**RETO01**

**Diseño e Implementación de un servicio de Anuncios Distribuido**

**Tópicos Especiales en Telemática**

**Alumnos:**

**Esteban Arango Medina**

**Sebastián Duque Jaramillo**

**Daniel J. Duque Tirado**

**Profesor:**

**Edwin Montoya Múnera**

**Fecha:**

**19 Febrero de 2012**

**UNIVERSIDAD EAFIT**

**MEDELLIN**

**2012**

Contenido

[Análisis y Diseño 3](#_Toc317435373)

[1.1. Arquitectura 3](#_Toc317435374)

[1.2. Modelo/Middleware sistema distribuido 3](#_Toc317435375)

[1.2.1. Definición y Especificaciones del Entorno 3](#_Toc317435376)

[1.2.2. Suposiciones del Entorno 3](#_Toc317435377)

[1.2.3. Vocabulario de Mensajes 3](#_Toc317435378)

[1.2.4. Codificación de Mensajes 3](#_Toc317435379)

[1.2.5. Reglas de Procedimiento 6](#_Toc317435380)

[1.3. Interacción asimétrica y sincrónica 6](#_Toc317435381)

[1.4. Manejo de sesión y estado 6](#_Toc317435382)

[1.5. Modelo de manejo de fallos 6](#_Toc317435383)

[1.6. Modelo de seguridad 7](#_Toc317435384)

[1.7. Niveles de transparencia 7](#_Toc317435385)

[1.7.1. Transparencia de acceso: 7](#_Toc317435386)

[1.7.2. Transparencia de ubicación: 7](#_Toc317435387)

[1.7.3. Transparencia de concurrencia: 7](#_Toc317435388)

[1.7.4. Transparencia frente a fallos 7](#_Toc317435389)

[1.7.5. Transparencia al escalado: 8](#_Toc317435390)

[1.8. Multiusuario 8](#_Toc317435391)

[1.9. Criterios o atributos de calidad 8](#_Toc317435392)

[1.9.1. Concurrencia: 8](#_Toc317435393)

[1.9.2. Extensibilidad: 8](#_Toc317435394)

[1.9.3. Escalabilidad: 8](#_Toc317435395)

[1.10. Detalles de implementación 9](#_Toc317435396)

[Referencias 9](#_Toc317435397)

# Análisis y Diseño

* 1. Arquitectura

Para el servicio de Anuncios Distribuido se implementó una arquitectura Cliente/Servidor en donde se construyó la lógica en el servidor que permite el paso de mensajes entre el publicador (AdFuente) y el suscriptor (AdCliente), sin que ellos se comuniquen directamente. La implementación se realizó en el lenguaje Ruby y se utilizaron sockets para la comunicación.

* 1. Modelo/Middleware sistema distribuido

Utilizamos un protocolo de aplicación para la implementación de este servicio, debido al hecho que el modelo de mensajería MOM requiere de unas entidades muy definidas, y en este caso, el ejercicio plantea una solución flexible, mediante un protocolo que permita dinamismo en las especificaciones, principalmente en las relaciones y comunicaciones entre entidades.

* + 1. Definición y Especificaciones del Entorno

Se desea implementar un protocolo que permita a un conjunto de clientes obtener mensajes de diferentes canales, estos creados por entidades fuentes. El cliente podrá suscribirse para recibir los mensajes automáticamente (PUSH) o realizar una petición al servidor para obtenerlos (PULL).

Características:

1. Orientado a la conexión.
2. Asimétrico.
3. Sincrónico.
4. No sesión, no estado.
5. Sin seguridad (no autenticación ni encriptación).
   * 1. Suposiciones del Entorno

El protocolo se soporta en un protocolo confiable, específicamente sobre TCP, con la implementación de la API de Sockets de Ruby. Se realiza una entrega acertada y validada.

