

• EXPLICACIÓN DE LOS MÉTODOS PARA DECODIFICACIÓN Y CODIFICACIÓN DE MENSAJES

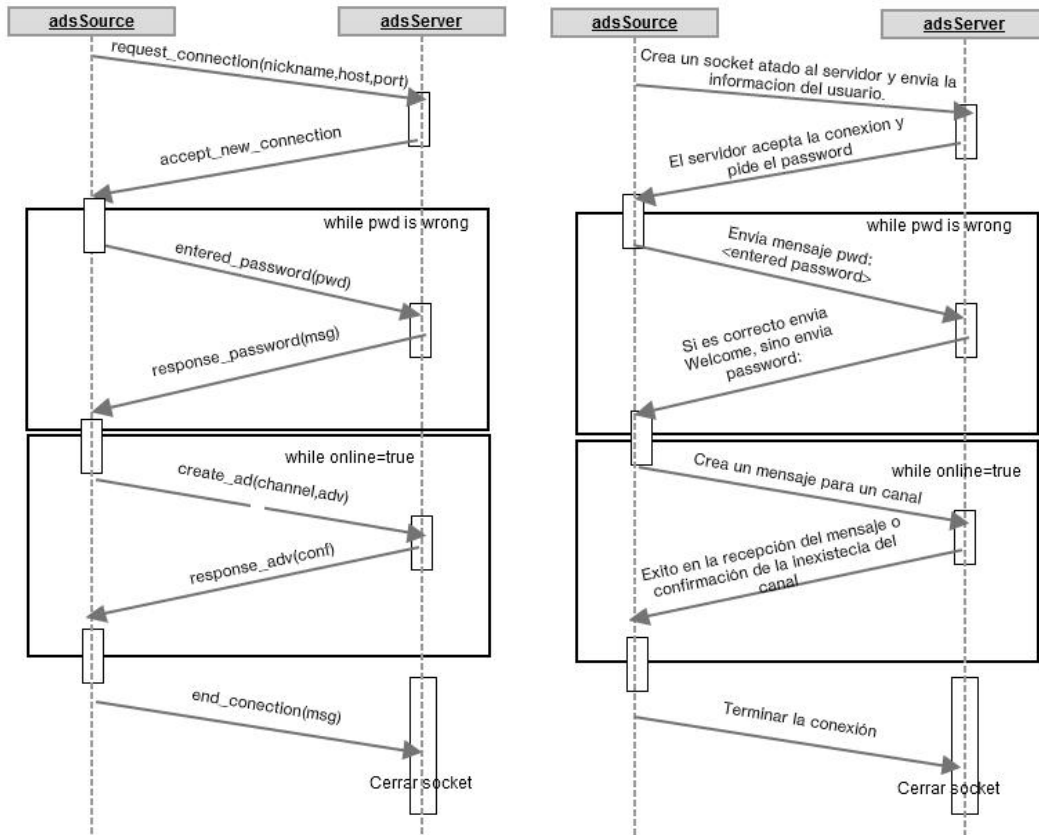
- request_connection (nickname, host, port). Operación que permite realizar la conexión TCP entre el cliente y el servidor, especificando el nombre de usuario, dirección IP del servidor y el puerto a través del cual se conectará.
- accept_new_connection(msg). El servidor acepta la nueva conexión y le pide al AdFuente que ingrese su contraseña.
- enter_password(pwd) Para poder ingresar adecuadamente al sistema, siendo un AdFuente, debe enviar el password para poder validarlo y tener acceso a la aplicación.
- response_password (msg) el servidor en caso de que la contraseña ingresada sea correcta, seguirá pidiendo la contraseña hasta que sea digitada correctamente y en ese momento dará el mensaje de bienvenida y estará disponible para recibir nuevos comandos.
- create_advice (channel, adv): una vez registrado el AdFuente tiene la posibilidad de crear mensajes, los cuales serán enviados al servidor y este se encargará de distribuirlos entre los usuarios que estén inscritos a estos canales y dependiendo del modo en el que se encuentre (push o pull). Además se especificará el canal al cual pertenece este mensaje.
- response_adv(conf) el servidor enviará un mensaje indicando que se pudo crear el mensaje correctamente o en caso contrario enviará un mensaje indicando la inexistencia del canal o algún error de sintaxis.
- Send_comand(msg). Dependiendo de la clase de usuario que sea se podrá tener acceso a una serie de comandos.

