Proyecto Fin de Carrera

Título: Desarrollo de servicios de presencia y mensajería instantánea en redes sociales

Autor: D. Aldo Gordillo Méndez

Tutor: D. Santiago Pavón Gómez

Departamento: Departamento de Ingeniería de Sistemas Telemáticos

Miembros del Tribunal Calificador

Presidente:	D. Juan Quemada Vives	Fdo.
Vocal:	D. Santiago Pavón Gómez	Fdo.
Secretario:	D. David Fernández Cambronero	Fdo.
Suplente:	D. Juan Carlos Yelmo García	Fdo.

Fecha de lectura y defensa en Madrid a .... de .... de 2012

Calificación

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR

DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Desarrollo de servicios de presencia y mensajería instantánea en redes sociales

Aldo Gordillo Méndez

Mayo 2012

Resumen

Uno de los proyectos del grupo “Internet de Nueva Generación” del Departamento de Ingeniería de Sistemas Telemáticos (DIT) es la plataforma *Social Stream*, un motor de *Ruby on Rails* de código abierto para el desarrollo de aplicaciones web con funcionalidades de redes sociales.

En la Internet actual se están extendiendo las aplicaciones en tiempo real, la *mensajería instantánea* ha emergido como una poderosa herramienta de comunicación, permitiendo a personas localizadas en cualquier parte del mundo comunicarse en tiempo real mediante mensajes de texto enviados a través de la red. Esta herramienta se apoya en el *servicio de presencia*, que proporciona información sobre el estado de disponibilidad de las personas, dispositivos y aplicaciones.
Las redes sociales también han jugado un papel importante, ya que en los últimos años muchos usuarios han abandonado el uso de los clientes de mensajería instantánea tradicionales por los chats de las redes sociales, que no son más que clientes de mensajería basados en el propio navegador Web, aunque con el paso del tiempo estos chats van ofreciendo nuevos servicios como videollamadas o juegos.

El objetivo principal de este proyecto es el diseño, implementación, documentación y puesta en producción de una gema llamada *Social Stream Presence* para proporcionar a la plataforma *Social Stream* de servicios de presencia y mensajería instantánea, incluyendo un chat completamente integrado.
Para lograr este objetivo se realizan diversas tareas, empezando por una investigación sobre las diferentes tecnologías, protocolos y arquitecturas para proporcionar servicios de presencia y mensajería instantánea en redes sociales, seguido por el diseño de la arquitectura del servicio, y por otras tareas como la implementación de los mecanismos de autenticación, configuración y ampliación de la funcionalidad del servidor de presencia, implementación de mecanismos de comunicación entre servidores, etc.
Finalmente se lleva a cabo el desarrollo de un chat plenamente integrado en *Social Stream*, accesible tanto desde el navegador como desde clientes de mensajería externos, y se implementan funcionalidades avanzadas como salas de chat, establecimiento de videollamadas y juegos multijugador.

Palabras clave

Presencia, mensajería instantánea, redes sociales, chat, web chat, XMPP, BOSH, WebSockets, ejabberd, Strophe, Ruby on Rails, Social Stream, videollamada, juegos.

Abstract

One project of the "Next Generation Internet" group of the Telematic Systems Engineering Department is the *Social Stream* platform, an open source engine for *Ruby on Rails* for building websites with social networking features.

In today's Internet, there is a growing demand for real-time applications, *instant messaging* emerges as a powerful communication tool, allowing people located anywhere in the world to communicate in real time through text-based messages sent over the Internet.
This tool is supported by the *presence service*, which provides information of the availability status of people, devices and applications.
Social networks have also played an important role, in the last years many users have abandoned the use of traditional instant messaging clients in benefit of social networks chats, which simply are browser-based instant messaging clients, although, as time goes by, these chats are offering new services like video calling or games.

The main objective of this project is the design, implementation, documentation and deployment of a gem called *Social Stream Presence* in order to provide presence and instant messaging services to the *Social Stream* platform, including a fully integrated chat.
In order to achieve this goal various tasks are carried out, starting with an investigation of the different technologies, protocols and architectures necessary to provide presence and instant messaging services in social networks, followed by the service architectural design, and followed by other tasks as the authentication mechanism implementation, configuration and functionality extension of the presence server, communication between servers mechanism implementation, etc.
Finally, we have carried out the development of a fully integrated with *Social Stream* chat, accessible from both the browser and external messaging clients, and we implement advanced features like chat rooms, video calls and multiplayer games.

Keywords

Presence, instant messaging, social networks, chat, web chat, XMPP, BOSH, WebSockets, ejabberd, Strophe, Ruby on Rails, Social Stream, video call, games.

AGRADECIMIENTOS

ANA MENDEZ CADENAS
& FLOREN

OSCAR GORDILLO

TANIA SANCHEZ

SANTIAGO PAVON GABRIEL HUECAS JUAN QUEMADA

ANTONIO TAPIADOR DIEGO CARRERA ALICIA DIEZ EDUARDO CASANOVA

LAURA SANCHEZ CARLOS BEJAR JESUS T. ANA FRANCES JUAN C

MIGUEL GOMEZ ARTURO CAROLINA MARI

MARINA

QUIQUE ISABELINOS FRAN BUGALLO

ALVARO (GAFAS) NESTOR DAVID

MAANU SERGIO ABEL MIWE

SANDRA TONY LALO SEBA VICTOR S.

JORGE VIR GONZ ADRI NANO MICHEL

DR.FLEX ROXY RUDY ALVARO ALONSO VICTOR H.

NICKY SMILE GONZALO MCLAREN CRISPIN

CATA RAFA RAQUEL GABO

PEDRO DORIAN MIGUEL HUANCAYO

ANNA SARA ALBERTO CRESPO

NACHO KIM JONG-IL JOSE & ROCIO IRENE ESTRELLA GORDILLO

ELENA VICTOR GORDILLO IRENA

DANIEL G. PABLO JUANMA ALEX & NURIA WILLY

RASTAS FRANCO FELIX GORDILLO

A MI FAMILIA,

Y GRACIAS A TI, QUE PESE A QUE SE ME HA OLVIDADO ESCRIBIR TU NOMBRE NO LE HAS DADO IMPORTANCIA

MUCHAS GRACIAS !

Índice

1. Introducción.......... 1
2. Estado del arte.......... 9
3. Arquitectura.......... 47
4. XMPP.......... 55
5. Servidor XMPP.......... 65
6. Servidor Web.......... 81
7. Cliente XMPP de Social Stream.......... 97
8. Clientes XMPP.......... 139
9. Despliegue.......... 147
10. Conclusiones.......... 159
11. Trabajo Futuro.......... 165
12. Bibliografía.......... 171
13. Planos.......... 175
14. Pliego de condiciones.......... 181
15. Presupuesto.......... 185

Glosario

AIM	America-On-Line Instant Messenger
AJAX	Asynchronous JavaScript And XML
AOL	America-On-Line
API	Application Programming Interface
BASH	Bourne Again SHell
BOSH	Bidirectional-streams Over Synchronous HTTP
BSD	Berkeley Software Distribution
CSS	Cascading Style Sheets
DIT	Departamento de Ingeniería de sistemas Telemáticos
DOM	Document Object Model
DRY	Don't Repeat Yourself
Emanagement	Ejabberd management
ETSIT	Escuela Técnica Superior de Ingenieros de Telecomunicación
FAQ	Frequently Asked Questions
GNU	GNU is Not Unix
GPL	General Public License
HiPE	High Performance Erlang
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IBM	International Business Machines
ICQ	"I seek you"
IETF	Internet Engineering Task Force
IM	Instant Messaging
IMPS	Instant Messaging and Presence Service
iOS	iPhone Operating System
IP	Internet Protocol
IQ	Info/Query
IRC	Internet Relay Chat
JDBC	Java DataBase Connectivity
JID	Jabber IDentifier
JS	JavaScript
JSON	JavaScript Object Notation
JSONP	JSON with Padding
LDAP	Lightweight Directory Access Protocol
MD5	Message-Digest Algorithm 5
MSN	MicroSoft Network
MSNP	MicroSoft Notification Protocol
MUC	Multi-User Chat
MVC	Modelo Vista Controlador
OAuth	Open Authorization

ODBC	Open DataBase Connectivity
OMA	Open Mobile Alliance
OSCAR	Open System for CommunicAtion in Realtime
OTP	Open Telecom Platform
P2P	Peer-to-Peer
PAM	Pluggable Authentication Modules
PFC	Proyecto Fin de Carrera
REST	REpresentational State Transfer
RFC	Request For Comments
RIA	Rich Internet Applications
RoR	Ruby on Rails
RSA	Rivest, Shamir y Adleman (Algoritmo criptográfico)
RTMP	Real Time Messaging Protocol
s2s	Server to Server
SASL	Simple Authentication and Security Layer
SASS	Syntactically Awesome StyleSheet
SI File Transfer	Stream Initiation File Transfer
SIMPLE	SIP for Instant Messaging and Presence Leveraging Extensions
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SSH	Secure SHell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
UIN	Universal Internet Number
UPM	Universidad Politécnica de Madrid
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTC	Universal Time Coordinated
VoIP	Voice over IP
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WBXML	WAP Binary XML
WebRTC	Web Real Time Communication
WINE	Wine Is Not an Emulator
XHR	XMLHttpRequest
XML	eXtensible Markup Language
XMLRPC	XML Remote Procedure Call
XMPP	eXtensible Messaging and Presence Protocol
XMPP4r	XMPP For Ruby
XSF	XMPP Standards Foundation

1. Introducción

Introducción

El objetivo de este primer capítulo del proyecto fin de carrera titulado *Desarrollo de servicios de presencia y mensajería instantánea en redes sociales,* es presentar al lector el marco en el cual se encuadra el proyecto, así como sus motivaciones , los objetivos que se pretenden alcanzar y la estructura de esta memoria.

La modalidad correspondiente es la de *proyecto clásico de ingeniería*: se ha diseñado, implementado y publicado un sistema software, el cual ha sido puesto en producción en diferentes entidades.

El proyecto se ha desarrollado dentro del Departamento de Ingeniería de Sistemas Telemáticos (DIT) de la Escuela Técnica Superior de Ingenieros de Telecomunicación (ETSIT) de la Universidad Politécnica de Madrid (UPM).

1.1 Contexto

La web 2.0, considerada como la transición de las páginas tradicionales a las aplicaciones web orientadas a los usuarios es ya una realidad, un claro ejemplo de ello son las redes sociales, aplicaciones web basadas en la compartición de información, la colaboración y el diseño centrado en el usuario.
Las redes sociales se han convertido en un auténtico fenómeno social y económico, constituyendo una de las principales herramientas de comunicación, de modo que actualmente ya forman parte del uso cotidiano de la red.
Podemos definir una red social como una estructura social compuesta de personas, organizaciones u otras entidades, las cuales están conectadas por uno o varios tipos de relaciones tales como amistad, parentesco, relaciones profesionales, intereses afines, asistencia a eventos, compartición de conocimientos, etc.
Sin embargo, este fenómeno no es tan novedoso como podría parecer, ya que desde que nacemos estamos inmersos en redes sociales (familia, compañeros de clase, vecinos,…), en cada una de las cuales construimos una identidad y desarrollamos una conducta en relación a los demás, por tanto, lo que ha cambiado con las redes sociales virtuales es el entorno y la manera en que nos relacionamos.
Este hecho se ve reflejado en la tendencia existente de sustitución del uso del teléfono móvil, fijo, email y mensajería instantánea tradicional por el uso de la red social.
Como consecuencia del auge de las redes sociales, el número de estas ha aumentado considerablemente, originando un proceso selección natural, de modo que algunas llegan para quedarse y otras acaban desapareciendo.

En cuanto al futuro, se predicen tendencias de especialización de redes sociales, dando lugar a redes sociales temáticas, interconexión de plataformas y socialización de los espacios.

Uno de los factores más importantes del éxito de las redes sociales, reside en los servicios de presencia y mensajería instantánea vía web.
En la Internet actual se están extendiendo las aplicaciones en tiempo real, donde es necesario enviar la información rápidamente y en el momento preciso.
Para ello es necesario conocer cuando las personas, dispositivos y aplicaciones se encuentran online, al servicio que proporciona esta información se le denomina *servicio de presencia*. Este servicio no se limita a proporcionar información sobre la disponibilidad, sino que también puede proporcionar datos de estado, identidad, ubicación o actividad.
La *mensajería instantánea* es una herramienta muy valiosa para la comunicación en tiempo real entre personas desde cualquier parte del mundo, que permite la comunicación entre dos o más personas basada en texto enviado a través de dispositivos conectados a internet.
Los principales motivos de pertenencia a las redes sociales que aducen los usuarios son mantenerse en contacto con amigos y conocidos y la comunicación gratuita.
Es en la comunicación gratuita donde los servicios de presencia y mensajería instantánea han jugado un papel fundamental que ha provocado que las personas hayan sustituido el uso del teléfono móvil, email y clientes de mensajería instantánea por el uso de las redes sociales. De hecho, en los últimos años muchas personas han abandonado el uso de los programas de mensajería instantánea tradicionales por los chats de las redes sociales, y a medida que las redes sociales incorporan nuevos servicios como llamadas o videoconferencia la migración de usuarios se agudiza aún más, dejando en segundo lugar a las aplicaciones de escritorio.
Prueba de ello es, que a día de hoy, la red social *Facebook* ha desbancado a la aplicación *Messenger* como primera herramienta de comunicación entre el conjunto de la población internauta.
Con el paso del tiempo, las redes sociales se han ido nutriendo de nuevas funcionalidades, tales como chats multiusuario, mensajería offline, videoconferencia, juegos, etc.
En el desarrollo de este proyecto, abarcaremos en mayor o menor detalle cada una de estas funcionalidades, no ciñéndonos exclusivamente a los servicios de presencia y mensajería instantánea, sino aventurándonos en la implementación de funcionalidades más avanzadas.

Con el objetivo de proporcionar una plataforma para la creación de redes sociales personalizadas de forma ágil nace el proyecto *Social Stream*.

Social Stream es una plataforma de código abierto para el desarrollo de aplicaciones web con funciones de redes sociales, proporciona las características básicas de una red social como *Facebook*, *Google+* o *LinkedIn*:

- Gestión de contactos en diferentes círculos (familia, amigos, miembros).
- Perfiles de usuario y grupo.
- Flujos de actividad de los contactos.
- Documentos, imágenes, audio y video compartido.
- Eventos.
- Sistema de mensajes y notificaciones.

Es un motor de *Ruby on Rails,* un entorno de desarrollo web de código abierto que usa el lenguaje de programación *Ruby* y que está diseñado de acuerdo con la estructura MVC. Su diseño de base de datos relacional se basa en el análisis de redes sociales y los flujos de actividad.
Social Stream está dividido en varios módulos, de modo que los desarrolladores pueden personalizar su red social con las funcionalidades necesarias en cada caso.
La distribución actual cuenta con los siguientes módulos:

- *Base*: funcionalidades básicas.
- *Documents*: Soporte para actividades con archivos: imagen, audio y video.
- *Events*: eventos programados con calendario.
- *Linkser*: gestión avanzada de hipervínculos.
- *Presence*: Soporte para chat basado en XMPP.

Los servicios de presencia y mensajería instantánea son proporcionados por el módulo Social Stream Presence, cuya lista completa de características es la siguiente:

- Servicio de presencia.
- Autenticación segura para clientes web y clientes de mensajería externos.
- Chat basado en XMPP.
- Servicio de chat multiusuario.
- Servicio de videollamada.
- Juegos multijugador.
- Soporte para servidor de presencia remoto.
- Soporte para multidominio.
- Instalación y configuración automática.

Todos los módulos de Social Stream son desarrollados en forma de *gema*.
Una gema es un plugin para el entorno Ruby on Rails que permite añadir nuevas funcionalidades o nuevas herramientas para el desarrollo.

1.2 Objetivos

El objetivo principal de este proyecto es el diseño, implementación, documentación y puesta en producción de la gema *Social Stream Presence* para proporcionar a la plataforma Social Stream de servicios de presencia y mensajería instantánea, incluyendo un chat completamente integrado con Social Stream.

A continuación vamos a definir concisamente los objetivos del proyecto:

- Diseño de una arquitectura que permita ofrecer los servicios requeridos.
- Configuración del servidor de presencia elegido, implementando nuevas funcionalidades si fuera necesario.
- Implementación de los mecanismos de comunicación entre servidor web y servidor de presencia.
- Proporcionar un servicio de presencia transparente para la aplicación web.
- Desarrollo de un chat plenamente integrado en Social Stream, accesible tanto desde el navegador como desde clientes de mensajería externos, y que proporcione el servicio de mensajería instantánea.
    - Desarrollar un cliente de chat para el navegador.
    - Proporcionar métodos de autenticación seguros para todos los clientes.
    - Salas de chat (conversaciones multiusuario).
- Estudio sobre la posibilidad de implementación de funcionalidades avanzadas:
    - Videollamadas.
    - Juegos multijugador.
- Generación de documentación de la gema.
- Evaluación de las tecnologías empleadas.

Para lograr estos objetivos se van a seguir una serie de fases en el desarrollo del proyecto:

- Investigación sobre las diferentes tecnologías, protocolos y arquitecturas para proporcionar servicios de presencia y mensajería instantánea en redes sociales.
- Diseño de la arquitectura.
- Implementación de los mecanismos de autenticación en el servidor de presencia.
- Desarrollo de nuevas funcionalidades del servidor de presencia.
- Implementación de los mecanismos de comunicación entre servidor Web y servidor de presencia.
- Desarrollo del cliente de mensajería instantánea basado en el navegador.
- Elaboración de tareas y scripts de instalación y configuración.
- Pruebas de integración.
- Elaboración de documentación: guías, tutoriales y memoria final.

1.3 Estructura del documento

Además de servir como memoria de este PFC, el objetivo de esta memoria es servir como ampliación de la documentación de la gema Social Stream Presence, incluyendo explicaciones más extensas de los aspectos teóricos y justificaciones de las decisiones adoptadas, de forma que sirva como guía para cualquier persona que desee continuar con su desarrollo facilitando la compresión de la arquitectura y su implementación.

La memoria se ha estructurado en tres partes:

Una primera parte que abarca el estado del arte, donde se analizan las diferentes tecnologías existentes, tanto las finalmente utilizadas en el proyecto como las tecnologías alternativas que se podrían haber empleado.
También se estudian brevemente algunos proyectos que ofrecen servicios de presencia y mensajería instantánea.

La segunda parte contiene la documentación relativa al diseño, desarrollo y puesta en producción del proyecto Social Stream Presence, es la parte más extensa y más importante de la memoria, ya que explica detalladamente el proceso seguido para el desarrollo del proyecto.
En primera instancia se describirá la arquitectura diseñada, y en los capítulos posteriores se describirá cada uno de los bloques que componen dicha arquitectura.

En la tercera y última parte, se encuentran las conclusiones del proyecto, que incluyen las conclusiones personales, los problemas encontrados y las lecciones aprendidas.
Finalmente también se incluyen los posibles trabajos futuros, la bibliografía empleada en la elaboración del proyecto, los planos (manual de usuario), el pliego de condiciones y el presupuesto.

2. Estado del arte

Estado del arte

En este capítulo se va a realizar un análisis detallado de las diferentes tecnologías y plataformas existentes que intervienen en el proyecto.
Para cada una de las tecnologías o sistemas empleados, vamos a analizar también las tecnologías o sistemas alternativos que se podrían haber empleado, comentando los puntos fuertes y débiles de cada una de ellas, y justificando nuestra elección.
Finalmente comentaremos brevemente las tecnologías empleadas en algunas aplicaciones que ofrecen servicios de presencia y mensajería instantánea.

2.1 Protocolo

Para la implementación del servicio se hace imprescindible la utilización de un protocolo de presencia y mensajería instantánea.
Existen multitud de protocolos para el intercambio de mensajes, algunos abiertos, entre los cuales podemos destacar *XMPP*, *SIMPLE* (basado en *SIP*) e *IMPS*, y otros propietarios, como los utilizados por *ICQ*, *Y!* (*Yahoo*) y *Windows Live Messenge*r.
Vamos a comentar las características principales de los más importantes.

2.1.1 XMPP

Extensible Messaging and Presence Protocol
Protocolo abierto de mensajería y comunicación de presencia, extensible y basado en XML, originalmente ideado para mensajería instantánea, anteriormente llamado **Jabber.**

Principales ventajas:

- Estándar abierto formalizado por el IETF (RFC 3920, RFC 3921 y *XSF Extensions*).
- Sistema descentralizado (federado).
- Existen multitud de implementaciones de los estándares XMPP para clientes, servidores, componentes y bibliotecas escritos en diferentes lenguajes de programación.
- Empleado en muchos proyectos *open-source.*
- Comunidad activa y abierta.
- Sistema de seguridad robusto (SASL y TLS).
- Permite la interoperabilidad con protocolos propietarios empleando un mecanismo de federación implementado mediante pasarelas.
- Existen un gran número de extensiones.

Inconvenientes:

- Sobrecarga de datos de presencia (elevado porcentaje de transmisiones redundantes). Actualmente se están estudiando posibles modificaciones para aliviar este problema.
- Sin datos binarios, se codifica como un único y extenso documento XML.

A continuación mostramos las características principales del protocolo:

- Conexiones persistentes.
- Con estado.
- Canal de cifrado y autenticación.
- Presencia y lista de contactos.
- Multitud de extensiones que proporcionan nuevas funcionalidades:
  - *Jingle* (Control de sesiones P2P. Servicios de videoconferencia y VoIP)
  - *Multi-User Chat* (Intercambio de mensajes multiusuario)
  - *XMPP Over BOSH*
  - *SOAP Over XMPP*
  - *OAuth Over XMPP*
  - *Service Discovery Extensions*
  - *JSON Encodings for XMPP*
  - *Publish-Subscribe* (Funcionalidades de publicación-subscripción)
  - *User Location* (Localización geográfica)
  - *SI File Transfer* (Transferencia de ficheros)

  En la página de la *XMPP Standards Foundation* (*XSF*) se listan todas las extensiones aprobadas, así como las que se encuentran en estado de evaluación: http://xmpp.org/xmpp-protocols/xmpp-extensions .

Otra de las características que cabe mencionar es que el protocolo XMPP utiliza HTTP para permitir el acceso a los usuarios que se encuentran tras un cortafuegos, ya que es frecuente que estos estén configurados de tal forma que permitan el paso del tráfico TCP dirigido al puerto empleado por el protocolo HTTP, mientras que se bloquea el puerto utilizado por XMPP.

Existen dos formas estandarizadas en las que XMPP puede utilizar HTTP:

- *Polling*: Consiste en emplear las funciones GET y POST de HTTP para enviar los mensajes al servidor de presencia a intervalos de tiempo regulares. Actualmente se encuentra en desuso.
- *Binding*: En este caso el cliente emplea conexiones HTTP prolongadas para recibir los mensajes tan pronto como son enviados.

Las principales aplicaciones del protocolo XMPP se pueden resumir en el siguiente listado:

- Presencia y lista de contactos.
- Mensajería instantánea.
- Redes sociales en tiempo real.
- Juegos.
- Voz y vídeo.
- Geolocalización.
- Servicios web.

XMPP ha sido utilizado por *Facebook* y *Tuenti*, entre otras redes sociales, para la implementación de su chat, así como *Google* también optó por utilizar XMPP para su servicio de mensajería *Google Talk*.
Otra de las aplicaciones más populares, *WhatsApp,* también utiliza este protocolo.
En Junio de 2011, *Skype* lanzó una versión beta que incluía un motor de XMPP, permitiendo la comunicación con los servidores de *Facebook*.
En septiembre de 2011, *Microsoft* anunció una interfaz XMPP para *Windows Live Messenger*, con el objetivo de permitir integrar *Messenger* en las aplicaciones de mensajería instantánea basadas en web, de escritorio o para móviles.

2.1.2 SIMPLE

SIP for Instant Messaging and Presence Leveraging Extensions

Es un protocolo abierto de presencia y mensajería instantánea, extensible (emplea componentes XML al igual que XMPP), basado en *SIP* e impulsado por el IETF.

2.1.2.1 SIP

Session Initiation Protocol

Es un protocolo desarrollado por el IETF con la intención de ser el estándar para la iniciación, modificación y finalización de sesiones interactivas de usuario donde intervienen elementos multimedia como el video, voz, mensajería instantánea, juegos en red y realidad virtual.
Tiene una sintaxis muy semejante a la de los protocolos *HTTP* y *SMTP*, ya que fue diseñado para hacer de la telefonía un servicio más de la red.
Actualmente SIP es empleado como uno de los protocolos de señalización para voz sobre IP.

SIMPLE emplea SIP para resolver los siguientes problemas:

- Registrar la información de presencia y recibir notificaciones de eventos, por ejemplo ante el inicio de sesión de un usuario.
- Envío de mensajes cortos.
- Gestión de sesiones de mensajes en tiempo real entre dos o más participantes.

SIMPLE define dos modos de operación:

- Modo mensaje *(“page-mode” messaging*): el concepto de conversación solo existe en la interfaz del cliente, cada mensaje intercambiado entre ambas entidades no se encuentra asociado con el resto.
- Modo sesión: Al modo que considera cada mensaje intercambiado entre dos o más participantes como parte de una sesión con un principio y un final definidos, se le denomina modo sesión *("session-mode" messaging*).

Ventajas:

- Estándar abierto formalizado por el IETF:
  - RFC 3428: *SIP Extension for Instant Messaging*
  - RFC 4975: *Message Session Relay Protocol*
- Existen dos modos de operación: modo mensaje y modo orientado a sesión.
  Esto hace que el protocolo sea flexible y que cada aplicación pueda escoger que variante implementar basándose en los requerimientos del usuario final.
- Arquitectura distribuida.

Inconvenientes:

- Los datos de presencia se codifican en XML y añaden mucha sobrecarga a los mensajes, de modo que pueden consumir anchos de banda significativos.

Las versiones más modernas de *Windows Messenger* emplean SIMPLE como protocolo de comunicación.

2.1.3 IMPS

OMA Instant Messaging and Presence Service

Es un protocolo abierto de presencia y mensajería instantánea desarrollado por la Open Mobile Alliance (OMA) que pretende ser un estándar para teléfonos móviles.
El consorcio *Wireless Village* desarrolló las primeras especificaciones, cuyo objetivo era crear un estándar de uso de los servicios de presencia y mensajería instantánea para los teléfonos y dispositivos móviles.
Tras la unión de la OMA y la Wireless Village, se crearon las especificaciones conocidas como OMA IMPS, con el objetivo de conseguir un protocolo abierto cuya implementación fuese posible tanto en dispositivos móviles como en el resto de dispositivos.
El protocolo IMPS se especifica dividido en cuatro partes bien diferenciadas: mensajería instantánea, presencia, contenido compartido y grupos.

Ventajas:

- Especialmente diseñado para entornos móviles: tiene un menor consumo de ancho de banda:
    - Es menos verboso que XMPP.
    - Existen estándares como *WBXML*, que permiten que los datos representados en XML sean transmitidos de un manera compacta a través de la redes de telefonía móvil.
- Existencia de clientes multiplataforma (java).

Inconvenientes:

- Escasa difusión
    - En un inicio fue ideado con el fin de que los operadores montaran sus servidores IMPS, pero pocos brindan este servicio.
    - Empleado por pocas aplicaciones.
- Pocos servidores IMPS. (Por ejemplo: *Mobjab* [http://www.mobjab.com])
- Menos funcionalidades que otros protocolos como XMPP.

IMPS está muy extendido, pero no ampliamente comercializado.
Muchos de los terminales a la venta hoy en día cuentan con clientes IMPS, especialmente los de Nokia, Sony Ericsson, Siemens o Motorola.

2.1.4 ICQ

"I seek you"

ICQ no es un protocolo, sino un cliente de mensajería instantánea, que permite, además de chatear con otros usuarios, el envío de archivos y la realización de videoconferencias y charlas de voz.

Los usuarios de la red ICQ son identificados con un número asignado en el registro, conocido como *UIN* ("*Universal Internet Number*" o "número universal de Internet").
El protocolo de comunicaciones utilizado por ICQ es conocido como *OSCAR*.

2.1.4.1 OSCAR

Open System for CommunicAtion in Realtime

Es un protocolo propietario de presencia y mensajería instantánea, empleado en la actualidad, principalmente, por los clientes de mensajería ICQ y AIM (*America-On-Line Instant Messenger*).
A pesar de los esfuerzos de AOL (*America-On-Line*) por impedir la implementación de clientes compatibles, en los últimos años gran parte de las especificaciones han sido determinadas mediante ingeniera inversa y están siendo implementadas por otros clientes. Posteriormente, AOL publicó partes de la documentación del protocolo.

2.1.5 Windows Live Messenger

Anteriormente llamado MSN Messenger y conocido popularmente como *MSN* o *Messenger*, es un cliente de mensajería instantánea creado por Microsoft, y que actualmente funciona en ordenadores con Microsoft Windows, dispositivos móviles con Windows Mobile, Windows Phone o iOS entre otros. Este cliente de mensajería instantánea forma parte del conjunto de servicios en línea denominado Windows Live desde 2005.

Windows Live Messenger utiliza el protocolo propietario MSNP (*Microsoft Notification Protocol*) sobre TCP para conectarse al servicio *.NET Messenger Service*.

Al contrario que en las arquitecturas descentralizadas, existe un servidor central. Microsoft reveló la versión 2 (*MSNP2*) en 1999, no obstante, la versión actual es la 15 (*MSNP15*) y los servidores de *.NET Messenger Service* solo aceptan versiones de protocolo superiores a la 8, por lo que la sintaxis de los nuevos comandos solo se conoce mediante el uso de analizadores de tráfico como *Wireshark*.

2.1.6 La elección: XMPP

En base a todo lo expuesto anteriormente, se ha optado por emplear el protocolo XMPP.

En primera instancia, necesitamos un protocolo abierto y ampliamente documentado. Por tanto, descartando los protocolos propietarios y el protocolo IMPS por su escasa difusión, la decisión queda acotada a elegir entre XMPP y SIMPLE.

Ambos protocolos son muy semejantes y ofrecen un conjunto de funcionalidades de presencia y mensajería instantánea muy parecidas.

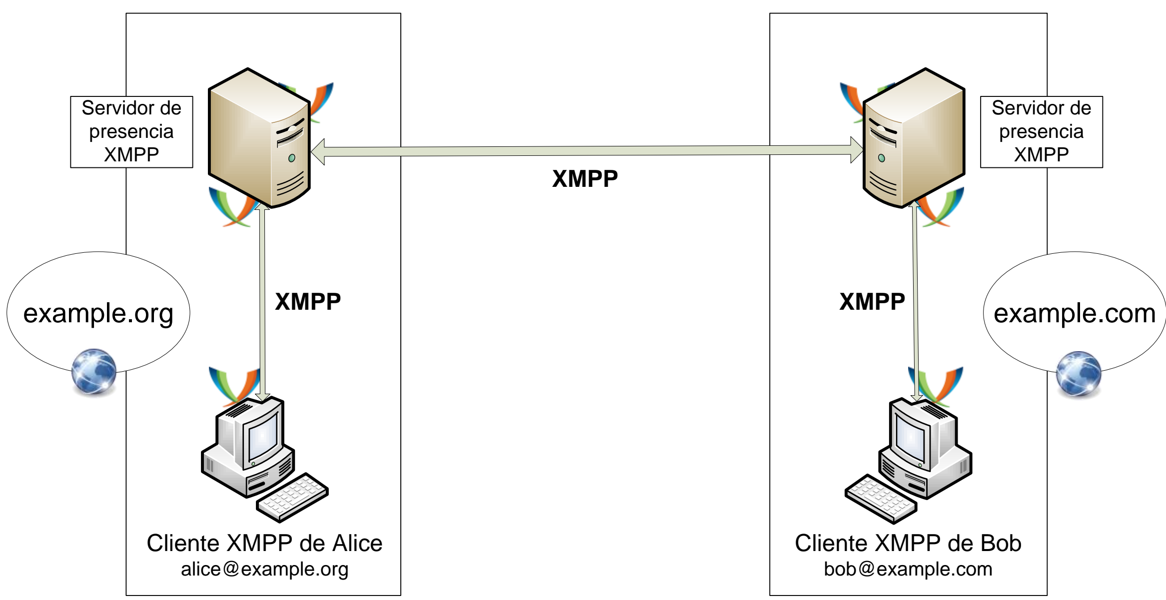
XMPP	SIMPLE
♦ Más ligero que SIMPLE. ♦ Mayor penetración en el mercado. Multitud de implementaciones de clientes, servidores y bibliotecas. ♦ Mayor escalabilidad. ♦ Mayor flexibilidad mediante la adición de extensiones.	♦ Recomendado para aplicaciones de VoIP, puesto que la mayoría de ellas emplean SIP como protocolo de inicio de sesión. ♦ Puede aprovechar todas las características que ya ofrece SIP (autenticación, compresión,…).

Considerando toda la información y como punto final podemos concluir que no parece existir un claro ganador, que tanto XMPP como SIMPLE seguirán teniendo su espacio en el mercado y que ambos protocolos coexistirán, la elección óptima del protocolo dependerá del tipo de aplicación y servicio a implementar.

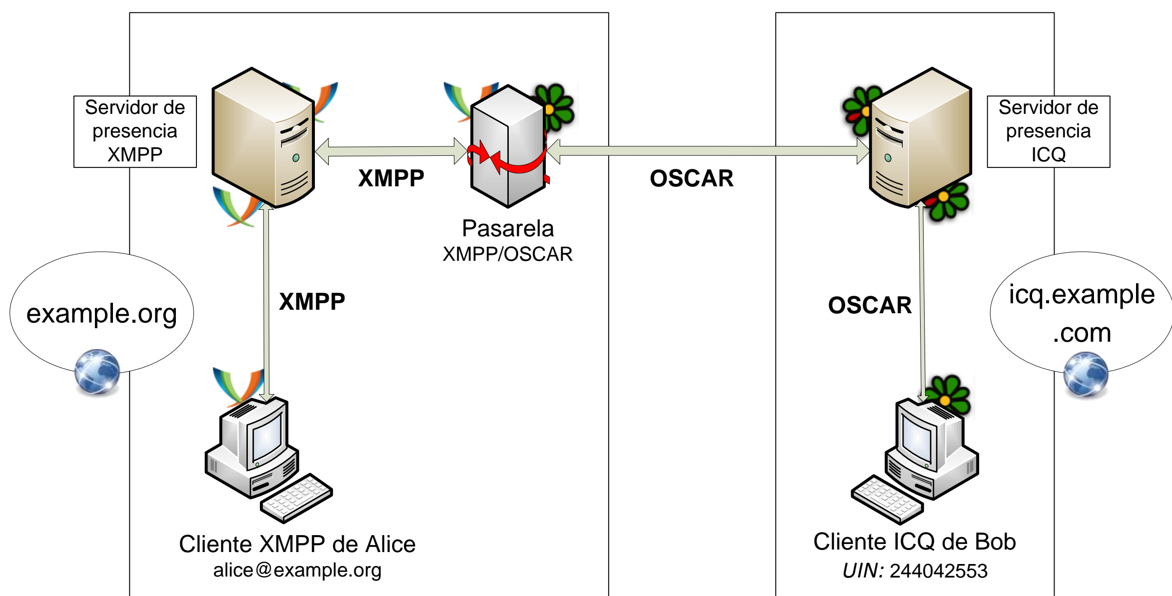
En este proyecto hemos optado por emplear XMPP, principalmente por su gran difusión, a lo largo de esta memoria veremos con detalle algunos servidores, clientes y librerías XMPP. Así mismo, el hecho de que otras aplicaciones como los chats de Facebook y Tuenti o Google Talk empleen XMPP es un factor muy favorable de cara a la posible interconexión futura de nuestro servicio.

2.1.7 Interconexión de servicios de mensajería

Cabe destacar que en el mercado también existen una serie de protocolos propietarios cuyo uso, como ya hemos descrito anteriormente, está bastante extendido.
En la actualidad se está avanzando hacia un escenario de interoperabilidad entre los diferentes protocolos de mensajería instantánea, tanto abiertos como propietarios.
Una de las formas en que se consigue esta interoperabilidad es mediante los mecanismos de federación, entendiéndose la federación como la característica de permitir la conexión de unos servidores con otros.
Por tanto, empleando un mecanismo de federación, es posible permitir que usuarios de diferentes sistemas de mensajería instantánea se comuniquen entre sí, empleando pasarelas para convertir el formato de un protocolo a otro.



Modelo de federación de presencia interdominio



Modelo de federación de presencia interdominio con pasarela XMPP/OSCAR

Para habilitar la interconexión necesitamos no solo una pasarela que traduzca de un protocolo a otro, sino unas especificaciones que determinen las asignaciones o mapeo entre los protocolos.
Por ejemplo, en el borrador titulado "*Interoperability between the Extensible Messaging and Presence Protocol (XMPP) and SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE)" se* propone una especificación sobre los mapeos entre los protocolos XMPP y SIMPLE, concretamente sobre direcciones, mensajes e información de presencia.
A continuación se muestra, a modo de ejemplo, el mapeo de estados de presencia propuesto por Cisco para la interoperabilidad entre los sistemas *Cisco Unified Presence* y *Microsoft Office Communications Server:*

Table 1. Cisco Unified Personal Communicator to Microsoft Office Communicator Presence Mapping

Cisco Unified Personal Communicator User Presence	Microsoft Office Communicator View of Cisco Unified Personal Communicator User Presence
Available	Available
Away	Away
Do not disturb	Busy
Idle	Away
On the phone	Busy
In a meeting	Busy
Invisible(Appear Offline)	Offline
Offline	Offline
Blocked	Offline*

Existen servidores de código abierto que pueden realizar las funciones de traducción.
Por ejemplo, *OpenSIPS*, que es un servidor SIP de código abierto, dispone de un módulo (*PUA_XMPP*) que proporciona una pasarela para presencia y mensajería instantánea entre SIP y XMPP.

2.2 Servidor de presencia

La elección de un servidor de presencia XMPP es una decisión muy importante, sin embargo, no es una decisión sencilla, ya que no existe mucha información publicada más allá de las comparaciones de las diferentes características.
En primera instancia resulta natural plantearse utilizar el servidor más popular, el que se encuentra escrito en el lenguaje de programación más extendido o aquel que ofrezca más facilidades en cuanto a mantenimiento y configuración.
A continuación vamos a realizar un breve recorrido por los servidores XMPP más populares, y posteriormente nos centraremos en los dos más importantes.

2.2.1 ejabberd

Erlang Jabber Daemon

Es un servidor XMPP de presencia y mensajería instantánea de código abierto (licencia GNU GPL) desarrollado por *ProcessOne*, escrito en Erlang y disponible para plataformas Unix y Microsoft Windows entre otras. Es software concurrente y distribuido.

Principales características:

- Base de datos distribuida: *Mnesia*.
    - Soporte para bases de datos externas (*MySQL, PostgreSQL,…*).
- Arquitectura modular
    - Posibilidad de ampliar la funcionalidad base mediante módulos.
- Diseñado para proporcionar:
    - Clustering.
    - Tolerancia a fallos.
    - Alta disponibilidad.
- Alta estabilidad y escalabilidad.
- Incorpora numerosas funcionalidades y extensiones del protocolo XMPP:
    - MUC (Multi-User Chat) Service.
    - Publish-Subscribe Service.
    - Service Discovery.
- Permite emplear autenticación interna, externa (vía script) y mediante otros mecanismos de autenticación: LDAP, PAM y ODBC.
- Soporte para SSL y TLS.
- Soporte para clientes web: servicios HTTP Polling y HTTP Binding (*BOSH*).
- Admite el protocolo de comunicación IRC (*Internet Relay Chat*).
- Mecanismos de interconexión con pasarelas.

2.2.2 Openfire

openfire™ Es un servidor XMPP de presencia y mensajería instantánea escrito en java y desarrollado por *Jive Software*, anteriormente conocido como *Wildfire*, se encuentra disponible tanto bajo la licencia propietaria como bajo la licencia *Apache 2.0*.

Es, junto con ejabberd, el servidor XMPP más popular.

La extensión propietaria de Openfire permite que múltiples instancias de servidor trabajen juntas en un único entorno o cluster. Actualmente esta extensión es de código abierto pero depende del producto comercial *Oracle Coherence*.

Principales características:

- Admite múltiples sistemas de gestión de bases de datos:
    - *Apache Derby*, sistemas con el driver *JDBC* 3 (*Java Database Connectivity*).
- Arquitectura modular
    - Posibilidad de ampliar la funcionalidad base mediante plugins.
- Cluster con múltiples servidores *(extensión propietaria)*.
- Alta escalabilidad.
- Incorpora numerosas funcionalidades y extensiones del protocolo XMPP:
    - MUC (Multi-User Chat) Service.
    - Publish-Subscribe Service.
    - *SI File Transfer*.
- Soporte para SSL y TLS.
- Autenticación vía certificados, Kerberos, LDAP, PAM y Radius.
- Soporte para VoIP, videollamadas y videoconferencias.
- Soporte de clientes de mensajería para navegadores web.
- Adición de pasarelas para la interconexión con otras redes de mensajería instantánea mediante plugins.

2.2.3 Tigase

Es un servidor XMPP de presencia y mensajería instantánea de código abierto (GPL) escrito en java.
Inicialmente el objetivo era desarrollar un servidor XMPP totalmente compatible, incluyendo compatibilidades con especificaciones extraoficiales.
Con el paso del tiempo el proyecto se ha dividido en partes más pequeñas:

- Implementación del servidor.
- Analizador de secuencias XML.
- Suite de pruebas con lenguaje de scripting.

Este servidor está mucho menos extendido que los dos anteriores, y el grado de implementación de las extensiones XMPP es notablemente inferior.

2.2.4 Comparativa: ejabberd y Openfire

Existen otros muchos servidores XMPP además de los mencionados anteriormente, la XSF ofrece una lista bastante completa y actualizada: http://xmpp.org/xmpp-software/servers .
Entre todos los servidores, ejabberd y Openfire, destacan en cuanto a su amplia difusión y grado de implementación de las diferentes funcionalidades del protocolo XMPP. Por tanto, vamos a realizar una breve comparativa entre ambos a raíz de la cual tomaremos una decisión final.

ejabberd	Openfire
♦ Menor consumo de CPU. ♦ Mayor estabilidad. ♦ Clustering. Openfire solamente ofrece esta funcionalidad en su extensión propietaria. ♦ Posibilidad de ejecutar comandos para gestionar el servidor vía consola, XMLRPC o REST. ♦ Contribuciones: numerosas nuevas funcionalidades disponibles mediante módulos de terceros. Openfire también tiene una comunidad activa y ofrece numerosas funcionalidades.	♦ Escrito en Java. Podemos considerar la amplia extensión del lenguaje Java como una ventaja, no obstante, pese a ser un lenguaje menos conocido, Erlang está bien documentado y es fácil de aprender. ♦ Curva de aprendizaje más suave. Más fácil de instalar y gestionar. La configuración de ejabberd es más compleja. ♦ Utilización de pasarelas de forma integrada. ejabberd no incluye pasarelas, pero proporciona mecanismos de interconexión.

Mucho se ha discutido y el debate parece no tener fin, existen numerosas discrepancias entre las valoraciones de los partidarios de ejabberd y las valoraciones de los partidarios de Openfire, sin embargo, podemos sacar algunas conclusiones en claro.

El servidor *ejabberd* es considerado, en general, como la mejor opción para grandes desarrollos dada su alta escalabilidad, estabilidad y gran rendimiento para cargas de trabajo intensivas.

Openfire es una buena opción si queremos disponer de un servidor de mensajería con poca carga de trabajo: uso personal o para pequeñas empresas, dada su facilidad para instalar, configurar y administrar.

Es frecuente la elección de Openfire como servidor con el único objetivo de evitar el lenguaje de programación Erlang, obviando el resto de factores en la decisión.
Esto no debería ser un factor crítico, puesto que Erlang, del que hablaremos posteriormente con más detalle, está bien documentado y es relativamente fácil de aprender.
Si bien es cierto que la curva de aprendizaje de ejabberd es más abrupta y que su configuración es más compleja que la de Openfire, no debemos descartarlo como opción, ya que estos inconvenientes se tratan de un pequeño precio a pagar por tener un sistema más robusto y escalable.
Otra funcionalidad muy interesante que ofrece ejabberd es el *clustering*, es decir, permite configurar múltiples servidores para trabajar conjuntamente de forma que, en muchos aspectos, pueden ser vistos como un único servidor.
Por tanto, el clustering permite que un dominio XMPP sea servido por uno más nodos de ejabberd, pudiendo estar funcionando estos nodos en diferentes máquinas conectadas a través de la red.

En base a todo lo expuesto, y dado que nuestro sistema se va a emplear para proporcionar servicios de presencia y mensajería instantánea en redes sociales, hemos considerado como factor clave la escalabilidad, por lo que hemos elegido ejabberd como servidor XMPP.

2.3 Medio de transporte.

La conexión del chat de la red social con el servidor de presencia se debe realizar mediante un cliente de mensajería instantánea, que deberá, necesariamente, funcionar en el navegador web.
A este tipo de clientes se les llama clientes de mensajería instantánea basados en web (en inglés *web-based IM clients*), o clientes XMPP basados en web (*web-based XMPP clients*) si particularizamos para aquellos que emplean el protocolo XMPP.
Posteriormente abordaremos el tema de los clientes y librerías XMPP para el navegador, en esta sección vamos a analizar las diferentes tecnologías y protocolos existentes que permiten el transporte del protocolo XMPP entre el navegador web y el servidor.

El protocolo empleado en la World Wide Web es HTTP, este protocolo define la sintaxis y semántica que utilizan los elementos software de la arquitectura web (clientes, servidores y proxys) para comunicarse.
Es un protocolo de petición-respuesta sobre conexiones TCP: el cliente inicia una conexión TCP, envía un mensaje HTTP de petición, el servidor devuelve un mensaje HTTP de respuesta y se cierra la conexión TCP. Además es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores.