* + 1. Vocabulario de Mensajes

Se agrupa el vocabulario de mensajes en dos categorías: Comunicación AdCliente-servidor y comunicación AdFuente-servidor.{AdCliente\_request, AdFuente\_request, server\_response}

* + 1. Codificación de Mensajes

*AdCliente\_request & (server\_response)*

Una conexión TCP a través de un socket comienza la comunicación, el servidor envía una respuesta preguntando por el nickname del usuario, y después de que el cliente ingrese su nickname, la interacción y las peticiones se soportarán en la siguiente expresión regular:

(?LIST CH|SUBSCRIBE|GETMSGS|UNSUBSCRIBE) ?(?\(.{1,}\))?

El contenido que el cliente envía contiene un match (aceptación) de esta expresión.

Posibilidades:

1. LIST CH: Este mensaje indica al servidor que el cliente solicita los canales existentes.

Respuesta del servidor: El servidor despliega en el cliente todos los canales disponibles con formato:

Actual Channels:

* CHANNEL NAME.
* CHANNEL NAME.
* CHANNEL NAME.

1. SUBSCRIBE (CHANNEL, n): Este mensaje indica al servidor que el cliente solicita suscribirse a uno o varios canales. (Se relaciona con el formato **PUSH** del servidor al cliente)

Respuesta del servidor: El servidor relaciona el cliente con el canal y desplegará alguno de los siguientes outputs dependiendo de la lógica realizada:

“Successfully subscribed to channel -> [channel name]”

“You are already subscribed to channel -> [channel name]”

1. GETMSGS (CHANNEL,..): Este mensaje indica al servidor que el cliente solicita los mensajes de uno o varios canales. (Se relaciona con el formato **PULL** del cliente al servidor).

Respuesta del servidor:

El servidor despliega los mensajes para cada canal con el formato:

CHANNEL NAME:

* “message1”
* “message2”

1. UNSUBSCRIBE (CHANNEL,n): Este mensaje indica al servidor que el cliente solicita anular su suscripción de uno o varios canales.

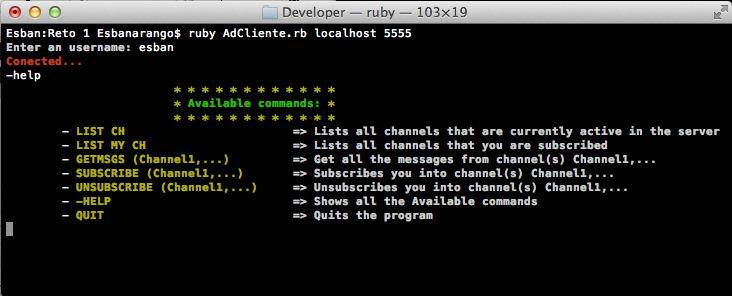
Respuesta del servidor:

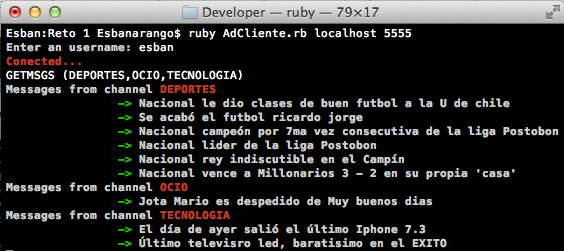
El servidor relaciona el cliente con el canal y desplegará alguno de las siguientes salidas dependiendo de la lógica realizada:

“Successfully unsubscribe from channel -> [channel name]”

“You were not subscribed to channel -> [channel name]”

Ejemplos de ejecución:





*AdFuente\_request & (server\_response):*

Una conexión TCP a través de un socket comienza la comunicación la interacción y las peticiones se soportarán en la siguiente expresión regular:

(?LIST CH|NEWMSG) ?(?\(.{1,}\))? ?(?.\*)?

Posibilidades:

1. LIST CH: Este mensaje indica al servidor que la fuente solicita los canales disponibles.

Respuesta del servidor:

El servidor despliega en la fuente todos los canales disponibles.