Así si es cliente tendrá las siguientes opciones: `channel_list` (le permitirá obtener la lista de los canales a los cuales puede suscribirse) `add_channel` (le permitirá seguir un canal recibiendo los mensajes de este canal) `my_channels` (listará los canales a los cuales está suscrito) `rm_channels` (le permitirá remover alguno de los canales a los cuales esté suscrito) `set_mode` (le permitirá cambiar los modos de recibir mensajes, push o pull, este último es el modo por defecto) `get_ads` (le permitirá recuperar los mensajes de un canal específico y para ello debe estar en modo pull). Para el editor las opciones que estará disponibles serán las siguientes: `add_channel` (le permitirá agregar un nuevo canal), `rm_channels` (removerá alguno de los canales existentes) `channel_list` (le permitirá listar los canales que actualmente se encuentran activos).

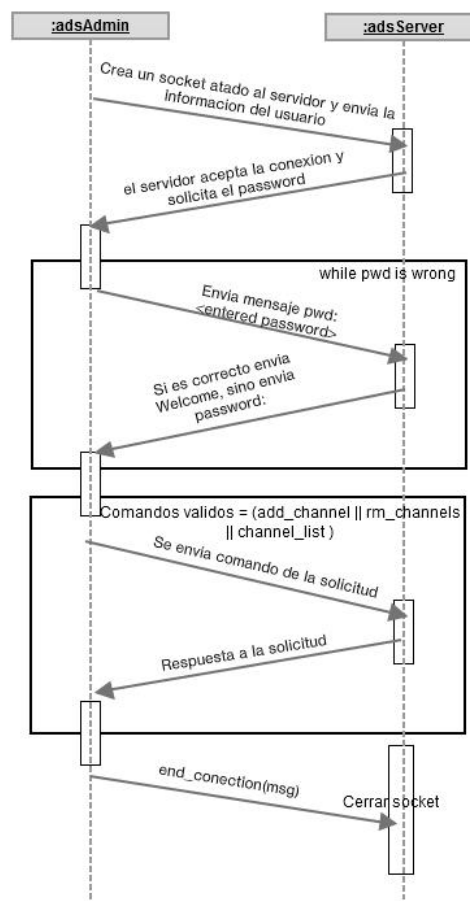
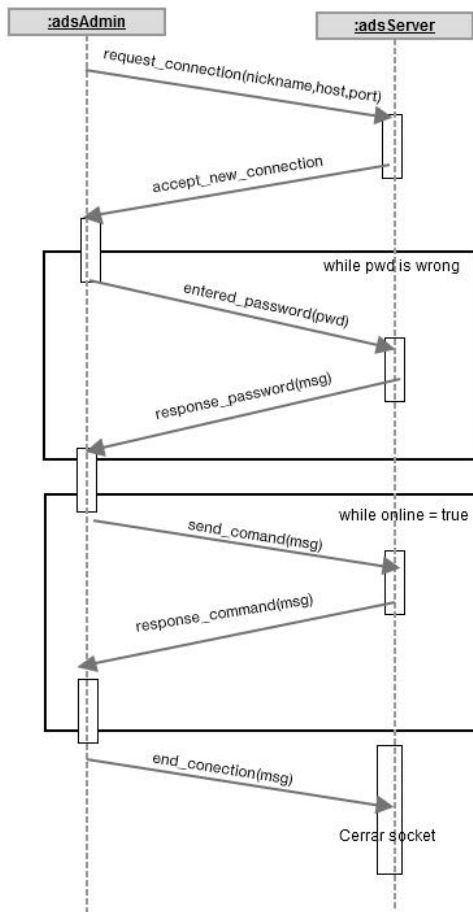
- `response_command(msg)`. Dependiendo de la operación que previamente se haya seleccionado se procederá a dar una respuesta indicando si la actividad se pudo llevar a cabo como en el caso de agregar o remover un canal, o listando los canales que existen o a los que está escrito, o la restauración del modo en el que se encuentra disponible para recibir los mensajes. Así mismo en caso de que el comando ingresado sea inválido se le indicará que esta mal realizada la operación y se le retornará un mensaje con la lista de los comandos válidos que puede digitar.
- `send_channel_msg(msg)`. En el caso de que el cliente desee recibir los mensajes en modo push, se ejecutará un hilo que le permitirá estar recibiendo dichos mensajes a mediada que son enviados por el AdFuente, y como son mensajes asincrónicos le permitirá enviar otros mensajes a través del mensaje `send_command()`, todo esto mientras esté activo dicho modo.
- `end_connection (msg)`. Con este mensaje el AdFuente terminará la conexión con el servidor y enviará la notificación del eof al servidor, quien la recibirá y al notar que un socket ha sido terminado procederá al cierre de este y actualización de los campos necesarios con respecto a la información almacenada del usuario.