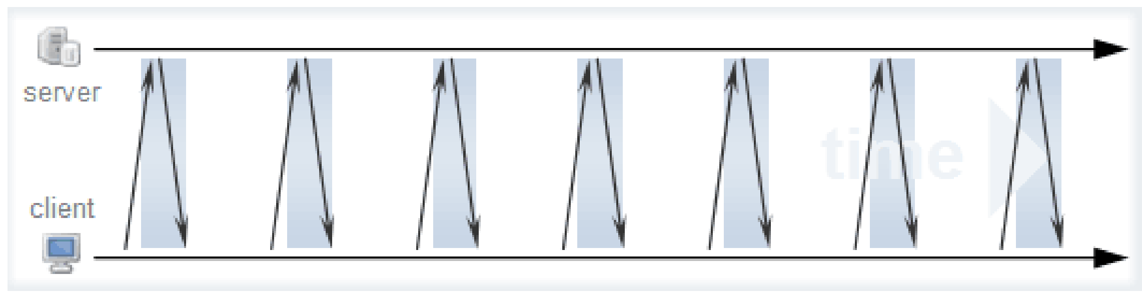
En contraposición con el modelo de petición-respuesta, XMPP necesita una conexión persistente, lo cual implica que para transportar XMPP sobre HTTP vamos a necesitar usar tecnologías adicionales.

Por otro lado TCP, que da soporte al protocolo de aplicación HTTP, es un protocolo de comunicación del nivel de transporte, fiable y orientado a conexión, es decir, que si proporciona conexiones persistentes.

Por tanto, existen principalmente dos formas de transporte, la primera consiste en simular conexiones persistentes sobre HTTP empleando distintas tecnologías como Comet o BOSH, y la segunda consiste en enviar la información empleando TCP mediante WebSockets.

2.3.1 Polling

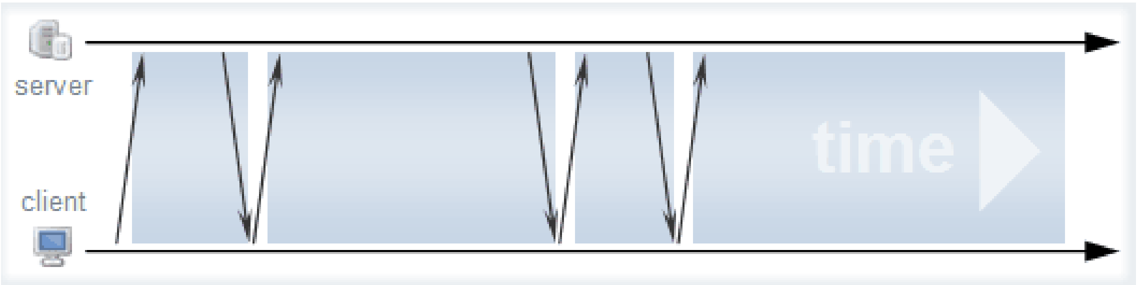
Esta técnica o estrategia de conexión fue el primer intento para conseguir que el navegador enviase información en tiempo real.
La técnica de polling consiste en realizar peticiones HTTP al servidor a intervalos regulares de tiempo con el objetivo de verificar actualizaciones.
Sin embargo, los eventos no son predecibles, por lo que inevitablemente se realizan peticiones innecesarias provocando como resultado que muchas conexiones sean abiertas y cerradas sin necesidad en situaciones de baja tasa de mensajes.
Los principales inconvenientes de esta técnica son el alto consumo de ancho de banda, su baja escalabilidad y el retardo, determinado por el intervalo entre peticiones.



Polling

2.3.2 Long Polling

Es una variación de la técnica tradicional de polling y permite emular el envío de información desde un servidor a un cliente. Con long polling, el cliente solicita información al servidor de una manera muy similar a como lo hacía con la técnica de polling, sin embargo, si el servidor no tiene información disponible para el cliente, en vez de enviar una respuesta vacía, el servidor mantiene la petición y espera a que alguna información esté disponible. Una vez la información está disponible (o después de un tiempo establecido), se envía una respuesta completa al cliente. Entonces el cliente realizará, normalmente inmediatamente después, una nueva petición de información al servidor, para que éste siempre tenga una petición en espera, que puede ser usada para enviar información como respuesta a un evento.
La principal ventaja de esta técnica frente a la tradicional es que es mucho más escalable. Long polling no es en sí misma una tecnología de push, pero puede ser usada bajo circunstancias donde un push real no es posible.



Long Polling

2.3.3 Comet

Es un término que describe un modelo de aplicación web en el que una petición HTTP de larga duración permite a un servidor web enviar datos a un navegador mediante *tecnología push*, sin que el navegador lo solicite explícitamente.

¿En qué consisten las tecnologías Push y Pull?

Push o *server push*, describe un estilo de comunicaciones de internet donde la petición de una transacción se origina en el servidor.

Por el contrario en la tecnología *pull*, la petición es originada en el cliente.

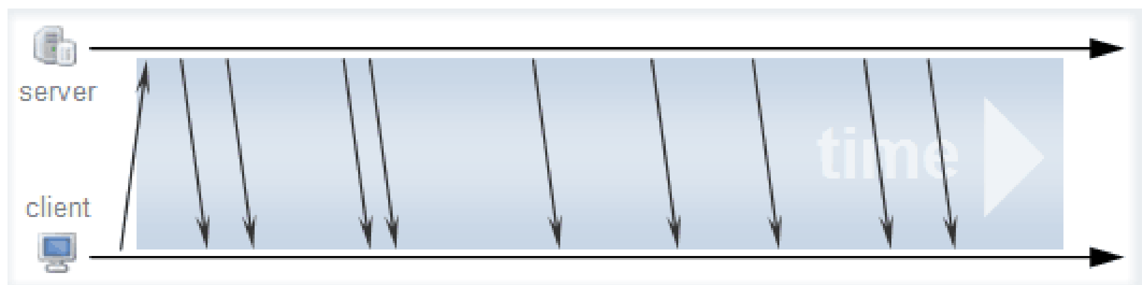
Comet es un término que engloba múltiples técnicas para conseguir esta interacción, todas estas técnicas se basan en funcionalidades incluidas por defecto en los navegadores, como *JavaScript*, en lugar de emplear plugins no disponibles por defecto.
El enfoque de Comet difiere del modelo original de la web, donde un navegador solicita una página web completa de una sola vez.
Estas técnicas ya se usaban en desarrollo web antes de que la palabra *Comet* fuera acuñada como termino para englobarlas a todas.
Comet es típicamente referido por diversos nombres como *Ajax Push*, *Reverse Ajax*, *HTTP Streaming* o *HTTP server push* entre otros.

Las aplicaciones basadas en Comet tratan de eliminar las limitaciones del modelo original de la web y el polling tradicional, ofreciendo interacción en tiempo real, empleando conexiones HTTP de larga duración entre el servidor y el cliente.
Dado que los navegadores y proxys no están diseñados para tratar eventos del servidor, se han desarrollado múltiples técnicas para conseguirlo, cada una de las cuales tiene sus ventajas e inconvenientes.
El mayor obstáculo radica en la especificación de HTTP 1.1, que establece que un navegador no debería tener más de dos conexiones simultáneas con un servidor, por lo tanto, mantener una conexión abierta para los eventos en tiempo real tiene un impacto negativo en la usabilidad del navegador, puesto que este no podrá realizar una nueva petición mientras esté esperando los resultados de una petición previa.
Esto se puede solucionar mediante la creación de un nombre de host distinto para la información en tiempo real, que es un alias para el mismo servidor físico.

Las diferentes técnicas englobadas por Comet se pueden dividir en dos categorías principales: *streaming* y *ajax con long polling*.

2.3.3.1 Streaming

Las aplicaciones que emplean *Streaming Comet* abren una única conexión persistente desde el navegador al servidor para todos los eventos.
Estos eventos se manejan e interpretan desde el cliente cada vez que el servidor envía un nuevo evento, sin que ninguna de las partes cierre la conexión.



Comet (HTTP Streaming)

Existen diversas técnicas específicas para llevar a cabo HTTP Streaming, entre las cuales podemos destacar:

Hidden iframe: Se basa en la utilización de un iframe oculto, que se envía como bloque fragmentado y que se declara implícitamente como de longitud infinita.
Cuando ocurre un evento, el iframe se va rellenando gradualmente con el código JavaScript a ejecutar en el navegador.

XMLHttpRequest (XHR): Consiste en emplear XHR, la herramienta principal empleada por las aplicaciones ajax para la comunicación entre navegador y servidor, para el envío de notificaciones de eventos del servidor al navegador.

¿Qué es XHR?

XHR es una interfaz empleada para realizar peticiones HTTP a servidores web, se pueden enviar datos empleando cualquier codificación basada en texto: texto plano, XML, JSON, HTML o codificaciones particulares específicas.

JSON (JavaScript Object Notation)

Es un formato de datos muy simple y ligero.
Muy utilizado en JavaScript debido a la extrema facilidad de análisis sintáctico.

Un inconveniente a destacar de esta técnica es que solo se puede realizar en navegadores basados en el motor de renderizado *Gecko*, como *Firefox* o *IceWeasel*.

2.3.3.2 Ajax con long polling

Ninguna de las técnicas de streaming funciona en todos los navegadores modernos sin efectos secundarios negativos.
Debido a ello, muchas aplicaciones emplean *ajax long polling*, que es más fácil de implementar en el navegador y funciona, como mínimo, en cualquier navegador que soporte XHR.
Este método consiste, simplemente, en una implementación en ajax de la técnica long polling explicada anteriormente.

¿Qué es Ajax?

***A**synchronous **J**avaScript **A**nd **X**ML*, es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (*Rich Internet Applications*).
Estas aplicaciones se ejecutan en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano.
De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que implica un aumento de la interactividad, velocidad y usabilidad en las aplicaciones.

Las llamadas a ajax se efectúan en lenguaje *JavaScript* y el acceso a los datos se realiza mediante *XMLHttpRequest*.

Existen diversas implementaciones en ajax de la técnica de long polling, entre las cuales podemos destacar:

XMLHttpRequest long polling: Emplea la interfaz XHR, lo cual permite que los datos enviados por el servidor en respuesta a un evento estén codificados en XML o JSON.

Script tag long polling: Otra implementación de esta técnica puede ser llevada a cabo mediante la creación dinámica de elementos *script*, que se agregan a la cabecera del documento, estableciendo su atributo origen (*src*) a la ubicación del servidor Comet.
Cuando el servidor quiere notificar al cliente de los nuevos datos envía una cadena de JavaScript (o *JSONP*) que invoca un método existente en el cliente.
Cada vez que se completa la petición de script, el navegador abre una nueva, al igual que ocurría en las otras implementaciones de long polling.
Este método tiene la ventaja de funcionar en todos los navegadores (*cross-browser*), mientras que sigue permitiendo la implementación multidominio (*cross-domain*).
El mayor inconveniente de esta implementación es la falta de control sobre la conexión, aunque funciona en todos los navegadores, ciertos *callbacks* no están disponibles en unos u otros navegadores.

2.3.4 BOSH

Bidirectional-streams Over Synchronous HTTP

Es una tecnología para la comunicación en ambos sentidos a través de HTTP empleando flujos bidireccionales sobre HTTP síncrono.
Emula muchas de las primitivas de transporte que son familiares del protocolo TCP.
BOSH está diseñado para transportar cualquier tipo de datos de forma eficiente y con el mínimo retardo en ambas direcciones.

¿Cómo funciona BOSH?

Se establece una conexión HTTP de larga duración (un minuto o dos) con el servidor XMPP.
Si llegan nuevos datos durante este tiempo, la petición HTTP devuelve los datos y se cierra, de lo contrario, simplemente expira.
De cualquier modo, una vez que se cierra una petición, otra se restablece.
Aunque el resultado es una serie de conexiones repetidas a un servidor, es un orden de magnitud más eficiente que la técnica de *polling*, en particular debido a que las conexiones se realizan a un servidor especializado en vez de directamente a una aplicación web.

Por tanto, BOSH está basado en una técnica de *long polling.*

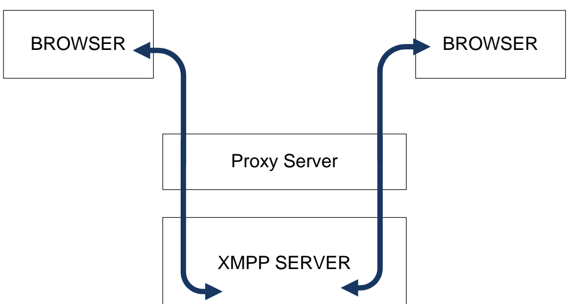
Para aplicaciones que requieren comunicaciones *push* y *pull* presenta una mayor eficiencia en cuanto al consumo de ancho de banda y una mayor sensibilidad que la mayoría de protocolos de transporte basados en HTTP bidireccional y que la mayoría de técnicas basadas en *polling*.
BOSH consigue esta eficiencia y baja latencia evitando el uso de *HTTP Polling,* y además lo hace sin recurrir a respuestas HTTP fragmentadas.
Con frecuencia esta técnica también es llamada *HTTP Binding*.
Hasta la fecha, BOSH ha sido utilizado principalmente como medio transporte para el intercambio de tráfico entre clientes y servidores XMPP, por ejemplo, para facilitar las conexiones de los clientes web y de clientes móviles en redes intermitentes.
Sin embargo, BOSH no está ligado únicamente a XMPP y puede ser utilizado para transportar otros tipos de tráfico.

Con el objetivo de construir aplicaciones XMPP dentro de los navegadores web, la XSF ha creado dos especificaciones en las que se define BOSH:

- XEP-0124: *Bidirectional-streams Over Synchronous HTTP*
  Esta especificación define el protocolo empleado por BOSH.

- XEP-0206: *XMPP Over BOSH*
  Esta especificación define como la tecnología BOSH puede ser empleada para transportar mensajes XMPP.
  El resultado es un enlace HTTP para las comunicaciones XMPP útil en aquellas situaciones en las que un dispositivo o cliente es incapaz de mantener una conexión TCP de larga duración con un servidor XMPP.

Por tanto, podemos decir que XMPP sobre BOSH permite al cliente web mantener la comunicación con el servidor XMPP a través de una conexión nativa.
Si empleamos BOSH para la comunicación entre cliente y servidor, es necesario un proxy que redirija los mensajes XMPP al servidor.



El cliente se conecta a través de una URL estándar mediante HTTP utilizando el puerto 80, lo cual le permite operar detrás de un firewall, posteriormente esta conexión es redirigida por el proxy a una URL HTTP en un puerto diferente, donde opera el servidor XMPP.

Si queremos emplear BOSH en nuestro servicio debemos utilizar un gestor de conexión BOSH (*BOSH connection manager*), del cual existen dos tipos: incorporado e independiente.

2.3.4.1 Gestores de conexión incorporados

Actualmente muchos servidores vienen con soporte de BOSH incorporado, concretamente, los tres servidores XMPP analizados anteriormente (ejabberd, Openfire y Tigase) incorporan soporte para BOSH.
Dicho soporte puede ser activado en la configuración del servidor y permite realizar conexiones a los clientes BOSH.

Gestores de conexión BOSH incorporados	
Ventajas	Desventajas
♦ No hay que preocuparse de la compatibilidad con el servidor. ♦ Más eficiente: al estar incorporado puede hablar de forma nativa con el servidor empleando sus protocolos de enrutamiento internos o llamadas directas al API. ♦ Muy fácil de configurar.	♦ Solo soporta conexiones locales, no se permite la conexión con otros servidores, por tanto, no puede ser empleado para aplicaciones web federadas. ♦ No todos los servidores XMPP tienen un gestor de conexión BOSH incorporado.

2.3.4.2 Gestores de conexión independientes

Existen varios gestores de conexiones BOSH que no están ligados a ninguna implementación de servidor XMPP en particular, por ejemplo: *Punjab*, *Araneo* o *JabberHTTPBind*.

La característica más importante de estos gestores es que, al contrario que los gestores incorporados, permiten la conexión a cualquier servidor XMPP, por tanto constituyen un elemento imprescindible para el desarrollo de aplicaciones web XMPP federadas.

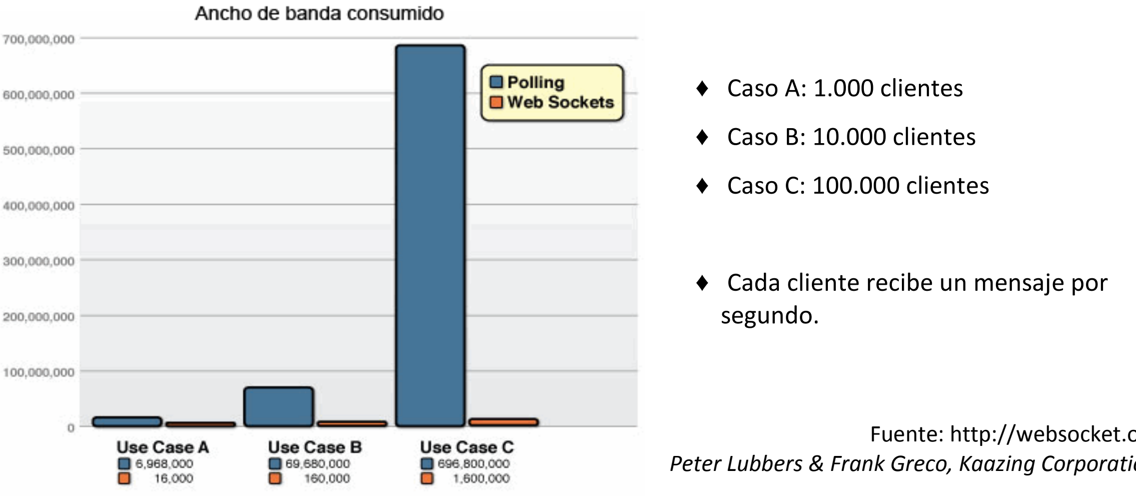
Gestores de conexión BOSH independientes	
Ventajas	Desventajas
♦ No están ligados a ningún servidor XMPP en particular. ♦ Permiten tanto conexiones locales como conexiones a otros servidores XMPP.	♦ Pueden surgir problemas de compatibilidad con el servidor. ♦ Menos eficientes. ♦ Más difíciles de configurar.

2.3.5 WebSockets

Es una tecnología web que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse en cualquier aplicación cliente/servidor.
La API de WebSocket está siendo estandarizada por el W3C, y el protocolo WebSocket, a su vez, está siendo estandarizado por el IETF (*RFC 6455*).
La especificación, desarrollada como parte de HTML5, define una interfaz JavaScript (*WebSocket API*) que permite a las aplicaciones web utilizar el protocolo WebSocket para la comunicación bidireccional con un host remoto.
El protocolo WebSocket es un protocolo independiente basado en TCP, su única relación con HTTP es que su negociación (*handshake)* es interpretada por los servidores HTTP como una petición de actualización.
El protocolo WebSocket emplea, por defecto, el puerto 80 para las conexiones habituales y el puerto 443 para las conexiones a través de TLS.
El empleo del puerto 80 permite evitar problemas con los cortafuegos, además, a costa de una pequeña sobrecarga del protocolo, se puede proporcionar una funcionalidad similar a la apertura de varias conexiones en distintos puertos pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP.
No sigue el modelo petición-respuesta, permite iniciar mensajes desde el servidor.
El objetivo de esta tecnología es proporcionar a las aplicaciones basadas en el navegador, un mecanismo de comunicación bidireccional en tiempo real con los servidores que no dependa de la apertura de múltiples conexiones HTTP.
Por lo tanto, los WebSockets no solo ofrecen una importante alternativa a todas las tecnologías de comunicación expuestas anteriormente, sino que representan el próximo paso evolutivo en la comunicación web, especialmente para las aplicaciones web en tiempo real dirigidas por eventos.

Los WebSockets ofrecen una reducción drástica del tráfico innecesario de red y de la latencia en comparación con las técnicas convencionales: polling, long polling y streaming.
Estas técnicas emplean peticiones y respuestas HTTP, cuyas cabeceras contienen una gran cantidad de datos adicionales innecesarios, lo cual introduce latencia y provoca un mayor consumo de ancho de banda.
Para simular una comunicación full-duplex sobre una conexión HTTP, muchas soluciones actuales emplean dos conexiones, una para cada sentido.
El mantenimiento y la coordinación de estas dos conexiones introduce complejidad y una sobrecarga significativa en cuanto al consumo de recursos.

A modo de ejemplo, se ofrece una comparativa del ancho de banda consumido por una aplicación concreta en tres situaciones diferentes, empleando polling y WebSockets.

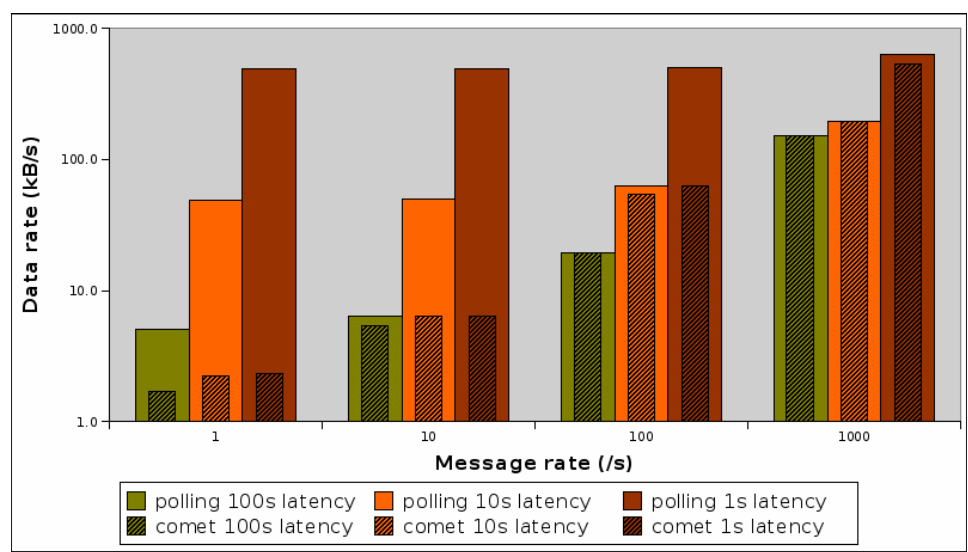


- Caso A: 1.000 clientes
- Caso B: 10.000 clientes
- Caso C: 100.000 clientes

- Cada cliente recibe un mensaje por segundo.

Fuente: http://websocket.org
Peter Lubbers & Frank Greco, Kaazing Corporation

No obstante, también debemos tener en cuenta que el ancho de banda consumido empleando la técnica de long polling es menor que el consumido empleando polling.



Fuente: Comet is Always Better Than Polling by Greg Wilkins

Sin embargo, a pesar de todas las ventajas de los WebSockets, cabe destacar que se trata de una tecnología muy poco madura, que aún se encuentra en proceso de estandarización, el último borrador del protocolo *(The WebSocket protocol),* es el *draft-ietf-hybi-thewebsocketprotocol-17,* publicado *el* 30 de Septiembre de 2011, así como la última especificación definida en el *RFC 6455* data de diciembre de 2011.
Este hecho se ve reflejado en el débil soporte que tienen los distintos navegadores de esta tecnología.
A modo de resumen en cuanto al estado de implementación de los WebSockets en los navegadores se ofrece la siguiente tabla, en la que se muestra, para cada navegador, el soporte de WebSockets de su versión actual y el soporte de las versiones futuras:

Estado de la implementación de WebSockets en los principales navegadores

Soportado	Parcialmente soportado	No soportado

Navegadores	Actualmente	Futuro cercano
Internet Explorer	9.0	10.0
Mozilla Firefox	11.0	12.0
Google Chrome	17.0	18.0
Safari	5.1	*6.0*
Opera	11.6	12.0
iOS Safari	5.0	
Opera Mini	6.0	
Opera Mobile	12.0	
Android Browser	4.0	

Fuente: When can I use... (Febrero 2012)

Los grados de implementación varían notablemente en función del navegador, de hecho, los únicos navegadores que soportan la última especificación (RFC 6455) del protocolo WebSocket son Chrome 16 *(Diciembre 2011)*, Firefox 11 *(Marzo 2012)* e Internet Explorer 10 (*Sin* fecha de lanzamiento confirmada).

Con el objetivo de ilustrar esta idea se muestran algunos fragmentos de los resultados de una batería de pruebas, realizada sobre diferentes navegadores, acerca de los diversos aspectos de los WebSockets.

3 Reserved Bits	AutobahnClient/0.4.11		Chrome/19.0.1054.0		Firefox/13.0a1-20120227		Safari/534.52.7	
Case 3.1	Pass	1002	Pass	None	Pass	None	Pass	None
Case 3.2	Pass	1002	Pass	None	Non-Strict	None	Pass	None
Case 3.3	Pass	1002	Pass	None	Non-Strict	None	Pass	None
Case 3.4	Pass	1002	Fail	Fail	Pass	None	Fail	Fail
Case 3.5	Pass	1002	Pass	None	Pass	None	Pass	None
Case 3.6	Pass	1002	Pass	None	Pass	None	Pass	None
Case 3.7	Pass	1002	Pass	None	Pass	None	Pass	None

9 Limits/Performance	AutobahnClient/0.4.11		Chrome/19.0.1054.0		Firefox/13.0a1-20120227		Safari/534.52.7	
9.6 Binary Text Message (fixed size, increasing chop size)								
Case 9.6.1	Missing		Missing		Missing		Missing	
Case 9.6.2	Missing		Missing		Missing		Missing	
Case 9.6.3	Pass 4777 ms	1000	Pass 4544 ms	None	Pass 4537 ms	1000	Pass 4544 ms	None
Case 9.6.4	Pass 2718 ms	1000	Pass 2487 ms	None	Pass 2486 ms	1000	Pass 2488 ms	None
Case 9.6.5	Pass 1688 ms	1000	Pass 1469 ms	None	Pass 1461 ms	1000	Pass 1463 ms	None
Case 9.6.6	Pass 1186 ms	1000	Pass 960 ms	None	Pass 947 ms	1000	Pass 955 ms	None

Fuente: WebSocket Implementation Test Report
Informe generado el 28 Febrero 2012

El estado parcialmente soportado quiere decir que la implementación de WebSockets emplea una versión obsoleta del protocolo y/o que se encuentra desactivada por defecto debido a cuestiones de seguridad.

Porcentaje de usuarios con soporte de WebSockets	
Soporte total:	43.26%
Soporte parcial	10.39%
Total	53.65%

Fuente: When can I use... (Febrero 2012)

Observando las estadísticas de uso, encontramos el mayor inconveniente de los WebSockets, y es que, en la actualidad, una aplicación web en tiempo real que emplee únicamente WebSockets como tecnología de comunicación solo funcionará correctamente en la mitad de los clientes.

Si bien es una tecnología poco madura, está destinada a convertirse en el estándar para la comunicación web del futuro, por lo que ya existen algunos proyectos que están empezando a trabajar con WebSockets.

Socket.IO es una librería JavaScript que tiene como objetivo posibilitar las aplicaciones web en tiempo real en cualquier navegador y dispositivo móvil, eliminando las diferencias entre los diferentes mecanismos de transporte.
Con el fin de proporcionar conectividad en tiempo real en cada navegador, Socket.IO selecciona el medio de transporte más competente en tiempo de ejecución, sin que ello afecte al API.

La jerarquía de medios de transporte es la siguiente:

- WebSocket
- Adobe Flash Socket
- AJAX long polling
- AJAX multipart streaming (XMLHttpRequest Streaming)
- Forever Iframe (Hidden iframe)
- JSONP Polling

Adobe Flash Socket

Adobe Flash Player admite sockets TCP que le permiten conectarse a otro proceso que actúe como servidor socket, sin embargo, no puede hacer de servidor.

EventMachine

Es una librería para Ruby, C++ y Java que provee entrada/salida basada en eventos haciendo uso del patrón reactor.

Está diseñado para satisfacer simultáneamente dos necesidades clave:

- Alta escalabilidad, rendimiento y estabilidad para los entornos de producción más exigentes.
- Un API que elimine las complejidades de la programación, permitiendo a los ingenieros concentrarse en la lógica de la aplicación.

Esta combinación hace que EventMachine sea una buena opción para desarrollar servidores web o proxys.

La gema EM-WebSocket proporciona un servidor de WebSockets escrito en Ruby y basado en EventMachine.

WebSocket Module for ejabberd

Este módulo añade soporte de WebSockets al servidor XMPP ejabberd.

Es una implementación del borrador *XMPP Over Websocket*, que define como transportar el protocolo XMPP a través de una capa de transporte WebSocket.

Plugin que proporciona soporte de WebSockets para el servidor XMPP Openfire.

Incluye un cliente web implementado como una extensión de Google Chrome.

2.3.6 ¿Qué medio de transporte utilizar?

El medio de transporte que ofrece mejores prestaciones son los WebSockets, sin embargo, el bajo soporte existente en la actualidad en los diferentes navegadores nos obliga a descartarlo como elección.
Podríamos pensar en utilizar Socket.IO para proporcionar mecanismos de fallback, pero esto obligaría utilizar un servidor de presencia o un gestor de conexión que lo soportase.
Ni el módulo de ejabberd ni el plugin de Openfire de WebSockets proporcionan en la actualidad mecanismos de fallback o soporte para Socket.IO.
No obstante, en un futuro próximo podríamos ver soporte para Socket.IO en ejabberd, puesto que ya existe una implementación de la librería en Erlang (*socket.io-erlang*) y los autores del módulo de WebSockets han establecido como prioridad la provisión de mecanismos de fallback, barajando la posibilidad de incluir soporte para Socket.IO.

Descartados los WebSockets la decisión se reduce a elegir entre BOSH y Comet.
Entre las diferentes técnicas que engloba Comet, vamos a quedarnos únicamente con *ajax long polling* ya que, como se ha explicado previamente, el resto no funciona en todos los navegadores sin efectos secundarios negativos, por lo que a partir de ahora cuando nos refiramos a Comet haremos alusión exclusivamente a este método.

BOSH y Comet son muy parecidos en cuanto a que ambos utilizan un mecanismo basado en long polling para la comunicación entre cliente y servidor.
Las principales diferencias son que BOSH ofrece detalles adicionales dentro de su protocolo y que fue desarrollado teniendo XMPP en mente, no obstante, cualquier servidor Comet podría ser desarrollado para cumplir con los estándares definidos en la especificación de BOSH. Por tanto, y sin afán de reinventar la rueda, vamos a emplear BOSH como medio de transporte.

¿Por qué BOSH?

- WebSockets todavía es una tecnología poco madura.
- Al contrario que BOSH, Comet no está diseñado específicamente para XMPP.
- Soportado por la mayoría de clientes y servidores XMPP.

Finalmente, tenemos que decidir que gestor de conexión utilizar.
Dado que el servidor ejabberd que vamos a utilizar tiene un gestor de conexión BOSH incorporado y que no es necesario dar soporte a la conexión con otros servidores XMPP, no resulta necesario utilizar un gestor de conexión independiente, por lo que para el proyecto emplearemos el gestor incorporado de ejabberd.
No obstante, debemos tener en cuenta que si en el futuro se desea hacer la aplicación web federada se deberá emplear un gestor de conexión independiente.

2.4 Clientes XMPP

Se considera cliente XMPP a cualquier aplicación que permita a un usuario conectarse a un servidor XMPP y utilizar los servicios de presencia y mensajería instantánea. Existen muchos clientes para diferentes sistemas operativos y dispositivos.

Cliente	Plataforma	Descripción
Pidgin	Multiplataforma	Anteriormente llamado Gaim
PSI	Multiplataforma	
Gajim	Windows, Linux, BSD	
Kopete	Multiplataforma	Diseñado para KDE
Empathy	Multiplataforma	
Jappix	Navegador	Versión móvil y de escritorio
Beem	Android	El soporte para salas está todavía en desarrollo.
Crosstalk	iOS	Diseñado para el iPAD, pero válido para el iPhone y compatible con iOS 4.
Agile Messenger	Móviles	Versión para iPhone, BlackBerry, Android, Nokia, Sony Ericsson, Windows Mobile y Palm Pre.

A pesar de que existen clientes basados en el navegador libres y gratuitos, como parte de este proyecto se va a llevar a cabo el desarrollo de un cliente XMPP basado en el navegador completamente integrado con la red social.
Esto nos va a permitir, en primer lugar, tener un cliente completamente adaptado a nuestras necesidades y plenamente integrado con la interfaz, y en segundo lugar, nos va a permitir la adición de funcionalidades avanzadas como videollamadas o juegos en red.

No obstante, el servicio de presencia y mensajería instantánea de la red social será accesible desde cualquier cliente XMPP.

2.5 Librerías XMPP

Existen multitud de librerías XMPP escritas en diferentes lenguajes de programación, a continuación vamos a comentar aquellas empleadas en el proyecto:

2.5.1 Strophe

Es una colección de librerías para hablar el protocolo XMPP.
Mientras que la mayoría de librerías XMPP se centran en aplicaciones de chat, Strophe abarca una visión mayor. Puede ser utilizada para la implementación de juegos en tiempo real, sistemas de notificaciones, motores de búsqueda así como mensajería instantánea tradicional.
La funcionalidad de Strophe puede ser extendida fácilmente mediante plugins.
Actualmente existen dos implementaciones: *libstrophe* escrita en C y la que vamos a emplear en este proyecto: *Strophe.js* escrita en JavaScript.

2.5.1.1 Strophe.js

Es una implementación en JavaScript orientada a clientes basados en el navegador, su objetivo principal es permitir a las aplicaciones XMPP en tiempo real basadas en web funcionar en cualquier navegador.
Incorpora soporte para BOSH de forma nativa.
Otras de las características a destacar de Strophe.js, es que ha sido probado durante más de dos años en los principales navegadores y que ha sido utilizado por miles de personas en diferentes aplicaciones web, por ejemplo, en *Chesspark*.
Como se ha comentado anteriormente, es posible extender la funcionalidad base mediante plugins, a continuación se listan algunos ejemplos:

- strophe.muc.js — Soporte para salas de chat
- strophe.roster.js — Facilidades para gestión de listas de contactos
- strophe.pubsub.js — Publicación-Subscripción XMPP
- websocket.js — Soporte para WebSockets
- socket-io.js — Soporte para Socket.IO
- strophe.flxhr.js — XHR multidominio mediante flash
- facebook.js — Conexión con chat de Facebook
- tuenti.js — Conexión con chat de Tuenti

2.5.2 XMPP4r

Es una librería XMPP escrita en Ruby cuyo objetivo es proporcionar un marco de trabajo completo para desarrollar aplicaciones o scripts de Ruby relacionados con XMPP.
Tiene soporte completo para XMPP y para una amplia gama de sus extensiones.

2.6 Lenguajes de programación

A lo largo de este capítulo se han realizado numerosas menciones a los diferentes lenguajes de programación que se van a emplear en el proyecto.
Esta sección está dedicada a comentar sus principales características.

2.6.1 HTML

HyperText Markup Language

Es un lenguaje de marcado escrito en forma de etiquetas empleado en la elaboración de páginas web, principalmente empleado para describir la estructura y el contenido en forma de texto, aunque también puede describir, hasta cierto punto, la apariencia.
Permite la inclusión de hojas de estilo y scripts en las páginas web.

2.6.2 CSS

Cascading Style Sheets

Es un lenguaje usado para definir la presentación de un documento estructurado escrito en HTML. El objetivo principal es separar la estructura de un documento de su presentación. La información de estilo puede ser adjuntada como un documento separado (hoja de estilo) o en el mismo documento HTML.
Una hoja de estilos CSS consiste en una serie de reglas, cada una de las cuales está formada por un bloque de estilos, formado a su vez por varias propiedades *clave:valor,* y un selector, que determina los elementos del documento a los que se les aplicarán los estilos del bloque.

2.6.3 SASS

Syntactically Awesome Stylesheet

Se basa en un lenguaje de script interpretado en el CSS, llamado *SaasScript*.
Sass extiende CSS proporcionando varios mecanismos disponibles en los lenguajes de programación tradicionales, pero no disponibles en el lenguaje CSS.
Cuando SaasScript es interpretado crea bloques de reglas CSS para diferentes selectores según la definición de un fichero Sass. El intérprete de Sass traduce de SassScript a CSS.

2.6.4 JavaScript

Es un lenguaje de script basado en prototipos, dinámico, débilmente tipado e interpretado. Es un lenguaje multi-paradigma que soporta programación orientada a objetos, imperativa y funcional.

La *programación basada en prototipos* es un estilo de programación orientada a objetos en el cual, las clases no están presentes y la herencia se obtiene a través de la clonación de objetos ya existentes, que sirven de prototipos, extendiendo sus funcionalidades.

Todos los navegadores modernos tienen un intérprete de JavaScript.
Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del *Document Object Model* (DOM), un API que proporciona un conjunto estándar de objetos para representar documentos HTML.

Tradicionalmente solo se empleaba en el lado cliente, con el objetivo de proporcionar mejores interfaces de usuario y sitios web dinámicos, sin embargo, en la actualidad está aumentando considerablemente el uso de JavaScript para aplicaciones web del lado servidor.

Dada la gran difusión de JavaScript, existe una inmensa cantidad de librerías.
A continuación, a modo de ejemplo, listamos algunas de las más populares y/o interesantes:

- *Prototype*: es un entorno de trabajo escrito en JavaScript orientado al desarrollo sencillo y dinámico de aplicaciones web. Es una herramienta que implementa las técnicas AJAX y cuyo potencial es aprovechado al máximo cuando se desarrolla con Ruby On Rails.
- *JQuery*: la biblioteca JavaScript más empleada en la actualidad, permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y realizar interacciones mediante ajax para desarrollo web ágil.

- *Script.aculo.us*: es una biblioteca JavaScript basada en Prototype que permite el uso de controles ajax, *drag and drop* y otros efectos visuales en una página web. Esta biblioteca está disponible en Ruby on Rails.
- *Modernizr*: detecta soporte para diferentes funcionalidades de HTML5 y CSS3.
- *Backbone.js*: Ofrece una estructura a las aplicaciones web proporcionando modelos, controladores, colecciones y vistas, y los conecta todos mediante un API existente sobre una interfaz JSON REST.
  Permite implementar el patrón de diseño MVC del lado cliente.

- *Node.js*: entorno JavaScript orientado a eventos que se utiliza para crear software de red escalable, como servidores web.
- *Hydra*.js: definición de sistemas modulares, tolerante a fallos.

De igual manera, también existen numerosas herramientas:

- Compiladores: el término no es estrictamente correcto, ya que en lugar de compilar de código fuente a código máquina, se compila de código JavaScript a un código JavaScript más eficiente.
    - *Closure Compiler*
    - *UglifyJS*
- Entornos de pruebas:
    - *QUnit (usado por el proyecto JQuery)*
    - *Jasmine*

2.6.5 Ruby

Es un lenguaje de programación interpretado, dinámico, reflexivo y orientado a objetos. Su implementación oficial es distribuida bajo una licencia de software libre.

2.6.6 Erlang

Es un lenguaje de programación funcional orientado a concurrencia y distribución, que incluye una máquina virtual.
Erlang tiene evaluación estricta, asignación única (no existen variables mutables) y tipado fuerte y dinámico.
Es un lenguaje interpretado, aunque también se puede compilar usando el compilador HiPE (*High Performance Erlang*).
Fue diseñado por la compañía Ericsson con el propósito de desarrollar aplicaciones distribuidas, tolerantes a fallos y con funcionamiento ininterrumpido.
Originalmente Erlang era un lenguaje propietario de Ericsson, pero fue cedido como software de código abierto en 1998.

Entre sus características podemos destacar:

- Concurrencia: utiliza procesos ligeros cuyos requisitos de memoria pueden variar de forma dinámica. Los procesos no tienen memoria compartida y se comunican por paso de mensajes asíncronos.
  Mientras que en muchos lenguajes de programación los hilos de ejecución se consideran un tema complicado y propenso a errores, la creación y gestión de procesos es trivial en Erlang, ya que toda concurrencia es explicita.

- Distribución: está diseñado para ejecutarse en un entorno distribuido. Una máquina virtual Erlang recibe el nombre de nodo Erlang.
  Un sistema distribuido de Erlang es una red de nodos, donde cada uno puede crear procesos paralelos que se ejecutan en otros nodos.
- Robustez: Los procesos pueden monitorizar el estado y las actividades de otros procesos. Un proceso de un sistema distribuido puede configurarse para conmutarse con otro en caso de error.
- Actualización de código en caliente: permite cambiar partes de una aplicación en funcionamiento, durante la transición, tanto el código antiguo como el nuevo pueden coexistir, por tanto, es posible instalar parches y actualizaciones en un sistema en funcionamiento.

La OTP (*Open Telecom Platform*) es una distribución de Erlang de código abierto que incluye, entre otros componentes, un intérprete, un compilador, un protocolo de comunicación entre servidores, un servidor de base de datos distribuido (*Mnesia*) y una colección de librerías.

2.6.7 Programación en Bash

Bash (*bourne again shell*) es un programa cuya función consiste en interpretar órdenes. Está basado en la consola de Unix y es el intérprete de comandos por defecto en la mayoría de las distribuciones de Linux.
Si bien al proceso de crear scripts de Bash se le denomina programación Bash, hay que tener en cuenta que Bash no es estrictamente un lenguaje de programación.

2.7 Ruby on Rails (*RoR*)

Es un entorno de desarrollo web de código abierto que usa el lenguaje de programación Ruby y que está diseñado de acuerdo al paradigma de la arquitectura Modelo Vista Controlador (MVC).
Rails se distribuye a través de *RubyGems*, que es el canal de distribución oficial de bibliotecas y aplicaciones Ruby.
En cuanto a su filosofía, destacan los siguientes principios:

- *DRY (Don't repeat yourself)*: significa que nunca se debería repetir información.
- *Convención sobre configuración*: significa que un desarrollador sólo necesita especificar los aspectos no convencionales de la aplicación.
  Permite al desarrollador disminuir el número de decisiones que debe tomar y escribir menos código para lograr el mismo resultado a cambio de conocer y seguir una convención establecida.
  Cuando la convención definida no es suficiente para lograr el comportamiento deseado, el desarrollador puede alterar el comportamiento por defecto y adaptarlo a sus necesidades.

Rails soporta diferentes sistemas de gestión de bases de datos relacionales tales como MySQL, PostgreSQL, SQLite, IBM y Oracle.
En cuanto a los servidores web utilizados, para desarrollo y pruebas es frecuente la utilización de *Mongrel* o *WEBrick*, y para la utilización de Rails en servidores de producción se está extendiendo el uso de *Passenger*, un módulo de Apache diseñado para facilitar el despliegue de aplicaciones Rails en este servidor.
Para el despliegue es usual la utilización de herramientas como *Capistrano*.
Los plugins y/o códigos añadidos a los proyectos de Ruby on Rails reciben el nombre de gemas, permiten añadir nuevas funcionalidades o herramientas para el desarrollo.
Existen muchos entornos de trabajo de RoR, entre los cuales podemos destacar *Aptana* (disponible como plugin de eclipse o de forma independiente), *Netbeans* y *TextMate* (solo disponible para Mac).
La versión más reciente, y la empleada actualmente en Social Stream, es *Ruby on Rails 3.2.0,* publicada el 20 de enero del 2012.
Entre las mejoras de Ruby on Rails 3.0 frente a su versión anterior, Ruby on Rails 2.3, podemos destacar las siguientes:

- Nueva API de enrutamiento.
- Nueva API de *Action Mailer (Módulo RoR para el envío de emails).*
- Nuevo lenguaje de consultas *Active Record.*
- *Helpers* de JavaScript no intrusivos con controladores para *Prototype* y JQuery.
- Gestión de dependencias con *Bundler.*

2.8 OpenTok

Proporciona una API gratuita que permite a cualquier desarrollador añadir funcionalidades de video chat a sus propias páginas web.
Actualmente existen versiones de la librería para JavaScript, ActionScript e iOS en el lado cliente, y para Ruby, Java, PHP y Python en el lado servidor.
El servicio básico de OpenTok es gratuito y han anunciado que seguirá siéndolo.
Actualmente el único servicio premium disponible es el servicio *Archiving*, que permite a los desarrolladores grabar secuencias de video chat, que serán almacenadas automáticamente en la nube, y la reproducción de videos mediante streaming.

2.9 API REST

Es un servicio web implementado mediante los principios de *REST*.
Consta de una colección de recursos, con cuatro aspectos definidos:

- La URI base para el servicio web.
- Tipo de datos soportados.
- El conjunto de funciones disponibles.
- El API debe estar orientada a hipertexto.

Puede ser descrita como una API (librería de funciones o métodos) a la que se accede mediante el protocolo HTTP, a través de URLs en las que enviamos los parámetros de nuestra consulta, obteniendo como respuesta a dicha consulta datos en diferentes formatos, como pueden ser texto plano, XML, JSON, etc.

2.10 Git

Es un sistema distribuido de control de versiones diseñado para manejar proyectos muy grandes con velocidad y eficiencia, pero igual de apropiado para repositorios pequeños. Es software libre distribuido bajo la licencia GNU, es especialmente popular en la comunidad *open source*, sirviendo como plataforma de desarrollo para proyectos como el núcleo de Linux, Ruby on Rails o WINE.

2.11 Web chats

Podemos definir el término *chat* como una comunicación escrita entre dos o más personas realizada de manera instantánea mediante el uso de software a través de Internet. Si añadimos a la comunicación las capacidades de audio y video, se emplea entonces el término de *video chat*.
Si el software empleado para realizar la comunicación está basado en una interfaz web, entonces hablamos de *web chats*, cuya principal característica es permitir a los usuarios el acceso a través del navegador sin exigir la instalación de software adicional.

Finalmente comentaremos brevemente las principales características de algunos de los web chats más populares.