1. NEWMSG (CHANNEL,..n) message: Este mensaje indica al servidor que el fuente solicita enviar a n canal(es) un mensaje.

Respuesta del servidor:

El servidor relaciona la fuente con el canal y adiciona el mensaje. El mensaje es desplegado en los clientes suscritos a los n canales parámetros con el siguiente formato:

“ A new message from channel [channelname]

* message1 ”

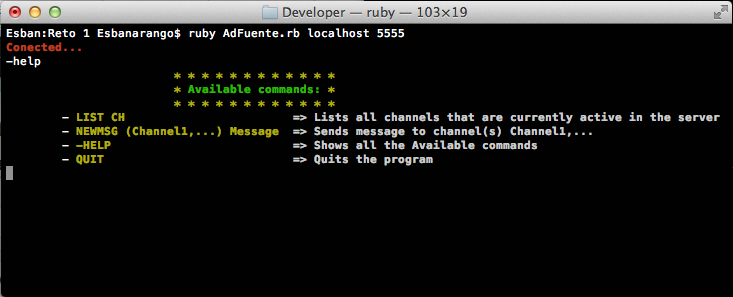
Si el mensaje es enviado con éxito se desplegará en la fuente el mensaje:

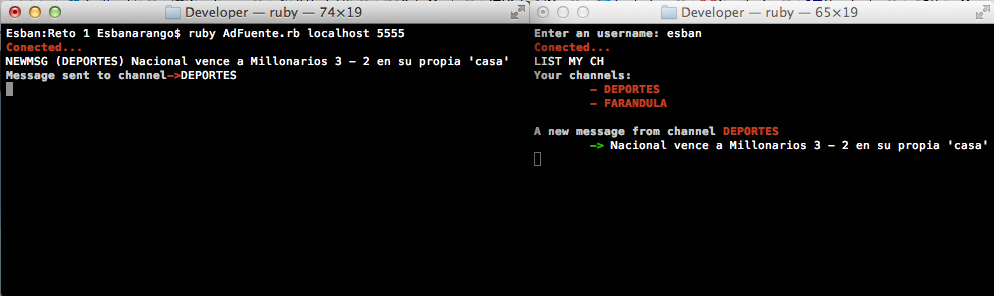
“Message sent successfully!”

Si no existe alguno de los canales se desplegará en la fuente el mensaje:

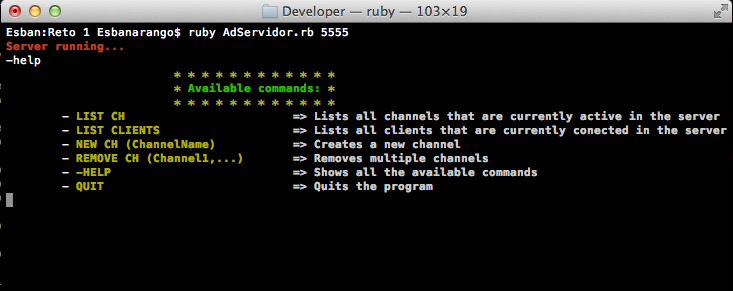
“Channel [channelname] does not exist!”

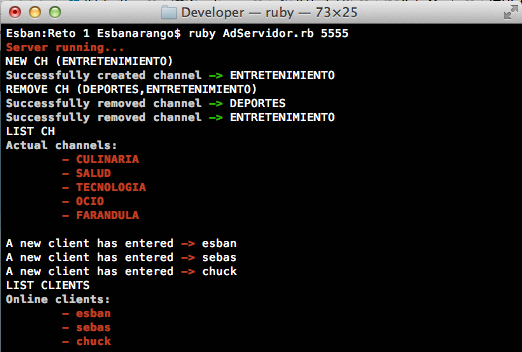
Ejemplos de ejecución:





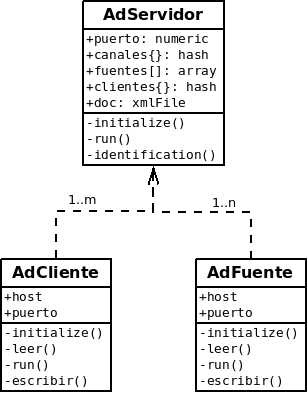
Ejemplo de ejecución para funcionalidad de AdServidor.rb, no se especifica un vocabulario de mensajes debido a que no existe una petición ni una respuesta con ninguna entidad (funcionalidades que no requieren comunicación con AdCliente ni AdFuente. Son internas y gestionan datos e información)