- **REGLAS DE PROTOCOLO**

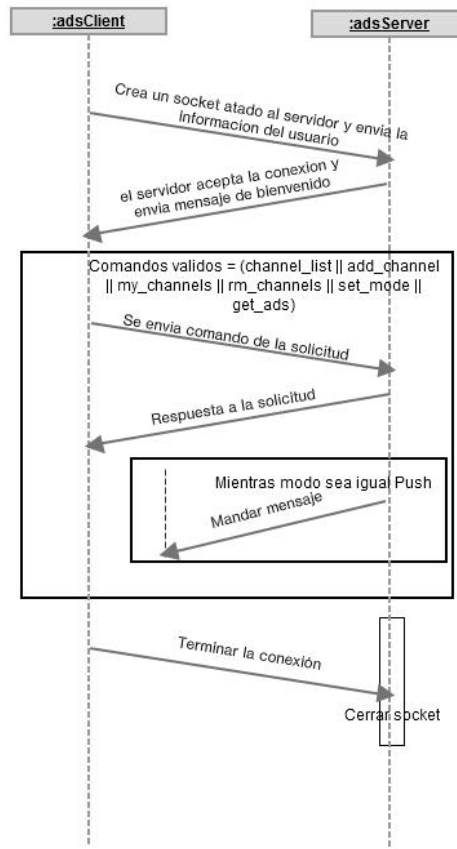
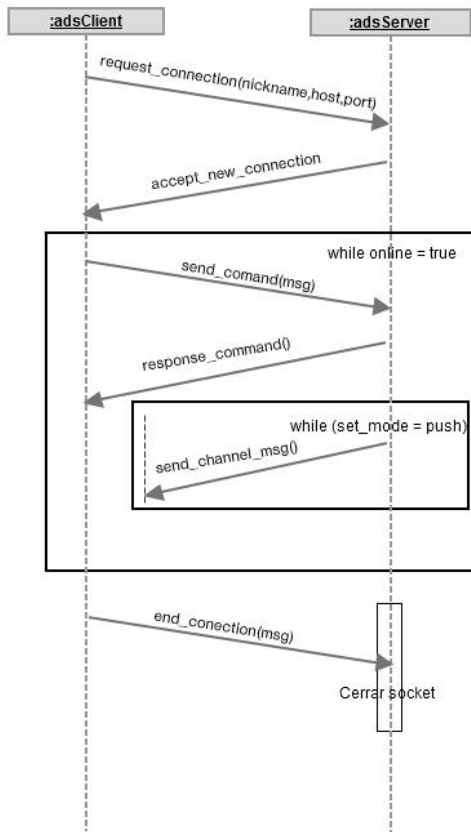
Sequence Diagram adsSource