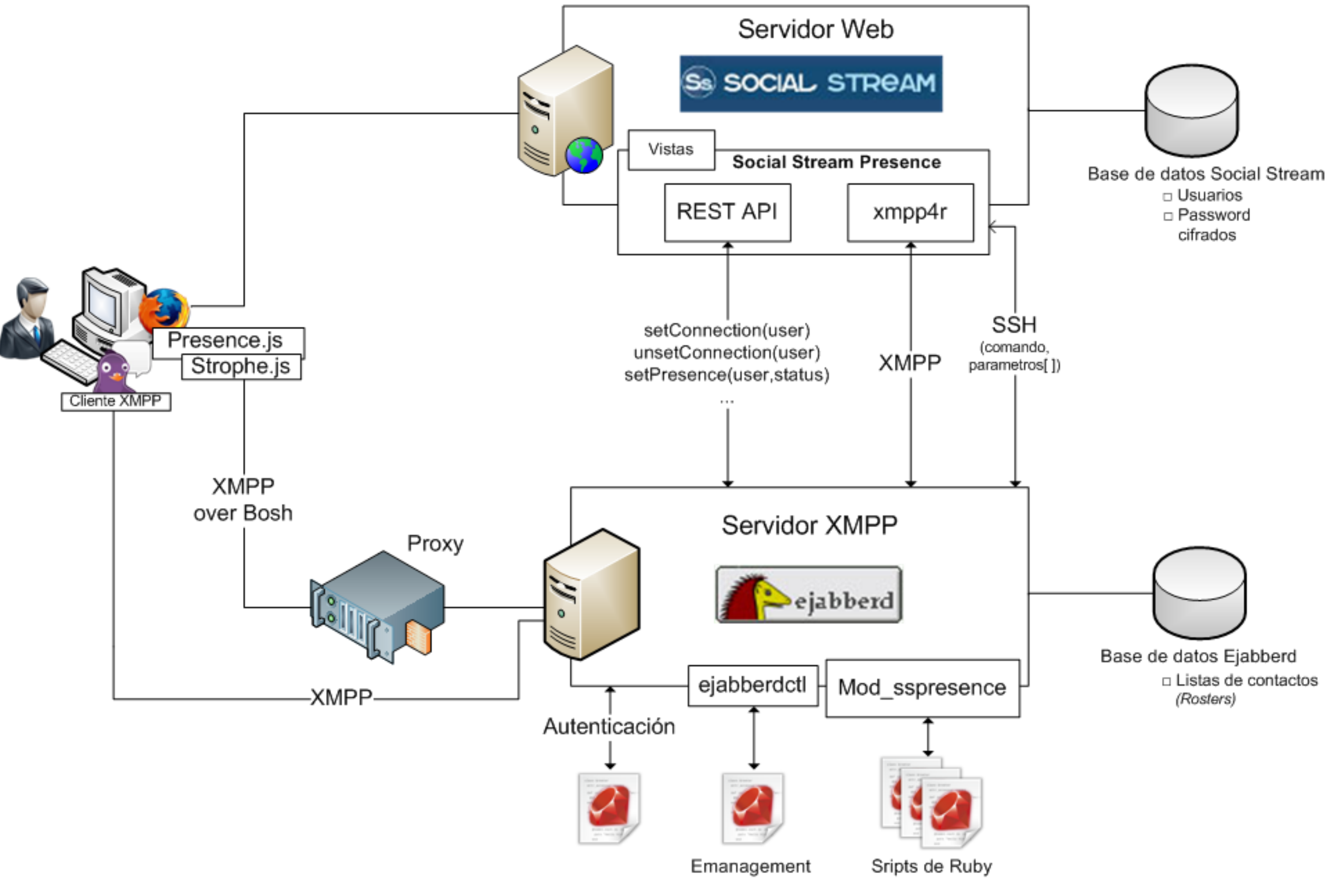
<table>
<tr><th>Web chat</th><th>Características</th></tr>
<tr><td>GMail Chat</td><td>♦ Proporcionado mediante el servicio Google Talk, que ofrece servicios de presencia y mensajería instantánea a millones de usuarios.
Google Talk sirve a diversos clientes (web, propios o de terceros, móviles) y se integra en diferentes aplicaciones.
♦ Los servidores Google Talk están implementados principalmente en Java, pero se emplean multitud de librerías escritas en C++.
♦ Gmail Chat se abstrae de la complejidad del sistema, de modo que no tiene conocimiento de la cantidad de servidores, balance de carga, etc.
♦ Está basado en el protocolo XMPP.
♦ Cliente JavaScript.
♦ Emplea técnicas de long-polling.
♦ Único mecanismo de autenticación admitido SASL PLAIN.
♦ Emplea extensiones XMPP para la señalización de voz y video y la comunicación p2p. Actualmente, la implementación difiere ligeramente de las especificaciones del borrador XMPP Jingle, pero Google lo está actualizando para alcanzar la plena conformidad.
♦ Soporta federación con cualquier servidor XMPP.
♦ No soporta algunas extensiones XMPP como vcard o User Nickname.
♦ No usa el protocolo XMPP para el envío de archivos, por lo que se hace imposible realizar esta tarea con otros clientes de mensajería.
♦ No usa cifrado extremo a extremo: conversaciones en claro.</td></tr>
<tr><td>Facebook Chat</td><td>♦ Basado en el protocolo XMPP.
♦ Cliente JavaScript.
♦ Emplea tanto técnicas de Ajax polling como de long polling (híbrido).
♦ Comunicación de servicios mediante Trift, un lenguaje de definición de interfaces utilizado para el desarrollo de servicios multilenguaje.
♦ A nivel web emplea PHP, pero la gestión de presencia y el almacenamiento de conversaciones están implementados en C++.
♦ Emplea Erlang para gestionar las colas y el envío de mensajes.
♦ Mecanismos de autenticación soportados:
• SASL X-FACEBOOK-PLATFORM: permite a los clientes conectarse al chat mediante la autenticación de Facebook.
• DIGEST-MD5: autenticación tradicional con usuario y contraseña.
♦ Facebook no proporciona acceso directo a un servidor XMPP, sino un API XMPP, por lo que no es posible tener acceso a todas las operaciones habitualmente disponibles en un servidor XMPP. Funcionalidad limitada.
♦ Puesto que los contactos de un usuario están basados en sus amigos de Facebook, no pueden ser modificados mediante mecanismos estándar XMPP.
♦ Algunos mensajes XMPP no pueden ser intercambiados entre clientes.
♦ Actualmente no soporta federación.
♦ Video chat mediante Skype video calling plug-in.
♦ No usa cifrado extremo a extremo: conversaciones en claro.</td></tr>
</table>

3. Arquitectura

Arquitectura

En este capítulo vamos a describir la arquitectura empleada para proporcionar el servicio de presencia y mensajería instantánea a la plataforma Social Stream.

El esquema general puede observarse en la siguiente figura:



Arquitectura Social Stream Presence

Todos los componentes, tecnologías y lenguajes de programación involucrados han sido introducidos en el capítulo anterior, no obstante, incidiremos nuevamente en algunos de ellos con más detalle, especialmente en el protocolo XMPP, al que dedicaremos un capítulo entero.

En primer lugar podemos ver que existen tres agentes principales involucrados en la arquitectura: el servidor Web, el servidor XMPP y el cliente, que puede tratarse de un cliente XMPP basado en el navegador o externo a la red social.

Tal y como se explicó previamente, se emplea BOSH para la comunicación entre los clientes XMPP basados en el navegador y el servidor XMPP, por lo tanto es necesario disponer de un proxy que redirija de forma adecuada estos mensajes XMPP al servidor. No es necesario emplear un proxy específico, en la documentación oficial se ofrece la configuración para *Apache* y *Nginx*, pero se puede emplear cualquier otro.
El único requisito es que este proxy debe tener la capacidad de funcionar como proxy inverso (*reverse proxy*), de forma que todo el tráfico entrante de internet y con destino el servidor Web pase a través de él.
De esta forma podemos redirigir las peticiones BOSH enviadas a una determinada ruta a través del puerto 80, a la ruta y puerto (generalmente el 5280) donde se encuentra escuchando el módulo *http bind* de ejabberd, responsable del soporte de BOSH.
Si accedemos con el navegador a la ruta configurada, obtendremos el siguiente mensaje informativo ofrecido por el módulo *http bind* de ejabberd:

ejabberd mod_http_bind

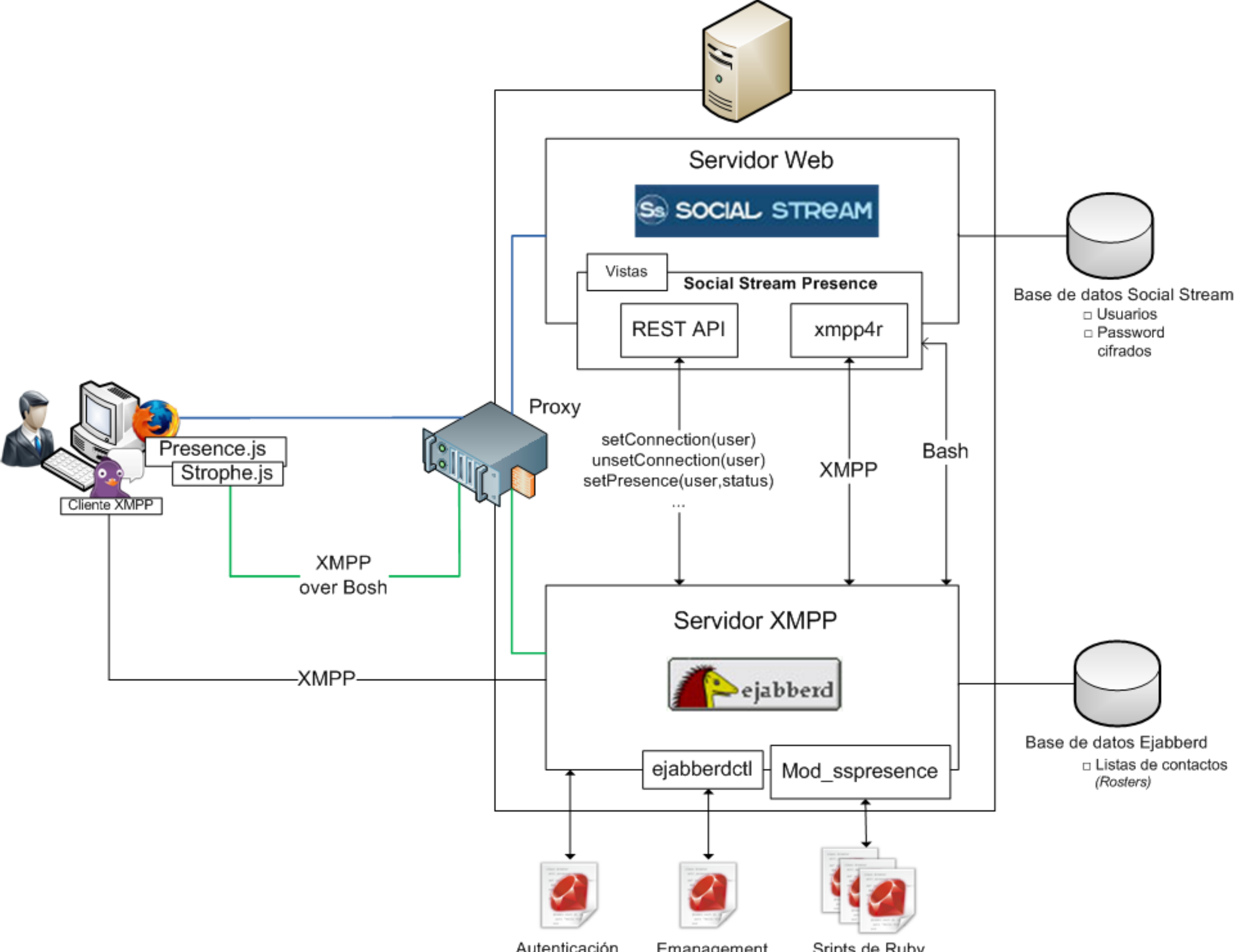
An implementation of XMPP over BOSH (XEP-0206)

This web page is only informative. To use HTTP-Bind you need a Jabber/XMPP client that supports it.

Otro aspecto importante a comentar sobre la arquitectura es la comunicación entre los servidores Web y XMPP.

Social Stream Presence tiene dos modos de funcionamiento: ***local*** y ***remoto***.

- La gema opera en modo local cuando el servidor Web y el servidor XMPP están hospedados en la misma máquina, utilizando el mismo sistema de ficheros.
- La gema opera en modo remoto cuando el servidor Web y el servidor XMPP están hospedados en diferentes máquinas.



Arquitectura Social Stream Presence: Modo Local

El servidor Web gestiona al servidor XMPP mediante órdenes que son llevadas a cabo mediante comandos de Bash, estos comandos son ejecutados directamente en Bash en modo local y mediante SSH en modo remoto.

SSH (Secure Shell)

Es el nombre de un protocolo y del programa que lo implementa, y sirve para acceder a máquinas remotas a través de una red. Permite manejar por completo la computadora mediante un intérprete de comandos, y también puede redirigir el tráfico de X (sistema de ventanas) para poder ejecutar programas gráficos.

En cambio, para la comunicación en el sentido servidor XMPP a servidor Web, se emplea una API REST.
En este caso, el funcionamiento entre el modo local y el modo remoto es idéntico, con la salvedad de que en modo local el tráfico se cursará por la interfaz de *loopback*, una dirección especial que los hosts pueden utilizar para dirigir el tráfico hacia ellos mismos.

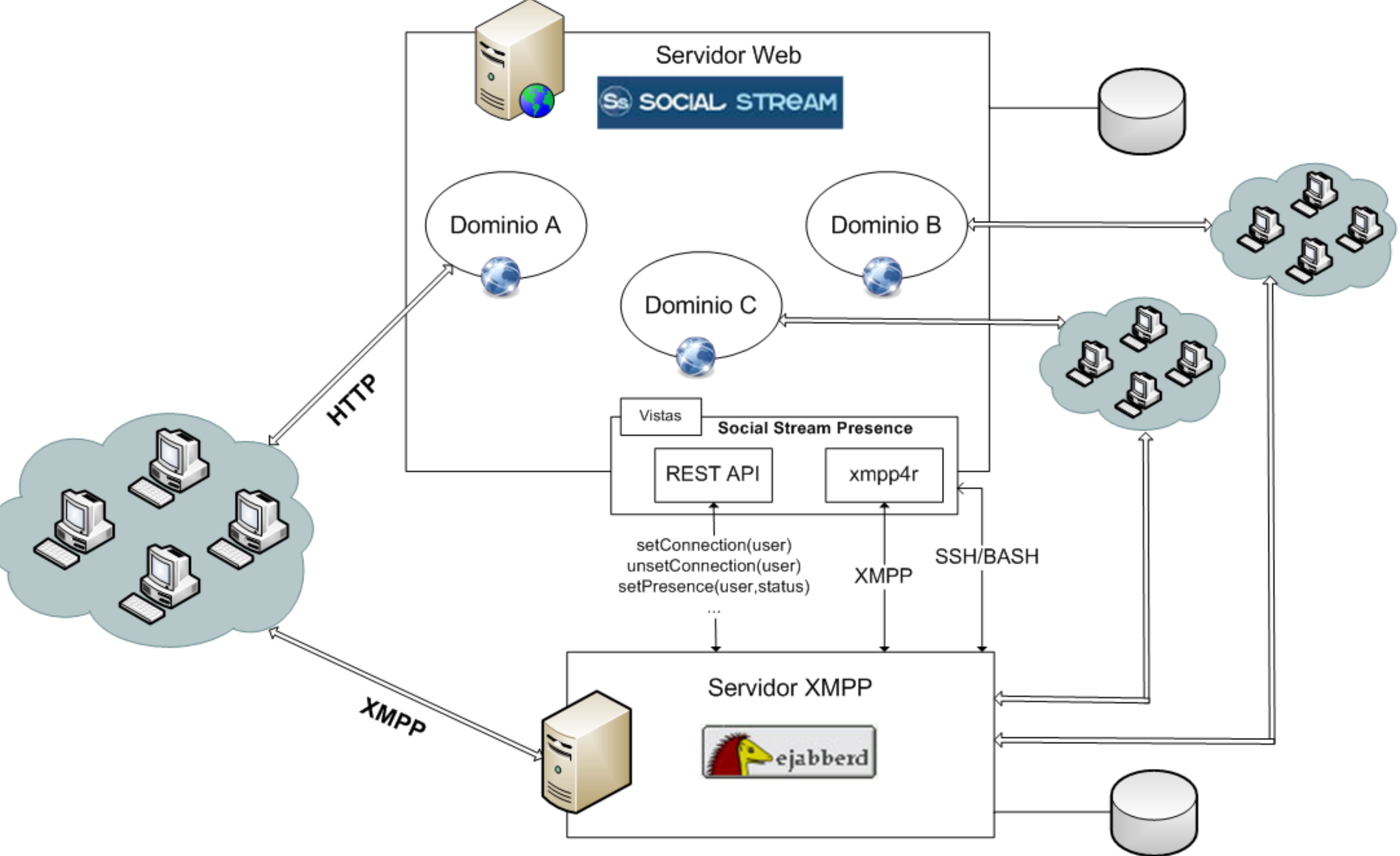
La autenticación en el servidor XMPP se realiza contra el servidor Web, de modo que la base de datos del servidor XMPP no guarda contraseñas, únicamente almacena información relacionada con los servicios de presencia y mensajería instantánea, como las listas de contactos, estados de presencia, mensajes no entregados (*mensajes offline*), historiales de las salas de chat, etc.

Con el objetivo de aumentar la funcionalidad del servidor XMPP, se han desarrollado dos nuevos componentes:

- *Mod_sspresence*: Un módulo de ejabberd, cuyo objetivo principal es la notificación de determinados eventos.
- *Emanagement*: es un script escrito en Ruby que hace uso de *ejabberdctl,* una interfaz de control del servidor ejabberd que permite la ejecución de diversas órdenes mediante línea de comandos.
  Su objetivo principal es extender la funcionalidad de *ejabberdctl*, especialmente para las labores de gestión de la base de datos.

Finalmente se ha desarrollado un cliente XMPP basado en el navegador completamente integrado con Social Stream escrito enteramente en JavaScript empleando *Strophe.js* como librería XMPP.

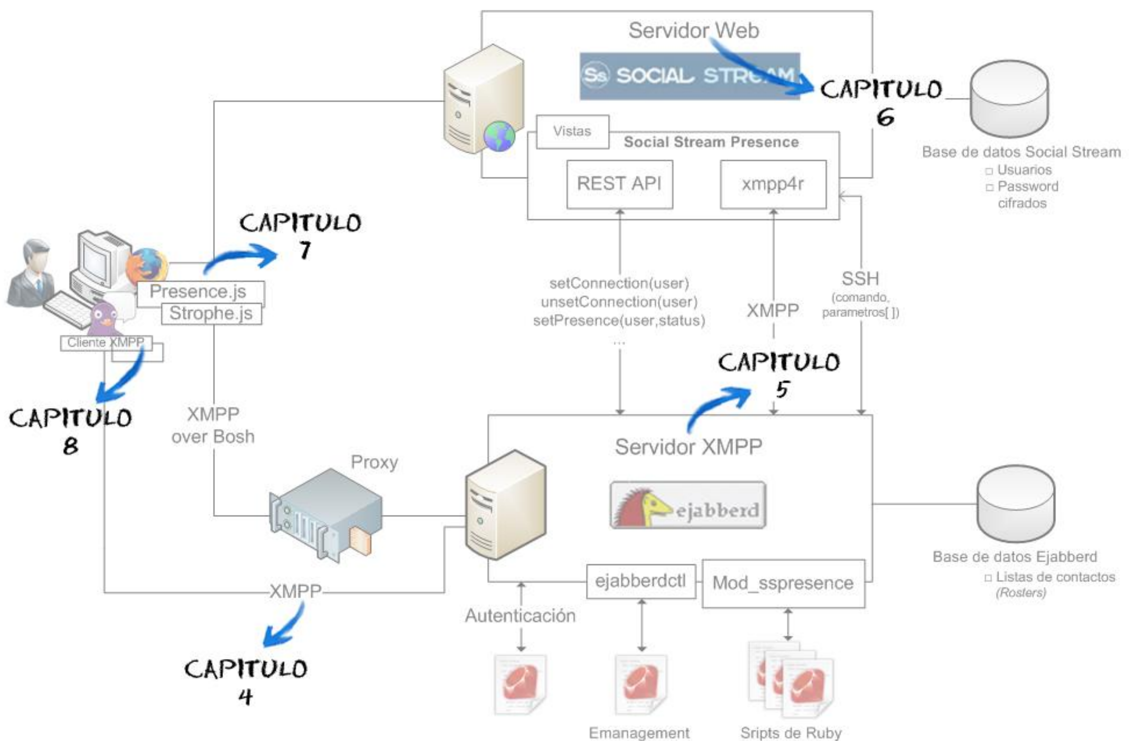
Social Stream Presence también ha sido diseñado para funcionar en una arquitectura multidominio, con la única restricción de que los dominios se encuentren funcionando bajo el mismo servidor Web.



Arquitectura Social Stream Presence: Multidominio

El servidor ejabberd permite la gestión de múltiples dominios de forma nativa, por lo que solo ha sido necesario adaptar los diferentes elementos adicionales (*Mod_sspresence, Emanagement, scripts,…)* para dotar al servicio de presencia y mensajería instantánea de soporte multidominio.
Pese a que actualmente Social Stream todavía no tiene soporte multidominio completo, la gema Social Stream Presence ofrece mecanismos para registrar y eliminar de forma automática dominios en el servidor XMPP.

En los próximos capítulos se va a explicar con detalle cada uno de los elementos de la arquitectura de Social Stream Presence:



4. XMPP

XMPP

En capítulos anteriores ya hemos comentado las principales características del protocolo XMPP, en este capítulo vamos a estudiarlo con un mayor nivel de detalle abordando aspectos más técnicos.

4.1 Modelo de mensajería

En primer lugar es necesario definir algunos conceptos fundamentales:

- *Entidad*

Se considera entidad a cualquier terminal capaz de comunicarse empleando XMPP.
El identificador de una entidad recibe el nombre de **JID** (*Jabber Identifier*).

- *XML Stream*

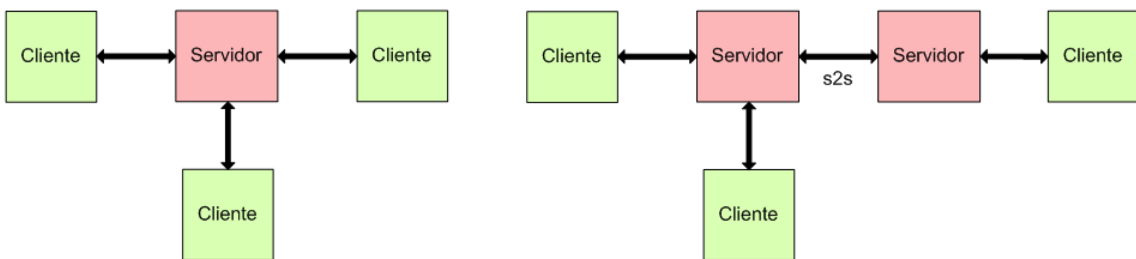
Es un contenedor para el intercambio de elementos XML entre dos entidades a través de una red. Mientras que un stream permanezca activo, la entidad que lo inició puede enviar un número ilimitado de elementos XML, tanto elementos de negociación (por ejemplo para negociar el uso de TLS o SALS) como *stanzas XML*.
El stream inicial permite comunicación unidireccional, para habilitar el intercambio de información en ambos sentidos, la entidad receptora debe negociar un stream en sentido contrario.

- *XML Stanza*

Es una unidad semántica discreta de información estructurada que es enviada de una entidad a otra sobre un stream XML. Una stanza puede contener otros elementos, que a su vez pueden venir acompañados de atributos, nuevos elementos o texto.
Los elementos XML enviados con el propósito de negociar el uso de TLS o SALS no se consideran stanzas.
Existen tres tipos de stanzas: ***message***, ***presence*** e ***iq***.

Por tanto, cuando un cliente se quiera conectar al servidor XMPP, iniciará un stream XML hacia el servidor, que este deberá responder iniciando un segundo stream XML hacia el cliente. Una vez finalizadas las negociaciones y la autenticación, el cliente podrá enviar un número ilimitado de stanzas a cualquier entidad de la red.
En esencia, un stream XML actúa como un envoltorio para todas las stanzas enviadas durante una sesión.
Por tanto y como conclusión, a partir de ahora emplearemos el término *stanza* para referirnos a cualquier mensaje XMPP intercambiado durante una sesión.

El servidor XMPP participa en todas las comunicaciones, es decir, las stanzas generadas por los clientes siempre son enviadas al servidor, incluidos los mensajes intercambiados entre dos entidades que también son enviados a través del servidor.



Las principales responsabilidades del servidor son:

- Gestionar las conexiones y sesiones de clientes autorizados, servidores y otras entidades.
- Enrutar adecuadamente las stanzas entre entidades.
- Almacenar información útil para los clientes: listas de contactos, estado, etc.

4.2 Identificadores

El identificador de una entidad recibe el nombre de JID (*Jabber Identifier*) y consta de tres partes:

- Nodo

Habitualmente representa a un usuario, sin embargo, puede representar otro tipo de entidades, por ejemplo, una sala de chat.

- Dominio

Es el identificador principal y el único obligatorio. Suele representar al servidor principal, pero en algunas ocasiones el identificador hace referencia a un servicio que se direcciona como un subdominio del servidor, por ejemplo un servicio de chat multiusuario o una pasarela hacia otros sistemas de mensajería.

- Recurso.

Generalmente representa una sesión específica, aunque también puede representar un objeto perteneciente a la entidad asociada al identificador de nodo, por ejemplo un participante en una sala de chat. Una entidad puede mantener múltiples recursos (sesiones) conectados de forma simultánea, siendo cada recurso (o sesión) diferenciado por un identificador distinto.

4.3 Stanzas: Atributos comunes

Existen una serie de atributos que son comunes a los tres tipos de stanzas.

- *To*

Especifica el JID del destinatario de la stanza.
Si una stanza debe ser manejada por el servidor, por ejemplo, una stanza de presencia que deba ser difundida a otras entidades, esta no debe poseer el atributo *to*.

- *From*

Especifica el JID del remitente de la stanza.
Cuando un servidor genera él mismo una stanza para enviársela a un cliente, la stanza debe no incluir un atributo from o bien incluir un atributo from cuyo valor sea el *bare JID* del cliente. Cuando un cliente recibe una stanza en estas condiciones, debe asumir que la stanza ha sido generada por el servidor con el que se encuentra conectado.
Este atributo es obligatorio para todas las stanzas generadas en los clientes.

- *Id*

Es opcional y puede ser usado por una entidad para el seguimiento interno de stanzas. Es de especial utilidad para el seguimiento de las interacciones petición-respuesta de las stanzas iq.

- *Type*

Especifica información detallada sobre el objetivo o contexto de la stanza.
Los valores permitidos varían en función del tipo de stanza, siendo el único valor común la stanza de tipo *"error".*

- *Xml:lang*

Especifica el idioma por defecto de cualquier tipo de datos legibles.
Toda stanza que contenga texto con el objetivo de ser presentado a una persona debería tener este atributo.

4.4 Espacios de nombres (XML Namespaces)

Proporcionan un método simple para clasificar los nombres de los elementos y atributos empleados en los documentos XML mediante su asociación con espacios de nombres identificados por referencias URI.
XMPP emplea espacios de nombres para extender las stanzas con el propósito de añadir nuevas funcionalidades.
En esta página http://xmpp.org/xmpp-protocols/protocol-namespaces se listan los diferentes espacios de nombres oficiales del protocolo XMPP, incluyendo enlaces a las especificaciones relevantes y los esquemas XML.

4.5 Intercambio de mensajes

Se realiza mediante stanzas de tipo mensaje (*message)*, que pueden verse como un mecanismo mediante el cual una entidad envía información a otra entidad.
Todas las stanzas de tipo *message* deben tener un atributo *to* especificando el destinatario del mensaje, de modo que el servidor, una vez recibida la stanza, sea capaz de enrutarla o enviarla a su destinatario.

Existen varios tipos de mensajes en función del valor tomado por el atributo *type*:

- *Chat*: utilizado para sesiones de chat entre dos entidades.
- *Groupchat:* mensajes intercambiados entre múltiples entidades en salas de chat.
- *Normal:* empleado para hilos de conversaciones que tienen lugar fuera de un contexto de tiempo real.
- *Headline:* mensajes, de los cuales no se debe esperar respuesta y típicamente generados por chatbots, diseñados para mostrar información en la barra de estado u otras partes de la interfaz de usuario.
- *Error:* para enviar mensajes de error.

Estas stanzas pueden contener, sin declaraciones adicionales, 4 tipos de subelementos:

- *<subject>*: título o tema del mensaje.
- *<thread>:* identificador de conversación generado por un cliente.
- *<body>:* cuerpo del mensaje.
- *<error>:* si procede, mensaje de error.

Los mensajes offline son idénticos, con la salvedad de que existe una recomendación *(XEP-0160: Best Practices for Handling Offline Messages)* según la cual los servidores deberían añadir un nuevo subelemento *<delay>* indicando que el mensaje fue generado cuando el cliente no se encontraba conectado.

```
<message from='demo@euphoria' to='ernest-ramirez@euphoria' type='chat'>
        <body>¡Hola Mundo!</body>
</message>
```

Mensaje de chat que envía el usuario demo al usuario ernest-ramirez.

```
<message from='gigashots@conference.euphoria/Demo' to='ernest-ramirez@euphoria'
type='groupchat' id='9150'>
        <body>Me encantan las salas multiusuario de Social Stream Presence</body>
        <x xmlns='jabber:x:event'>
                <composing/>
        </x>
</message>
```

Mensaje enviado por el ocupante Demo a la sala de chat “gigashots”.
Se muestra la stanza recibida por el ocupante ernest-ramirez.

4.6 Presencia

El intercambio de información de presencia se hace relativamente sencillo en XMPP empleando stanzas de presencia (*presence*).

El protocolo de presencia se usa principalmente en dos contextos:

- Actualizaciones de presencia

Originadas por cambios en el estado o la disponibilidad de los usuarios.

- Gestión de subscripciones

Con el fin de proteger la privacidad de los usuarios de mensajería instantánea y de cualquier otra entidad, la presencia se da a conocer solo a aquellas entidades aprobadas por el usuario.
Cuando un usuario autoriza a una entidad a ver su presencia, se dice que la entidad tiene una **subscripción** a la información de presencia del usuario.
Las subscripciones son gestionadas mediante el envío de stanzas de presencia con atributos especiales. Estas stanzas siguen un protocolo de petición-respuesta, se requiere autorización explícita de un usuario para subscribirse a su presencia.
Una subscripción de presencia puede encontrarse en los siguientes cuatro estados:
- *none*: no existe ninguna subscripción.
- *to*: El usuario tiene una subscripción a la información de presencia del contacto.
- *from*: El contacto tiene una subscripción a la información de presencia del usuario.
- *both*: tanto el usuario como el contacto tienen subscripciones a la información de presencia del otro.

En ambos casos el servidor XMPP actúa como intermediario entre el emisor de la actualización de presencia y los destinatarios de la misma.
El servidor tiene la obligación de hacer llegar la actualización de presencia de un usuario a todos sus contactos autorizados para recibirla.
En XMPP, a la lista de contactos de un usuario se le llama **roster**, y consiste en un numero de elementos identificados por un JID (generalmente de la forma *usuario@dominio*).
El roster de cada usuario es almacenado por su servidor, de modo que el usuario puede acceder a la información del roster desde cualquier terminal.

La ausencia del atributo *type* se emplea para indicar al servidor que el usuario está conectado y disponible para la comunicación.
En caso de especificar un valor, estos son los diferentes tipos de stanzas de presencia:

- *available*: indica que el usuario está disponible para la comunicación.
- *unavailable* indica que el usuario no está disponible para la comunicación.
- *subscribe*: mensaje de petición de subscripción de presencia, el usuario que lo envía desea subscribirse a la presencia del destinatario.
- *unsubscribe*: mensaje de cancelación de subscripción de presencia, el usuario que lo envía desea cancelar su subscripción a la presencia del destinatario.
- *subscribed*: mensaje enviado por un usuario para comunicar que permite al destinatario la subscripción a su presencia.
- *unsubscribed*: mensaje para denegar una petición de subscripción o para cancelar una subscripción previamente concedida.
- *error*: mensaje estándar de error.
- *probe*: petición de un servidor para conocer la presencia actual de una entidad.

Estas stanzas pueden contener, sin declaraciones adicionales, 4 tipos de subelementos:

- *<status>:* texto libre con información detallada del estado de disponibilidad.
- *<priority>:* prioridad numérica del mensaje.
- *<error>:* Si procede, mensaje de error.
- *<show>:* Estado estándar. Hay cuatro posibilidades:
    - *chat*: el usuario está interesado en chatear.
    - *away*: el usuario está ausente por un corto periodo de tiempo.
    - *xa (extended away)*: el usuario está ausente por un largo periodo de tiempo.
    - *dnd (do not distrub)*: el usuario está ocupado.

```
<presence from='demo@euphoria/3652519' to='ernest-ramirez@euphoria/1335202'>
        <status>Vuelvo en 5 minutos</status>
        <show>away</show>
</presence>
```

Stanza de presencia recibida por el usuario ernest-ramirez.
Indica que el usuario demo ha cambiado su estado a "away" (ausente).

```
<presence from='demo@euphoria/3652519' to='ernest-ramirez@euphoria/13352' type='unavailable'/>
```

En este caso la stanza recibida indica al cliente del usuario ernest-ramirez que el usuario demo ya no se encuentra disponible para la comunicación, está desconectado.

4.7 IQ (Info/Query)

Las stanzas IQ proporcionan un mecanismo de petición-respuesta extensible.
Cuando un cliente envía una stanza de tipo IQ siempre espera recibir una respuesta del destinatario, aunque sea de tipo error.
El contenido de los datos de la petición y la respuesta se define mediante el espacio de nombres declarado en un subelemento directo de la stanza IQ mediante el atributo *xmlns*, y la interacción es seguida mediante el uso del atributo *id*. Existen múltiples extensiones IQ y cualquier desarrollador puede implementar las suyas propias.

Las interacciones IQ siguen un patrón común de intercambio de datos estructurados acorde a las siguientes reglas:

- El atributo *id* es obligatorio para las stanzas IQ.
- El atributo *type* también es obligatorio y su valor debe ser alguno de los siguientes:
    - *get*: La stanza es una petición de información.
    - *set*: La stanza es una petición que proporciona datos requeridos, establece nuevos valores o reemplaza valores existentes.
    - *result*: la stanza es una respuesta a una petición *get* o *set*.
    - *error*: la stanza es una respuesta que indica que se produjo un error relacionado con el procesamiento o la respuesta de la petición.
- Una entidad que reciba una petición IQ de tipo *get* o *set* debe responder con una respuesta IQ de tipo *result* o *error* cuyo atributo id sea igual al de la petición.
- Una entidad que reciba una respuesta IQ no debe responder con otra respuesta, sin embargo, puede responder con otra petición IQ.
- Una petición IQ debe contener un único subelemento que especifique la semántica de la petición o respuesta particular.
- Una stanza IQ de tipo *result* debe incluir un o ningún subelemento.
- Una stanza de tipo error debe incluir un subelemento *<error>, y* debería incluir también el subelemento contenido en la petición asociada.

```
<iq from='demo@euphoria' to='ernest-ramirez@euphoria' type='get' id='versionID'>
        <query xmlns='jabber:iq:version'/>
</iq>
```

El usuario ernest-ramirez recibe una petición IQ de demo para conocer los datos de su cliente de mensajería instantánea.

```
<iq from='ernest-ramirez@euphoria' to='demo@euphoria' type='result' id='versionID'>
        <query xmlns='jabber:iq:version'>
                <name>Pidgin</name>
                <version>2.6.6 (libpurple 2.6.6)</version>
        </query>
</iq>
```

El cliente de ernest-ramirez responde mediante una respuesta IQ con los datos solicitados.

4.8 MUC (XEP-0045: Multi-User Chat)

Finalmente vamos a describir muy brevemente algunos conceptos del protocolo XMPP necesarios para comprender el funcionamiento de las salas multiusuario.

El identificador de una sala de chat, o **room**, es de la forma **roomName@service**, en ejabberd habitualmente toman la forma *roomName@conference.domain*.
A cualquier usuario que esté dentro de una sala se le llama **ocupante**, y su identificador (*occupant JID*) es de la forma **roomName@service/Nick**, donde *Nick* es el apodo del usuario en dicha sala.

Las principales acciones que puede realizar un usuario en una sala son:

- Unirse a la sala: enviando una presencia *available* a roomName@service/Nick.
- Enviar mensajes a toda la sala: mensajes de tipo *groupchat* enviados a la propia sala (roomName@service). El usuario debe haberse unido a la sala previamente.
- Enviar mensajes privados: enviando mensajes de tipo *chat* al *occupant JID*.
- Cambio de apodo y/o estado disponibilidad: enviando una presencia con el nuevo estado a roomName@service/nuevoNick.
- Abandonar la sala: enviando una presencia *unavailable* a roomName@service/Nick.

Como se puede ver, para unirse y abandonar un grupo se usa el protocolo de presencia y para enviar mensajes el protocolo habitual de mensajes.

Cada ocupante puede tener un rol y/o una afiliación.
Las afiliaciones son asociaciones de larga duración con una sala: dueño, miembro, etc.
Un rol es una posición temporal dentro de una sala, que se pierde al abandonarla.
Los posibles roles son *moderator*, *participant* y *visitor*.
En este breve resumen no vamos a incidir sobre los diferentes permisos que tiene un usuario en función de su afiliación y/o rol, sin embargo, en caso de mayor interés se puede encontrar una explicación detallada en la especificación XEP-0045.

En cuanto a los tipos de salas podemos destacar los siguientes:

- Salas Ocultas: No pueden ser encontradas por ningún usuario mediante los medios habituales, como el servicio de descubrimiento. Antónimo: Salas Públicas.
- Salas Temporales: son destruidas cuando su último ocupante las abandona. Antónimo: Salas Persistentes.
- Salas sólo para miembros: un usuario necesita estar en la lista de miembros para acceder a la sala. Antónimo: Salas abiertas.
- Salas protegidas por password: es necesario ingresar una contraseña para acceder.

Otra utilidad de las salas de chat es su posibilidad de ser empleadas como salas de juego multijugador, donde los jugadores intercambian stanzas IQ.

5. Servidor XMPP

Servidor XMPP

En este capítulo vamos a centrarnos en los diferentes aspectos del servidor XMPP.

5.1 Autenticación

La arquitectura se ha diseñado de modo que solo el servidor Web almacena información de autenticación (usuarios y contraseñas cifradas), por lo que el servidor XMPP tendrá que autenticar a los usuarios contra el servidor Web.
En el caso de la arquitectura multidominio, la autenticación se realiza contra el dominio web específico del cliente.

Ejabberd permite habilitar un modo de autenticación externo mediante la ejecución de un script de autenticación propio.
Básicamente consiste en un demonio, cuyo número de instancias en ejecución se puede configurar, al que ejabberd solicitará una respuesta de la forma *true/false* cada vez que tenga que autenticar o autorizar a un usuario.
Al margen de estas restricciones, la implementación del script es libre, incluido el lenguaje de programación. En nuestro caso se ha escrito en Ruby empleando la gema *REST Client*, un cliente HTTP y REST muy sencillo, para realizar las consultas.

La idea es que la conexión al chat de la red social desde el navegador sea completamente transparente al usuario, y que además este tenga la posibilidad de conectarse desde cualquier cliente XMPP externo empleando el mismo nombre de usuario y contraseña con el que se registró en la red social.

Para lograr este objetivo *Social Stream Presence* da soporte a dos mecanismos de autenticación, uno basado en cookie cifrada, pensado especialmente para la conexión desde el propio navegador, y un segundo método tradicional basado en usuario y contraseña, disponible tanto para el cliente basado en el navegador como para clientes externos.

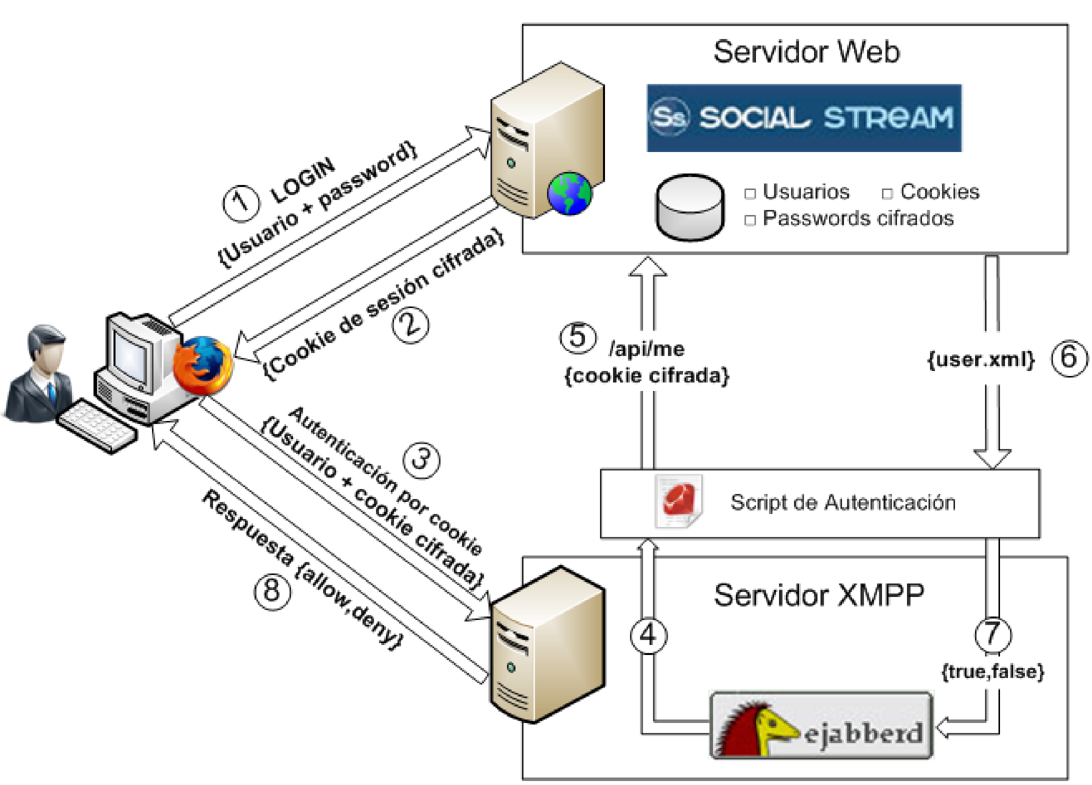
Strophe.js emplea de forma automática el mecanismo de autenticación más seguro ofrecido por el servidor, soporta autenticación *SASL PLAIN* (contraseña en claro) y *SALS DIGEST-MD5,* pero no soporta *TLS*.
Por otra parte, ejabberd solo da soporte *SALS DIGEST-MD5* para el mecanismo de autenticación interna, para autenticación externa ofrece *SASL PLAIN, TLS y SSL.*
Está combinación de factores da como resultado que la autenticación entre el cliente XMPP basado en Strophe.js y ejabberd se realice enviando la contraseña en claro.
La solución adoptada, consiste en emplear como contraseña la cookie de sesión cifrada proporcionada por el servidor web al iniciar la sesión.
Esta solución es mucho más segura ya que no se expone la contraseña del usuario (eliminando el riesgo de usurpación de cuenta), y la cookie de sesión va cifrada por lo que no puede ser recuperada.

Frente a un ataque del tipo "*Man in the Middle*" el mayor daño posible sería la suplantación de identidad en el servidor XMPP (no en el servidor web), sin embargo estaría limitada al periodo de validez de la cookie de sesión.

El método de autenticación por defecto del cliente XMPP basado en el navegador es mediante cookie, sin embargo, se ofrece la posibilidad a los desarrolladores de activar el método de autenticación mediante contraseña.
Ambos métodos son transparentes de cara al usuario final, ya que la cookie cifrada se inyecta en el código JavaScript devuelto, y la contraseña se captura cuando el usuario inicia sesión en la red social, almacenándose en el navegador mediante *SessionStorage*.

LocalStorage y SessionStorage

LocalStorage es un mecanismo que permite almacenar datos persistentes en el cliente, mediante una lista clave-valor alojada en el navegador.
SessionStorage funciona de la misma forma con la salvedad de que los datos son eliminados una vez terminada la sesión del navegador.
Aunque el acceso a la información está restringido al dominio web que la almacenó, SessionStorage no fue diseñado para almacenar datos sensibles.

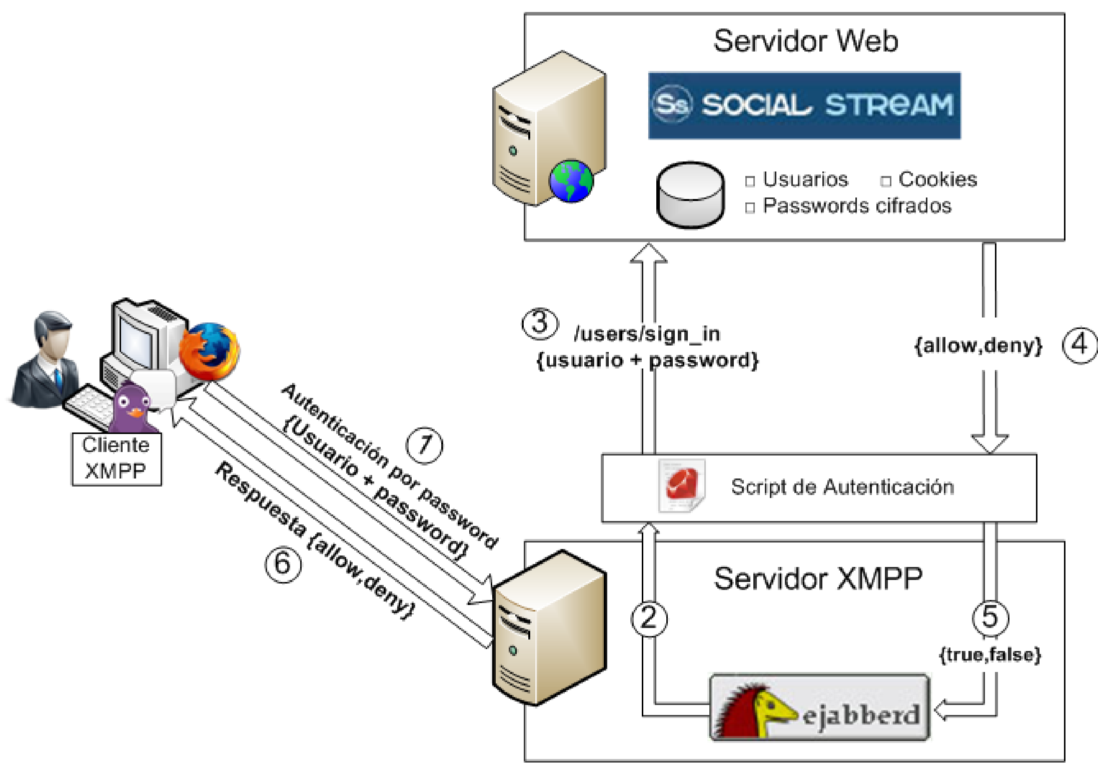


Autenticación basada en cookie cifrada

Para la autenticación basada en cookie cifrada el script realiza una consulta a un API de Social Stream pasando dicha cookie como parámetro, el API devuelve los datos del usuario al que pertenece la cookie en formato XML, por lo que solo tenemos que comprobar si el usuario que intenta autenticarse y el devuelto por el API coinciden.