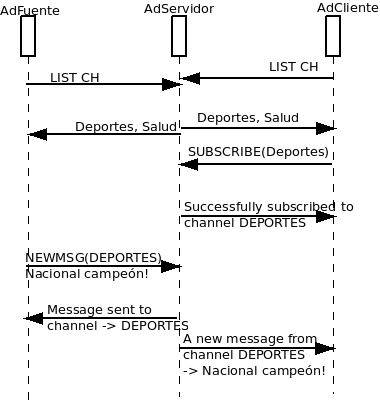
* + 1. Reglas de Procedimiento

Diagrama de clases de alto nivel:



(Otras funcionalidades están presentes en los módulos designModules.rb y mainModules.rb, además del archivo XmlHandler.rb)

Diagrama de secuencias: ejemplo sencillo del flujo de la aplicación con 1 cliente y 1 fuente.



* 1. Interacción asimétrica y sincrónica

Aunque en un principio este tipo de servicios publicador/suscriptor tiende a una interacción asincrónica, al adicionar un servidor ya no se tiene este modelo, por consecuencia las comunicaciones deben pasar por medio del servidor, convirtiendo cada entidad en una entidad bloqueante que espera la finalización de su transacción con el servidor. Con respecto a la interacción asimétrica, será así debido a que el AdCliente y el AdFuente utilizan al servidor como un medio para realizar sus acciones, mientras que el servidor no usa funcionalidades de AdCliente o AdFuente, es decir, para las dos primeras entidades el servidor es fundamental e imprescindible en la arquitectura planteada.

* 1. Manejo de sesión y estado

No se tiene un manejo de estado en el protocolo, ni una sesión de aplicación, aunque existe una implementación que permite el manejo de persistencia. Para cada cliente se utiliza un nombre de registro rápido que es obligatorio y que permite guardar en un archivo .XML los canales a los cuales está suscrito al igual que la relación con el socket. Estos se cargan cada vez que la aplicación se ejecute junto con los canales y mensajes.

* 1. Modelo de manejo de fallos

El servidor contiene la mayoría de la lógica del servicio, y se cuenta con un solo servidor, convirtiendo el manejo de fallos en un tópico delicado. No se tiene planeación de carga ni servidores de soporte, así que si el servidor cae, el servicio caerá. Se podría plantear un segundo servidor con la misma lógica, y crear comunicación entre las fuentes y los clientes con él, realizando rutinas de chequeo del servicio, si este no se encuentra disponible, chequear por el servidor de soporte.

* 1. Modelo de seguridad

La seguridad e integridad en la aplicación se ve estructurada en la diferenciación de clientes y fuentes, además de entre ellos mismos. Los datos son vistos por quien los solicita solamente. Los datos son confiables, ningún usuario ni fuente puede saber a qué canales están suscritos otros usuarios ni cuando solicitan información. La integridad será cuidada, los mensajes creados en las fuentes llegan sin alteración a los clientes, no existe corrupción de datos por ningún cliente, o en el servidor mismo. La disponibilidad es brindada en forma de que un cliente puede realizar PULL de mensajes cuando lo desee, además de recibir al instante nuevos mensajes provenientes de las fuentes por medio de push.

* 1. Niveles de transparencia

La transparencia se identifica especialmente en que aunque el servicio tiene una tendencia publicador/suscriptor, al adicionar el servidor, estos pierden comunicación directa, así, las fuentes y los clientes no se conocen, pero realizan sus comunicaciones indirectas. Para el usuario son transparentes las comunicaciones.