Sequence Diagram adsAdmin



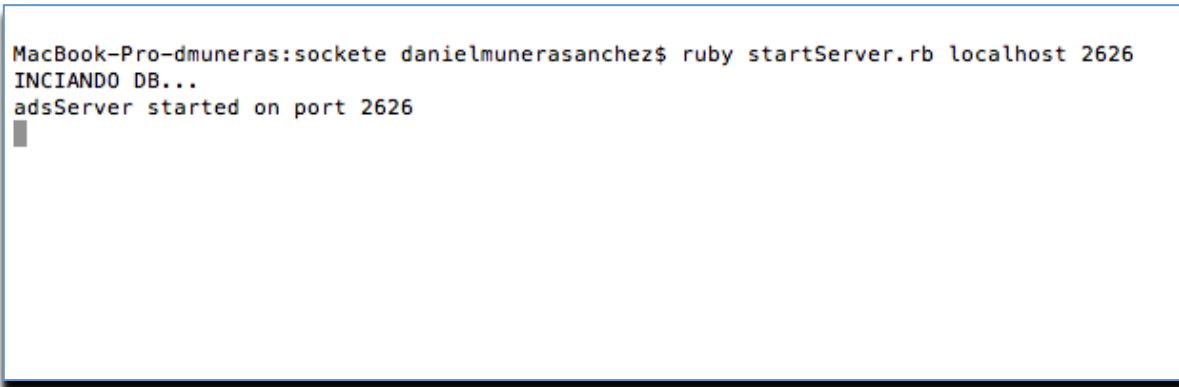
Sequence Diagram adsClient



6. DESPLIEGUE DE LA APLICACIÓN

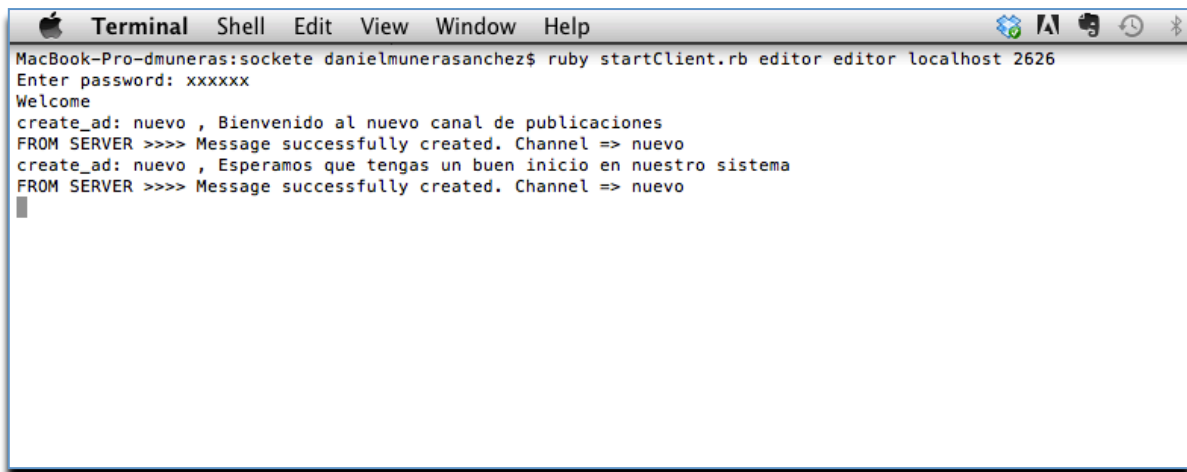
Los siguientes screen shots muestran la manera como corre la aplicación y que nos permiten identificar de manera gráfica los clientes que interactúan el sistema y el despliegue del intercambio de mensajes.

En esta primera imagen podemos ver como se ejecuta el servidor. Al momento de correr la aplicación debe indicársele la dirección IP de la máquina que actuará como servidor y el puerto a través del cual atenderá las diversas peticiones. También se ve claramente como al momento de ejecutarse el servidor se carga la base de datos que tiene la información relacionada con los canales y suscripciones, tal como se explicó anteriormente.