Dado que al emplear BOSH todos los mensajes son enviados mediante HTTP, otra opción para proveer autenticación segura a los clientes basados en el navegador es emplear **HTTPS**, de esta forma toda la información intercambiada entre el cliente y el servidor XMPP iría cifrada.
No obstante, dado que no es aconsejable almacenar información sensible mediante *SessionStorage*, seguiría siendo aconsejable el método de autenticación por cookie.
El servidor ejabberd soporta *https bind*, para activarlo solo es necesario realizar algunos cambios en la configuración, y dado que las peticiones deben dirigirse al puerto 5281 en lugar del 5280 como ocurría con *http bind*, se deberán realizar cambios en la configuración del proxy.
En este caso se ha optado por no emplear HTTPS, ya que actualmente Social Stream no lo utiliza, sin embargo, se prevé su soporte para un futuro cercano, por lo que es una solución muy interesante a tener en cuenta.

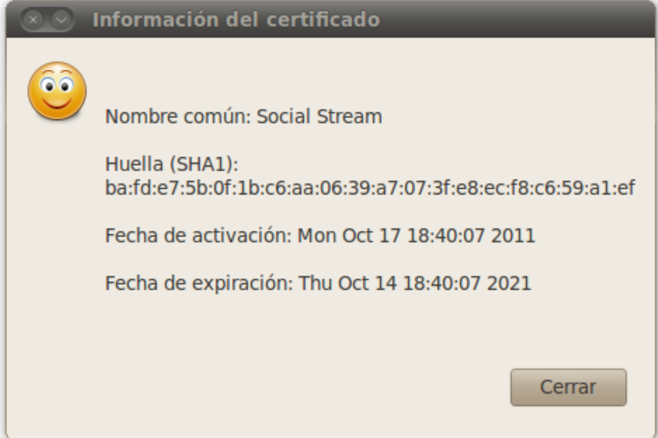
La autenticación de los clientes XMPP externos se realiza siempre por usuario y contraseña. Actualmente prácticamente todos los clientes XMPP soportan TLS, por lo que la autenticación se puede realizar de forma segura enviando la contraseña cifrada.



Autenticación basada en usuario y contraseña

Para la autenticación por usuario y contraseña el script llama al mismo método que utiliza el servidor Web para autenticar a los usuarios que acceden a la red social.

Para emplear TLS el servidor ejabberd debe disponer de un certificado digital que debe de ser aceptado por el cliente.
En este caso se ha generado un certificado autofirmado.



Es importante resaltar que, dado que el servidor de presencia autentica contra el servidor Web, el identificador de nodo de una sesión en ejabberd coincide con el identificador del usuario en la aplicación Ruby on Rails.
Este identificador recibe el nombre de **slug**, y es único para cada uno de los actores (usuarios y grupos) de la red social.

Slug
Es la parte de un URL que identifica a una página empleando palabras legibles. Se utilizan para construir URLs limpias (sin cadenas de consulta) y fáciles de escribir y de recordar. Se usan con frecuencia para construir *permalinks*.

Como consecuencia los *bare JIDs* de los usuarios toman la forma **slug@dominioWeb**.
Estos identificadores facilitan la comunicación entre el servidor Web y el servidor de presencia, ya que para el servidor Web resulta trivial obtener el usuario a partir del JID enviado por el servidor de presencia.

En la siguiente página se ofrecen algunas capturas de tráfico realizadas con Wireshark en diferentes situaciones.

Autenticación en claro con un cliente XMPP externo.

Para este caso desactivamos la opción de cifrado TLS en el servidor, y nos conectamos mediante Pidgin activando la opción de permitir autenticación en claro sobre canales no cifrados.

```
14    Jabber/XML    Request: <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl' mechanism='P

⊞ Frame 14: 277 bytes on wire (2216 bits), 277 bytes captured (2216 bits)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:0
⊞ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ Transmission Control Protocol, Src Port: 44431 (44431), Dst Port: xmpp-client (5222),
⊟ Jabber XML Messaging
  ⊟ eXtensible Markup Language
    ⊟ <auth
        xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
        mechanism='PLAIN'
        xmlns:ga='http://www.google.com/talk/protocol/auth'
        ga:client-uses-full-bind-result='true'>
        AGphY3F1ZWxpbmUtd2lsbGlhbXNvbgBkZW1vbnN0cmF0aW9u
        </auth>
```

La información está en claro y codificada en *Base64*, al decodificar la cadena de texto *AGphY3F1ZWxpbmUtd2lsbGlhbXNvbgBkZW1vbnN0cmF0aW9u* se obtiene el valor:

“*jacqueline-williamson demonstration*”

que se corresponde con el nombre y contraseña del usuario con el que hemos iniciado la sesión.

Autenticación con un cliente XMPP externo empleando TLS.

Volvemos a utilizar Pidgin pero activando TLS tanto en cliente como en servidor.

No.	Protocol	Info
13	Jabber/XML	Request: <?xml version='1.0' ?>
17	Jabber/XML	Response: <stream:features><starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
19	Jabber/XML	Request: <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
21	Jabber/XML	Response: <proceed xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
23	Jabber/XML	Request: \200m\001\003\001\000T\000\000\000\020\001\000\200\003\000\200\a\000
25	Jabber/XML	Response: \026\003\001\0001\002\000\000-\003\001N\234]6y w\227F 3 \004}w\227
27	Jabber/XML	Request: \026\003\001\000\206\020\000\000\202\000\200| \222\024v%@ \202\037

Una vez finalizada la negociación para acordar la utilización de TLS, todos los mensajes son cifrados y ya no podemos entender su contenido.

Autenticación con Strophe.

```
107       HTTP       POST /http-bind HTTP/1.1  (text/plain)

⊞ Frame 107: 814 bytes on wire (6512 bits), 814 bytes captured (6512 bits)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00
⊞ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ Transmission Control Protocol, Src Port: 58730 (58730), Dst Port: http-alt (8080),
⊞ Hypertext Transfer Protocol
⊟ Line-based text data: text/plain
    <body rid='1906588418' xmlns='http://jabber.org/protocol/httpbind' sid='89754ccc3
      <auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl' mechanism='PLAIN'>
        ZGVtb0Bsb2NhbGhvc3QAZGVtbwBkZW1vbnN0cmF0aW9u
      </auth>
    </body>
```

Al igual que en el primer caso, la información va en claro y codificada en Base64. Si la decodificásemos obtendríamos la cookie cifrada o bien el usuario y contraseña, según el método de autenticación empleado.

5.2 Módulo SSpresence

Entre los objetivos del proyecto figura conseguir que el servicio de presencia sea transparente para la aplicación web, esto quiere decir que tenemos que proporcionar a los desarrolladores una forma de acceder al estado de disponibilidad de un usuario desde el entorno Ruby on Rails ajena al funcionamiento del servidor XMPP, es decir, sin la necesidad de realizar consultas explícitas al mismo.
Para conseguir este objetivo la información de presencia de un usuario es almacenada como un campo más en la base de datos (*connected* y *status*), de modo que desde la aplicación Rails, la obtención del estado de disponibilidad de un usuario se lleva a cabo mediante una consulta tradicional a la base de datos.
Por ejemplo, podríamos obtener mediante una consulta a la base de datos, todos los usuarios inscritos a un evento de videoconferencia que se encuentren conectados en ese momento, para notificarles vía mensajería instantánea de alguna novedad o incidencia del evento.
Esta forma de acceso también resulta de utilidad en el renderizado, ofreciendo la posibilidad de alterar el aspecto en función de la información de presencia, por ejemplo, podríamos resaltar a nuestros contactos conectados dentro de una lista de asistentes a un evento con un enlace para iniciar un chat.

Por tanto, la idea es que el servidor de presencia notifique al servidor Web aquellos eventos que le interesen, para esto es necesario dotar de nuevas funcionalidades a ejabberd, y aquí es donde entra en juego el módulo interno *SSpresence*.

Los módulos de ejabberd funcionan como *plugins* y se utilizan para extender la funcionalidad del servidor. Están escritos en Erlang y deben de seguir un comportamiento específico y proveer un API determinada.
Los módulos interactúan con ejabberd empleando uno o varios de los siguientes mecanismos:

- Utilización de los módulos básicos.
  Módulos principales y funcionalidades de ayuda y soporte.
- Eventos y Hooks.
  Cada módulo puede subscribirse a eventos y un *hook* (callback) del módulo será llamado cuando alguno de ellos ocurra.
- Controlador IQ.
  Similar al mecanismo de eventos, pero con mensajes de tipo IQ.
- Tabla de rutas.
- Controlador de peticiones HTTP.
  Un servidor web integrado en ejabberd puede ser extendido mediante sus propios módulos.

En nuestro caso emplearemos el mecanismo de eventos para escuchar aquellos que nos interesen, y algunos módulos básicos que proporcionan funcionalidades de ayuda, como facilidades para la manipulación de paquetes XML.

A modo de ejemplo se incluye a continuación un fragmento de la implementación del módulo correspondiente a la escucha y tratamiento del evento de desconexión de un usuario.

```
start(Host, _Opts) ->
  ?INFO_MSG("mod_sspresence starting", []),
  ejabberd_hooks:add(sm_remove_connection_hook,  Host, ?MODULE, on_remove_connection, 50),
  [...]
  ok.
```

En primer lugar tenemos que añadir el hook, esto se realiza en la función *start* que es invocada al iniciar el módulo, el cual es iniciado al arrancar ejabberd.

El evento *sm_remove_connection_hook* se produce cuando se desconecta un usuario, lo cual, una vez añadido el hook, provocará la llamada a la función *on_remove_connection*.

```
on_remove_connection(_SID, JID, _SessionInfo) ->
  {_A,User,Domain,_C,_D,_E,_F} = JID,
  UserJid = string:join([User, Domain ], "@"),
  ?INFO_MSG("mod_sspresence: on_remove_connection (~p)", [UserJid]),
  Connected = isConnected(UserJid),
  case Connected of
           true -> ok;
           _ -> Rest_api_script_path = string:concat(getOptionValue("scripts_path="), "/rest_api_client_script"),
      os:cmd(string:join([Rest_api_script_path, "unsetConnection", UserJid ], "  "))
  end,
  ok.
```

En primera instancia obtenemos el JID del usuario que ha originado el evento, comprobamos que el usuario no tenga más sesiones activas mediante la llamada *isConnected(UserJid),* ya que el servidor Web solo considera que un usuario está desconectado cuando no tiene ninguna sesión activa, y en caso de no tener más sesiones activas se ejecuta el script *rest_apli_client_script* recibiendo como parámetros el hook "*unsetConnection*" y el JID del usuario.

Esta ejecución origina una llamada al API REST para notificar el evento al servidor Web.

En el próximo capítulo abordaremos con detalle la implementación del cliente y de la interfaz REST.

A modo de resumen se ofrece el siguiente esquema, donde se muestran las diferentes acciones que se llevan a cabo cuando un usuario cambia su estado de disponibilidad.

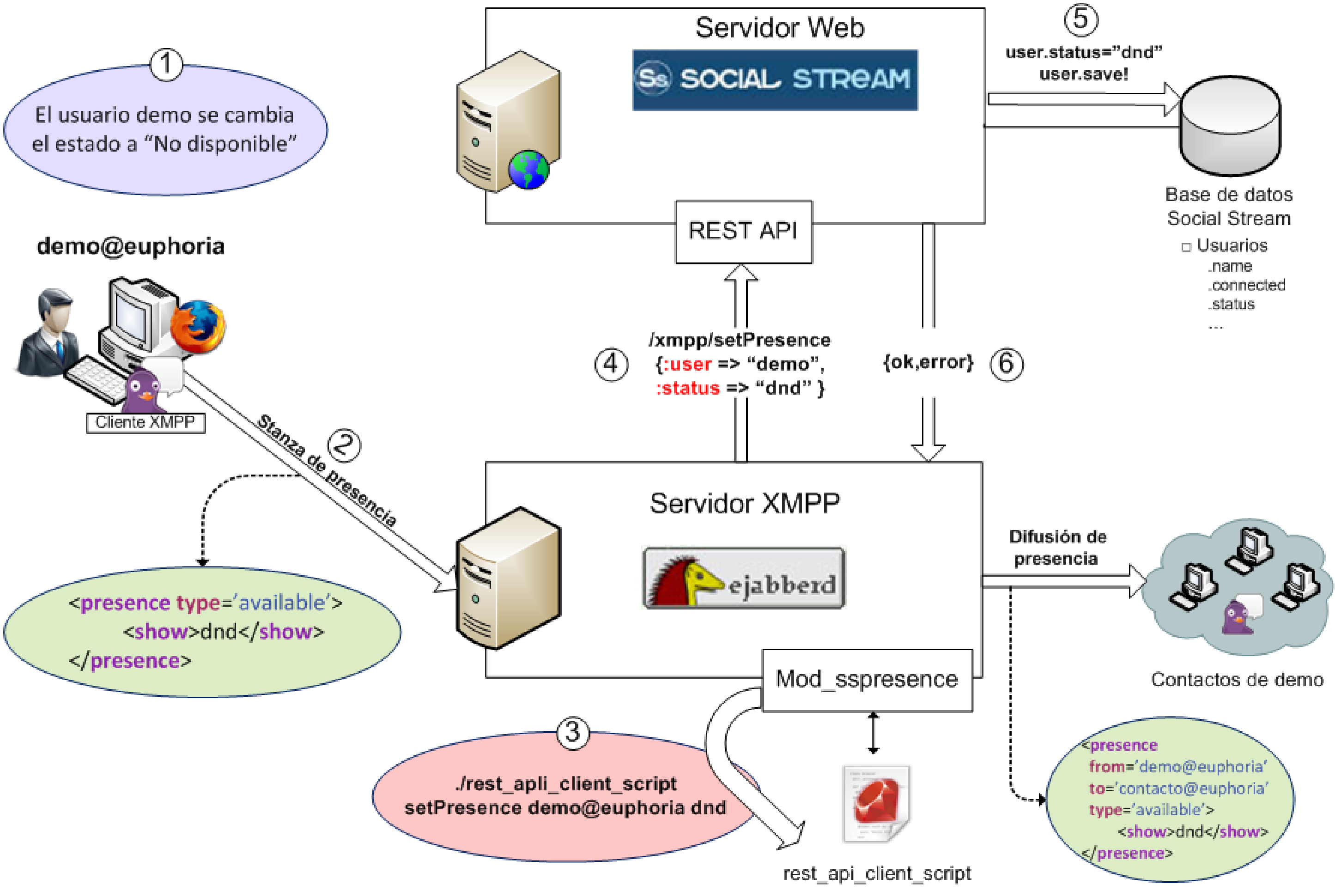


Diagrama de flujo de actividad: cambio de estado de disponibilidad

Otro aspecto a tener en cuenta es que el chat de Social Stream utiliza sus propios estados de disponibilidad, esto no afecta a la interoperabilidad entre clientes, ya que tanto cliente como servidor realizan un mapeo entre los estados estándar y propios:

Estados XMPP	Estados SS
Ninguno	'available'
'chat'	'available'
'away'	'away'
'xa'	'away'
'dnd'	'dnd'

El módulo se distribuye junto con la gema Social Stream Presence y su instalación, parte automática, es muy sencilla y está documentada en la guía oficial.
En el capítulo de despliegue incidiremos más en estos aspectos.

5.3 Emanagement (Ejabberd management)

Ejabberdctl es una interfaz de control del servidor ejabberd, que permite la ejecución de diversas órdenes mediante línea de comandos.
Existen dos módulos que añaden comandos adicionales a ejabberdctl:

- *ejabberdctl-extra:* nuevos comandos para mantenimiento, gestión de usuarios y rosters.
- *mod_muc_admin:* incorpora comandos para la gestión de salas de chat.

En nuestro caso ejabberd empleará ambos módulos, sin embargo, el servidor Web necesita ejecutar ciertas órdenes para gestionar el servidor XMPP no disponibles, en primera instancia, mediante ejabberdctl.
Con el objetivo de posibilitar la ejecución de dichas órdenes, se ha desarrollado el script de Ruby Emanagement que, internamente, interacciona con ejabberd mediante ejabberdctl. Podríamos decir que es una interfaz de mayor nivel de abstracción que extiende la funcionalidad base proporcionada por ejabberdctl.
Cabe plantearse el motivo por el cual se ha desarrollado Emanagement como un script externo en lugar de cómo un módulo plenamente integrado con ejabberdctl, caso de los módulos anteriores.
Esta decisión se ha tomado fundamentalmente por dos motivos:

- Algunos comandos de ejabberdctl tienen una sintaxis muy complicada cuando se incluyen cadenas de texto como parámetro, el entrecomillado y el escapado de caracteres se hacen excesivamente complejos, siendo incluso necesario variar la expresión del comando en función de la máquina donde se ejecuta.
  La interfaz proporcionada por Emanagement abstrae al desarrollador de esta complejidad, permitiendo la ejecución de los comandos con una sintaxis mucho más sencilla e independiente de la máquina donde se ejecuta.
- Como veremos posteriormente, muchos de los comandos son particulares de Social Stream, por lo que no tendría ningún sentido incluirlos en un módulo de ejabberd abierto a la comunidad.

5.3.1 Reflexión de los contactos de Social Stream

Para comprender el funcionamiento del servicio de presencia es importante entender cómo se reflejan los contactos que tiene un usuario de Social Stream en la red social en su lista de contactos almacenada en el servidor de presencia.

Para esta explicación es necesario introducir algunos conceptos básicos de Social Stream, en esta memoria vamos a explicar lo mínimo necesario, pero se puede encontrar una explicación más extensa en la siguiente dirección:

http://rubydoc.info/gems/social_stream-base/frames

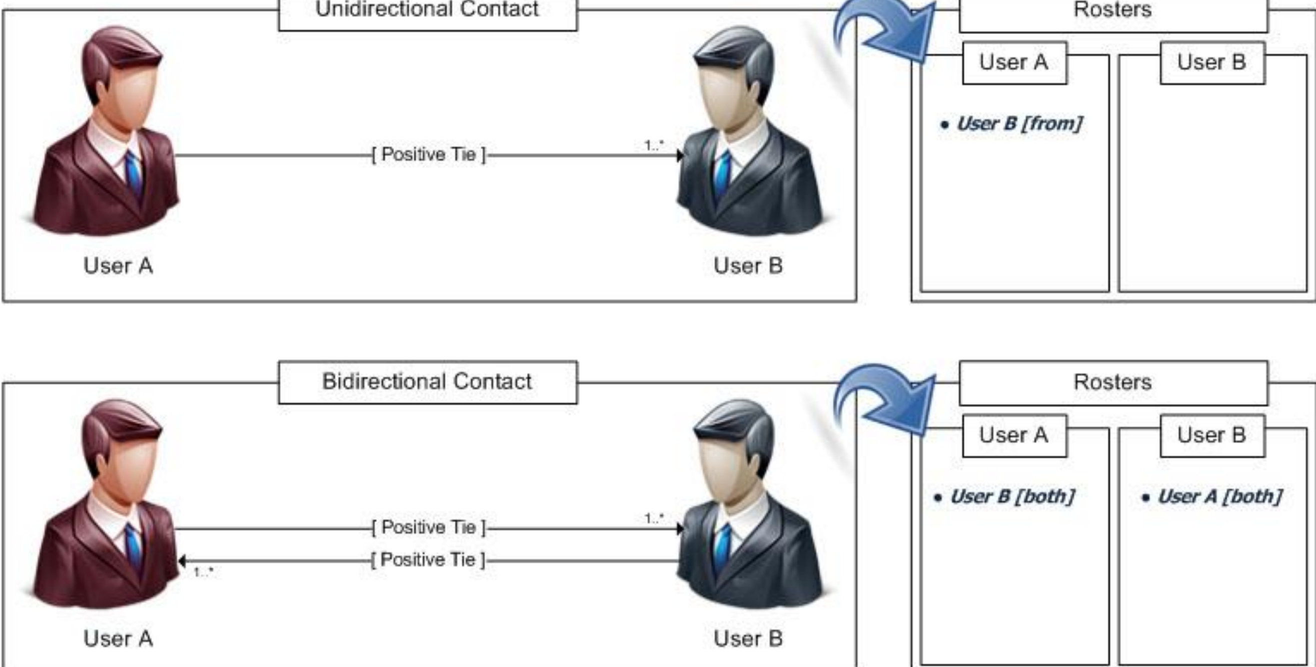
Modelos básicos de Social Stream

- **Actor** es cualquier entidad de la red social, pueden ser usuarios individuales pero también grupos u organizaciones.
- Un contacto (**contact**) es un enlace unidireccional entre dos actores.
  Por lo tanto tienen siempre un emisor y un receptor.
  Se llama **contacto inverso** al contacto establecido en sentido contrario.
- Cada contacto tiene muchos lazos (**ties**), que determinan el tipo de enlace mediante relaciones (**relations**).
- Las relaciones pueden ser afectivas (amigo, conocido,…), biológicas (familiar) o de afiliación a alguna organización (miembro, socio,…).
  A su vez las relaciones se pueden clasificar en positivas y negativas (de rechazo).
  Cada actor puede definir sus propias relaciones.
- Los ties son siempre unidireccionales y pueden ser positivos o negativos, en función de su relación. Cuando un actor establece un tie con otro, implica la concesión de un conjunto de permisos (leer y/o escribir en su muro, etc).
- Se considera que un contacto es positivo cuando tiene ties positivos.

Para esta explicación vamos a diferenciar solamente entre **dos tipos de contactos**:

- **Unidireccionales**: cuando el contacto inverso no está establecido o es negativo.
- **Bidireccionales**: cuando el contacto inverso está establecido y es positivo.

En base a la gestión de subscripciones XMPP vista anteriormente, podemos establecer una correspondencia entre los contactos unidireccionales y bidireccionales de Social Stream y los posibles estados de las subscripciones.



Reflexión de los contactos de Social Stream en las listas de contactos

Es importante el hecho de que esta reflexión solo se lleva a cabo para los usuarios, y no para el resto de actores.
Cuando un usuario añade a otro (***buddy***) como contacto, esto se refleja en su roster como la adición del buddy con el estado de subscripción *from*.
La concesión de permisos del usuario al buddy que tiene lugar en Social Stream se ve reflejada mediante el estado de subscripción *from,* que indica que el buddy tiene subscripción a la información de presencia del usuario.
Se ha optado por no incluir en el roster del buddy al usuario con subscripción *to*, ya que esto provocaría que el buddy recibiese información de presencia que quizás no está interesado en recibir.
Dado que ningún usuario recibe nueva información de presencia tras este cambio, su única utilidad es mantener sincronizadas las bases de datos.

Cuando el contacto es replicado positivamente (convirtiéndose en bidireccional) esto se refleja en la base de datos del servidor de presencia mediante la adición de cada uno de los contactos al roster del otro con subscripción *both*, lo cual permite a cada usuario tener una subscripción a la información de presencia del otro, habilitando la comunicación mediante el servicio de mensajería instantánea.

Hemos visto anteriormente que las subscripciones XMPP se gestionan mediante el envío de stanzas de presencia que siguen un protocolo de petición-respuesta, requiriendo la autorización explícita del usuario. Sin embargo, en nuestro caso la autorización ya se produce explícitamente cuando el usuario envía o acepta una petición de amistad en la red social, por lo que sería redundante volver a exigir al usuario su autorización. Esto nos deja dos opciones posibles, suplantar la identidad del usuario desde el servidor Web para enviar dichas stanzas siguiendo el protocolo XMPP o bien escribir directamente en la base de datos del servidor los datos deseados.
La primera opción añade serios inconvenientes en cuanto a seguridad, dado que el servidor no puede obtener la contraseña en claro de un usuario habría que barajar opciones tales como la inclusión de una contraseña universal o llave maestra.
Por tanto tomamos la segunda opción, la cual implementaremos utilizando Emanagement para la gestión de la base de datos mediante las siguientes funciones:

Emanagement: Funciones de reflexión de contactos	
Nombre	Descripción
addBuddyToRoster	Añade un contacto al roster de otro.
removeBuddyFromRoster	Elimina un contacto del roster de otro.
setBidirectionalBuddys	Refleja un contacto bidireccional en la base datos.
unsetBidirectionalBuddys	Refleja el paso de un contacto bidireccional a unidireccional en la base de datos.
checkBidirecctionalBuddys	Comprueba si un contacto está reflejado en la base de datos como bidireccional.

5.3.2 Reflexión de los grupos de Social Stream

En Social Stream un usuario puede representar a un grupo, lo cual implica que es capaz de realizar acciones en su nombre y con sus permisos, tales como postear en muros o enviar mensajes privados. Sin embargo, no se contempla la posibilidad de que un grupo pueda emplear el servicio de presencia y mensajería instantánea, es decir, no es posible autenticarse como grupo en el servidor de presencia, por lo que no es posible chatear en representación de un grupo ni recibir estados de disponibilidad del mismo.
No obstante, **en Social Stream existe una sala de chat persistente para cada grupo**, de modo que el reflejo en la base de datos es inmediato, cuando se crea un nuevo grupo se crea una sala de chat persistente y cuando se elimina el grupo se borra.
Las salas de chat funcionan acordes a la especificación MUC de XMPP.
Como las salas de chat creadas son persistentes y no tienen dueño, estas nunca desaparecen, ni siquiera al quedarse sin ocupantes, y la única forma de eliminarlas es borrándolas directamente de la base de datos.
Al igual que antes, empleamos Emanagement para gestionar la base de datos:

Emanagement: Funciones de reflexión de grupos	
Nombre	Descripción
createPersistentRoom	Crea una sala de chat persistente.
destroyRoom	Elimina una sala de chat.

El JID de las salas de chat sigue la estructura ***slugGrupo@conference.dominioWeb*** , es decir, el nombre de cada sala de chat creada coincide con el identificador del grupo al que pertenece.

5.3.3 Funcionalidades de Emanagement

Además de proporcionar funciones para la reflexión de los contactos y grupos de Social Stream en la base de datos, Emanagement proporciona muchas más funcionalidades, en el *Capítulo 13: Planos,* se ofrece el listado completo de la interfaz accesible, aunque realmente existen más funciones internas.
Las funciones, de las que a continuación incluiremos algún ejemplo, se pueden clasificar, a grandes rasgos, en los siguientes grupos:

- Funciones de mantenimiento y administración: Impresión de información sobre usuarios, listas de contactos,… con la posibilidad de filtrar por dominios.
- Gestión avanzada de listas de contactos.
- Gestión avanzada para salas de chat.
- Generador de stanzas: permite generar stanzas de presencia y mensajes.
- Otros: cierre de sesiones, anuncios mediante difusión de mensajes, etc.

Obtención del roster del usuario con JID *demo@localhost*:

```
admin@localhost:~/ejabberd_scripts$ ./emanagement getRoster demo@localhost

pamela-hall@localhost    Pamela   both    none    SocialStream
terry-alvarez@localhost  Terry    both    none    SocialStream
lois-johnson@localhost   Lois     both    none    SocialStream
```

Obtención de todos los rosters del dominio *localhost*:

```
admin@localhost:~/ejabberd_scripts$ ./emanagement printAllRostersByDomain localhost

-------------------------------------
jacqueline-moore Roster
-------------------------------------
pamela-hall@localhost    Pamela  both     none     SocialStream
-------------------------------------

-------------------------------------
pamela-hall Roster
-------------------------------------
jacqueline-moore@localhost     Jacqueline     both     none     SocialStream
demo@localhost                 Demo           both     none     SocialStream
-------------------------------------
[...]
```

Comprobar si un contacto está reflejado en la base de datos como bidireccional:

```
admin@localhost:~/ejabberd_scripts$ ./emanagement checkBidirecctionalBuddys demo@localhost
pamela-hall@localhost

true
```

Envío de una stanza de presencia de tipo *available* en nombre de *demo@localhost*:

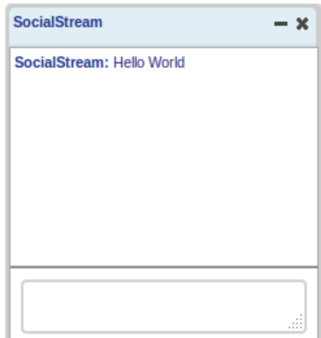
```
admin@localhost:~/ejabberd_scripts$ ./emanagement setPresence demo@localhost

Done
```

Y en la consola del servidor XMPP se obtiene:

```
=INFO REPORT==== 30-Nov-2011::17:43:09 ===
I(<0.665.0>:mod_sspresence:50) : mod_sspresence: on_presence ("demo@localhost")
=INFO REPORT==== 30-Nov-2011::17:43:09 ===
I(<0.665.0>:mod_sspresence:56) :
mod_sspresence: set_presence_script call with  userJid ("demo@localhost") and status ("chat")
```

Envío de un mensaje al usuario *demo* del dominio *localhost* en nombre de *SocialStream*:

```
admin@localhost:~/ejabberd_scripts$ ./emanagement sendMessageToUser
SocialStream demo@localhost "Hello World"

Done
```



5.3.4 Comunicación con el servidor Web

Ya hemos visto como Emanagement puede ser utilizado, entre otras tareas, para reflejar los diferentes actores de la red social y sus relaciones en la base de datos del servidor de presencia. Sin embargo, no hemos explicado todavía cómo se implementa la comunicación entre el servidor Web y Emanagement.

De momento, basta saber que existe una clase llamada **XmppServerOrder** que ofrece una interfaz a la aplicación Ruby on Rails para ejecutar comandos de Emanagement, en el próximo capítulo se explicará en detalle.

En el siguiente diagrama se muestran las acciones realizadas por el servicio cuando un usuario añade un nuevo contacto:

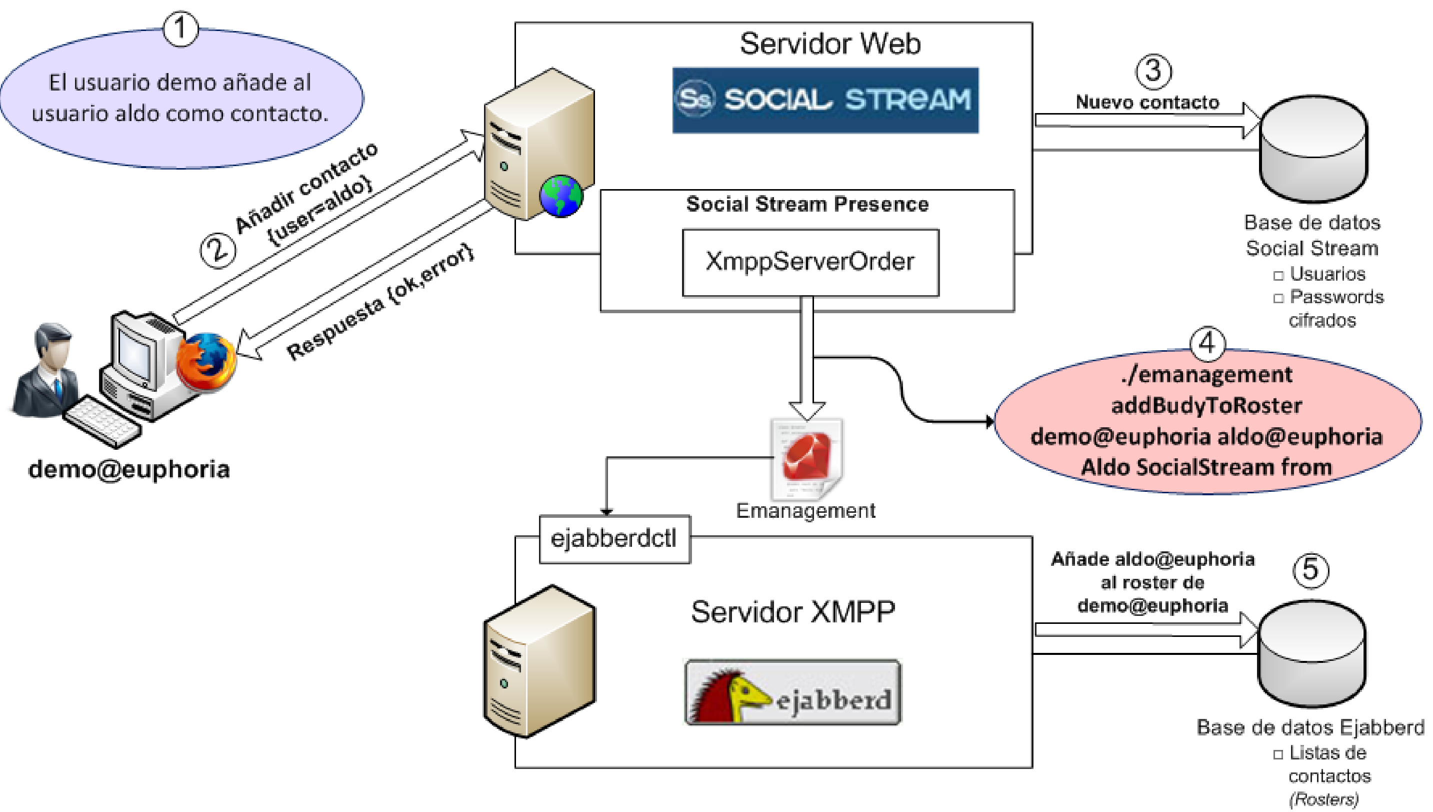


Diagrama de flujo de actividad: creación de un nuevo contacto

6. Servidor Web

Servidor Web

En este capítulo vamos a centrarnos en la configuración de la gema Social Stream Presence, la sincronización de las bases de datos y, especialmente, en los mecanismos de comunicación entre el servidor Web y el servidor de presencia: el *API REST*, la clase *XmppServerOrder* y el cliente XMPP basado en la librería *Xmpp4r*.

6.1 Configuración Social Stream Presence

Existen, básicamente, tres ficheros de configuración:

En el servidor Web:

- El *initializer* de Social Stream Presence.

Un initializer es un fichero responsable de procesar la configuración inicial de una aplicación Ruby on Rails, asignando valores a unas variables de configuración estáticas que, posteriormente, serán usadas por la aplicación.
Al instalar la gema se genera una plantilla del initializer, que el desarrollador deberá completar con sus opciones de configuración específicas.
En este fichero se debe incluir toda la configuración necesaria para el servidor Web. Incluye la dirección del servicio BOSH provisto por ejabberd, el modo de funcionamiento: local/remoto, el método de autenticación para los clientes basados en el navegador: password/cookie, configuración del API REST, etc.
Incluye también una opción para deshabilitar la gema, esto resulta de interés si vamos a utilizar Social Stream (que por defecto incluye Social Stream Presence) en un entorno donde no disponemos de un servidor de presencia, por ejemplo, en desarrollo.

En el servidor de presencia:

- *Ejabberd.cfg*.

Es el fichero de configuración general de ejabberd.
Se utiliza para configurar la autenticación externa, definir los dominios Web servidos por ejabberd, habilitar los diferentes módulos (mod_http_bind, mod_sspresence,…), configurar el servicio MUC, el cifrado TLS, etc.

- *SSconfig.cfg.*

Es creado al instalar la gema y recoge la configuración específica de Social Stream, es accedido tanto por ejabberd como por los diferentes scripts.
Incluye la ruta donde se encuentran los scripts, información sobre los dominios web (identificador y URI base del API REST), el nombre de la cookie empleada en el servidor web, configuración del API REST y configuración de Emanagement.
Existen tareas que permiten configurar el servidor de presencia de forma prácticamente automática desde el servidor Web, esto permite al desarrollador especificar toda la configuración necesaria en un único lugar: el initializer.

6.2 API REST

El objetivo principal del API REST es actuar como medio de comunicación entre los servidores, permitiendo al servidor de presencia notificar los diferentes eventos al servidor Web.
No obstante, también se utiliza el API REST para proporcionar servicios adicionales a los usuarios, como configuración personalizada del chat, videollamadas o juegos multijugador.

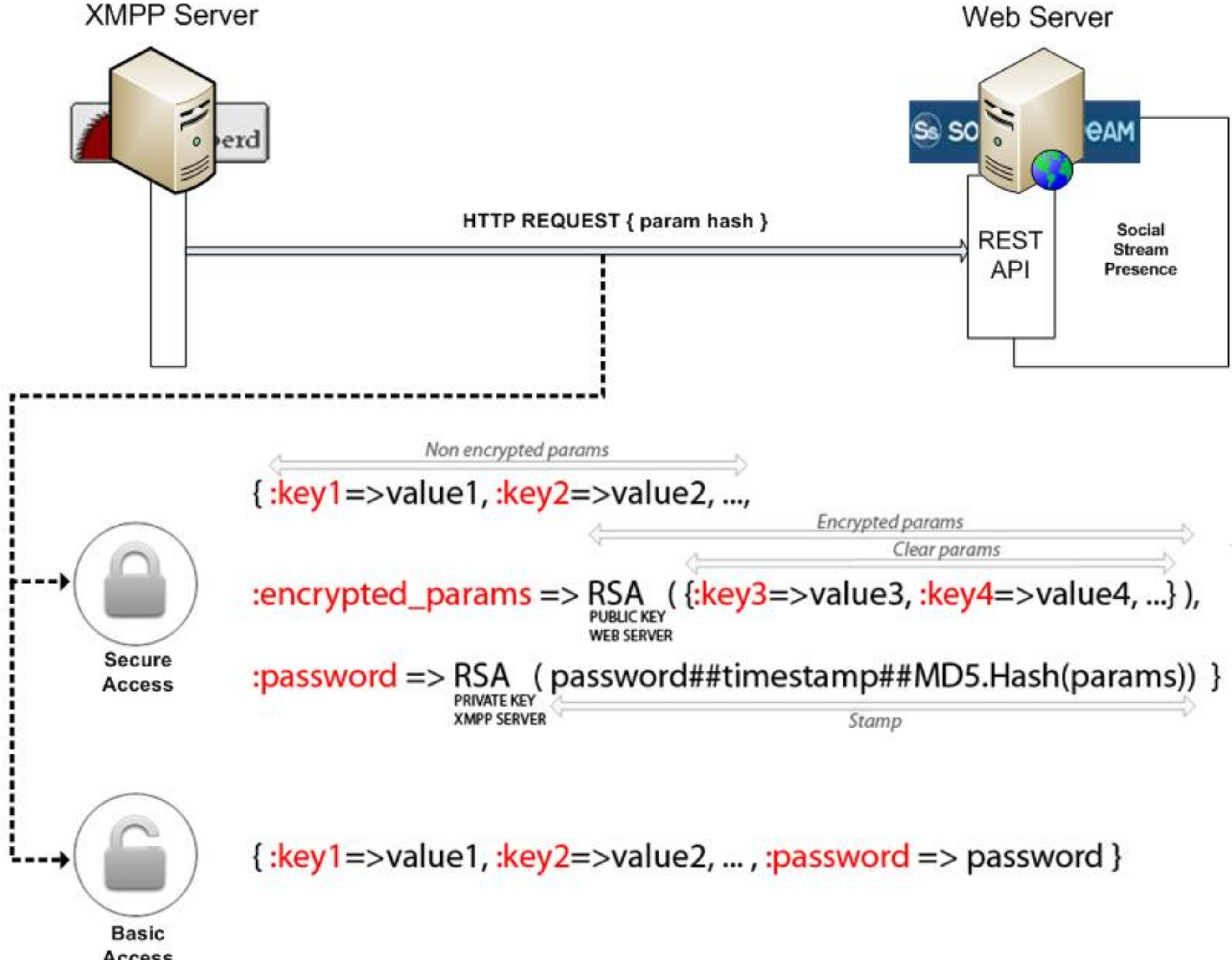
La interfaz REST ofrece, principalmente, las siguientes funciones:

API REST	
Función	Descripción
setConnection	Notifica la conexión de un usuario.
unsetConnection	Notifica la desconexión de un usuario.
setPresence	Notifica el cambio de estado de un usuario.
unsetPresence	Notifica el envío de una stanza de presencia de tipo unavailable.
resetConnection	Marca a todos los usuarios como desconectados.
synchronizePresence	Sincroniza la información de presencia en el servidor Web.
updateSettings	Actualiza la configuración del chat de un usuario.
requestVideoChat	Solicita un identificador de sesión y dos tokens para iniciar una sesión de videoconferencia.
requestGames	Solicita una lista con los juegos disponibles.

La URI base de la interfaz REST es */xmpp/*, y en cuanto a los datos soportados, maneja tanto cadenas de texto como datos en XML.
Los mensajes de confirmación o error se devuelven como cadenas de texto, y las estructuras de datos más sofisticadas, como los datos de videoconferencia o las listas de juegos, se devuelven en formato XML.

Un aspecto muy importante del API REST es la seguridad, las llamadas a las funciones de notificación de eventos de presencia deberían estar restringidas, ya que de lo contrario cualquier usuario podría invocarlas introduciendo datos erróneos en la base de datos.

Social Stream Presence proporciona dos tipos de acceso al API REST sobre HTTP: *básico* y *seguro*, de modo que el formato de las peticiones dependerá del tipo de acceso configurado, en la siguiente página se representa el formato de las peticiones para las dos alternativas.



- *Acceso básico*

Se basa simplemente en la inclusión de una contraseña que acompaña a cada petición al API. Es altamente inseguro, dado que la captura de una petición proporciona acceso total, por lo que solo debe ser empleado en entornos donde no sea posible capturar el tráfico intercambiado. Tanto la contraseña como los parámetros se envían en claro. Está pensado para trabajar con la gema en modo local, donde el tráfico es muy difícil de capturar puesto que se cursa por la interfaz de *loopback,* aunque también puede ser utilizado para trabajar en modo remoto cuando el intercambio de tráfico entre los servidores se realiza a través de redes de confianza. El acceso *básico* también podría ser utilizado si se implementa el API REST sobre HTTPS.

- *Acceso seguro*

Está pensado para trabajar con la gema en modo remoto cuando el intercambio de tráfico entre los servidores se realiza a través de redes desconocidas. También es necesario emplear el *acceso seguro* si queremos permitir la utilización del API REST por parte de terceros, esto puede tener utilidad, por ejemplo, en una arquitectura donde varios nodos de Social Stream emplean un mismo servidor de presencia.

El *acceso seguro* se basa en la inclusión, en cada petición, de un sello (*stamp*) formado por una palabra de control (*password*), una marca de tiempo (*timestamp*) y un resumen (*hash*) de los parámetros, todo ello cifrado con la clave privada del servidor de presencia.

- La palabra de control es un valor constante, que no tiene por qué ser privado.
- El formato de la marca de tiempo es UTC (*Universal Time Coordinated*).
- La función hash utiliza el algoritmo MD5 y se implementa de la siguiente forma:

```
def calculateHash(request_params)
        unless request_params
                request_params = {};
        end

        hash = "";
        request_params.each do |key,value|
              hash = hash + key.to_s + value.to_s
        end
        return Digest::MD5.hexdigest(hash)
end
```

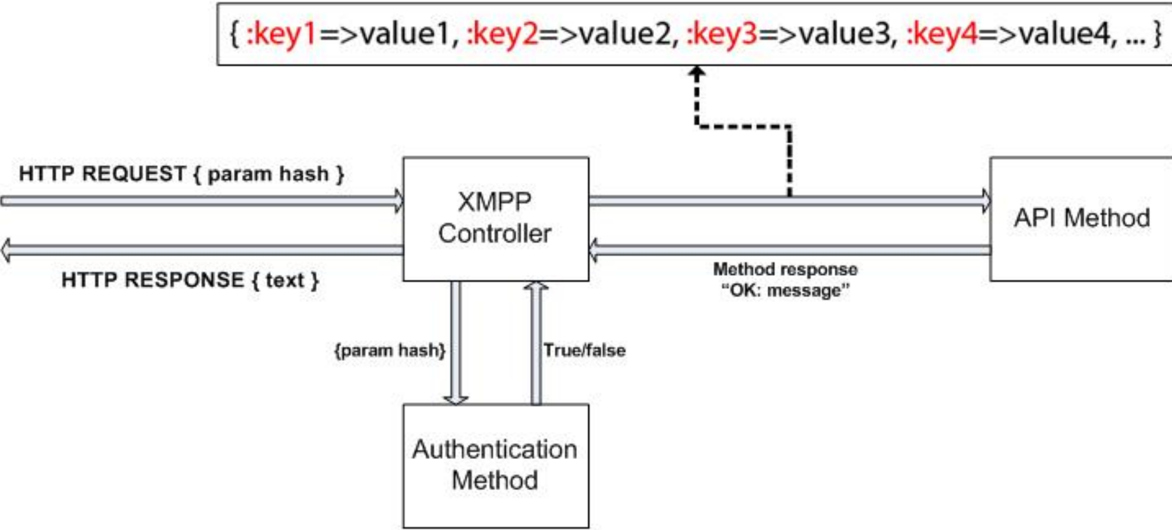
Donde *request_params* es un hash de la forma:

{:key1=>value1,:key2=>value2,...,:encrypted_params=>encryptedValue}

Pese a que la palabra de control, el formato de las marcas de tiempo y el algoritmo de hash son públicos, solo el servidor de presencia puede generar sellos, al estar estos firmados con su clave privada.
Frente a un ataque "*Man-In-The-Middle*", por una parte el atacante no podrá alterar el contenido de las peticiones, ya que una variación de los parámetros produciría que la petición fuese rechazada, y por otra parte, el envío de peticiones repetidas (previamente interceptadas) por parte del atacante está limitado a la duración permitida por la marca de tiempo.
Por tanto, lo único que puede hacer el atacante es replicar peticiones y enviarlas inmediatamente después que el servidor de presencia, produciendo que el servidor Web reciba varias peticiones iguales en espacios muy cortos de tiempo, lo cual, no debería ocasionar ningún problema ya que no altera el contenido de la base de datos.