* + 1. Transparencia de acceso:En este servicio los clientes acceden a un recurso local, data.xml, en donde se cargan datos claves de la aplicación, como canales, mensajes y clientes existentes, y además se obtienen recursos que no están guardados, si no que son generados en ejecución, sin que el cliente note la diferencia.
    2. Transparencia de ubicación:Al utilizar una arquitectura cliente/servidor todos los recursos están centralizados. Y al utilizar sockets se debe conocer la ubicación de la entidad con la que se quiere establecer la conexión. Para esto, el cliente debe conocer la IP y el puerto en donde se están gestionando las conexiones. Esto es lo único que un cliente debe saber para acceder a los recursos, no necesita ninguna dirección de directorio.
    3. Transparencia de concurrencia:En la aplicación múltiples usuarios acceden a mensajes publicados en diferentes canales y por distintas fuentes sin ningún problema. Un cliente puede recibir un mensaje de un canal mientras realiza un PULL del mismo, siempre y cuando este suscrito; las acciones son ejecutadas en su orden de petición. Los recursos no son alterados en ningún momento.
    4. Transparencia frente a fallos***:*** Frente a los fallos, por su arquitectura cliente/servidor y la ausencia de herramientas como balanceo de cargas y replicación de servidores, si el servidor cae, el servicio caerá. Pero, si un cliente o una fuente falla, la ejecución del servicio continua eficientemente*.*
    5. Transparencia al escalado:Debido a la diferenciación de entidades, el servicio puede incrementarse sin cambiar la estructura del sistema en general, nuevas funcionalidades pueden ser agregadas en cada uno de los componentes sin reemplazar la lógica de comunicación.
  1. Multiusuario

La aplicación es multiusuario, debido a que para cada usuario nuevo un hilo es creado. Los mensajes son transmitidos a todos aquellos usuarios suscritos (PUSH), además pueden obtener los mensajes mediante una solicitud al servidor (PULL). La implementación de sockets en un protocolo como TCP es un problema cuando existen grandes cantidades de usuarios debido al consumo de los recursos y su orientación a la conexión, pero provee una gran confiablidad en el envío y recibo de mensajes, indiscutiblemente se podrá tener multiusuario.

* 1. Criterios o atributos de calidad
     1. Concurrencia: Como se comentaba anteriormente el servicio no tiene problemas de concurrencia, múltiples usuarios pueden acceder a mensajes de múltiples canales, que a su vez son proveídos por diferentes fuentes. La información no es alterada y el orden de petición indica el orden de ejecución.
     2. Extensibilidad: La implementación permite agregar nuevas funcionalidades sin cambios en la arquitectura. Al identificar entidades y mapearlas a clases, se permite que nuevos procedimientos puedan ser agregados sin cambiar las especificaciones de comunicación y relación soportadas en la arquitectura.
     3. Escalabilidad: La arquitectura cliente/servidor soportada en TCP mediante sockets restringe en medida el número de usuarios que pueden estar interactuando con el servicio en cierto momento. El buen rendimiento de la aplicación dependerá del número de usuarios conectados en el mismo instante. Con los recursos la aplicación no tendrá problemas, podrán ser agregados nuevos recursos y estos podrán ser accedidos. En conclusión, la utilización de sockets y un protocolo como TCP, que en su esencia es orientado a la conexión, no permiten tener un concepto amplio de escalabilidad.
  2. Detalles de implementación

La implementación fue realizada en Ruby. Se crearon las clases AdFuente, AdCliente, AdServidor. Para la persistencia se utiliza un archivo XML ‘data.xml’ y para la manipulación de este utilizamos ‘Nokogiri’ una gema de Ruby que simplifica el manejo de este tipo de archivos. Se tienen otros tres archivos que contienen funcionalidades llamadas desde las clases, estos son designModules.rb, mainModules.rb y XmlHandler.rb.

Para explicación procedimental de instalación y ejecución echar un vistazo a los archivos readme.txt e install.txt.

# Referencias

* TutorialsPoint, 2012. Ruby Socket Programming, Ruby Advanced. En TutorialsPoint “Simply easy learning”. <http://www.tutorialspoint.com/ruby/ruby_socket_programming.htm>. Visitado en febrero de 2012.
* Britt, James. Neurogami Company. TCPServer,Ruby Documentation. En Ruby Documentation Organization. <http://ruby-doc.org/stdlib-1.9.3/libdoc/socket/rdoc/TCPServer.html>. Visitad en febrero de 2012.
* Shteflyuk, Dmytro. 2007. Colorizing console Ruby-Script Output, Ruby&Rails. En Dmytro Shtefluyk’s Home, “thoughts about programming”. <http://kpumuk.info/ruby-on-rails/colorizing-console-ruby-script-output/.Visitado> en Febrero de 2012.