A terminal window with a white background and a blue border. It shows the command 'MacBook-Pro-dmuneras:socket danielmunerasanchez\$ ruby startServer.rb localhost 2626' being executed. The output consists of three lines: 'INCIANDO DB...', 'adsServer started on port 2626', and a small grey cursor block on the line following the second output line.

```
MacBook-Pro-dmuneras:socket danielmunerasanchez$ ruby startServer.rb localhost 2626
INCIANDO DB...
adsServer started on port 2626
█
```

Esta imagen permite visualizar la interacción de un usuario tipo editor con el sistema. Nuevamente vemos que para establecer la conexión debe indicarse el nickname y role con el cual se va a logear (en este caso son editor), host y port del servidor para poder establecer el socket. Una vez establecida la conexión, el server identificando el tipo de usuario, le pide la contraseña para poder hacer uso de las funciones brindadas por el servidor. En este caso creamos un nuevo mensaje haciendo uso del siguiente mensaje “*create_ad: nuevo, Bienvenido al nuevo canal de publicaciones*” en este, a través del comando create se le indica que se va a crear un nuevo mensaje que pertenecerá al canal nuevo, y cuyo contenido será *Bienvenido...* como respuesta encontramos un mensaje de confirmación del server acerca de la creación del mensaje. Después se realiza la creación de un nuevo mensaje.



```
MacBook-Pro-dmuneras:sockete danielmunerasanchez$ ruby startClient.rb editor editor localhost 2626
Enter password: xxxxxx
Welcome
create_ad: nuevo , Bienvenido al nuevo canal de publicaciones
FROM SERVER >>>> Message successfully created. Channel => nuevo
create_ad: nuevo , Esperamos que tengas un buen inicio en nuestro sistema
FROM SERVER >>>> Message successfully created. Channel => nuevo
```

A continuación veremos la interacción de un usuario tipo cliente con el sistema de servicio de anuncios. Nuevamente para poder establecer la conexión es necesario que se envíe la información del nickname (carlos en este caso), el rol bajo el cual va a interactuar (client), y la dirección IP y puerto del servidor (localhost y 2626 en este caso). A continuación se muestra un mensaje en el que notifica que se está tratando de realizar la conexión con el servidor y posteriormente se recibe el mensaje de bienvenida. Para este caso no es necesario el uso de password. Con el envío del mensaje my_channels, se señala que el cliente desea recuperar la lista de los canales a los cuales está suscrito. El server retorna el mensaje en el cual se muestra la lista de canales, encabezada con el título MY CHANNELS y el canal al cual está suscrito, nuevo. Con el envío del mensaje set_mode: push se notifica al servidor que se desea cambiar el modo de recibir mensaje al modo push, por defecto se encuentra en modo pull. Mientras ejecuta nuevamente el comando my_channels, el servidor le trae los mensajes que se encuentran disponibles en los canales a los cuales se ha suscrito. Es importante ver el funcionamiento de este modo, pues a través de este se puede ver cómo es el server el que se encarga de enviar los mensajes, y en el cliente se establece un hilo para poder recibir estos mensajes intrusivos y luego si continua con el hilo que anteriormente se estaba ejecutando que era el de recuperar los canales a los cuales está suscrito.

```
dmuneras@sistemas:~ ruby... ruby ruby
MacBook-Pro-dmuneras:sockete danielmunerasanchez$ ruby startClient.rb carlos client localhost 2626
Trying to connect to: localhost
FROM SERVER >>>> Welcome carlos
my_channels
FROM SERVER >>>> | MY CHANNELS      |
FROM SERVER >>>> | -----          |
FROM SERVER >>>> => nuevo
set_mode: push
FROM SERVER >>>> Now your mode is: push
my_channels
===== Advice from server =====
Advice from channel nuevo : Bienvenido al canal de publicaciones
Advice from channel nuevo : Esperamos que tengas un buen inicio en nuestro sistema
=====
FROM SERVER >>>> | MY CHANNELS      |
FROM SERVER >>>> | -----          |
FROM SERVER >>>> => nuevo
```