Podemos incluir en la petición parámetros en claro (si se considera que la información no es sensible), parámetros cifrados con la clave pública del servidor Web, de manera que este sea el único que pueda descifrarlos, o una combinación de ambos.
En teoría es posible recuperar el hash de los parámetros descifrando el sello de la petición con la clave pública del servidor de presencia, no obstante, resulta imposible obtener los parámetros ya que una propiedad de los algoritmos de hash es que no se puede recuperar el mensaje a partir de su resumen.
El acceso seguro garantiza que solamente el servidor de presencia puede acceder a las funciones del API, y además, garantiza que la información sensible irá cifrada y no podrá ser entendida por terceros.

La ejecución de los diferentes métodos del API REST es gestionada por el controlador *XmppController* proporcionado por Social Stream Presence.
El tipo de acceso al API REST es completamente transparente, ya que el controlador proporciona siempre los parámetros en claro, de modo que el desarrollador no tiene que preocuparse del tipo de acceso utilizado al escribir la implementación de las funciones del API REST.



6.2.1 Configuración del acceso seguro

Para habilitar el acceso seguro tanto el cliente (servidor de presencia) como el servidor (servidor web) deben de poseer funcionalidades de cifrado y descifrado, para proveer dichas funcionalidades se emplea la librería *OpenSSL*.
Para activar el acceso seguro solo tenemos que configurar dicha opción en el initializer:

```
config.secure_rest_api = true
```

y configurar las claves RSA.

Rake
Es una herramienta de gestión de tareas de software escrita en Ruby.
Permite especificar tanto tareas como grupos de tareas en un espacio de nombres.
La definición de una tarea consiste en un identificador, una descripción, y el código Ruby a ser ejecutado por la tarea.
Pueden especificar parámetros de entrada y ejecutar otras tareas.

Existe una tarea de rake en el servidor web que permite generar y distribuir de manera automática las claves pública y privada de los servidores Web y de presencia:

```
rake presence:install:generate_RSA_keys
```

Esta tarea, que será ejecutada automáticamente por el instalador inicial si hemos especificado el tipo de acceso seguro, facilita en gran medida la configuración.
Para la generación de las claves se emplea también la librería OpenSSL.

6.2.2 Cliente API REST

La generación de consultas con acceso seguro puede resultar laboriosa, con el objetivo de facilitar esta tarea se proporciona un script escrito en Ruby para generar y enviar peticiones al API REST de Social Stream Presence llamado *rest_apli_client_script.*
Este script recibe como primer parámetro la función del API que se quiere ejecutar, y los sucesivos parámetros serán interpretados como parámetros de la propia función.
El script generará las peticiones de tipo acceso básico o acceso seguro en función de la configuración de la gema.
Este script es utilizado principalmente por el módulo *SSpresence* para notificar eventos al servidor Web.
En nuestro caso el script se ha escrito en Ruby, empleando la gema *REST Client* que proporciona un cliente HTTP y REST muy sencillo, y la librería *openssl* que proporciona funcionalidades de cifrado y descifrado.
A continuación incluimos un ejemplo (disponible en la documentación oficial) de cómo escribir nuestras propias llamadas al API empleando *rest_apli_client_script,* no obstante, el desarrollador es libre de utilizar cualquier otro script escrito en el lenguaje de programación que prefiera.

```
def myHook(param1,param2)
    log($script_title,"Call #{getMethodName}(#{param1},#{param2})")
   url = "http://" + getWebDomainUrlFromDomain(domain) + "/xmpp/hookRoute"

   params = {}
   encrypted_params = {}
   #Add params to sent in clear
   params[:param1_in_server] = param1
   #Add params to sent cipher
   encrypted_params[:param2_in_server] = param2

   return [getMethodName,generic_api_call(url,params,encrypted_params)]
end
```

○ En el hash *params* debemos incluir aquellos parámetros que serán enviados en claro.
○ En el hash *encrypted_params* debemos incluir los parámetros que queremos cifrar, no obstante, si el acceso seguro está deshabilitado los parámetros serán enviados en claro.
○ La cadena de texto "*hookRoute*" debe ser reemplazada por el nombre de la función del API REST que queremos llamar.
○ Finalmente, para realizar la llamada debe ejecutarse el siguiente comando:

```
./rest_api_client_script myHook param1 param2
```

Si definimos *myHook* de forma que coincida con *hookRoute* la relación entre las funciones del script y las del API es directa.

6.3 XmppServerOrder

Es una clase que ofrece una serie de funciones a la aplicación Ruby on Rails para la ejecución de órdenes en el servidor de presencia, estas órdenes son llevadas a cabo mediante la ejecución de comandos de Bash.
Una de sus principales aplicaciones es permitir al servidor Web la ejecución de comandos de Emanagement para gestionar la base de datos del servidor de presencia. Gracias a la interfaz provista por XmppServerOrder el modo de funcionamiento de la gema (local/remoto) es transparente para el desarrollador, ya que este no tiene que preocuparse de la localización del servidor de presencia al llamar a las funciones, la propia clase se encargará de ejecutar los comandos directamente en la consola local o mediante SSH en función del modo de funcionamiento configurado.

A continuación se ofrece una lista de las principales funciones ofrecidas:

XmppServerOrder	
Función	Descripción
setRosterForBidirectionalTie	Refleja en la base de datos del servidor de presencia la creación de un contacto bidireccional.
unsetRosterForBidirectionalTie	Refleja en la base de datos del servidor de presencia el paso de un contacto bidireccional a unidireccional.
addBuddyToRoster	Refleja en la base de datos del servidor de presencia la creación de un contacto unidireccional.
removeBuddyFromRoster	Refleja en la base de datos del servidor de presencia la eliminación de un contacto unidireccional.
createPersistentRoom	Crea una sala de chat persistente.
destroyRoom	Destruye una sala de chat.
synchronizePresence	Sincroniza la información de presencia obteniendo los contactos conectados del servidor de presencia.
resetPresence	Marca a todos los usuarios como desconectados.
synchronizeRosters	Sincroniza las listas de contactos en base a los usuarios.
synchronizeRooms	Sincroniza las salas de chat en base a los grupos.
copyFolderToXmppServer	Copia un directorio o fichero al servidor XMPP.
executeCommands	Ejecuta un comando en el servidor XMPP y devuelve la respuesta.
generateRSAKeys	Genera pares de claves RSA pública/privada para cada servidor.
addWebDomain	Añade la configuración necesaria en el servidor de presencia para dar servicio a un nuevo dominio web.
removeWebDomain	Elimina un dominio web.
updateWebDomain	Actualiza la información de un dominio web existente.

Para ilustrar el funcionamiento de la clase incluimos en la siguiente página un diagrama que representa las diferentes acciones realizadas cuando se ordena la ejecución de una orden mediante XmppServerOrder.

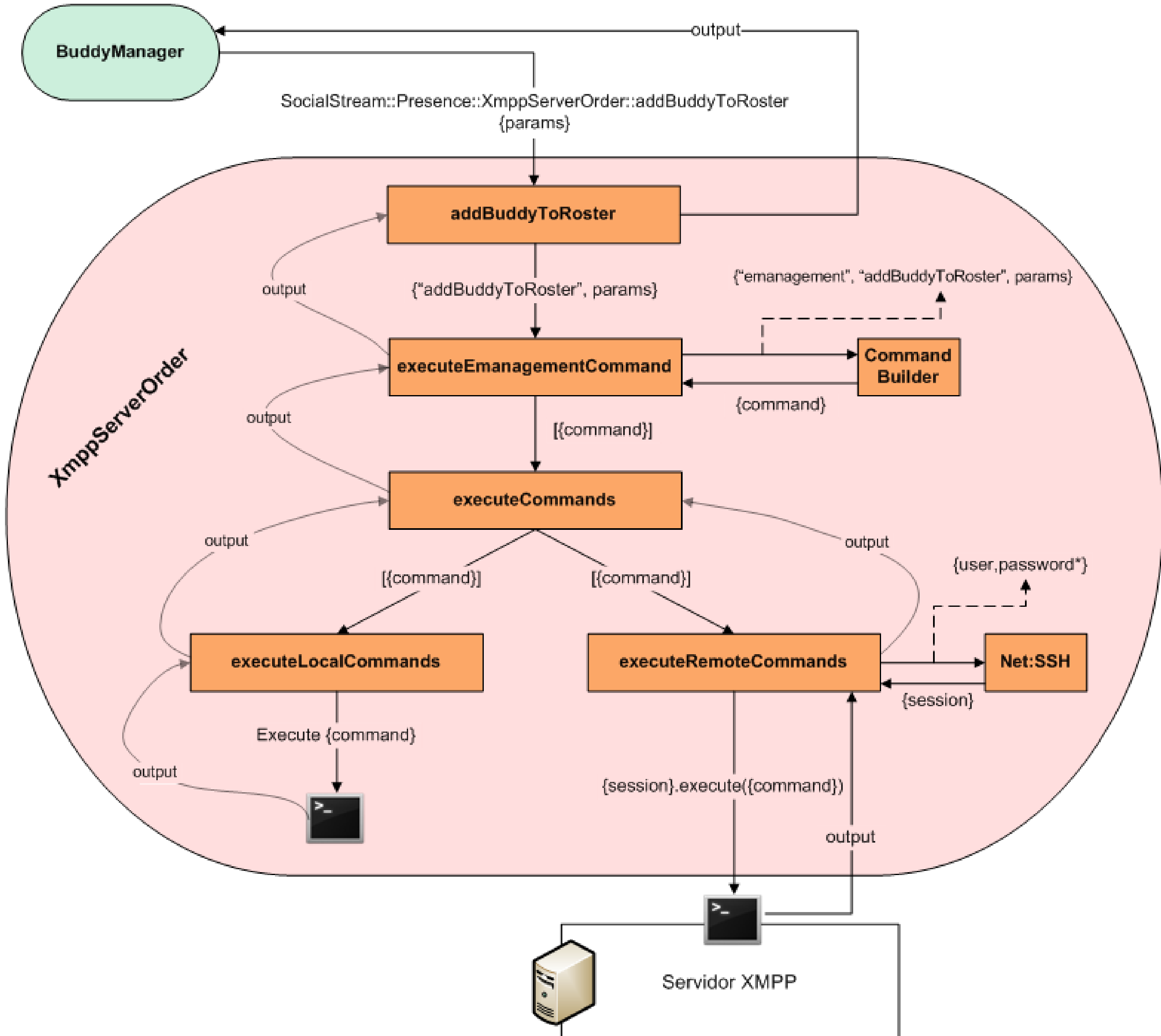


Diagrama de flujo de actividad: Ejecución de una orden mediante XmppServerOrder

Cuando la gema funciona en modo local los comandos se ejecutan directamente en la consola del sistema, sin embargo, cuando la gema funciona en modo remoto se inicia una sesión SSH para acceder al servidor de presencia a través de la red, y ejecutar los comandos mediante su consola. Esto es transparente para el componente que realiza la llamada, en este caso para el módulo *BuddyManager*.
La función *executeCommands* recibe la salida de la consola producida por la ejecución del comando independientemente de la localización del servidor de presencia.
Para permitir la comunicación remota se debe especificar la configuración SSH en el initializer, Social Stream Presence admite autenticación con usuario y contraseña o autenticación mediante usuario y clave SSH.

6.3.1 Captura de eventos

Llegados a este punto podemos realizar una nueva lectura mucho más precisa del diagrama de creación de un nuevo contacto:

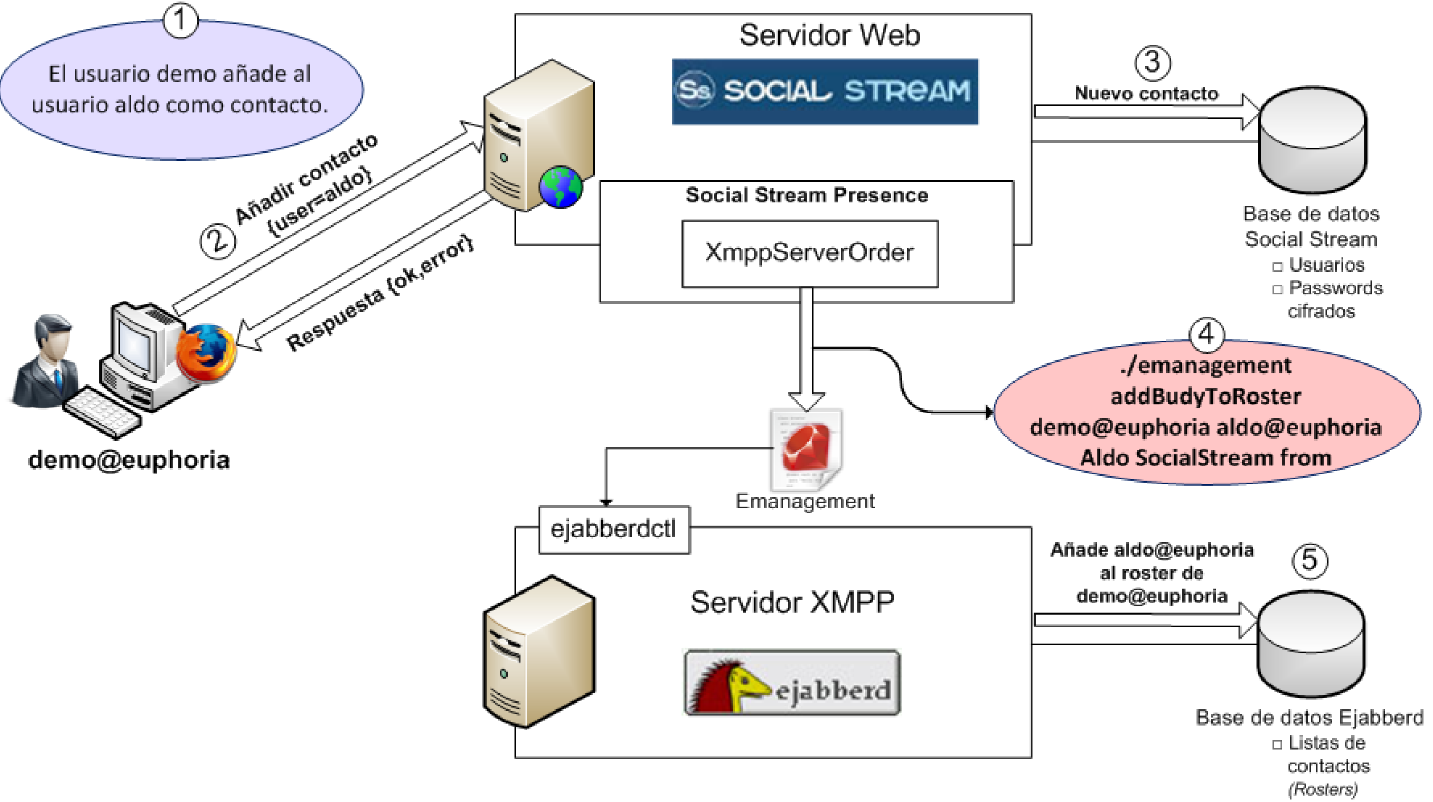


Diagrama de flujo de actividad: creación de un nuevo contacto

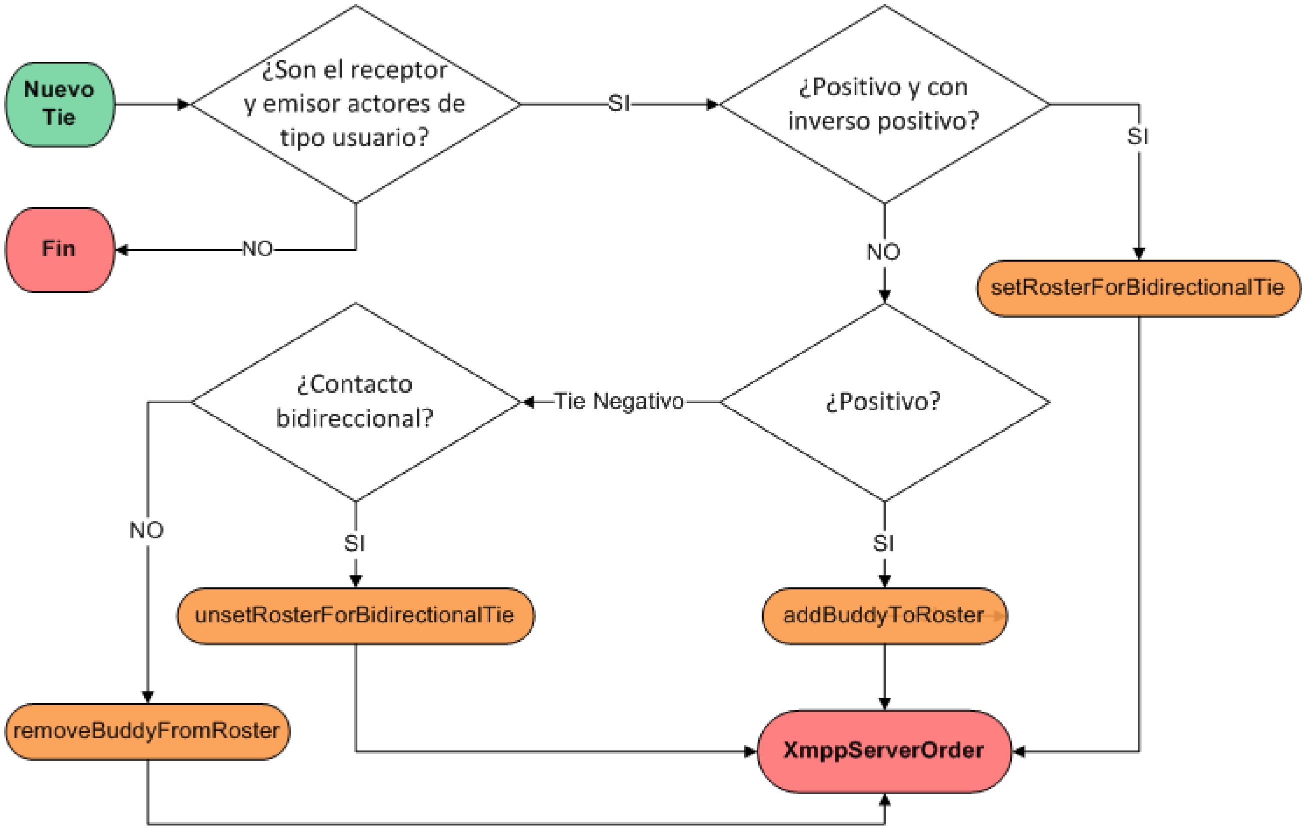
Para comprender el funcionamiento completo del servicio solo falta explicar cómo se implementa la captura de eventos, es decir, cuando se van a llevar a cabo las llamadas a la clase XmppServerOrder y quién las va a realizar.
Social Stream Presence, empleando un mecanismo que ofrece Ruby on Rails, proporciona dos modelos: ***BuddyManager*** y ***GroupManager***, que extienden respectivamente a los modelos **Tie** y **Group** de Social Stream Base.
Estos modelos añaden nuevas funciones y *callbacks* a los modelos originales.

En el caso de GroupManager, se definen dos callbacks: *after_create* y *after_destroy,* que son invocados cuando se crea o destruye un grupo respectivamente.
El callback *after_create* llamará a la función *createPersistentRoom* de XmppServerOrder y el callback *after_destroy* llamará a la función *destroyRoom*.
De esta forma se realizan las llamadas a la clase XmppServerOrder para la creación y destrucción de las salas de chat acorde a los grupos de la red social.

El caso de BuddyManager sigue el mismo principio, pero es ligeramente más complejo, ya que se debe tener en cuenta que los ties pueden ser positivos o negativos y que la creación de un tie puede implicar la creación de un contacto unidireccional o bidireccional.

También se debe tener en cuenta que, en Social Stream, la destrucción del último tie positivo de un contacto siempre va seguida de la creación de un tie negativo.

BuddyManager solo define un callback de tipo *after_create*, que es invocado cada vez que se crea un nuevo tie y cuyo funcionamiento se representa en el siguiente diagrama:



BuddyManager: after create tie callback

6.4 Xmpp4r

Otro mecanismo de comunicación consiste en disponer de un cliente XMPP en el propio servidor Web. Esto se puede conseguir utilizando la librería Xmpp4r, la cual nos va a permitir disponer de un cliente XMPP con plena funcionalidad.
Para iniciar una sesión con el cliente necesitamos un usuario válido, con el cual podamos superar el proceso de autenticación, disponemos de dos opciones:

- Registrar un usuario específico (o varios) en la aplicación web.
- Modificar el script de autenticación para que autentique localmente una serie de cuentas, a las que nos vamos a referir como *cuentas de administración*.

En el primer caso ninguna modificación es necesaria, ya que el cliente podrá autenticarse mediante usuario y contraseña de forma segura empleando TLS.

En el segundo caso debemos introducir un chequeo adicional en el script de autenticación, de forma que antes de consultar al servidor web compruebe localmente si se trata de una cuenta de administración, y en caso contrario continúe con el proceso habitual. Social Stream Presence proporciona facilidades para esta tarea.

Ninguna de las dos soluciones es estrictamente conforme a la arquitectura, ya que en el primer caso estamos registrando como usuario una entidad que realmente no es un actor de la red social, y en el segundo caso el servidor de presencia debe almacenar los credenciales de las cuentas de administración, lo cual rompe con el modelo de autenticación visto hasta ahora.
Otra posible solución, no implementada, sería la utilización de un tercer mecanismo de autenticación, implementado por funciones diferentes del servidor Web, y de forma que este almacenase los credenciales de las cuentas de administración como un nuevo modelo, y no como usuario.

Llegados a este punto cabe plantearse que podemos conseguir con un cliente XMPP en el servidor Web, a continuación se ofrecen algunos ejemplos:

- Notificaciones vía mensajería instantánea.
- Implementación de *bots*: *chatbots*, *game bots*, etc. Ejemplos:
    - *Chatbot* para ofrecer información sobre un evento.
    - Establecimiento de configuración a través de un *chatbot*.
    - Realización de encuestas o estadísticas vía chat.
    - Moderador automático para salas de chat y/o salas de juego.
    - Juegos multijugador con jugadores controlados por un *game bot*.
- Ofrecer la posibilidad a distintos nodos de Social Stream de comunicarse vía XMPP. Cada nodo tendría su propio usuario (que podría ser un nuevo tipo de actor que representase a la red social en su conjunto) con el cual conectarse.
  Podría existir un servidor central o bien interconectar los servidores de presencia de las diferentes redes sociales.

6.5 Sincronización de las bases de datos

Al existir un mapeo entre los actores de la red social y las listas de contactos y salas de chat, existe una relación entre la información almacenada en las bases de datos de los servidores Web y de presencia. Para que esta relación sea coherente y ambas bases de datos representen la misma situación, se deben mantener las bases de datos sincronizadas.

Una forma de conseguir esto es garantizar que los comandos de Emanagement siempre se ejecutan correctamente, y en caso contrario deshacer toda la operación, como si se tratase de una transacción (conjunto de órdenes que se ejecutan formando una unidad de trabajo indivisible).

Sin embargo, Social Stream Presence funciona como un plugin y no es conveniente que interfiera en las operaciones básicas de Social Stream, por lo que deshacer toda la operación no es una opción permitida.

Otra complejidad añadida es que el servidor de presencia puede no estar hospedado en la misma máquina (modo remoto), por lo que la comunicación con Emanagement puede resultar notablemente más lenta, aumentando la latencia en la respuesta del servidor Web.

Una solución posible es emplear un sistema de encolamiento de procesos, de modo que cada vez que queramos ejecutar un comando de Emanagement lo añadiremos a la cola de procesos y será ejecutado en cuanto se tenga ocasión.

Por un lado resuelve el problema de la latencia en la petición, y por otro permite que el comando se ejecute aunque el servidor de presencia haya estado caído durante un cierto periodo de tiempo, ya que una vez se reinicie los comandos se ejecutarán con normalidad.

Social Stream emplea *resque*, una biblioteca respaldada en *Redis* (motor de base de datos basado en el almacenamiento de tablas clave-valor) para la creación, encolamiento y procesado de trabajos en segundo plano.

Si Social Stream Presence va a funcionar en modo local, dada la alta estabilidad de ejabberd y la existencia de tareas administrativas para sincronizar las bases de datos, puede no resultar imprescindible la utilización de un sistema de encolamiento de procesos.

Otro aspecto a tener en cuenta, es que los usuarios pueden modificar sus rosters siguiendo los mecanismos estándar de XMPP, si un usuario añadiese un contacto manualmente desde un cliente XMPP externo la base de datos se desincronizaría.

Esto puede ser inhibido a través del script de autenticación, ya que ejabberd no solo lo invoca cuando un cliente se autentica, sino cuando tiene que autorizarlo para realizar determinadas operaciones como la adición de contactos.

Por tanto, podemos denegar todas las operaciones que modifiquen los rosters.

En cuanto a las salas de chat se dan algunas diferencias.
Cuando un usuario se **une** a una sala de chat, si no existía previamente la **crea** de forma automática, y en caso contrario se une de la forma habitual.
Por tanto, si especificamos que los ocupantes de las salas no tengan permisos de administrador, los usuarios no podrán modificar las salas de los grupos.
Podemos permitir que creen sus propias salas temporales/permanentes, o podemos denegarles dicho permiso configurando correctamente los permisos del módulo *mod_muc* de ejabberd en el fichero *ejabberd.cfg*.

Otra información que conviene mantener sincronizada, aunque es mucho menos crítica, es la información de presencia.
Todas las notificaciones de presencia enviadas por el servidor XMPP requieren de confirmación por parte del servidor Web, además, se implementan dos mecanismos de sincronización:

- Cada vez que se despliega el servidor Web este solicita al servidor de presencia los usuarios conectados y sincroniza su base de datos.
- Cada vez que se reinicia el servidor de presencia, este le indica al servidor Web que marque a todos los usuarios como desconectados.
  La llamada la realiza el módulo *SSpresence* mediante el API REST.

Finalmente cabe mencionar que el servidor Web ofrece al desarrollador tres *tareas de rake* que permiten la sincronización de las bases de datos.

Sincronización de la información de presencia:

```
rake presence:synchronize:connections
```

- Sincronización de las listas de contactos:

```
rake presence:synchronize:rosters
```

- Sincronización de las salas de chat:

```
rake presence:synchronize:rooms
```

Estas tareas se utilizan generalmente en dos circunstancias:

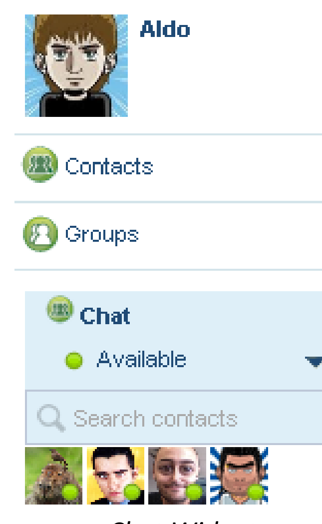
- Poblar la base de datos del servidor de presencia tras instalar Social Stream Presence en una aplicación web en producción.
- Restaurar la base de datos del servidor de presencia en caso de incidencias.

7. Cliente XMPP de Social Stream

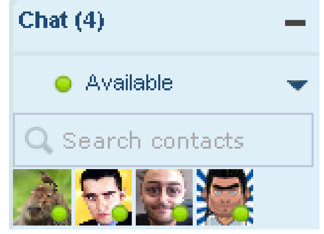
Cliente XMPP de Social Stream

Como parte del proyecto se ha desarrollado desde cero un cliente XMPP basado en el navegador completamente integrado con la red social llamado *Social Stream XMPP Client*.
El cliente se ha escrito completamente en JavaScript, haciendo uso de las múltiples facilidades ofrecidas por la biblioteca *JQuery*, y empleando *Strophe.js* para hablar el protocolo XMPP, para dotar a Strophe de soporte para el servicio MUC se ha utilizado el plugin *Strophe.muc.js*.
El cliente emplea BOSH para la comunicación con el servidor XMPP, dado que Strophe incorpora soporte para BOSH de forma nativa, no es necesario emplear plugins adicionales.

Al estar el cliente escrito completamente en JavaScript el usuario no tiene necesidad de instalar ningún software adicional, además, como hemos visto anteriormente, los credenciales para la autenticación se obtienen de forma automática, tanto para la autenticación mediante usuario y contraseña como para la autenticación basada en cookie cifrada, por lo que la conexión al chat desde el navegador es completamente transparente para el usuario.
Esto implica que cuando el usuario acceda a la red social desde su navegador, el chat aparecerá integrado de forma automática.
Social Stream Presence ofrece al desarrollador la posibilidad de definir en qué páginas desea que aparezca el chat y su modo de visualización.
Existen dos modos: *Chat Widget* y *Chat Float*.
El modo *Chat Widget* está pensado para renderizar el chat en la toolbar, situada en los laterales de la página.
El modo *Chat Float* permite renderizar el chat de forma flotante en la esquina inferior derecha de la pantalla.
De esta forma el desarrollador puede elegir el modelo que más le guste o incluso, renderizar el chat de forma diferente según la página visitada.



Chat Widget



Chat Float

Para añadir el chat a las páginas de Social Stream Base (home, perfil, grupos,…) se utiliza un mecanismo de Ruby on Rails que permite extender las vistas, de forma que podemos renderizar en estas páginas un HTML definido en la gema Social Stream Presence, pasándole parámetros para determinar el modo de visualización.

En la primera sección vamos a realizar una explicación de la arquitectura interna del cliente, y en las secciones posteriores del capítulo iremos recorriendo sus diferentes funcionalidades, incluyendo capturas para ilustrar los resultados conseguidos.

SOCIAL STREAM | Search | Home • Profile • Messages 1 • Notifications 25 • Aldo

Aldo

Contacts

Groups

Files

Chat

Available

Search contacts

You are here | Home: Aldo

Last groups (4)

Social Cheesecake | test123123 | HTML5Party! | Social Stream

Activities

What are you doing?

Post | File | Your contacts | Share

HTML5Party!
MI gato come comida de gato
3 days ago · Comment · I like · Restricted

Rafael García Gallego
How Linux is Built
http://www.youtube.com/watch?v=yVpbFMhOAwE
Infografía-slash-video. Se deja ver y es interesante.
3 days ago · Comment · I like · Restricted
Antonio Tapiador likes this

March 2012

26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22

April 2012

Suggestion

shane X
0 contacts in common
+ Add contact

kevin koym X
1 contact in common
+ Add contact

Chat Widget

SOCIAL STREAM | Search | Home • Profile • Messages 1 • Notifications 25 • Aldo

Aldo

Information

Files

18 Contacts

You are here | Profile: Aldo

Group (4)

Social Cheesecake | test123123 | HTML5Party! | Social Stream

Activities

What are you doing?

Post | File | Your contacts | Share

HTML5Party!
MI gato come comida de gato
3 days ago · Comment · I like · Restricted

Eques and Aldo are now connected.
8 days ago · Comment · I like · Restricted

Aldo added Eques as contact.
8 days ago · Comment · I like · Restricted

Aldo
Social Stream Presence GameCore demo.... http://www.youtube.com/watch?v=SY6KmvDn5S4&feature=youtu.be
26 days ago · Comment · I like · Restricted · Delete

March 2012

26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22

April 2012

Suggestion

Ali X
0 contacts in common
+ Add contact

Pavlo Kuda X
0 contacts in common
+ Add contact

Chat (4)

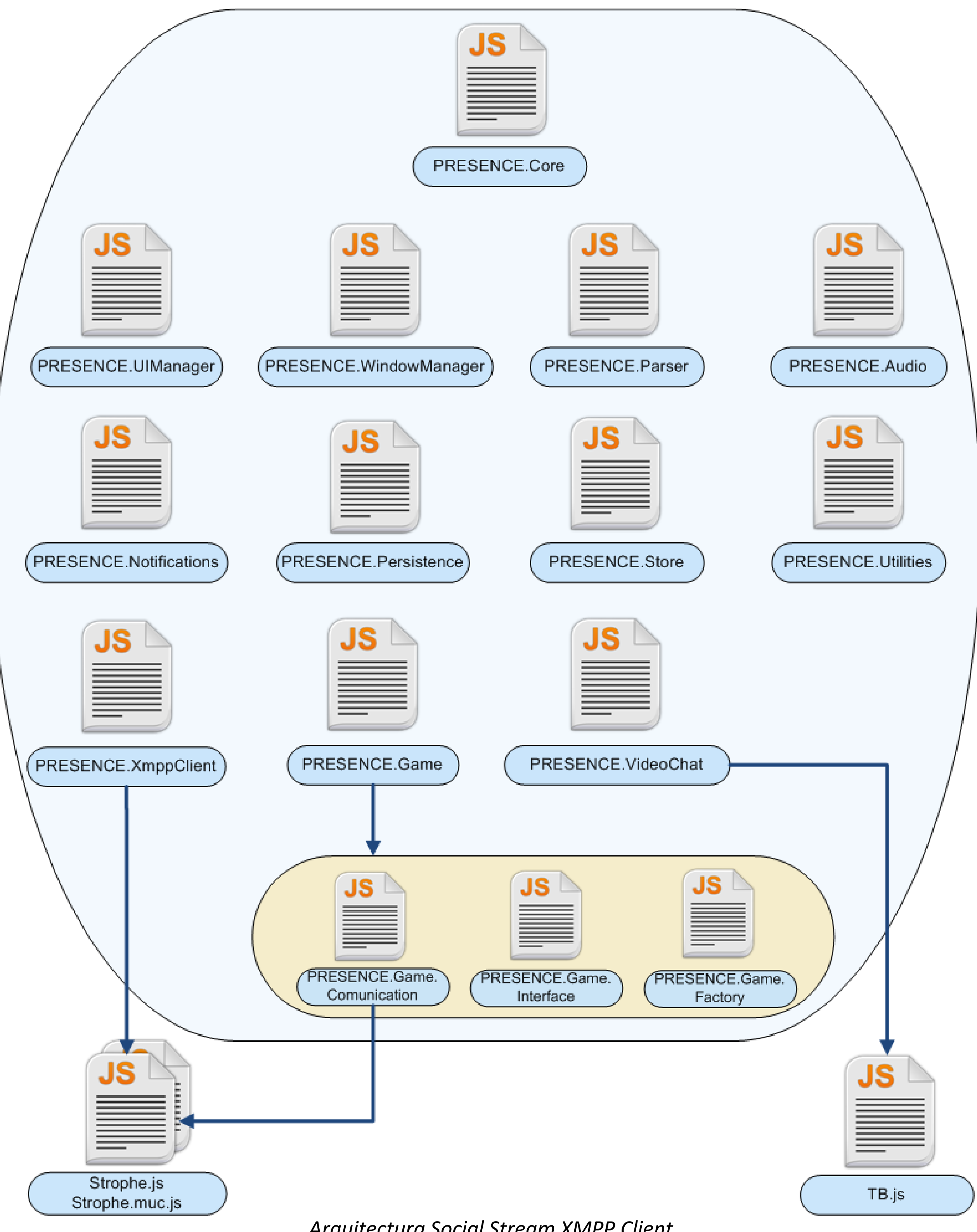
Available

Search contacts

Chat Float

7.1 Arquitectura

Social Stream XMPP Client se ha desarrollado en JavaScript siguiendo el patrón módulo para conseguir una arquitectura modular, en el siguiente esquema se representan los diferentes módulos que lo constituyen:



Arquitectura Social Stream XMPP Client

El **patrón módulo** se basa en la creación de módulos JavaScript mediante unos elementos, llamados ***closures***, que actúan como contenedores, permitiendo la creación en su interior de métodos y atributos públicos y privados.
Un **método privado** es una función a la que solo tienen acceso otros métodos dentro del *closure*, siendo invisible de cara al exterior.
Un **método público** es una función que puede ser llamada desde fuera del *closure* y que define la interfaz del mismo.
Buscando un símil con los términos típicos de la programación orientada a objetos, podríamos decir que un *closure* es una clase, sus atributos son propiedades y sus funciones son métodos.

Las principales ventajas de la utilización de este patrón son las siguientes:

- Extremadamente útil para desarrollar aplicaciones modulares.
- Permite la reutilización de código de una manera prácticamente transparente.
- Código más fácil de mantener: una aplicación JavaScript desarrollada sin ningún tipo de patrón termina convirtiéndose en una serie de variables y funciones repartidas a lo largo del código sin un orden estricto, complicando las labores de mantenimiento, a este tipo de código se le conoce como *código spaghetti*.
- Permite mantener el espacio de nombres global limpio de variables y funciones. Un código lleno de variables globales implica un alto riesgo de colisión, tanto con bibliotecas de terceros como con nuestro propio código.

Desventajas:

- Se necesita emplear un prefijo para llamar a un método público.
  Es decir, la llamada a una función varía si esta forma parte de un *closure*.
- La depuración se hace más complicada.

El patrón sigue el siguiente esquema:

```
// Espacio de nombres para el closure.
var NuevoModulo = {};

// Definición del closure (clase)
NuevoModulo = (function (parametro, undefined) {

  // Variables privadas (propiedades)
  var p1, p2;

  // Métodos
  var unMetodoPrivado = function() {
  }

  var unMetodoPublico = function() {
  }

  // Interfaz pública
  return {
    unMetodoPublico: unMetodoPublico
  }
})(valorParametro);
```

A modo de ejemplo se muestra a continuación la definición del *closure* PRESENCE y del módulo PRESENCE.CORE.

```
var PRESENCE = PRESENCE || {};

PRESENCE.VERSION = '<%=SocialStream::Presence::VERSION%>';
PRESENCE.AUTHORS = 'Aldo Gordillo';


PRESENCE.CORE = (function(P,$,undefined){

  var init = function(){
     PRESENCE.UIMANAGER.init();
     PRESENCE.UTILITIES.init();
     PRESENCE.VIDEOCHAT.init();
     PRESENCE.XMPPCLIENT.init();
     PRESENCE.GAME.init(PRESENCE.XMPPCLIENT.getConnection());
     PRESENCE.AUDIO.init();
  };

  return {
    init: init
  };
}) (PRESENCE, jQuery)
```

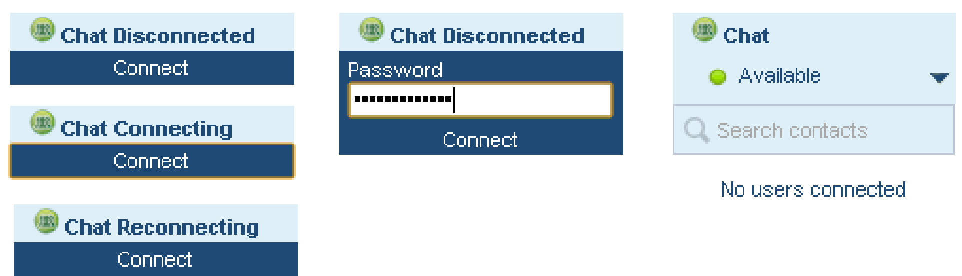
7.2 Módulos

7.2.1 Core

Se encarga de inicializar todos los módulos.
El control de la interfaz se delega principalmente en el módulo UI Manager y la comunicación XMPP se delega en el módulo Xmpp Client.

7.2.2 UI Manager

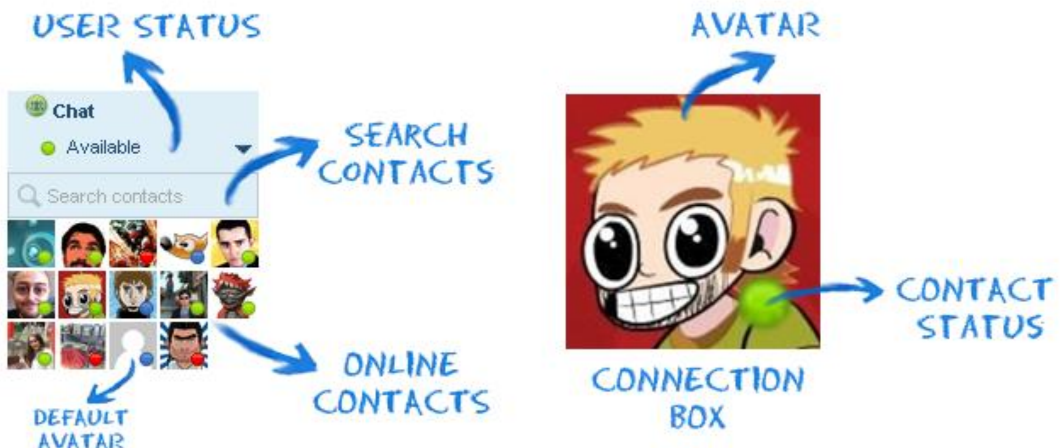
Es el módulo responsable de gestionar la interfaz gráfica:
○ Realiza la carga de paneles y botones (panel de conexión, desconexión,...).
○ Controla los widgets: cambio de estado, búsqueda de contactos, etc.
○ Renderiza los avatares de los usuarios conectados.
Estos elementos reciben el nombre de *connection boxes*.
○ Controla los tooltips.
○ En modo *Chat Float* maneja la caja de chat flotante (*MainChatBox)*.
○ Invoca al módulo Window Manager para crear o abrir las ventanas de chat.
○ Escribe los mensajes en las ventanas de chat.

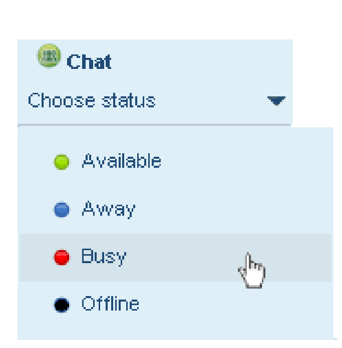


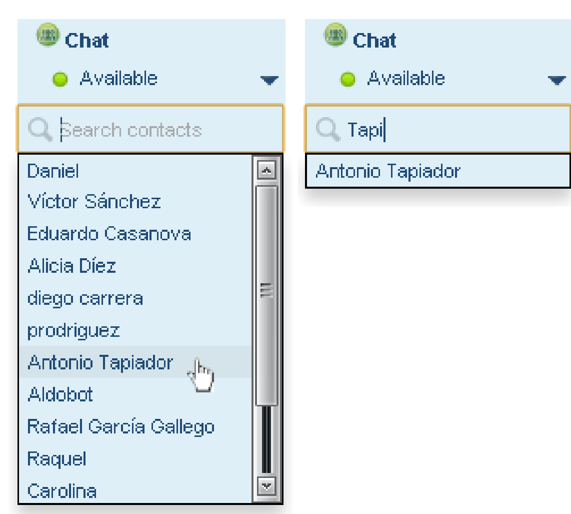
Paneles de conexión

El panel de conexión que incluye un input de tipo password se renderiza cuando se emplea el mecanismo de autenticación basado en usuario y contraseña y esta no ha podido ser obtenida, bien porque el cliente no tiene soporte para SessionStorage o bien porque no hay ningún valor almacenado (este hecho se puede dar, por ejemplo, si el usuario borra los datos almacenados en el cliente).

Ante una pérdida de conectividad con el servidor de presencia, el cliente XMPP realizará periódicamente una serie de intentos de reconexión.







Widget de cambio de estado *Widget de búsqueda de contactos*

Tooltips

7.2.3 Window Manager

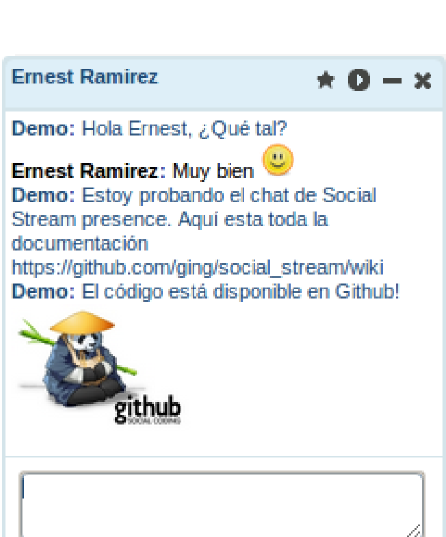
Tiene dos funciones principales:

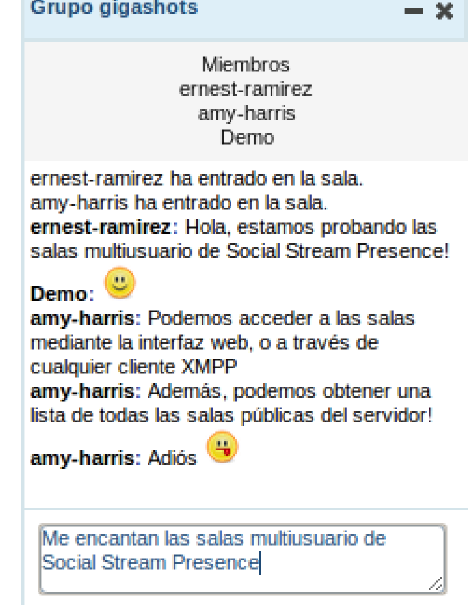
○ Funciona como una factoría para crear ventanas de chat (*chatboxes)*.

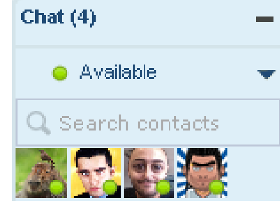
Existen tres tipos de ventanas: para usuarios, para grupos y la *MainChatBox*.

○ Gestiona la colocación de las ventanas en la pantalla, incluye mecanismos de priorización y algoritmos de sustitución.

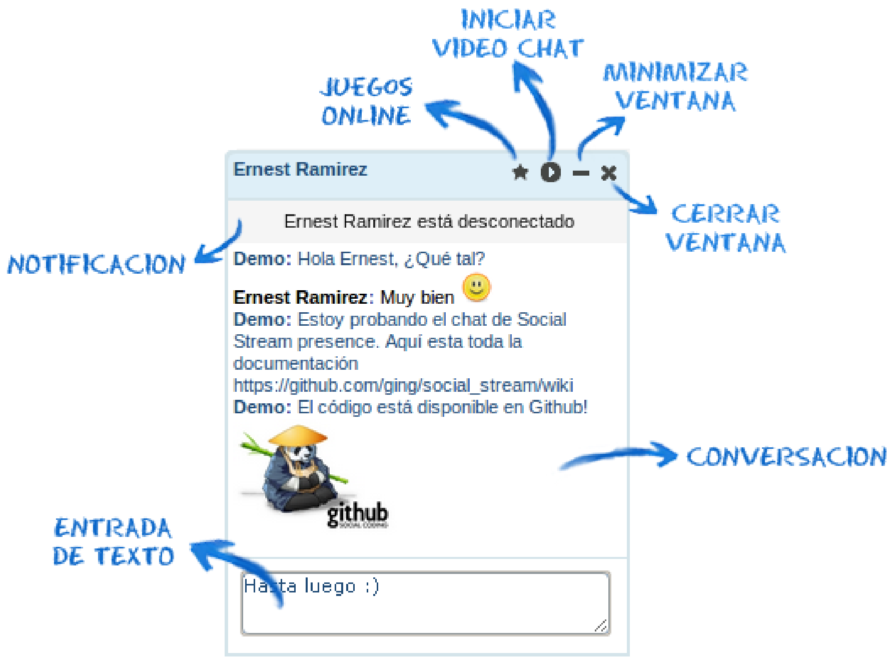
Por tanto, cuando un módulo quiera crear o mostrar una ventana de chat tendrá que contactar con él.







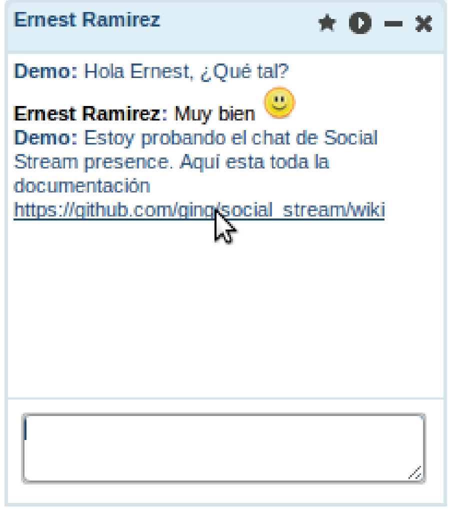
Ventana de chat tipo Usuario *Ventana de chat tipo Grupo* *MainChatBox*



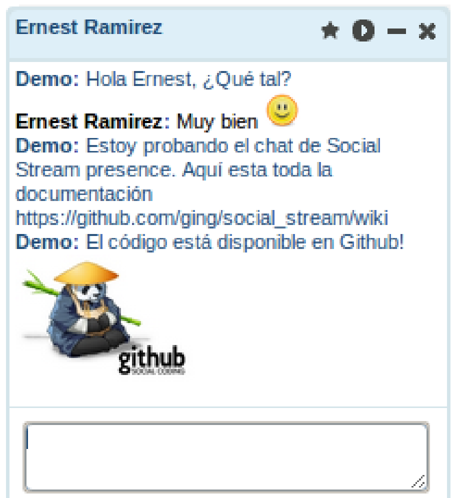
7.2.4 Parser

Analiza el texto de las ventanas de chat.
Funciona como una caja donde se introduce una cadena de texto y sale un código HTML que contiene la cadena de texto enriquecida con elementos multimedia, hipervínculos o metadatos.
Entre sus principales funciones destacan las siguientes:

- ○ Representación de emoticonos mediante imágenes.
- ○ Reconocimiento de urls.
- ○ Inclusión de imágenes.
- ○ Consultas a APIs de terceros: Youtube.
- ○ Protección frente a inyección de código HTML/JavaScript.

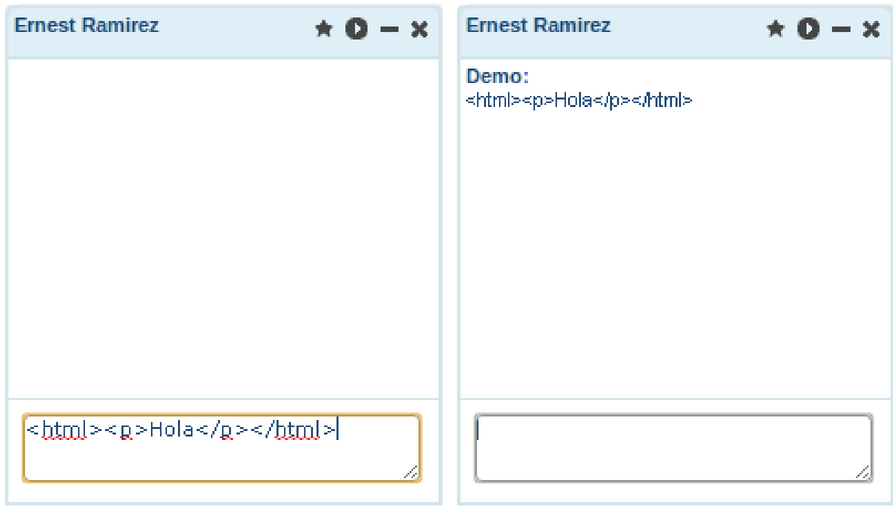


Reconocimiento automático de enlaces



Emoticono e imagen

Consulta al API de Youtube



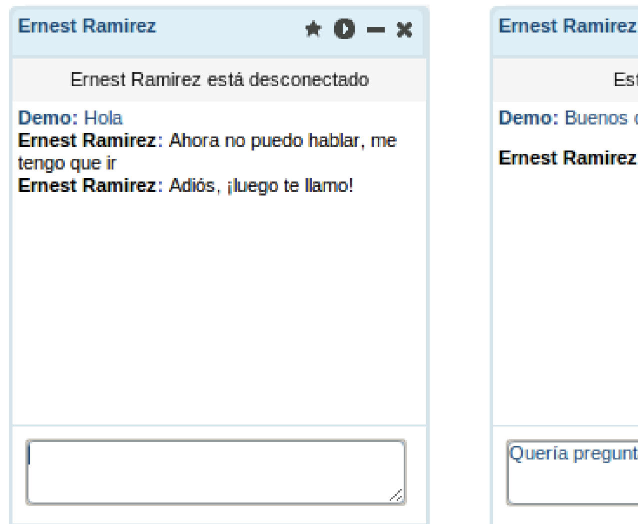
Protección frente a inyección de código HTML

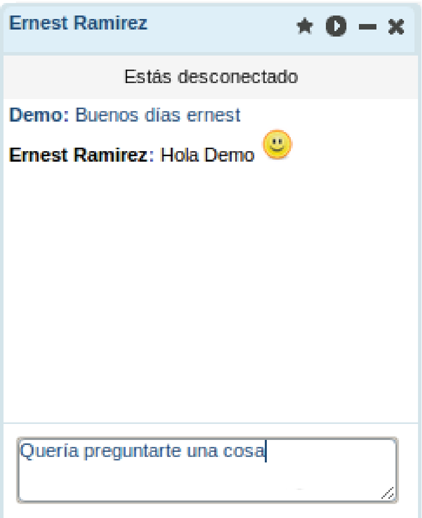
7.2.5 Audio

Controla la reproducción de sonidos del chat.
Los sonidos son emitidos, por ejemplo, al recibir un nuevo mensaje.
Se implementa haciendo uso de la etiqueta Audio de HTML5, por lo que los sonidos se proporcionan en diferentes formatos (mp3 y wav).

7.2.6 Notifications

Gestiona las notificaciones de las ventanas de chat.
Las notificaciones son mensajes cortos que aparecen en la parte superior de las ventanas de chat para informar al usuario de ciertos eventos, como la desconexión de la persona con la que se está conversando. Pueden interaccionar con el usuario, por ejemplo, desapareciendo cuando este las selecciona.
También se utilizan notificaciones, aunque con otra filosofía, para renderizar los ocupantes de las salas de chat.





Notificaciones de desconexión

7.2.7 Persistence

El módulo de persistencia tiene la misión de **almacenar** toda la información necesaria para conservar el estado del chat ante el cierre de la página, y de **restaurar** dicho estado cuando el usuario cargue nuevamente el chat.
Guarda información sobre las ventanas de chat (orden, posición, estado), las conversaciones abiertas, el estado de presencia y las salas de chat a las que se ha unido el usuario.
Los datos se almacenan de manera persistente en el navegador empleando el mecanismo *SessionStorage*, por lo que los datos son eliminados una vez terminada la sesión, es decir, no se guarda información entre sesiones.

Cuando se cierra o actualiza una página web, se termina forzosamente la ejecución de todo el código JavaScript, perdiéndose todos los datos y conexiones establecidas.
El módulo *Persistence* permite que al reiniciar la aplicación del chat se restaure el estado, de forma que el usuario conserve sus ventanas abiertas, conversaciones en curso y estado de disponibilidad.

Esto es de gran utilidad para aplicaciones web donde la carga de páginas no se realiza vía Ajax, por lo que cada vez que el usuario pulsa sobre algún enlace se finaliza la ejecución de código JavaScript.
Para las aplicaciones web donde todas las cargas y actualizaciones de páginas se realizan mediante Ajax el módulo de persistencia simplemente no se invoca.
En este caso no aportaría ninguna utilidad ya que nunca se finaliza la ejecución del JavaScript por lo que no se pierde el estado ni las conversaciones.

La versión actual de Social Stream no emplea de forma generalizada Ajax para la carga de nuevas páginas, por lo que hace uso del módulo de persistencia.
Hay que destacar que aunque se restaure el estado, cada vez que se recarga una página se pierden las sesiones establecidas, tanto la sesión con el servidor de presencia como las posibles sesiones de videoconferencia que el usuario pudiera tener establecidas.
En el caso de la sesión con el servidor de presencia el cliente se reconecta de forma automática y, generalmente, muy rápida al recargar la página, por lo que la experiencia de usuario es altamente satisfactoria.
El establecimiento de las sesiones de videoconferencia, como veremos posteriormente, es más complejo ya que requiere de una negociación previa que involucra a los dos clientes, al servidor Web y a un tercer servidor.
Por tanto, el módulo de persistencia no da soporte a la restauración de las sesiones de videoconferencia.

Cabe plantearse si sería posible que Social Stream Presence proporcionase un mecanismo para que todas las recargas se hicieran mediante Ajax.
En un principio podría confinarse el chat en un iframe y modificar todos los enlaces del resto de la página para que realizasen peticiones Ajax, recargando el contenido mediante JavaScript.
Esto puede resultar bastante complejo, ya que los enlaces, botones u otros elementos HTML pueden tener eventos JavaScript asociados sobre los que la gema no puede tener conocimiento ni control.
Debemos tener en cuenta que Social Stream Presence debe funcionar como un plugin que no debe interferir en el funcionamiento estándar de Social Stream, por lo que no puede aplicar una solución tan intrusiva con la aplicación base.
Sin duda es una opción interesante, pero si se decide aplicar debe ser añadida en el módulo base Social Stream Base.

7.2.8 Store

Se utiliza únicamente cuando se emplea el método de autenticación basado en usuario y contraseña, su función es capturar, almacenar y actualizar la contraseña del usuario mediante SessionStorage para permitir al cliente XMPP iniciar la sesión en el servidor de presencia de forma automática.

7.2.9 Utilities

En este módulo se incluyen todas aquellas pequeñas funcionalidades que no se consideran lo suficientemente importantes como para constituir un módulo independiente.
También se incluyen funciones de soporte o ayuda utilizadas por la aplicación.

A continuación ofrecemos algunos ejemplos de funciones incluidas en el módulo de utilidades:

○ Gestión de trazas de depuración.
○ Parpadeo del título de la página: utilizado cuando se reciben nuevos mensajes y la página no tiene el foco.
○ Control de tormentas de texto: recibidas y enviadas.
○ Control de las animaciones de las ventanas de chat: se inhiben bajo determinadas circunstancias.

7.2.10 Xmpp Client

Es el módulo responsable de la comunicación mediante XMPP, tanto con el servidor de presencia como con otros clientes XMPP. Hace uso de la librería Strophe.js.

Entre las múltiples tareas que realiza podemos destacar las siguientes:

○ Gestiona la sesión con el servidor, que es única para toda la aplicación.

○ Proporciona las funciones de conexión, reconexión y desconexión para todos los tipos de autenticación.

○ Gestiona los callbacks de conexión: conectado, error de conexión, error de autenticación, desconectado, etc.

○ Almacena los datos del usuario: estado de disponibilidad, lista de contactos con información detallada de cada uno (recurso, tipo de cliente, sesiones de videoconferencia establecidas, etc.) y salas de chat abiertas con información detallada de cada una de ellas (ocupantes, apodo, rol y afiliación).

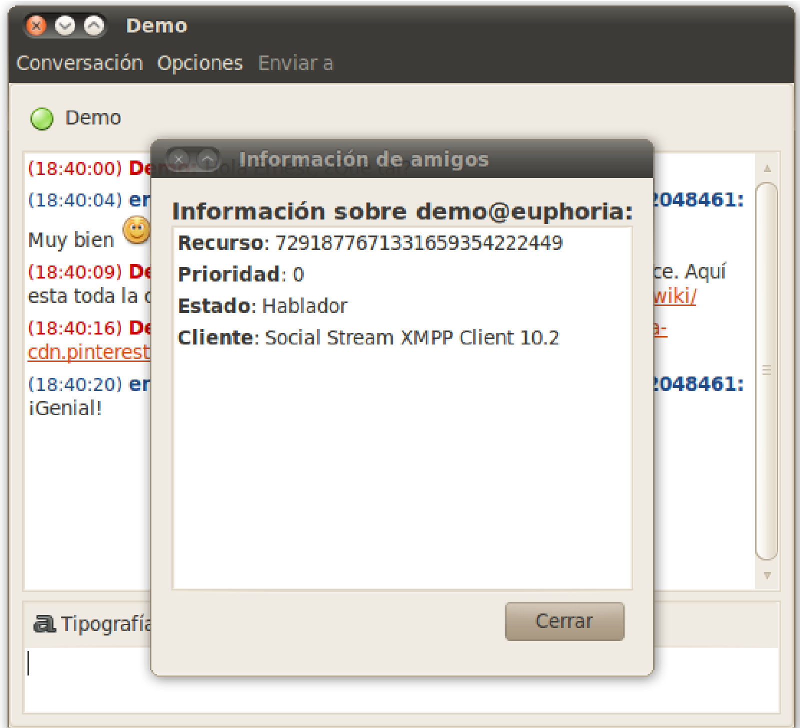
○ Genera y envía las stanzas de tipo mensaje, presencia e iq.

○ Recibe y procesa las stanzas de tipo mensaje, presencia e iq.

○ Gestiona el servicio MUC: unión/abandono de salas de chat, envío/recepción de mensajes, envío/recepción de presencias, etc.

○ Implementa la interacción de peticiones y respuestas IQ que sirve como base para el establecimiento de las sesiones de video chat.

Por ejemplo, si se recibe una consulta IQ para conocer la información del cliente, el módulo XMPP Client responderá con otra stanza IQ portando la información solicitada: nombre del cliente y versión (cuyo valor coincide con la versión de la gema).

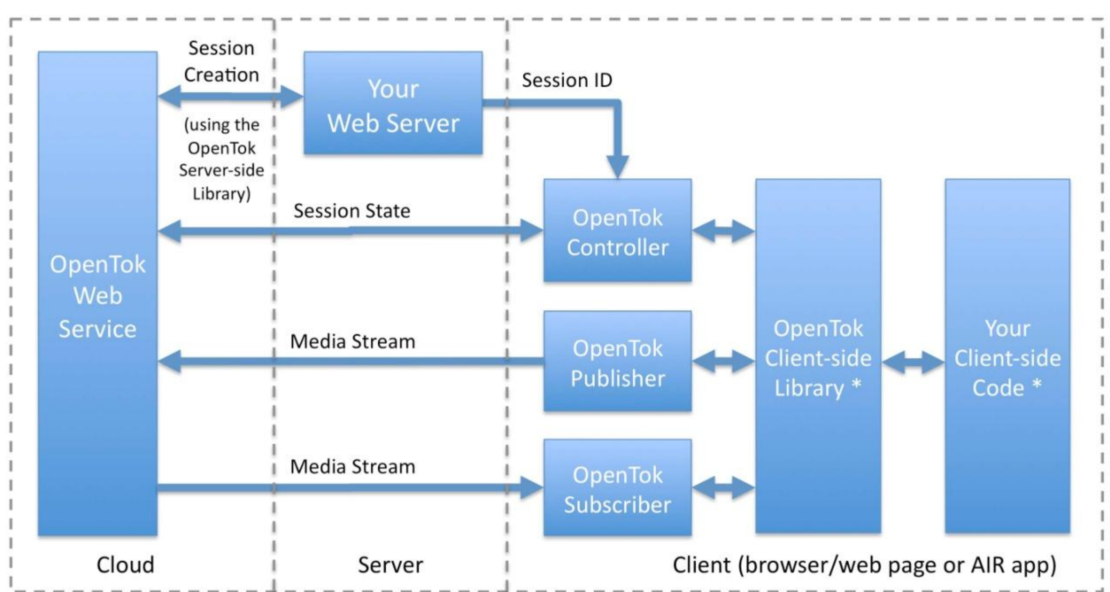


Obtención de información con Pidgin de un usuario conectado con Social Stream XMPP Client

7.2.11 Video Chat

Este módulo proporciona la funcionalidad de video chat, entendiéndose como tal la comunicación en tiempo real entre dos personas con capacidades de audio y video.
Para proporcionar esta funcionalidad Social Stream Presence utiliza el servicio **OpenTok**, que proporciona una API diseñada para añadir funcionalidades de videoconferencia a las páginas web.
Aunque OpenTok da soporte para videoconferencias con múltiples participantes, en nuestro caso, la comunicación siempre se va a realizar entre dos personas.

OpenTok



La plataforma OpenTok se basa en tres conceptos clave: streams, conexiones y sesiones.

Stream

Un stream es una señal individual de audio y vídeo.
El caso más habitual de un stream es aquel cuyas fuentes son la webcam y el micrófono de un usuario.

Conexión

Es el mecanismo a través del cual el navegador publica streams en una sesión y se subscribe a los streams publicados por otros en una sesión.

Sesión

Es una colección de conexiones que publican y se subscriben a streams.
También se encarga de distribuir los eventos que representan cambios en la sesión.

La plataforma OpenTok emplea Flash para acceder a la cámara y para transmitir y mostrar el audio y video. No obstante, todos los objetos Flash se crean y destruyen de forma transparente al desarrollador, que solo necesita emplear las librerías del cliente para acceder al API de OpenTok, que proporciona un control completo de las sesiones de video.
La conexión entre el cliente y el servidor OpenTok se realiza mediante Flash Sockets, empleando el protocolo RTMP (Real Time Messaging Protocol) que ofrece descriptores para transmitir audio y video sobre un socket TCP, y define el formato de codificación de los mensajes para reutilizar el socket para control y señalización.

Las sesiones se identifican mediante un *sessionId* y deben ser creadas por el servidor Web utilizando para tal propósito las librerías de servidor de OpenTok.
De igual forma, el servidor Web también debe generar los *tokens* con los que se tienen que autenticar los navegadores al conectarse a una sesión.
Los tokens tienen un periodo de validez, de modo que cada vez que un navegador interactúa con el servicio web OpenTok, este comprueba que el token de autenticación sigue siendo válido y que la acción solicitada está permitida, en caso de que el token haya expirado la acción no se lleva a cabo.
En el lado cliente se va a emplear la librería JavaScript de OpenTok (*TB.js*), y para el lado servidor se va a emplear la versión Ruby.
Por tanto, desde el cliente gestionaremos todos los objetos Flash mediante JavaScript.

Generalmente cada usuario abrirá una sola conexión a una sesión dada desde su navegador. A través de esta conexión, el navegador puede subscribirse a uno, algunos o todos los streams disponibles y, empleando esta misma conexión, puede publicar un stream a la sesión.
En nuestro caso uno de los usuarios (*Emisor*) solicitará la creación de una sesión al servidor Web e invitará a otro (*Receptor*) a iniciar una sesión de video chat.
Una vez creada ambos se unirán a la sesión y publicarán (previa autorización del usuario) su stream.

Integración de OpenTok

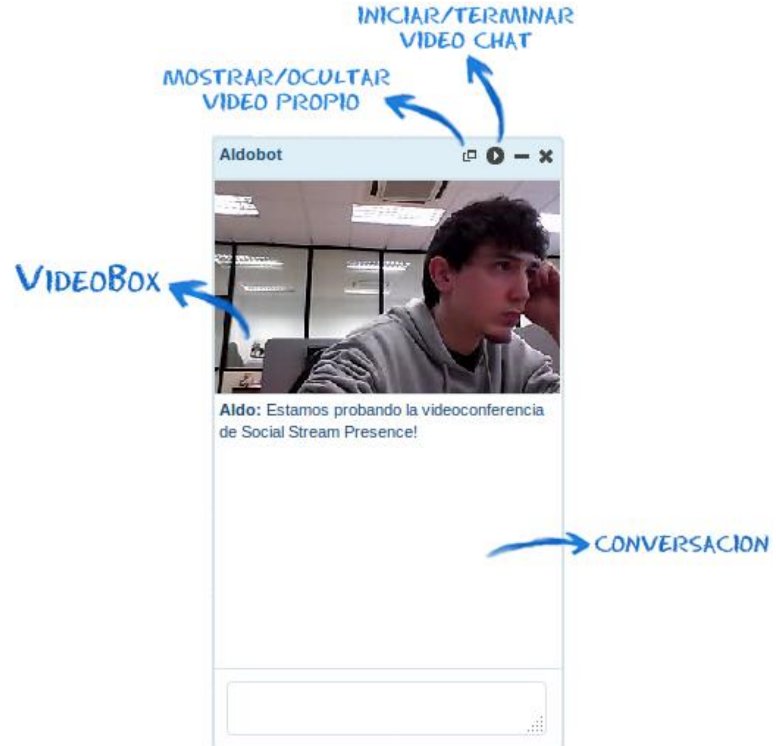
Una de las principales funcionalidades del módulo es la integración de OpenTok con el chat de Social Stream. Para ello lleva a cabo diversos cometidos:

Almacenamiento de información

Guarda información relacionada con la negociación y establecimiento de sesiones de video chat con todos los usuarios: estado de conexión del servicio, identificadores y objetos de sesión, tokens, etc.

Gestión de la interfaz del servicio de video chat.

Al contrario que sucedía hasta ahora, la interfaz relacionada con el servicio de video chat será controlada de manera independiente en lugar de por el módulo UI Manager. Para evitar conflictos, el ámbito de este módulo está limitado a modificar el contenido de un recinto cerrado llamado *VideoBox*, consistente en una extensión de la ventana tradicional de chat. El acceso y control de este recinto, al igual que ocurre con el resto de áreas de las ventanas de chat, se realiza de manera exclusiva a través del módulo Window Manager.
Dado que el módulo Video Chat solo tiene jurisdicción sobre un recinto de la interfaz, para realizar cambios sobre otros componentes (*avatares, tooltips, notificaciones,…*) deberá utilizar los mecanismos tradicionales ofrecidos por los otros módulos.
Este patrón de diseño permite modularizar no solo la funcionalidad, sino también la gestión de la interfaz, de este modo mantenemos lo más limpio posible el módulo principal de gestión de interfaz UI Manager.



Establecimiento de las sesiones de video chat

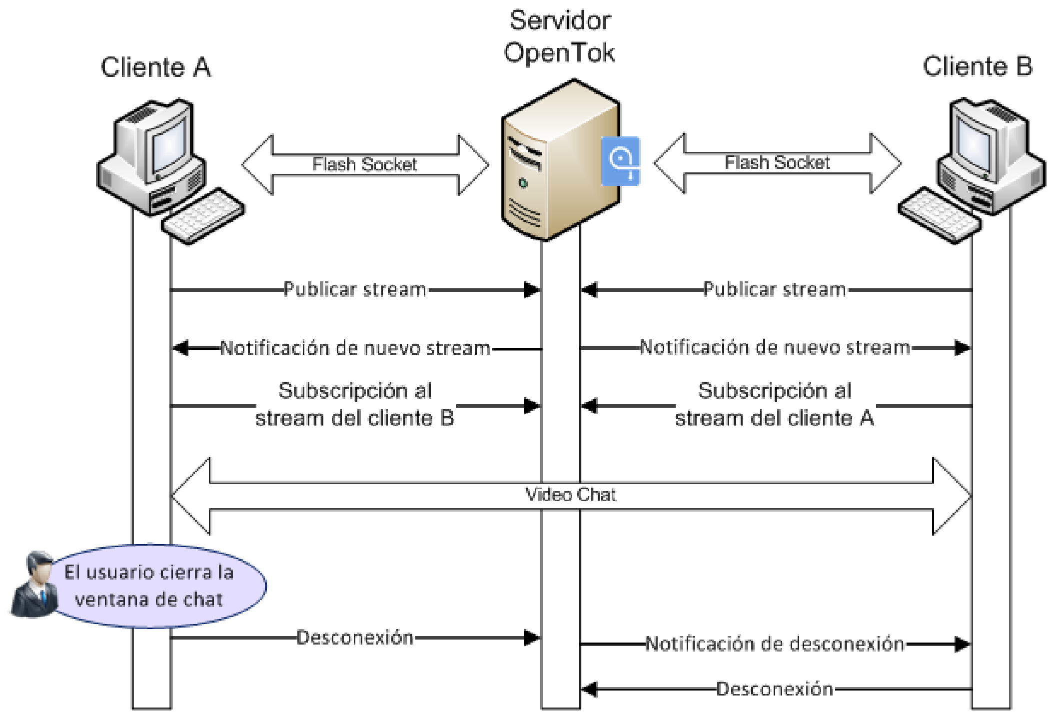
Aprovechando la sesión establecida con el servidor de presencia, el establecimiento de la sesión de video chat se llevará a cabo mediante XMPP, los parámetros necesarios para conectarse a las sesiones de OpenTok (identificadores de sesión y tokens para ambos participantes) se obtendrán del servidor Web mediante peticiones al API REST.
Existe una extensión del protocolo XMPP llamada *Jingle*, que permite la transferencia p2p de datos multimedia permitiendo la adopción de servicios de videoconferencia. Sin embargo, *Strophe.js* no tiene soporte para Jingle y no existe en la actualidad ningún plugin que lo proporcione, además el servicio de video chat, al emplear OpenTok, solo funcionaría entre clientes XMPP de tipo Social Stream, por lo que emplear un protocolo estándar no aportaría ninguna ventaja.
Por tanto, se ha optado por implementar unos mecanismos propios de establecimiento de sesión y negociación basados en el envío y respuesta de stanzas XMPP de tipo IQ.
Los procesos de generación, envío e interacción de las peticiones y respuestas IQ son proporcionados por el módulo Xmpp Client.

El módulo implementa el mecanismo de establecimiento de sesión como una máquina de estados, estableciendo una serie de acciones y transiciones en respuesta a determinados eventos para cada estado.
Nos referiremos al cliente que envía la petición de video chat como *emisor* y al cliente que la recibe como *receptor*.
Es una distinción a tener en cuenta ya que los posibles estados de conexión al servicio de video chat son diferentes para el cliente *emisor* y el *receptor.*

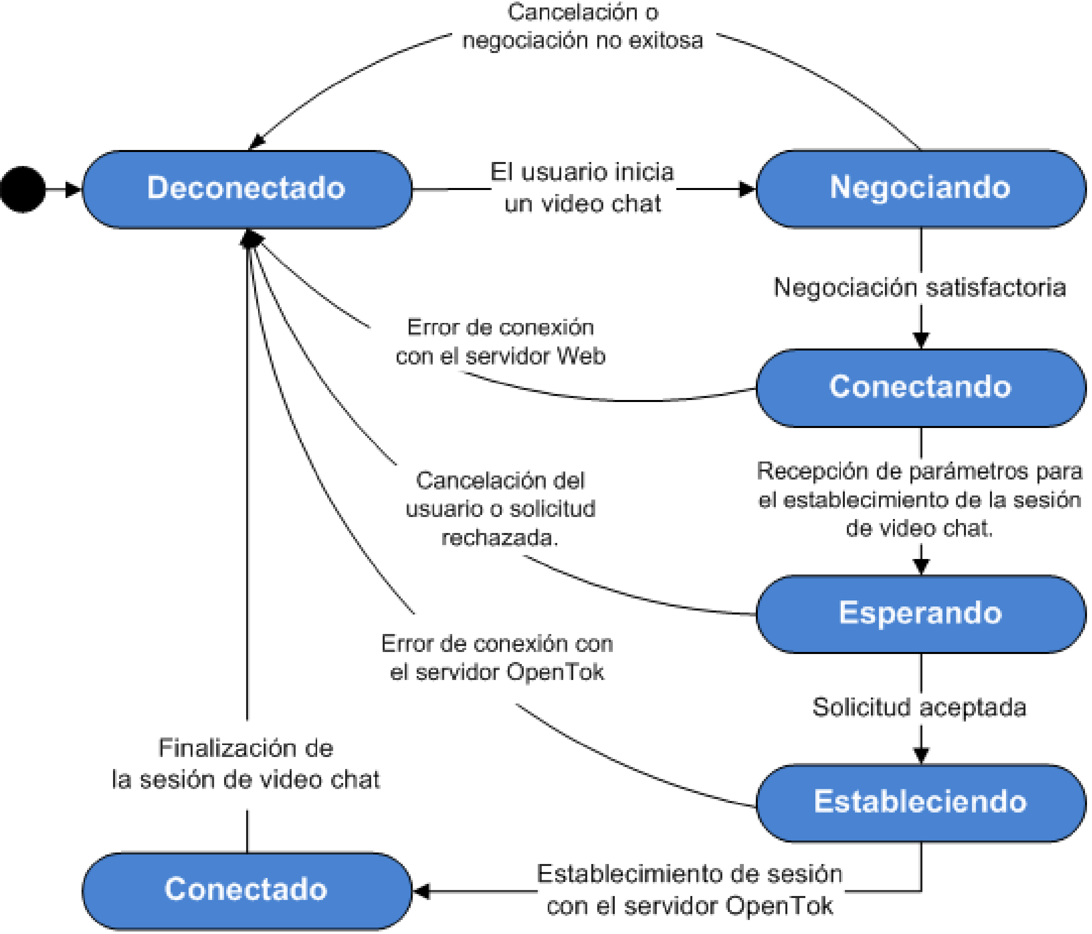
Por otra parte el proceso de negociación implementado es muy simple, su funcionamiento se puede describir como una secuencia de consultas realizadas por el cliente emisor sobre determinadas capacidades del cliente receptor para comprobar que es capaz de establecer un video chat.
En la versión actual se comprueba únicamente el tipo de cliente del receptor, abortando la operación si no es de tipo Social Stream, pero se puede extender fácilmente el funcionamiento para consultar otras características como por ejemplo el soporte flash, el acceso a cámara, etc.

En la siguiente página se representa todo el proceso llevado a cabo para el establecimiento de una sesión de video chat.

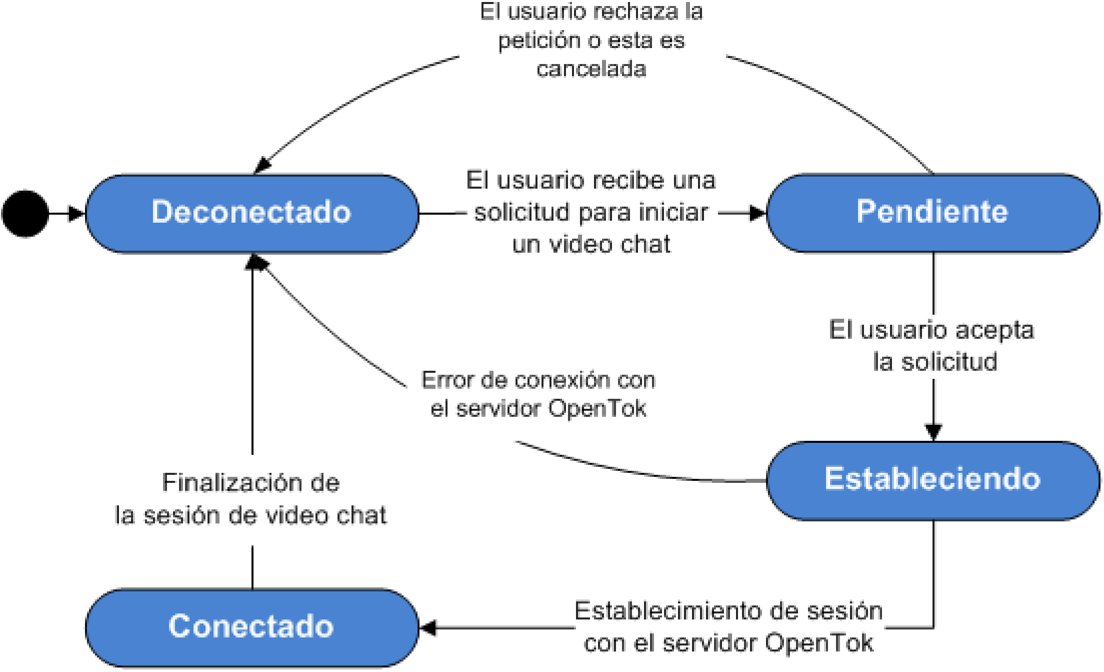
Servidor OpenTok
Servidor Web
Cliente A (Emisor)
Cliente B (Receptor)
HTTP
HTTP
XMPP
El usuario pulsa el botón para iniciar un video chat
Inicio de negociación
Negociación
Solicitud de parámetros para el establecimiento de la sesión de video chat
Generación de una sesión de videoconferencia y de tokens de autenticación *{OpenTok API Key, OpenTok API Secret}*
{sessionId,tokenA,tokenB }
{sessionId, tokenA, tokenB }
Petición de video chat *{sessionId, tokenB }*
El usuario acepta la petición
Petición aceptada *{ true }*
Establecimiento de sesión de videoconferencia *{sessionId,tokenA}*
Establecimiento de sesión de videoconferencia *{sessionId,tokenB}*



Establecimiento de una sesión de video chat



Posibles estados de conexión al servicio de video chat del cliente Emisor



Posibles estados de conexión al servicio de video chat del cliente Receptor

Como se puede apreciar, el establecimiento de la sesión de video chat comienza con el proceso de negociación.
La principal utilidad de este proceso es comprobar que el cliente receptor es capaz de establecer un video chat, en caso contrario, se aborta el establecimiento.

Inicio de sesión de video chat con un usuario que emplea el cliente Pidgin

Posteriormente el cliente emisor solicita al servidor Web los parámetros necesarios para el establecimiento de la sesión de video chat mediante una petición al API REST.

El servidor web contacta con el servidor de OpenTok para generar una nueva sesión y un par de tokens de autenticación adjuntando dos parámetros:

- *API Key*: Es una clave pública que se emplea como identificador de la aplicación web dentro de la plataforma OpenTok.
- *API Secret*: Es una clave privada que solo debe ser conocida por el servidor web para generar sesiones y tokens de autenticación.

Ambas claves son proporcionadas al registrarse en la plataforma OpenTok y se pueden configuran en el initializer de la gema, son la única configuración necesaria para habilitar el servicio de video chat en Social Stream.
Si especificamos un *API Key* en el initializer automáticamente se activará el servicio de video chat. Al final de este capítulo hablaremos sobre la adaptación automática de la interfaz al activar nuevas funcionalidades.

El cliente emisor recibe los parámetros para el establecimiento de la sesión de video chat por HTTP (en formato XML) y se los envía al cliente receptor mediante XMPP:

```
<iq xmlns="jabber:client" from="ernest-ramirez@euphoria" to="demo@euphoria"
      type="get"  id="videochatRequestID">
          <query xmlns="urn:ietf:params:xml:ns:xmpp-stanzas">
                  <session_id>"2_MX4xMTc2M41M34"</session_id>
                  <token>"T1==cGFydG5lcl9pZD0xMDQ1MjkzODU5Mg=="</token>
          </query>
</iq>
```

Una vez que el usuario contesta, la respuesta también es enviada mediante XMPP:

```
<iq xmlns="jabber:client" from="ernest-ramirez@euphoria" to="demo@euphoria"
     type="result"  id="videochatRequestID">
          <query xmlns="urn:ietf:params:xml:ns:xmpp-stanzas">
                  <response>yes</response>
          </query>
</iq>
```

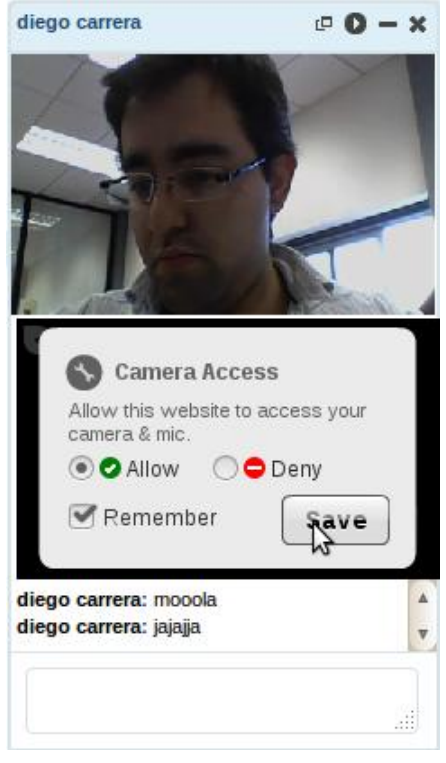
Tras esta respuesta finaliza el intercambio de mensajes XMPP entre ambos clientes y cada uno de ellos procede a realizar el establecimiento de sesión con el servidor OpenTok, que se realiza mediante Flash Sockets.

Las acciones que tienen lugar una vez establecida la sesión con el servidor OpenTok se han representado en la segunda imagen de la figura "*Establecimiento de una sesión de video chat".*

Como conclusión podemos decir que Social Stream Presence ofrece un servicio de video chat con una alta calidad y una experiencia de usuario muy satisfactoria con una configuración muy sencilla.

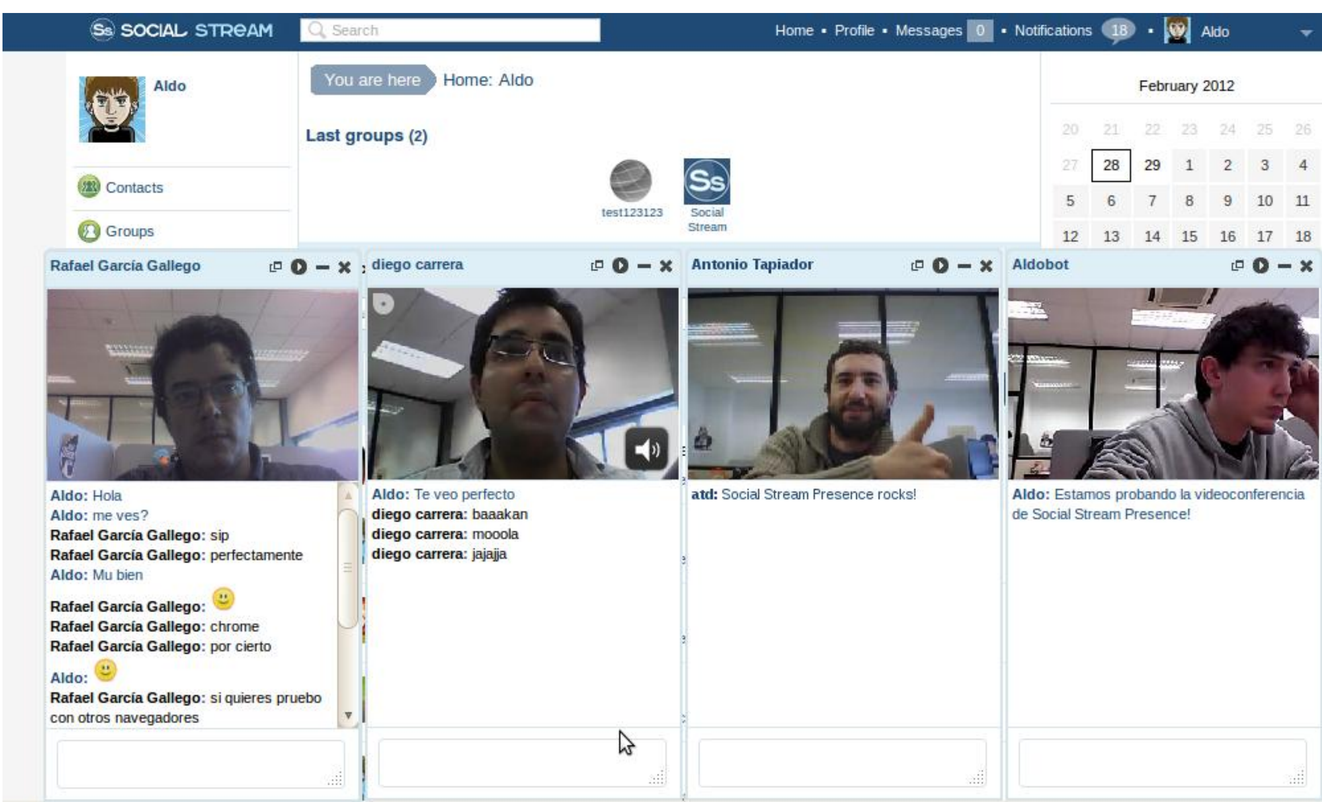
En cuanto a los dispositivos soportados, cualquier webcam soportada por flash es apta para el servicio, y en cuanto a los dispositivos móviles, OpenTok está empezando a dar soporte tanto para Android como para iOS.

Como mayores inconvenientes podemos destacar que al emplear implementaciones propias el servicio no es compatible con otros clientes, y por otra parte, existe una fuerte dependencia con un servicio ofrecido por terceros: OpenTok, de modo que si este dejara de estar disponible obligaría a reimplementar el servicio casi por completo.

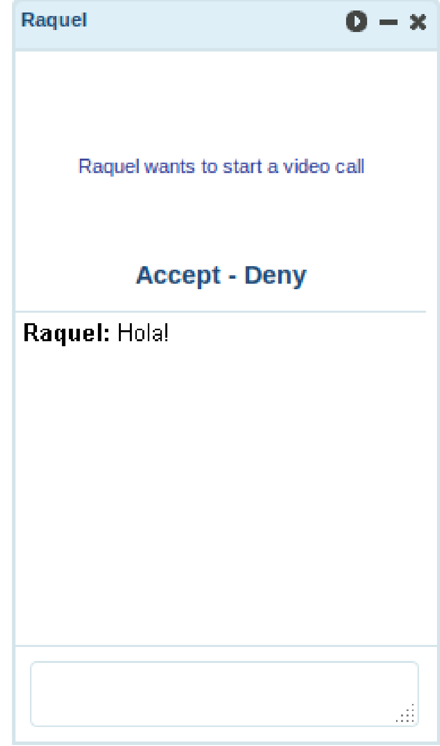


Autorización de acceso a la webcam y micrófono

Visualización de video propio en la conversación



Múltiples sesiones de video chat simultáneas



Invitación a video chat

Cancelación de invitación

7.2.12 Game (Multiplayer Games Core)

Este módulo proporciona mecanismos para jugar en línea a juegos multijugador a través del chat de Social Stream Presence.

Aunque la implementación de este módulo se escapaba ligeramente del ámbito del proyecto y no figuraba en los objetivos iniciales, se ha desarrollado con la misión de probar las capacidades del protocolo XMPP para diferentes tipos de juegos: por turnos, tiempo real, etc.
Adicionalmente, para probar el sistema se han desarrollado dos juegos JavaScript:

- *Tres en Raya*: juego por turnos.
- *JavaScript Fighter*: juego de pelea en tiempo real, basado en el clásico videojuego *Street Fighter*.

Dado que el módulo aún se encuentra en fase de desarrollo, la versión actual *Multiplayer Games Core 1.2* tiene algunas limitaciones:

- Solo permite el establecimiento de partidas para dos jugadores.
- Solo permite una partida simultánea para cada tipo de juego.
- El catálogo de juegos tiene solamente tres opciones.

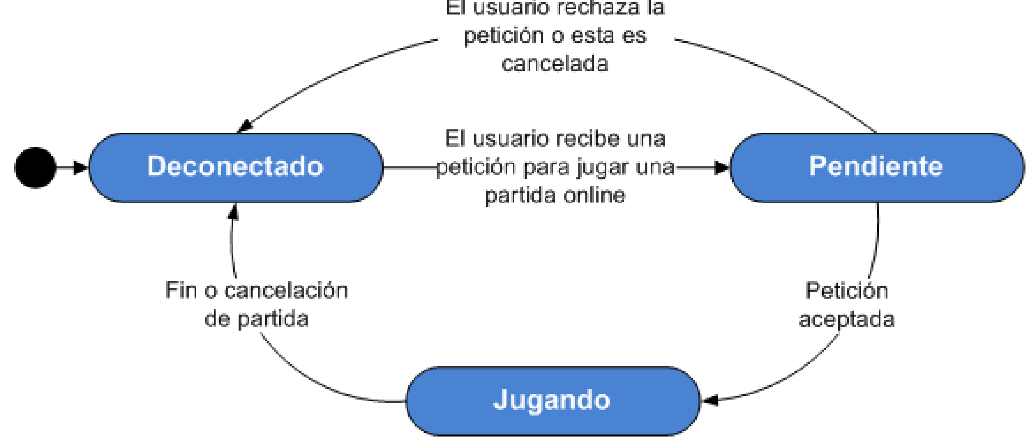
Debido a estas limitaciones, el módulo todavía no está disponible en las versiones desplegadas y solo puede probarse en desarrollo.
No obstante, la funcionalidad del servicio es suficiente para nuestro objetivo: probar las capacidades del protocolo XMPP para aplicaciones en tiempo real.