Finalmente esta imagen muestra las interacciones entre un cliente y el servidor de una manera un poco más extensa. En las primeras líneas vemos como un editor se loguea en el sistemas e intenta ejecutar un comando inválido, el server responde enviando una lista con los comandos válidos: ["create_ad:", "rm_ad", "pwd", "ads_list"] a ejecutar, finalmente el editor abandona la aplicación gracias al comando quit. A continuación vemos las interacciones del cliente carlos con el sistema, en ellas a su vez se muestra el mensaje del servidor cuando el cliente ingresa un comando que no es válido para su rol o que es incorrecto y se responde con la lista de comandos válidos: ["set_mode:", "get_ads:", "cannel_list:", "rm_channel:", "add_channel:", "my_channels"]. En seguida vemos el usuario como interacciona con el servidor y le envía algunos comando como son set_mode: push, cannel_list (para listar los canales que se encuentran disponibles), add_channel para suscribirse a un canal, también el uso del comando get_ads: que permite recuperar los mensajes de un canal específico, en cuyo caso el server notifica que debe encontrarse en modo pull para poder hacer uso de esta función.

```
dmuneras@sistemas:~ ruby ... ruby
MacBook-Pro-dmuneras:socket danielmunerasanchez$ ruby startClient.rb
Usage: startClient.rb <nickname> <role> <host> <port>
role = client | editor | admin
MacBook-Pro-dmuneras:socket danielmunerasanchez$ ruby startClient.rb editor editor localhost 2626
Enter password: xxxxxx
Welcome
a
FROM SERVER >>> Invalid command: a
FROM SERVER >>> Valid Commands:["create_ad:", "rm_ad:", "pwd:", "ads_list"]
quit
MacBook-Pro-dmuneras:socket danielmunerasanchez$ ruby startClient.rb carlos client localhost 2626
Trying to connect to: localhost
FROM SERVER >>> Welcome carlos
a
FROM SERVER >>> Invalid command: a
FROM SERVER >>> Valid Commands:["set_mode:", "get_ads:", "channel_list", "rm_channel:", "add_channel:", "my_channels"]
set_mode: push
FROM SERVER >>> Now your mode is: push
channel_list
FROM SERVER >>> Channels: channel,sports,nuevo
add_channel: nuevo
FROM SERVER >>> Channel nuevo added
my_channels
FROM SERVER >>> | MY CHANNELS |
FROM SERVER >>> | ----- |
FROM SERVER >>> => nuevo
get_ads
FROM SERVER >>> Invalid command: get_ads
FROM SERVER >>> Valid Commands:["set_mode:", "get_ads:", "channel_list", "rm_channel:", "add_channel:", "my_channels"]
get_ads: nuevo
FROM SERVER >>> You have to change your mode to pull
set_mode: pull
FROM SERVER >>> Now your mode is: pull
get_ads: nuevo
FROM SERVER >>> Nothing to show
get_ads: channel
FROM SERVER >>> You have to change your mode to pull
█
```

7. RECURSOS UTILIZADOS

1. Medios magnéticos:

- <http://www.glify.com/> . Herramienta para la creación de diagramas
- <https://www6.software.ibm.com/developerworks/education/l-rubysocks/l-rubysocks-a4.pdf> , Documento sobre el funcionamiento de los sockets y uso de funciones como el select para implementar un servidor.
- Documentos disponibles en EAFIT interactiva en la materia Tópicos Especiales en Telemática.

2. Medios físicos:

- Apuntes de clase Tópicos de Telemática
- Consultas de clase realizadas para la materia de Tópicos Especiales en Telemática y Telemática.

8. CONSIDERACIONES DEL DESARROLLO

- Lenguaje de Programación Ruby versión 1.9.2
- SQLite
- Highline , biblioteca utilizada para el ingreso de password por consola.
- Manejador de versiones github donde se encuentra el repositorio del proyecto:
<https://github.com/dmuneras/sockete>