A continuación se explican las funcionalidades ofrecidas por el módulo Game:

Establecimiento de partidas

Proporciona la negociación para el establecimiento de las partidas mediante XMPP: inicio de partida, cancelación, configuración de opciones, etc.
La implementación es muy similar a la del servicio de video chat ya que también se basa en el envío y respuesta de stanzas XMPP de tipo IQ, sin embargo en este caso es el propio módulo Game quien se encarga de la generación, envío e interacción de las stanzas IQ en lugar del módulo Xmpp Client.
Como la conexión al servidor de presencia es única, ambos módulos deberán de compartir la conexión (proporcionada por *Strophe*).
Para cada partida, el módulo Game añadirá a la conexión un controlador para capturar sus mensajes, escuchando aquellas stanzas IQ con un atributo *id* igual al identificador de la partida. Al finalizar la partida el controlador será eliminado.
De esta forma no se interfiere con el funcionamiento del módulo Xmpp Client, y aunque este módulo capture todas las stanzas de tipo IQ, ignorará (al no reconocerlas) las relativas al servicio multijugador.

Al igual que ocurría con el servicio de video chat, el módulo implementa el mecanismo de establecimiento de partidas como una máquina de estados.



Posibles estados del establecimiento de una partida para el cliente Emisor

Posibles estados del establecimiento de una partida para el cliente Emisor

En este caso, el establecimiento es todavía más simple ya que no existe negociación previa. Si el cliente no es de tipo Social Stream no entenderá la petición y generalmente responderá con un mensaje de error o bien con una stanza IQ vacía. En versiones futuras sería interesante incluir un proceso de negociación para comprobar características del cliente como su versión, el soporte de HTML5, la velocidad de conexión, etc.

Análogamente al servicio de video chat, el módulo almacena información relacionada con el establecimiento de las partidas multijugador.

A continuación se muestra una petición para iniciar una partida online:

```
<iq from='demo@euphoria' to='ernest-ramirez@euphoria'  type='get'  id='gameRequestID'>
	<query xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
		<game id='TER' name='Tic Tac Toe'>
			<players>
				<player id='demo'>Demo</player>
				<player id='ernest-ramirez'>Ernest Ramirez</player>
			</players>
			<options>
				<option name='theme'>Modern</option>
			</options>
		</game>
	</query>
</iq>
```

En la petición se indica el juego (identificador y nombre), los jugadores y las diferentes opciones de la partida. En este caso el usuario *demo* está invitando al usuario *ernest-ramirez* ha iniciar una partida de *Tic Tac Toe (Tres en Raya)* en versión *Modern*.

Si el usuario *ernest-ramirez* está conectado desde Social Stream y acepta la petición, su cliente enviaría un mensaje como el siguiente:

```
<iq xmlns="jabber:client" from="ernest-ramirez@euphoria" to="demo@euphoria"
    type="result"  id="gameRequestID">
	<query xmlns="urn:ietf:params:xml:ns:xmpp-stanzas">
		<response>yes</response>
	</query>
</iq>
```

Por el contrario, si *ernest-ramirez* estuviera conectado mediante otro cliente XMPP, su cliente respondería automáticamente con un mensaje como el siguiente:

```
<iq id='gameRequestID' to='demo@euphoria' from='ernest-ramirez@euphoria'  type='error' >
	<query xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
		<game id='TER' name='Tic Tac Toe'>
			<players>
				<player id='demo'>Demo</player>
				<player id='ernest-ramirez'>Ernest Ramirez</player>
			</players>
			<options>
				<option name='theme'>Modern</option>
			</options>
		</game>
	</query>
	<error type='cancel' code='501'>
		<feature-not-implemented xmlns='urn:ietf:params:xml:ns:xmpp- stanzas'/>
	</error>
</iq>
```

En este caso el cliente *Pidgin* indica que no dispone de la funcionalidad solicitada.

Gestión de la interfaz del servicio de juegos multijugador.

Al igual que ocurría con el servicio de video chat, el módulo Game gestionará de forma autónoma la interfaz vinculada al servicio de juegos multijugador: galerías de selección de juegos, renderizado de peticiones, etc.

Por cuestiones de diseño, se ha impuesto un límite al tamaño de las ventanas de chat, por lo que como máximo podrán tener un solo área adicional a la de conversación.

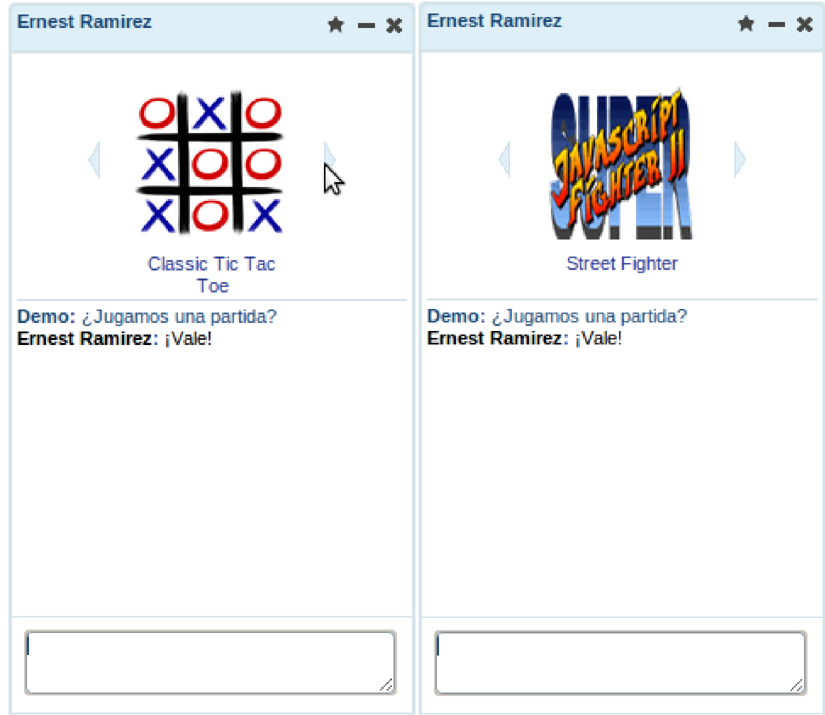
Esto implica que por motivos de diseño el servicio de juegos y el servicio de video chat no se van a poder ofrecer simultáneamente en la misma ventana de chat.

Por tanto, cuando se active el video chat se ocultará el servicio de juegos de la ventana de chat y viceversa, volviendo al estado original una vez finalizado.

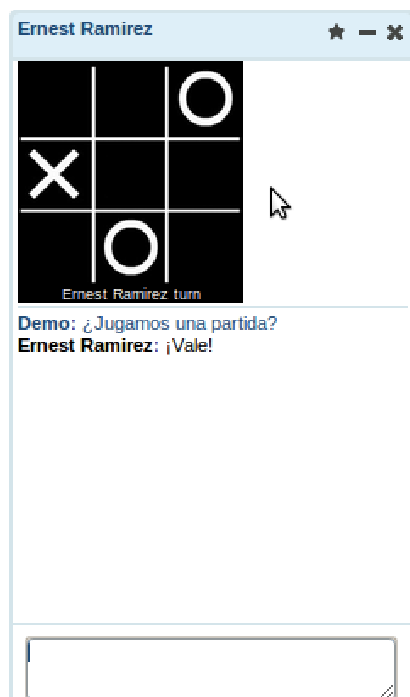
El módulo aprovechará como recinto para trabajar el mismo que se utilizó para el video chat, es decir, el recinto *VideoBox*.

Dado que ambos servicios nunca funcionarán de forma simultánea en la misma ventana de chat no se producirán colisiones entre sus gestores de interfaz.

El renderizado de los gráficos de los juegos no es responsabilidad de este módulo, cada juego será responsable de sus propios gráficos, no obstante como veremos posteriormente, los juegos solo podrán pintar dentro del recinto *VideoBox*.



Galería de juegos



Partida de Tres en Raya

API para Juegos.

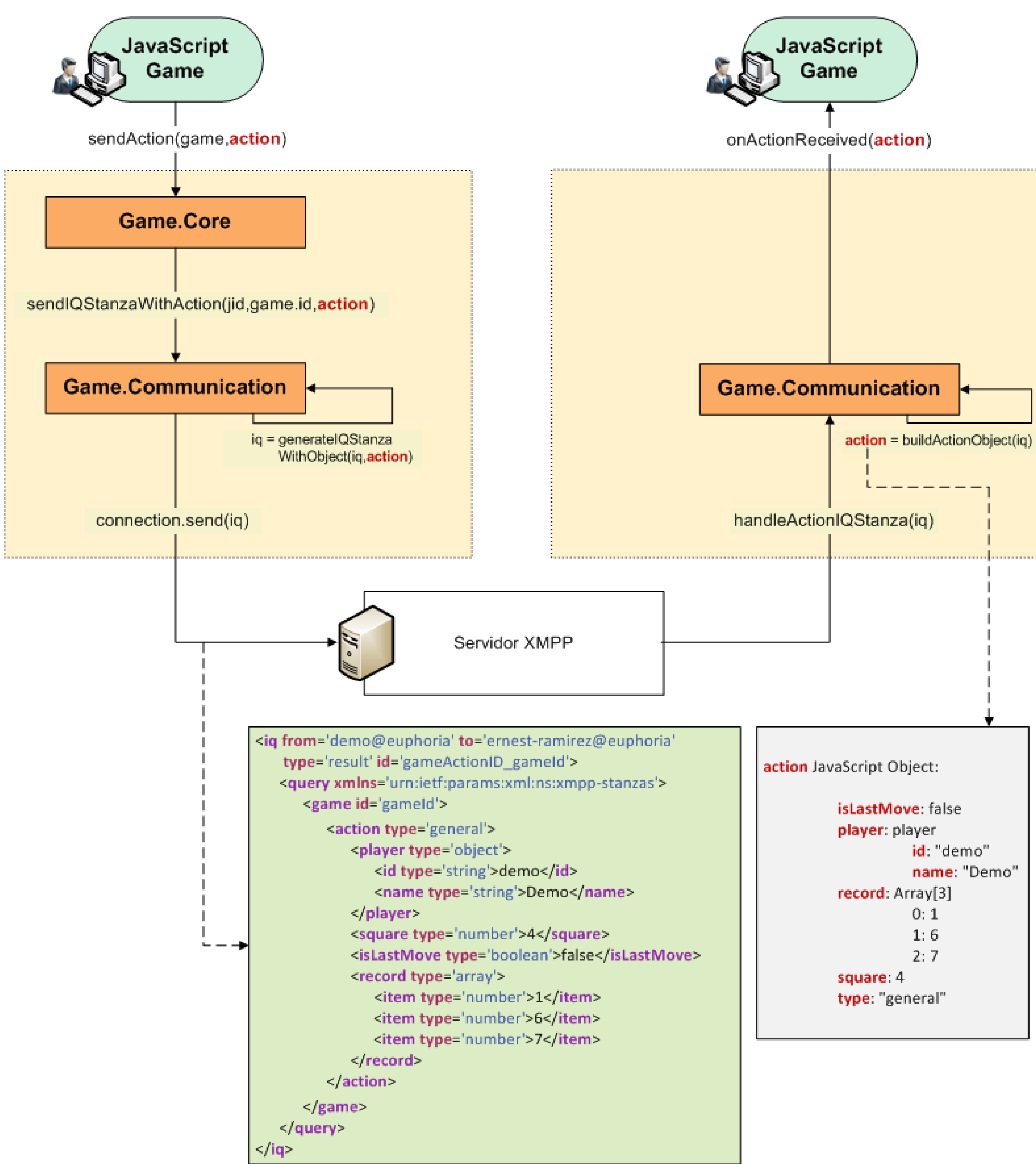
El módulo *Multiplayer Games Core* ofrece un mecanismo basado en un API para construir juegos JavaScript multijugador online.
El API abstrae al desarrollador de la gestión de la comunicación a través de la red, de modo que puede centrarse exclusivamente en la lógica del juego.
Este mecanismo puede ser empleado tanto para crear juegos desde cero como para adaptar juegos JavaScript ya existentes.
La información intercambiada entre jugadores se lleva a cabo mediante el intercambio de objetos JavaScript de tipo **accion**.
Por tanto, su funcionamiento se basa en el envío de objetos JavaScript mediante stanzas XMPP, de forma que dichos objetos vuelven a ser reconstruidos en el destino.
La generación de las stanzas XMPP a enviar a partir de los objetos JavaScript, y la reconstrucción de los objetos JavaScript a partir de las stanzas XMPP recibidas, se realiza de forma automática y transparente al desarrollador.
Un juego queda completamente definido por un identificador y un fichero JavaScript que implementa toda la lógica y gráficos del juego.
En teoría el identificador puede ser tanto un nombre que identifica un fichero interno como una URL, no obstante, la versión actual no soporta juegos externos.
Para que un juego sea conforme al API debe implementar dos funciones fundamentales:

- ***sendAction****:* Sirve para notificar una acción al resto de jugadores.
- ***onActionReceived***: Se invoca cada vez que se recibe una acción de alguno de los jugadores de la partida.

La implementación que cada juego haga del objeto JavaScript "**accion"** es completamente libre, puede contener *cadenas de texto, números, booleans*, *arrays* u incluso otros *objetos*.
No obstante, se recomienda que una acción siempre tenga asociados dos parámetros: el jugador que la ha realizado y el tipo de acción.
Debe emplearse el objeto jugador (*player*) ofrecido por el *CORE* del módulo, ya que este gestiona internamente para cada jugador información de conectividad.
Concretamente en Social Stream cada jugador tendrá un campo *id* cuyo valor coincide con el slug del usuario, y un campo *name* cuyo valor coincide con el nombre del usuario, este nombre será el que emplearán los juegos para nombrar al jugador.
El diseño del objeto acción puede variar completamente en función del juego, una acción puede ser la elección de un personaje de combate o de un escenario de batalla, la pulsación de un botón de pausa, el movimiento de una ficha en una partida de ajedrez o la ejecución de un puñetazo.

En cuanto a la gestión de la interfaz, al iniciar una nueva partida el *CORE* le debe indicar al juego el identificador del contenedor HTML sobre el cual puede trabajar.
En Social Stream el contenedor es el recinto *VideoBox*.

A continuación se ofrece un diagrama que ilustra el funcionamiento del módulo *Multiplayer Games Core.*



Multiplayer Games Core API

7.2.12.1 Arquitectura de Game

Game

También referido como *CORE*.
Gestiona el establecimiento de las partidas entre usuarios de Social Stream.
Determinar el JID para cada jugador.
Proporciona modelos (*jugador*, *partida*) y funciones de soporte (*validación*, *evaluación de requisitos*,…).

Game.Communication

Es el encargado de enviar los objetos JavaScript como stanzas XMPP y reconstruirlos.

Game.Interface

Actualiza la interfaz a raíz de los callbacks invocados por el core:
updateInterfaceAfterGameRequestReceived, updateInterfaceBeforeStartGame, updateInterfaceAfterFinishGame, updateInterfaceOnInformationMessage, etc.
También incluye la galería de juegos.

Game.Factory

Genera los objetos JavaScript de tipo partida.
También proporciona métodos de depuración, por ejemplo, partidas de prueba.

Aunque se ha desarrollado exclusivamente para Social Stream, los diferentes componentes de su arquitectura son piezas independientes, de modo que por ejemplo, podríamos emplear el módulo Communication en cualquier otra aplicación para transportar objetos JavaScript mediante XMPP.

7.2.12.2 Juegos

Para probar el funcionamiento del módulo se han desarrollado dos juegos JavaScript.

Tic Tac Toe (Tres en Raya)

Este juego basado en turnos ha sido creado desde cero.
Para ilustrar el funcionamiento del módulo vamos a ir comentando algunos pasos de la implementación realizada:

```
PRESENCE.GAME.TER = (function(P,$,undefined){ […] }
```

El juego se define como un submódulo de Game, cuyo nombre coincide con su identificador.

```
function action(player,square) {
    this.player = player;
    this.square = square;
    this.type = "general";
}
```

En este caso una acción, además de tener un jugador y tipo asociado, está definida por la casilla (*square*) seleccionada.

```
//The core will invoke this method automatically to start the game.
var init = function(myPlayer,myGame,myDivID){
  player = myPlayer;
  game = myGame;
  divID = myDivID;
  //Setting options
  game.options = settingOptions(myGame.options);
  //Init Graphics
  drawBoard();
  //Init Logic
  settingBoard();
  //Init Events
  setBoardEvents();
  //Init game variables
  currentPlayer = game.players[0];
  playing=true;
  //Start Play! […]
  };
```

El método de inicialización recibe como parámetros al jugador propio (*myPlayer*), el objeto partida (*myGame*), que es igual para todos los jugadores, y el identificador del contenedor donde se va a renderizar el juego (*myDivID*).
Al tratarse de un juego basado en turnos, necesitamos mantener una referencia del jugador que está jugando actualmente: *currentPlayer*.
El objeto partida puede incluir una serie de opciones (*myGame.options*), en este caso las opciones permiten elegir entre dos *temas* diferentes.
En función del tema elegido se cargará un CSS u otro cambiando por completo el aspecto del juego, aunque la lógica será la misma.

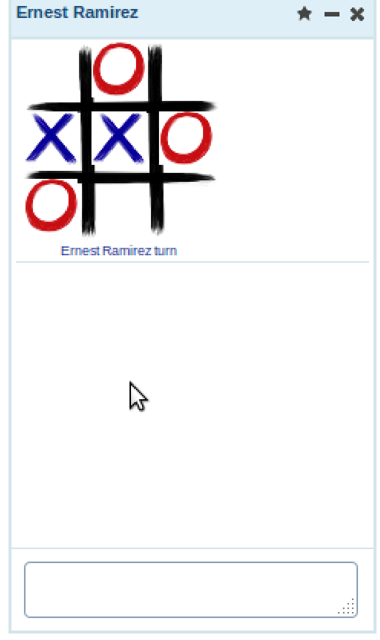
```
//The core will invoke this method automatically in the validation process
var validateParams = function(myPlayer,myGame,myDivID){
  if(myGame.players.length != 2){
    return new CORE.createValidationResult(false,"This game needs two players");
  }
  return new CORE.createValidationResult(true,"Ok");
}
```

Los juegos pueden definir métodos adicionales de validación que serán ejecutados automáticamente por el CORE durante el proceso de validación. También es posible definir métodos para especificar los *requisitos* del juego, por ejemplo, se pueden especificar los navegadores compatibles o la necesidad de soporte HTML5.

```
  // GAMECORE API
  var sendAction = function(action){
    CORE.sendAction(game,action)
    return;
  }
  var onActionReceived = function(action){
    processAction(action);
    return;
  }
```

Aquí especificamos las dos funciones del API.
El método *processAction* actualiza la lógica y la interfaz del juego y, si la acción ha sido ejecutada por nuestro jugador llama al método *sendAction* para notificársela al resto de jugadores. También resulta interesante incluir un método que compruebe que las acciones procesadas son legales.

En este caso la experiencia de juego es excelente, el intercambio de acciones se realiza siempre de manera rápida y transparente.
Por tanto, podemos concluir que el protocolo XMPP es perfectamente válido para la realización de juegos basados en turnos.



Partida de Tres en Raya : Tema Clásico

Partida de Tres en Raya: Tema Moderno

JavaScript Fighter

Se trata de un juego de pelea en tiempo real basado en el clásico *Street Fighter*.
Se trata de una versión reducida, en la que solo existen dos personajes, un escenario de batalla y un número reducido de movimientos.
En este caso el juego no ha sido desarrollado desde cero, sino que ha se adaptado uno ya existente.
La aplicación original era una *demo* cuyos luchadores eran controlados de forma automática, por tanto la adaptación ha consistido principalmente en sustituir la inteligencia artificial por el control humano mediante el teclado.
Además se han añadido algunas acciones extra como *bloquear* y *morir*.
Finalmente se ha adaptado el juego para emplear el API del CORE y poder jugar mediante el chat de Social Stream.
La aplicación original puede encontrarse en la página de *GameQuery:* http://gamequeryjs.com, que es un motor de creación de juegos JavaScript para *JQuery* con el que se ha desarrollado este juego.
La implementación interna del juego queda fuera del ámbito de este proyecto, por lo que simplemente vamos a comentar algunos aspectos de la integración con el API:

```
function action(player,move) {
    this.player = player;
    this.move = move;
    this.type = "movement";
}
```

En este caso la acción se especifica por un movimiento en lugar de por una casilla.
Los movimientos se identifican mediante un número y representan acciones del juego: puñetazo, patada, caída, etc.

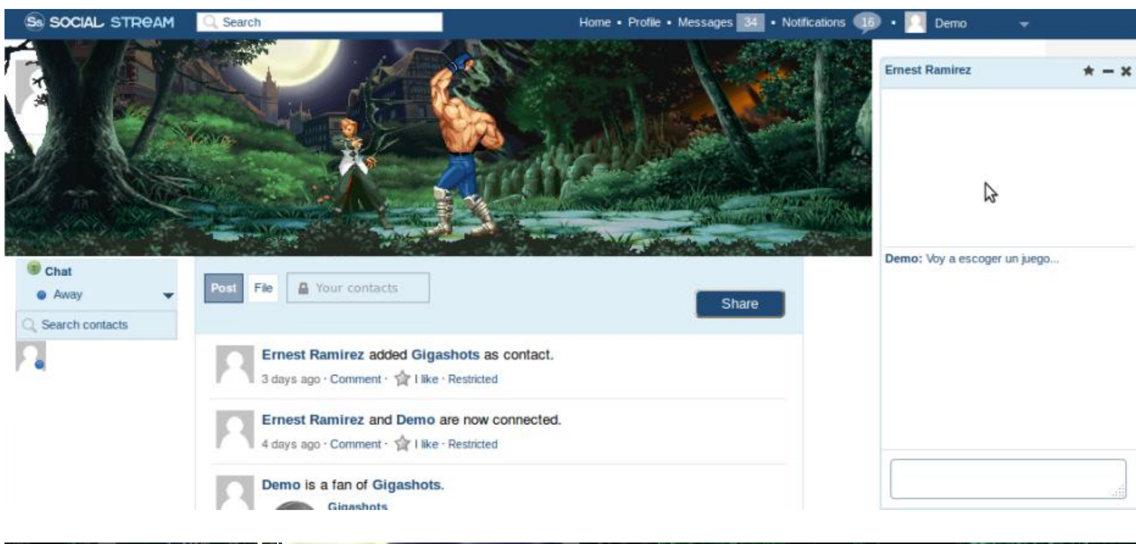
```
  // GAMECORE API

  var sendAction = function(action){
    CORE.sendAction(game,action)
    return;
  }

  var onActionReceived = function(action){
    lastAction = action;
    return;
  }
```

Cada vez que realizamos una acción la notificamos mediante sendAction, y cada vez que recibimos una se almacena en una variable (*lastAction*) que luego será accedida por *GameQuery*.

En este caso la experiencia de juego no es satisfactoria ya que se pierde la sincronía entre ambos clientes, no existe la coordinación adecuada entre ambos.
Si bien el protocolo XMPP cumple con su cometido: informar de las acciones del adversario, el juego no es capaz de gestionarlas correctamente.
Por tanto, sería necesario emplear algún mecanismo de sincronización basado en XMPP para conseguir una experiencia de usuario satisfactoria.
Una posibilidad sería implementar los objetos de tipo acción de la siguiente forma:

```
function action(id, myPlayerMov, myEnemyMove) {
    this.id = id;
    this.myPlayerMove = myPlayerMov;
    this.myEnemyMove = myEnemyMove;
    this.type = "synchronized";
}
```

Las acciones deberían ejecutarse en orden (siguiendo el atributo *id*), y además cada acción contendría los movimientos realizados por ambos jugadores.
De esta forma se pondría conseguir la sincronía buscada.
No obstante, en un caso real, esto sería mucho más complejo, ya que no todas las acciones tienen la misma duración, pueden ser canceladas o pueden basarse en parámetros aleatorios.

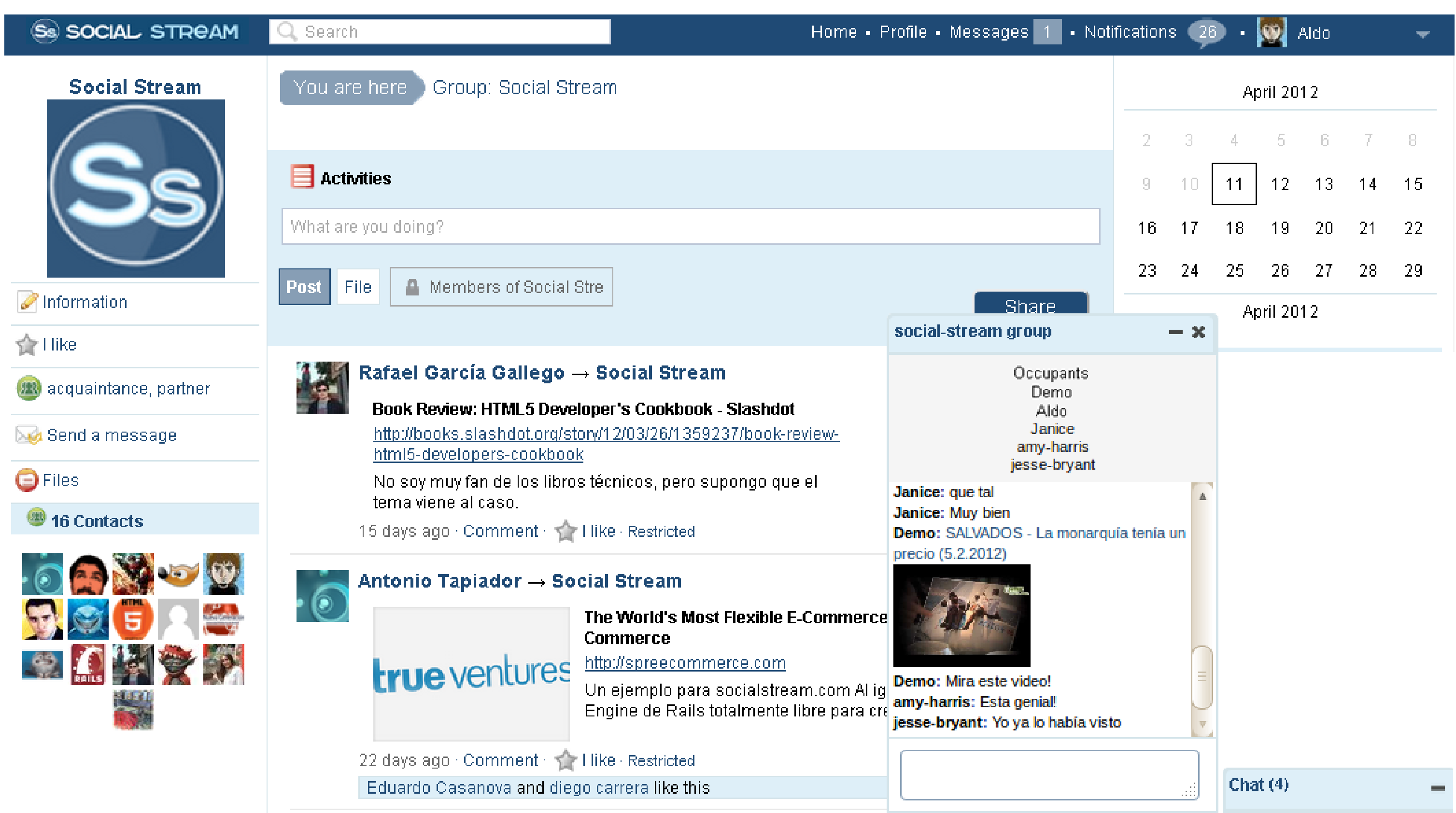


JavaScript Fighter

7.3 Servicio MUC

En la versión actual el servicio de chat multiusuario se utiliza únicamente para proporcionar salas de chat a los diferentes grupos de Social Stream.
Sin embargo, en un futuro podría emplearse para ofrecer otros servicios tales como la adición de invitados a conversaciones o salas de juegos.
No existe un módulo específico responsable del servicio MUC, sino que su funcionalidad se encuentra repartida entre los diferentes módulos.
Las ventanas de chat para grupos son creadas por Window Manager, el sistema de notificaciones para mostrar los ocupantes recae en el módulo Notifications, el módulo UI Manager gestiona la interfaz de las ventanas y escribe los mensajes, y Xmpp Client se encarga de la unión y abandono de las salas de chat, el envío y recepción de mensajes y presencias, etc.

La orden de renderizado del chat permite la inclusión de parámetros, por ejemplo para determinar el modo de visualización: *Widget* o *Float*.
Si se incluye como parámetro un grupo de Social Stream automáticamente se creará una ventana de chat para la sala asociada dicho grupo y el usuario se unirá a ella.
Esto permite a los usuarios acceder de manera automática a la sala de chat de un grupo cuando visitan su página principal.



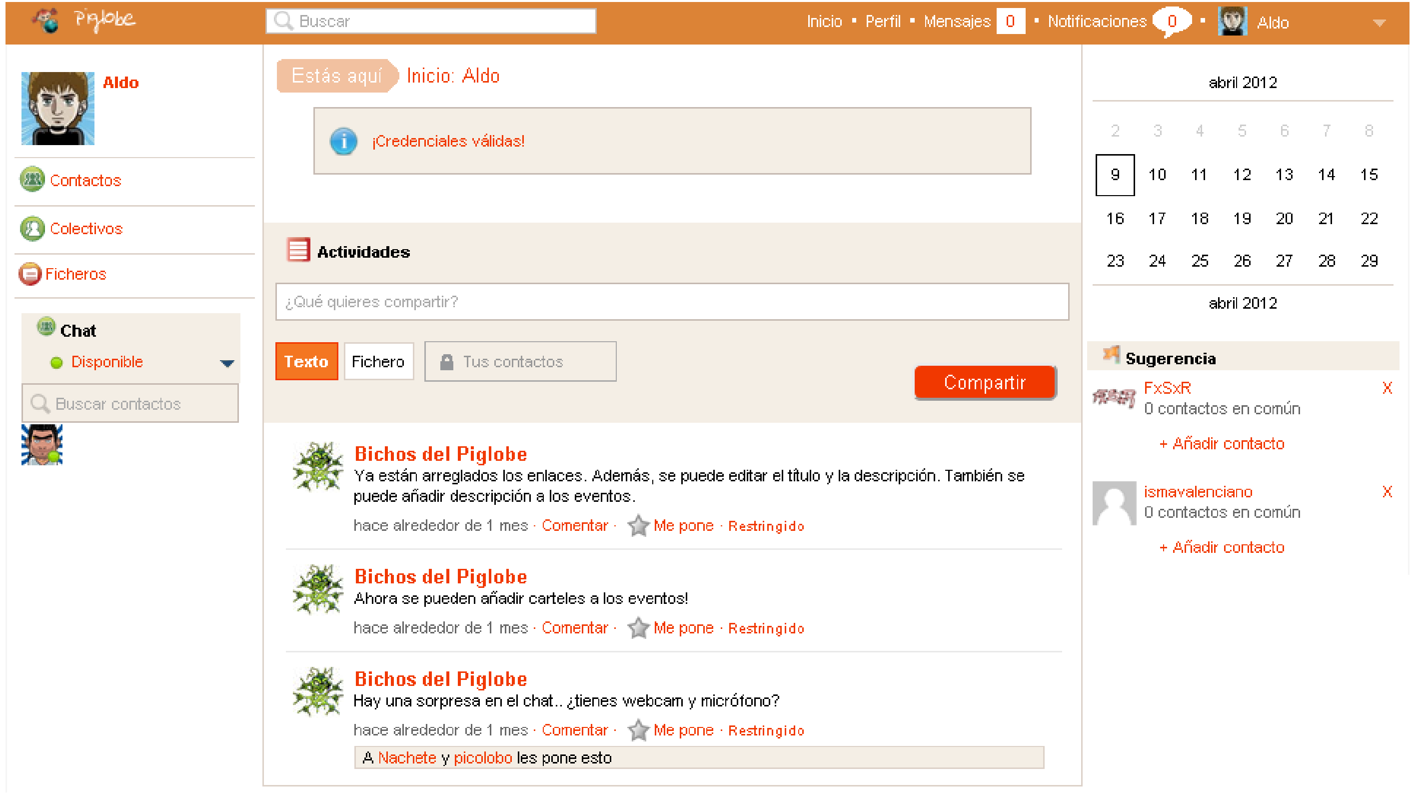
Sala de chat del grupo Social Stream

7.4 Adaptación de la interfaz

Social Stream tiene la capacidad de adaptar su estilo a la aplicación web donde se ejecuta.
Para lograr este objetivo se emplea *SASS*, que permite usar variables en las hojas de estilo CSS en lugar de valores constantes.
El desarrollador solo tiene que rellenar un fichero que asigna el valor a las variables principales, que son, fundamentalmente, colores.
El resto de hojas de estilo de la aplicación toman como referencia las variables cuyo valor se ha definido en el fichero anterior, de modo que el desarrollador no necesita realizar cambios adicionales.
Por supuesto, el desarrollador puede modificar las hojas de estilo existentes y cambiar la estética como el desee mediante la dinámica habitual.

Siguiendo esta filosofía, Social Stream Presence también utiliza SASS, y además acapara en las hojas de estilo toda la información de estética, de modo que cuando se generen elementos de interfaz dinámicamente el aspecto será obtenido exclusivamente de los ficheros CSS (o SCSS).
Por tanto, la interfaz gráfica del chat de Social Stream Presence también tiene la capacidad de adaptar su estilo a la aplicación web donde se ejecuta.

Las capturas ofrecidas a lo largo del proyecto proceden del entorno de desarrollo o del servidor http://demo-social-stream.dit.upm.es , utilizado como escenario de pruebas y como *demo* de Social Stream.
A continuación se van a mostrar algunas capturas de otra aplicación web que emplea Social Stream: http://piglobe.lapiluka.org , en este caso se trata de una red social para un centro cultural.
Los colores empleados en las dos páginas son muy diferentes, sin embargo, podemos observar que el aspecto del chat combina con el resto de la página en ambas situaciones.



Chat Widget

Piglobe
Buscar
Inicio • Perfil • Mensajes 0 • Notificaciones 0 • Aldo

Aldo
Información
Ficheros
1 Contacto

Estás aquí Perfil: Aldo
Actividades
¿Qué quieres compartir?
Texto Fichero Tus contactos
Compartir

David → Bichos del Piglobe
ai1.jpeg
Ver detalles...
hace 4 meses · Comentar · Me pone · Restringido

mikel
cartel grupo consumo .jpg
Ver detalles...
hace 5 meses · Comentar · Me pone · Público

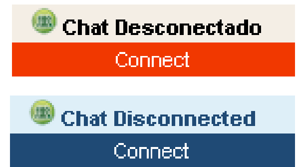
abril 2012
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
abril 2012

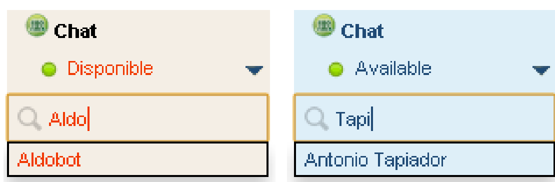
Sugerencia
Bichos del Piglobe
1 contacto en común
+ Responder contacto
Raúl Griot
0 contactos en común
+ Añadir contacto

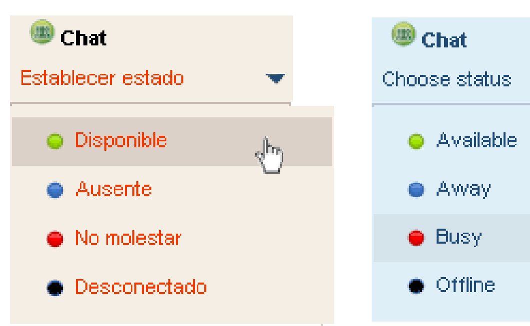
Chat (1)
Disponible
Buscar contactos

2012 © Piglobe

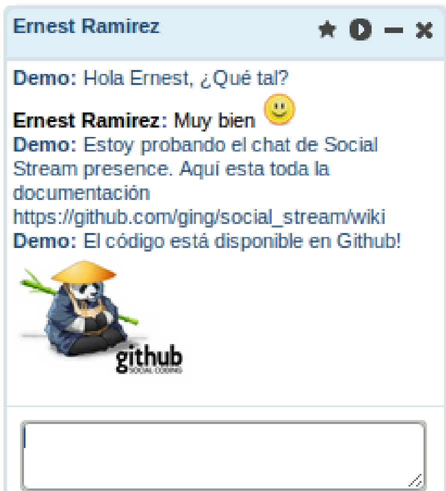
Chat Float


Chat Desconectado
Connect
Chat Disconnected
Connect


Chat
Disponible
Aldo
Aldobot
Chat
Available
Tapi
Antonio Tapiador


Chat
Establecer estado
Disponible
Ausente
No molestar
Desconectado
Chat
Choose status
Available
Away
Busy
Offline

Aldobot
Aldobot: Hola
Aldo: Estamos probando Social Stream en
http://piglobe.lapiluka.org
Aldobot: El aspecto del chat se adapta al nuevo interfaz
Aldo:


Ernest Ramirez
Demo: Hola Ernest, ¿Qué tal?
Ernest Ramirez: Muy bien
Demo: Estoy probando el chat de Social Stream presence. Aquí esta toda la documentación https://github.com/ging/social_stream/wiki
Demo: El código está disponible en Github!
github

7.4.1 Funcionalidades del chat

No solo el aspecto puede variar de una aplicación web a otra, también pueden existir variaciones en cuanto a las funcionalidades disponibles.
En la versión actual de Social Stream Presence existen dos funcionalidades opcionales: el video chat y los juegos online. Será el desarrollador quien decida cuáles incluir en su aplicación web.
A partir de la configuración especificada en el initializer, Social Stream Presence mostrará automáticamente en las ventanas de chat aquellas funcionalidades que hayan sido activadas y ocultará el resto.

Por ejemplo, en la última gráfica de la página anterior se observa que la ventana de chat de estilo naranja tiene el icono para iniciar video chat pero no tiene el de juegos online mientras que la otra ventana muestra ambos iconos.
Esto es debido a que la aplicación http://piglobe.lapiluka.org no tiene activado el módulo *Game* o a que está empleando una versión de Social Stream Presence que no incluye dicho módulo.
Por el contrario, la segunda ventana proviene de un entorno de desarrollo donde sí se encontraba activo el módulo *Game*.

8. Clientes XMPP

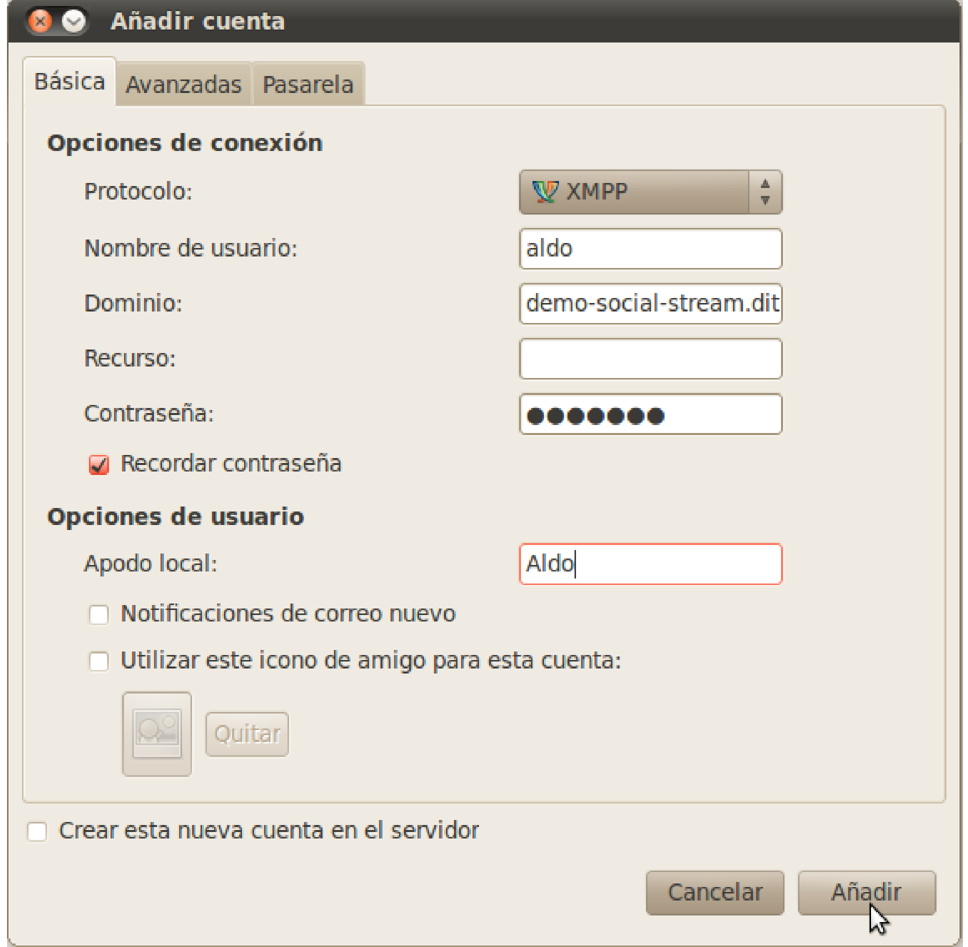
Clientes XMPP

Social Stream Presence proporciona a los usuarios la posibilidad de conectarse al chat desde cualquier cliente XMPP externo empleando el mismo nombre de usuario y contraseña con el que se registraron en la red social.

8.1 Configurando un cliente externo

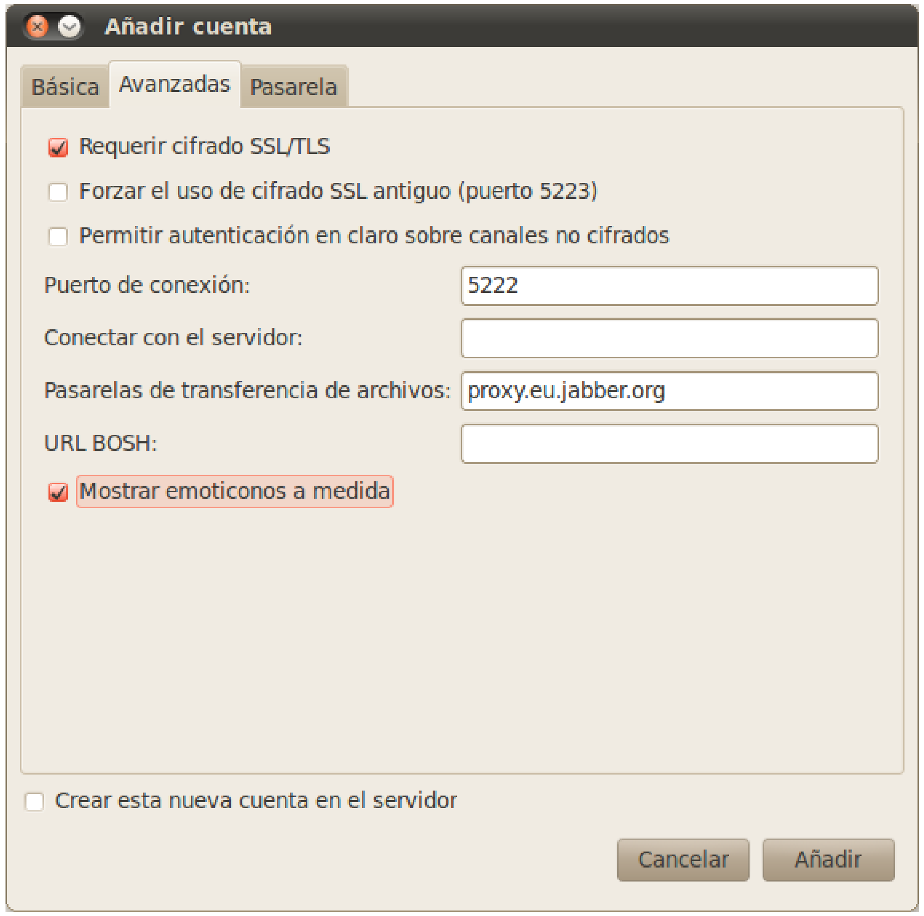
En esta sección vamos a explicar cómo configurar el cliente XMPP *Pidgin* para acceder al chat de la red social http://demo-social-stream.dit.upm.es .
La configuración para otros clientes XMPP es muy similar.

- Abrir el Pidgin y acceder a: *Cuentas –> Gestionar cuentas -> Añadir*



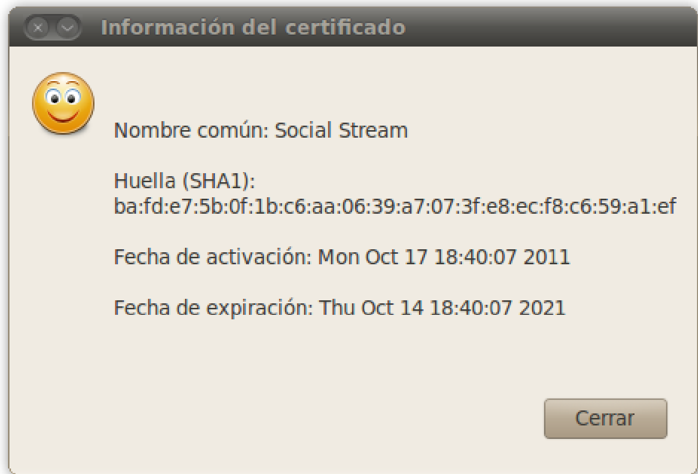
- Escoger el protocolo **XMPP**.
- Como nombre de usuario debemos elegir nuestro *slug* de Social Stream.
  Puede consultarse en http://demo-social-stream.dit.upm.es/api/me .
- En este caso el dominio es: **demo-social-stream.dit.upm.es**.
- Finalmente la contraseña es la misma que utilizamos en Social Stream.

- Ahora seleccionamos la pestaña *Avanzadas.*

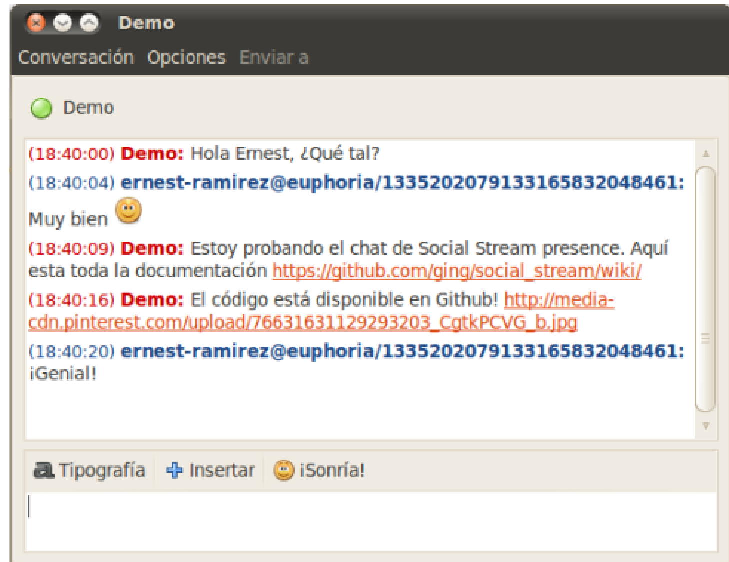


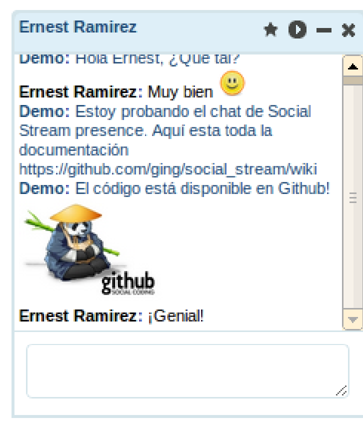
  - Podemos habilitar el cifrado SSL/TLS para autenticarnos de forma segura.
  - Puerto de conexión: **5222**.
  - Si quisiéramos emplear BOSH como medio de transporte tendríamos que especificar la URL BOSH: http://demo-social-stream.dit.upm.es/http-bind . En este caso no podríamos emplear cifrado.

- Finalmente solo tenemos que aceptar el certificado SSL.

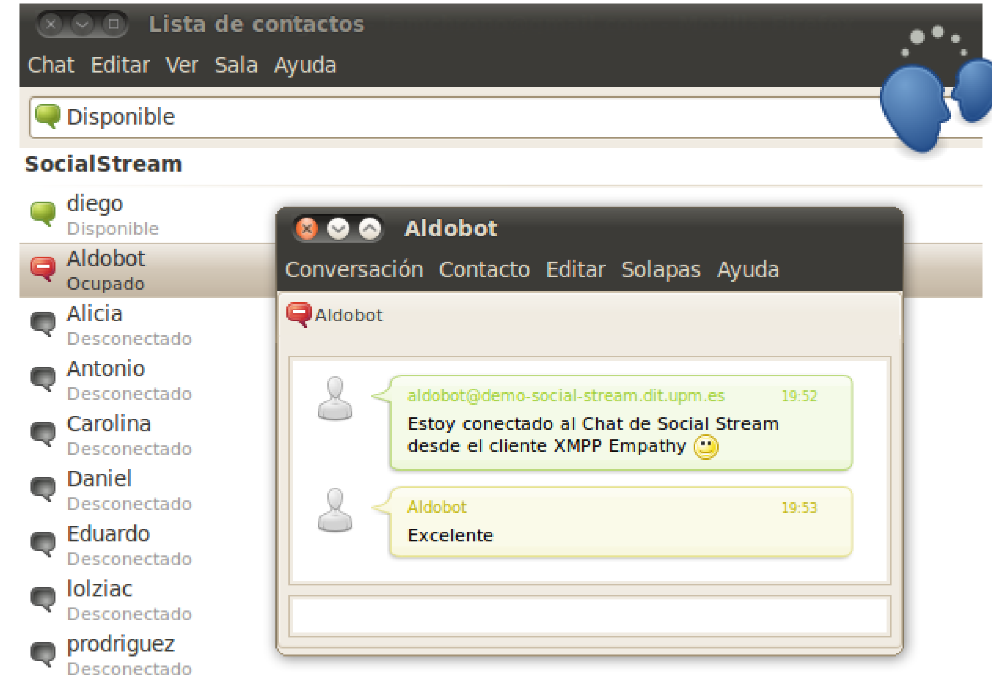


8.2 Ejemplos

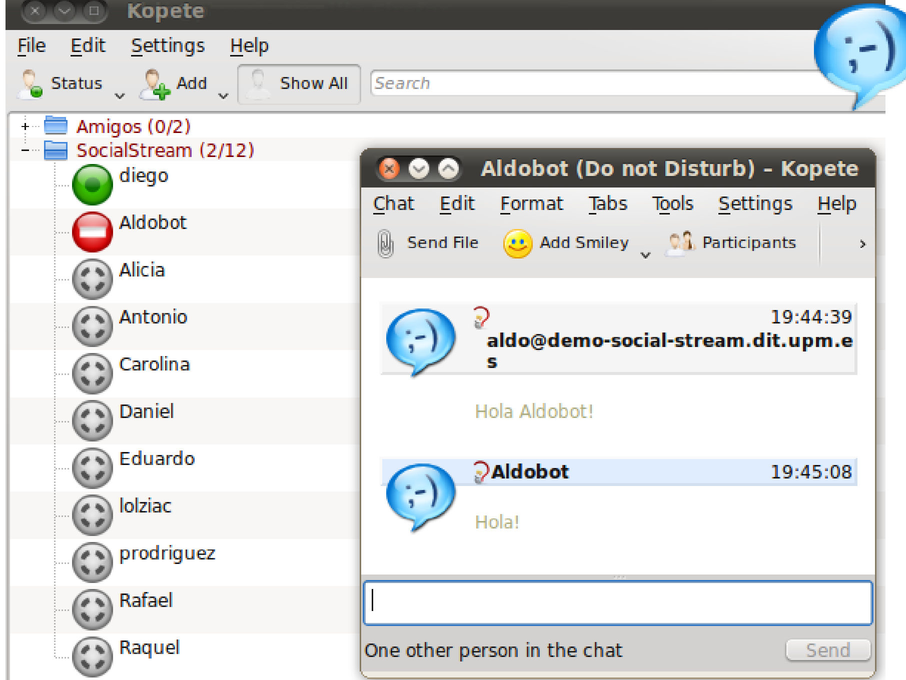




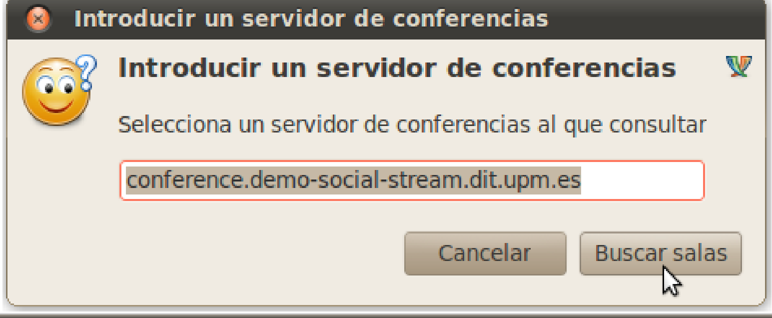
Pidgin

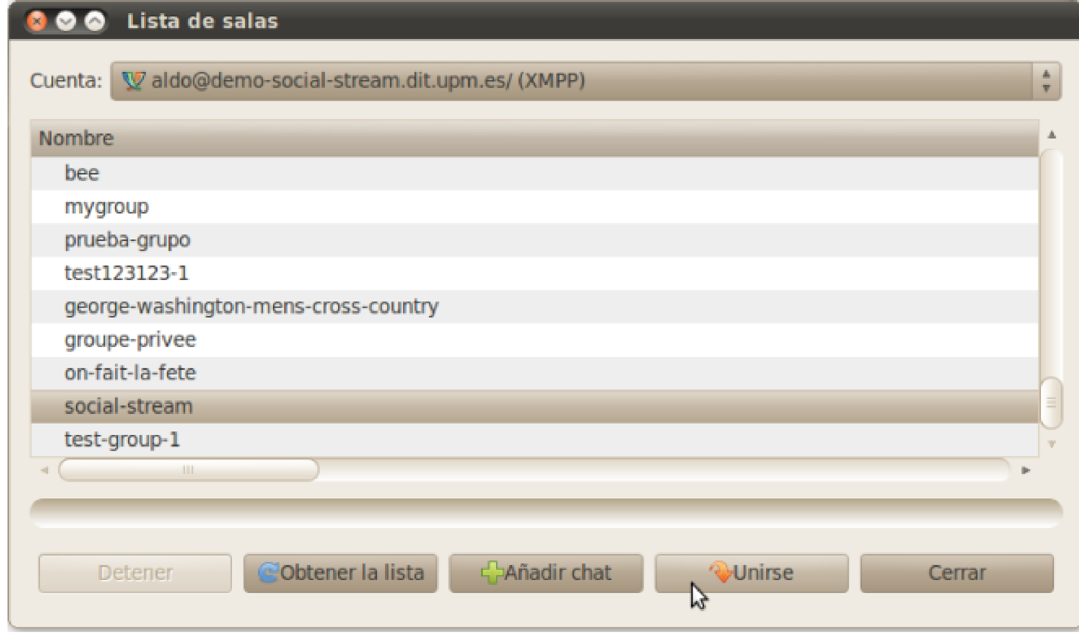


Empathy

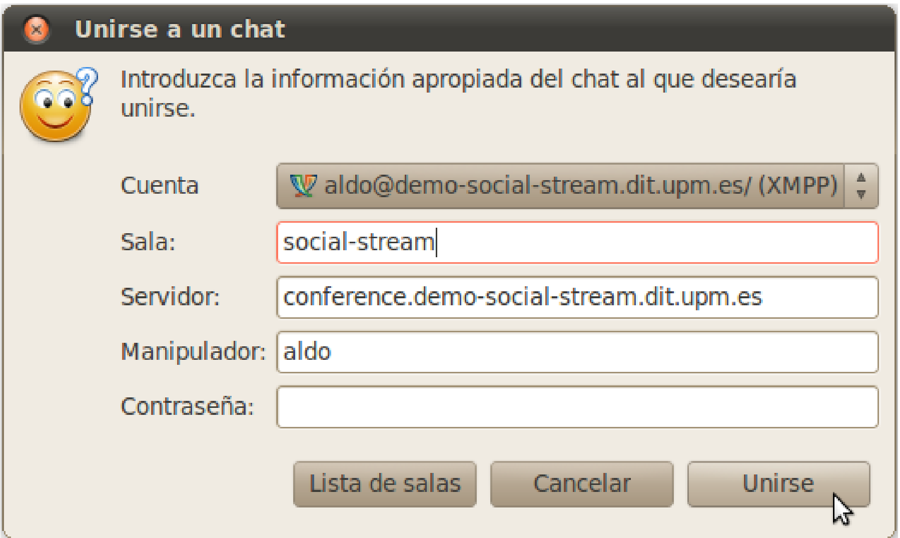


Kopete

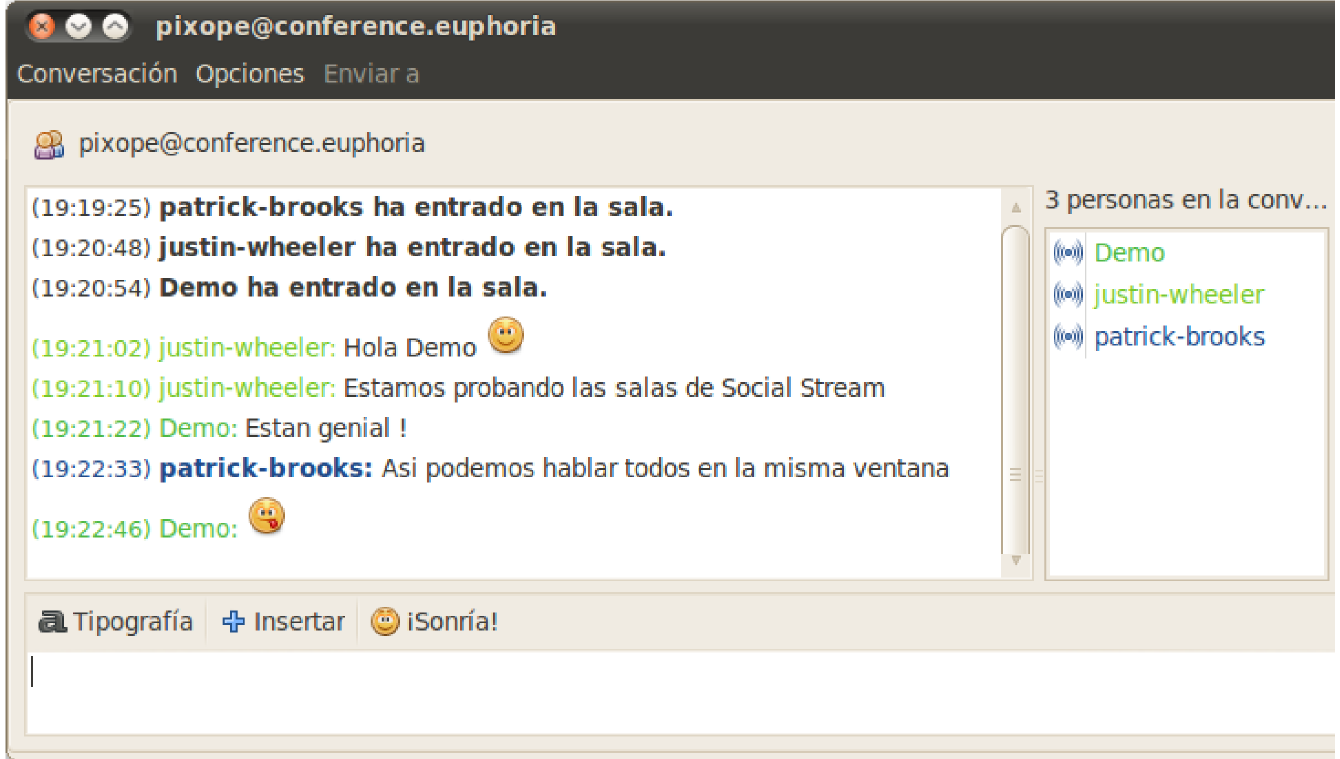




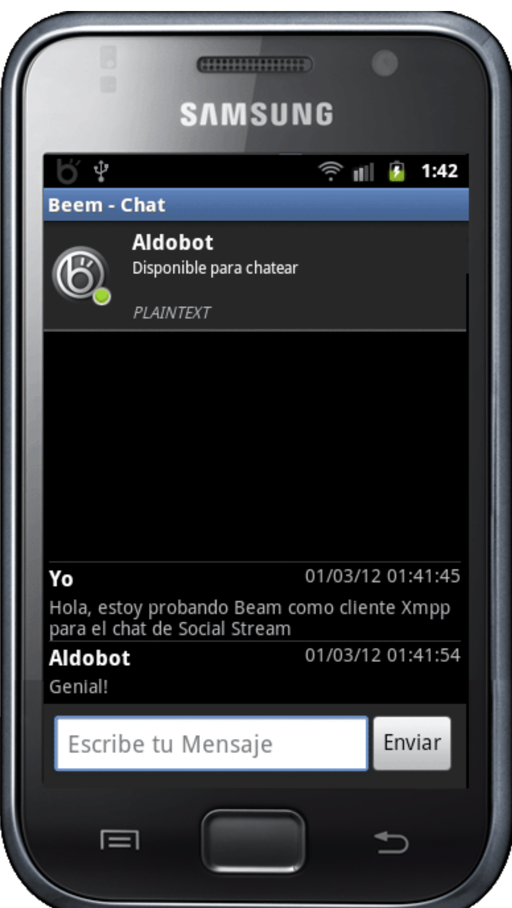
Descubrimiento de salas de chat con Pidgin



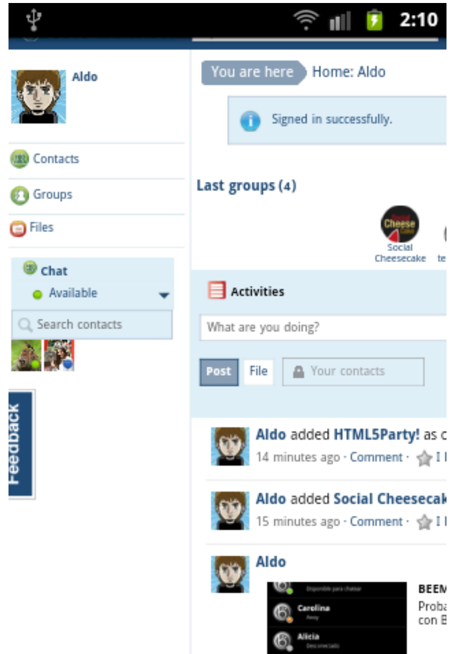
Unión a una sala de chat con Pidgin



Conversando en una sala de chat con Pidgin



Cliente Beem en Android



Android Browser

9. Despliegue

Despliegue

Este capítulo consiste en una guía de instalación y configuración de *Social Stream Presence.* Se trata, a grandes rasgos, de una traducción de la guía oficial:

Getting Started With Social Stream Presence

https://github.com/ging/social_stream/wiki/Getting-Started-With-Social-Stream-Presence

La gema se encuentra publicada en el repositorio oficial RubyGems y actualmente su última versión es la *0.12.0*, publicada el 31 de Marzo de 2012.

Descripción de la gema Social Stream Presence en RubyGems

El proceso de instalación parte de una aplicación web donde la gema *Social Stream* se encuentra correctamente instalada, en caso de no haber realizado dicha instalación en la siguiente web se detalla el proceso: http://rubydoc.info/gems/social_stream/frames .

Instalación

Incluir la gema en el gemfile (si solo tenemos instalado Social Stream Base):

```
gem 'social_stream-presence'
```

y ejecutar

```
bundle update
```

Ejecutar los siguientes comandos que copiarán las migraciones y los initializers:

```
rails g social_stream:presence:install
```

Aplicar las migraciones de la base de datos:

```
rake db:migrate
```

Configuración del servidor Web: Initializer

El initializer de Social Stream Presence está localizado en *yourApp/config/initializers/social_stream_presence.rb* **:**

```
#Configures Web Domain served by XMPP Server
config.domain = "localhost"
#Configures Bosh Service Path
#config.bosh_service = "http://xmpp-proxy/http-bind"
#Configures Authentication Method: "cookie" or "password"
config.auth_method = "cookie"
#Configures XMPP Server Password
config.xmpp_server_password = "autogenerated random password"
#Uncomment to enable REST API Secure Access
#config.secure_rest_api = true
#Remote or local mode
config.remote_xmpp_server = false
#Scripts path to execute ejabberd scripts: local or remote
config.scripts_path = "/scripts_path"
#Ejabberd module path in the xmpp server
config.ejabberd_module_path = "/usr/lib/ejabberd/ebin"
#Uncomment to enable Social Stream Presence
#config.enable = true
```

En *config.domain* debemos poner nuestro dominio web.
El valor por defecto de *config.bosh_service* es *http://root_url/http-bind/* lo cual debería ser válido para modo local, para modo remoto debemos reemplazar *root_url* por el dominio del servidor XMPP.
Por defecto, el método de autenticación es mediante cookie, pero podemos cambiarlo a autenticación basada en usuario y contraseña **:** *config.auth_method = "password"*.
El campo *config.xmpp_server_password* define la palabra de control del API REST y se rellenará automáticamente con una contraseña aleatoria, pero se puede cambiar por cualquier otra.
Por defecto el API tiene configurado el *acceso básico*, para activar el *acceso seguro* solo es necesario descomentar la línea *config.secure_rest_api = true*.
Necesitamos almacenar los scripts en un directorio (en la máquina del servidor XMPP) a nuestra elección, a partir de ahora nos referiremos a este directorio como ***scripts_path***.
El path absoluto de este directorio se debe especificar en la variable config.scripts_path.
Para permitir la instalación automática del servidor XMPP necesitamos especificar el path de los módulos de ejabberd en *config.ejabberd_module_path* .

También debemos especificar en la variable *config.remote_xmpp_server* si la gema va a funcionar en modo local o remoto.
Finalmente, dado que por defecto se encuentra desactivada, necesitamos activar la gema descomentando la siguiente línea: config.enable = true.

Acceso Remoto

```
#Parameters for remote mode
#SSH Login
#config.ssh_domain = "domain"
#config.ssh_user = "login"
#Comment to allow SSH authentication with key instead of password
#config.ssh_password= "password"
```

El acceso remoto es provisto mediante SSH, si especificamos una contraseña en el campo *config.ssh_password* la autenticación será mediante usuario y contraseña, si descomentamos esta línea la autenticación se realizará automáticamente mediante usuario y clave SSH.
Para permitir la autenticación mediante clave SSH del usuario esta debe ser incluida en la lista de claves autorizadas (*ssh/authorized_keys*) del servidor XMPP.

Configuración del servidor XMPP

Debemos instalar el paquete *ejabberd* para UNIX:

```
sudo apt-get install ejabberd
```

Si queremos ejecutar ejabberd en modo *live* (consola en tiempo real), tendremos que instalar ejabberd a partir de los ficheros fuente.

Instalación de Ruby y Rubygems

Para ejecutar los scripts de Ruby necesitamos instalar Ruby en el servidor XMPP.

```
sudo apt-get install ruby-full
```

Además para poder instalar gemas debemos instalar rubygems.

Instalación de la gema rest-client

```
gem install rest-client
```

Instalación de los ficheros de ejabberd de Social Stream Presence

Social Stream Presence proporciona una tarea de rake que copia automáticamente los ficheros al servidor XMPP y lleva a cabo la instalación.

```
rake presence:install:xmpp_server
```

Si la gema trabaja en modo remoto, el acceso SSH debe estar correctamente configurado antes de ejecutar la tarea.
Si lo único que queremos es copiar los ficheros (sin realizar la instalación), podemos emplear la siguiente tarea:

```
rake presence:install:copy_xmpp_server_files
```

O descargar los ficheros de:
https://github.com/ging/social_stream/blob/master/presence/ejabberd/ejabberd_files.zip
En lugar de utilizar el instalador podemos realizar una instalación manual. Los pasos detallados se pueden encontrar en la guía oficial: *Getting Started With Social Stream Presence*
https://github.com/ging/social_stream/wiki/Getting-Started-With-Social-Stream-Presence

Configuración de ejabberd

1. Edición del fichero de configuración ejabberd.cfg

1.1 Activar los módulos de ejabberd

Tenemos que añadir o descomentar las siguientes líneas en la sección de módulos:

```
%%
%% Modules enabled in all ejabberd virtual hosts.
%%
{modules,
 [
  {mod_...,    []},
  {mod_http_bind, []},
  {mod_sspresence, []},
  {mod_admin_extra, []},
  {mod_muc_admin, []},
  {mod_...,    []},
 ]}.
```

1.2 Configurar el servicio MUC (Multi-User Chat)

Debemos configurar los permisos del servicio:

```
%%
%% Modules enabled in all ejabberd virtual hosts.
%%
 {modules,
 ...
 {mod_muc,     [
        %%{host, "conference.@HOST@"},
        {access, muc},
        {access_create, muc_create},
        {access_persistent, muc_create},
        {access_admin, muc_admin}
       ]},
  ...
  }
```

Definir las reglas de acceso:

```
%%%.   ============
%%%'   ACCESS RULES
[...]
%% MUC permissions
%% All users can create rooms:
{access, muc_create, [{allow, all}]}.
%% Admins of this server are also admins of the MUC service:
{access, muc_admin, [{allow, admin}]}.
%% All users are allowed to use the MUC service:
{access, muc, [{allow, all}]}.
```

Para denegarles a los usuarios el permiso de crear salas de chat:

```
%% No user can create rooms:
{access, muc_create, [{deny, all}]}.
```

1.3 Definir los dominios web servidos por ejabberd

```
{hosts, ["social-stream-node.com"]}.
```

1.4 Método de autenticación

Para permitir a ejabberd realizar la autenticación mediante un script externo necesitamos cambiar el método de interno (opción por defecto) a externo. Para ello debemos comentar la línea *%%{auth_method, internal}* y descomentar la línea *{auth_method, external}*.

Debemos especificar la ruta absoluta al script de autenticación:

{extauth_program, "/scripts_path/authentication_script"}.

Además, podemos especificar el número de instancias del script que operarán simultáneamente, por ejemplo para activar 3: *{extauth_instances, 3}.*

```
%%%.   ==============
%%%'   AUTHENTICATION

%% auth_method: Method used to authenticate the users. The default method is the internal.
%% If you want to use a different method, comment this line and  enable the correct ones.
%%{auth_method, internal}.

%% Authentication using external script
%% Make sure the script is executable by ejabberd.
%%
{auth_method, external}.
{extauth_program, "/scripts_path/authentication_script"}.
{extauth_instances, 3}.
```

1.5 Permitir comunicación segura mediante TLS

1.5.1 Generación de un certificado SSL autofirmado simple para ejabberd

Instalamos *make-ssl-cert*:

```
apt-get install ssl-cert
```

Modificamos la plantilla localizada en */usr/share/ssl-cert/ssleay.cnf* añadiendo la línea *days = 7000* para generar un certificado de 20 años de duración y ejecutamos:

```
make-ssl-cert /usr/share/ssl-cert/ssleay.cnf /etc/ejabberd/ejabberdCertificate.pem
```

1.5.1 Activación de TLS

Debemos especificar la ruta absoluta al certificado SSL en la siguiente línea:

```
%%%.   ===============
%%%'   LISTENING PORTS

%%
%% listen: The ports ejabberd will listen on, which service each is handled
%% by and what options to start it with.
%%
{listen,
 [
  {5222, ejabberd_c2s, [
         %%{certfile, "/path/to/ssl.pem"}, starttls,
         {certfile, "/etc/ejabberd/ejabberdCertificate.pem"}, starttls,

         {access, c2s},
         {shaper, c2s_shaper},
         {max_stanza_size, 65536}
           ]},
[...]
```

2. Edición del fichero de configuración ssconfig.cfg

Si se ha ejecutado la instalación automática, la mayoría de los parámetros (como el *cookie_name* o el *ejabberd_password*) ya estarán correctamente configurados, por lo que solo necesitamos especificar los siguientes:

2.1 Especificar el dominio Web para el API REST

```
web_domains=[social-stream-node.com]
```

Si la gema va a trabajar en modo local se puede emplear el valor *localhost*.

Si el nombre del dominio Web y su dirección no coinciden, debemos especificar la dirección:

```
social-stream-node.com=social-stream-node:3000.com
```

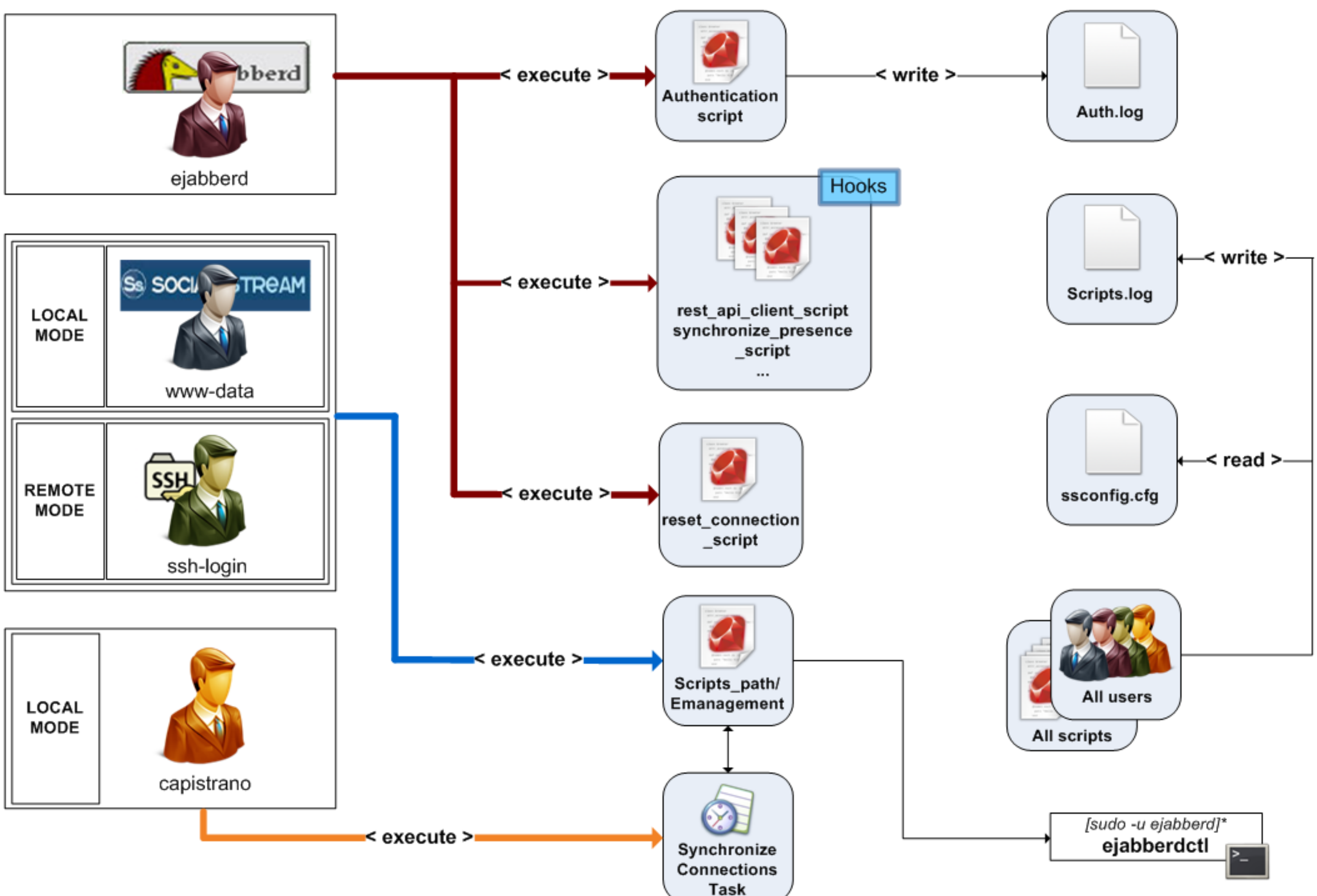
2.2 Especificar el usuario de ejabberd

Si hemos instalado ejabberd desde el repositorio no es necesario modificar nada ya que el nombre del usuario será '*ejabberd*'. Si hemos realizado una instalación para otro usuario debemos especificarlo:

```
ejabberd_server_user=ejabberd
```

Configuración de Permisos

Los permisos deben ser configurados acorde al siguiente esquema:



Configuración de permisos en Social Stream Presence

La instalación por defecto de ejabberd vía repositorio creará un usuario llamado '*ejabberd*', y el servidor ejabberd siempre accederá a los ficheros con este usuario.
Por tanto necesitamos darle permisos al usuario de ejabberd para leer los ficheros de configuración, escribir en los logs y ejecutar los scripts contenidos en scripts_path.
Si la gema funciona en <u>modo local</u>, el usuario de la aplicación rails también necesita permisos.
Si la gema funciona en <u>modo remoto</u> accediendo al servidor XMPP vía SSH, el usuario empleado para el acceso SSH también necesita tener permisos.
Por otro lado, Emanagement ejecuta los comandos de *ejabberdctl* mediante *sudo -u ejabberd* (donde ejabberd es el usuario del servidor), por lo tanto, es necesario realizar un paso adicional (para todos los usuarios menos *'ejabberd'*):

2.3 Permitir la ejecución de sudo sin requerir contraseña.

```
sudo visudo
```

Añadir esta línea al final (cambiando *user* por el nombre del usuario):

```
user ALL = (ejabberd) NOPASSWD:ALL
```

De esta forma, cuando *user* ejecute un comando mediante *sudo –u ejabberd* no se le requerirá la contraseña.

Configuración del Proxy

En primer lugar debemos asegurarnos de que ejabberd tiene el servicio *http-bind* activo y funciona correctamente. Debemos comprobar la presencia de las siguientes líneas en el fichero ejabberd.cfg añadiéndolas en caso de ausencia:

```
{listen,
 [
 [...]
{5280, ejabberd_http, [
          captcha,
          http_bind,
          http_poll,
          web_admin
         ]}
 ]}.
```

Si todo funciona correctamente, al visitar la dirección http://dominioServidorXmpp:5280/http-bind deberíamos ver una página como la siguiente:

demo-social-stream.dit.upm.es:5280/http-bind

ejabberd mod_http_bind

An implementation of XMPP over BOSH (XEP-0206)

This web page is only informative. To use HTTP-Bind you need a Jabber/XMPP client that supports it.

En caso de error debemos asegurarnos de que el módulo de ejabberd *http-bind* está correctamente incluido y activado, y debemos comprobar que el servidor de ejabberd esta arrancado, si no se encuentra en funcionamiento podemos iniciarlo ejecutando:

sudo /etc/init.d/ejabberd start

Finalmente debemos configurar el proxy, existen muchas opciones posibles, en esta guía vamos a explicar a modo de ejemplo cómo configurar *Apache* y *Nginx* como proxys inversos, pero si escogemos cualquier otra opción la configuración será similar.

1 Configurando Apache como proxy inverso

Debemos editar el fichero */etc/apache2/sites-available/mySocialStreamRailsApp* y añadir las siguientes líneas:

```
<VirtualHost *:80>
  [...]
  ProxyRequests Off
  ProxyPass /http-bind/ http://127.0.0.1:5280/http-bind
  ProxyPassReverse /http-bind/ http://127.0.0.1:5280/http-bind
</VirtualHost>
```

2 Configurando Nginx como proxy inverso

Debemos editar el fichero */etc/nginx/nginx.conf* y añadir las siguientes líneas:

```
http {
    [...]

    server {
    listen 8080;
    server_name localhost;

      location /http-bind {
      proxy_buffering off;
      tcp_nodelay on;
      keepalive_timeout 55;
      proxy_pass http://localhost:5280;
      }

    }
}
```

Si la configuración se ha realizado correctamente, al acceder a la dirección http://dominioServidorXmpp /http-bind deberíamos ver nuevamente la página anterior.

ejabberd mod_http_bind

An implementation of XMPP over BOSH (XEP-0206)

This web page is only informative. To use HTTP-Bind you need a Jabber/XMPP client that supports it.

Puesta en marcha y despliegue

Arrancamos el servidor Web (la aplicación Rails que usa Social Stream):

```
rails server
```

Arrancamos el servidor de presencia:

```
sudo /etc/init.d/ejabberd start
```

Ahora necesitamos sincronizar las bases de datos de ambos servidores.
Sincronización de las listas de contactos:

```
bundle exec rake presence:synchronize:rosters
```

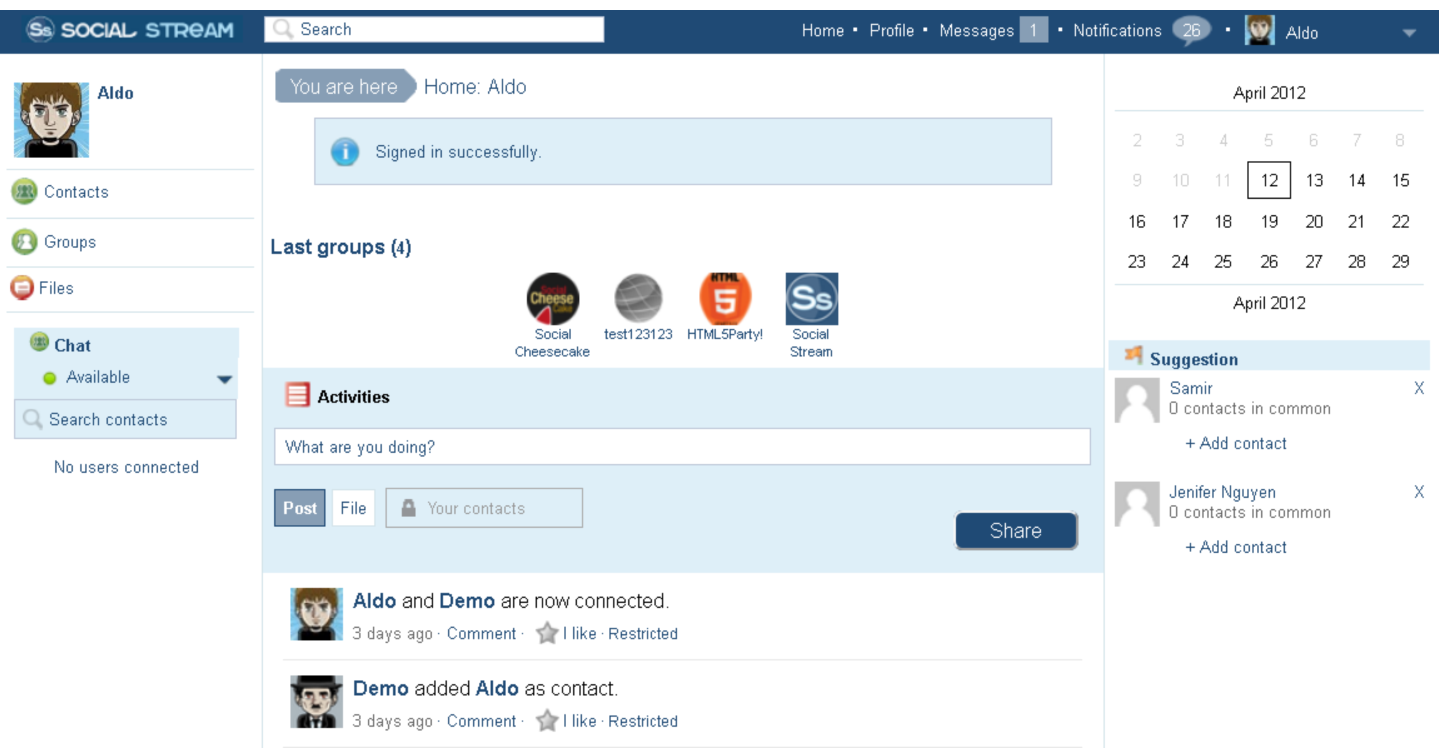
Sincronización de las salas de chat:

```
bundle exec rake presence:synchronize:rooms
```

Para mantener la información de presencia sincronizada es recomendable ejecutar la tarea *presence:synchronize:connections* en cada reinicio de la aplicación.
Si empleamos capistrano para desplegar la aplicación, podemos ejecutar dicha tarea en cada despliegue añadiendo las siguientes líneas al fichero *deploy.rb* :

```
before 'deploy:restart', 'deploy:synchronize_presence'

namespace(:deploy) do
  task :synchronize_presence do
    run "cd #{current_path} && bundle exec \\"rake presence:synchronize:connections RAILS_ENV=production\\""
  end
end
```

Si todo ha ido bien, al acceder nuevamente a la página de inicio, veremos el Widget del chat de Social Stream en la toolbar.

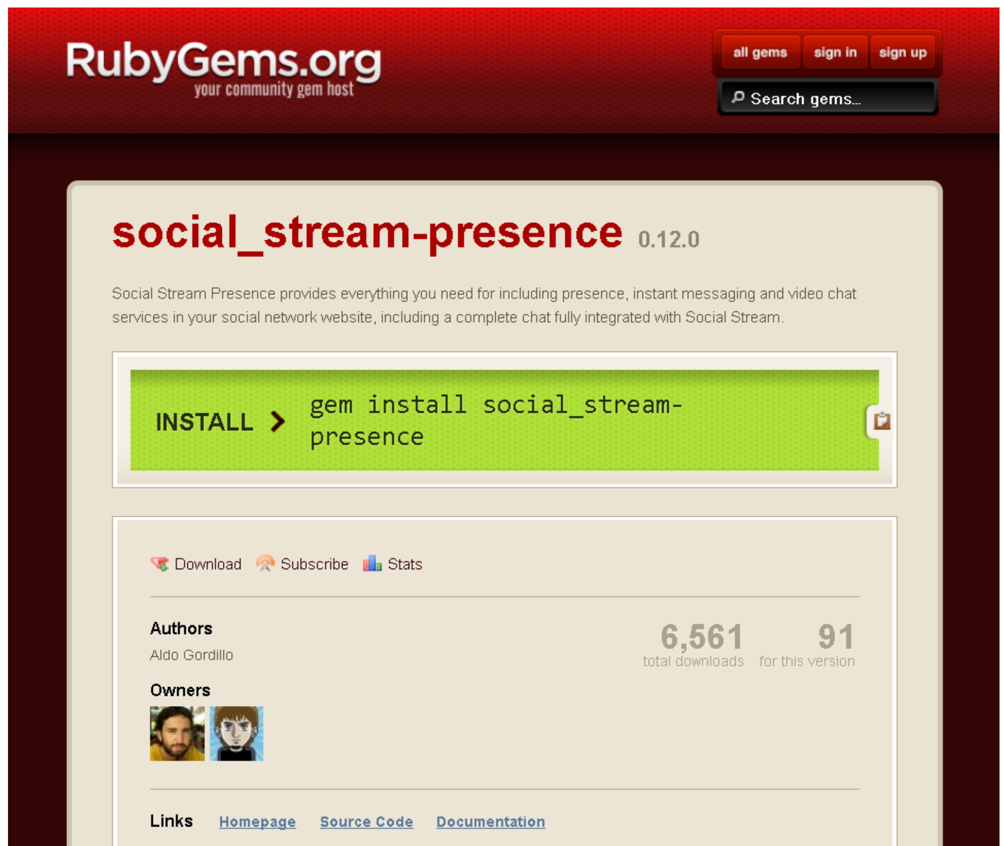


10. Conclusiones

Conclusiones

10.1 Objetivos cumplidos

Observando la última versión publicada de la gema *Social Stream Presence* podemos afirmar que se han cumplido todos los objetivos propuestos al inicio del proyecto.
Se ha conseguido proporcionar a la plataforma Social Stream de servicios de presencia y mensajería instantánea, desarrollando además un cliente basado en el navegador completamente integrado con la red social.
Se ha elaborado amplia documentación (guías, tutoriales, *FAQs*) explicando detalladamente la arquitectura empleada en el servicio, las tecnologías y protocolos utilizados y la instalación, configuración y despliegue. La documentación también incluye ejemplos prácticos y consejos para implementar nuevas funcionalidades.
En cuanto a la internacionalización, la gema proporciona versiones en inglés y castellano.
De forma adicional a los objetivos iniciales, se han desarrollado los servicios de video chat y juegos multijugador.
La gema ha sido publicada en RubyGems y actualmente está siendo utilizada en dos servidores: http://demo-social-stream.dit.upm.es y http://piglobe.lapiluka.org.



Descripción de la gema Social Stream Presence en RubyGems

10.2 Problemas encontrados

El aspecto más problemático en la realización del proyecto ha sido la ausencia de una definición clara de los requisitos de la aplicación así como de la arquitectura necesaria para proporcionar el servicio.
En efecto, existe muy poca documentación y referencias sobre como implementar servicios de presencia y mensajería instantánea para redes sociales, por lo que el diseño de la arquitectura, la elección de los protocolos y tecnologías a utilizar, y la configuración de los diversos componentes (cliente, proxy, ejabberd, etc.) constituyó una tarea muy laboriosa, que requirió emplear grandes cantidades de tiempo para la lectura de documentación y la realización de pruebas.
En la configuración del servidor ejabberd, encontramos la mayor dificultad en la falta de documentación, pese a que ejabberd cuenta con una guía oficial, esta trata ciertos aspectos de forma muy escueta y carece de ejemplos prácticos. Por otro lado, existen pocas fuentes de información adicionales por lo que suele acabar siendo necesario consultar foros oficiales y extraoficiales, blogs especializados o directamente el código de otros proyectos.
Para la implementación del nuevo módulo fue necesario aprender Erlang, un lenguaje con el cual no se había tenido experiencia previa.
Por tanto, la escritura del script de autenticación y la implementación e interconexión del nuevo módulo de ejabberd resultaron tareas complicadas.
Otra dificultad relacionada con la arquitectura, se dio en la interconexión de los diferentes componentes, un claro ejemplo de ello es la discordancia explicada en el capítulo 5 entre los mecanismos de autenticación del cliente Strophe.js y el servidor ejabberd, a consecuencia de la cual hubo que implementar un mecanismo de autenticación especial basado en cookie cifrada para que los clientes basados en el navegador pudieran autenticarse de forma segura.

La gema trabaja conjuntamente con Social Stream, de modo que pueden producirse cambios en la aplicación base, tanto de diseño como de funcionalidad, que obliguen a realizar modificaciones sobre el proyecto.
Por ejemplo, el modo de visualización *Chat Float* se añadió por una decisión de diseño, ya que había ciertas páginas como los perfiles, donde la *toolbar* estaba sobrecargada.
Otro ejemplo podemos encontrarlo en una ocasión donde la adición de una librería JavaScript en Social Stream dio origen a una colisión de variables con otra de librería de terceros utilizada por Social Stream Presence, provocando el fin de la ejecución del código JavaScript.

Otra dificultad añadida, es que no solamente puede haber cambios en Social Stream, sino que también pueden producirse cambios en las gemas utilizadas, las librerías JavaScript o en los navegadores.

Por ejemplo, la actualización de la librería JQuery trajo consigo la necesidad de realizar algunos cambios en el código JavaScript.
La compatibilidad con los diferentes navegadores es también uno de los mayores problemas a los que nos hemos enfrentando en el proyecto.
No tenemos que tratar solamente con nuestros fallos, sino también con los de las librerías utilizadas. Para implementar el servicio MUC hubo que corregir algún fallo de la librería *strophe.muc.js*, que ocasionaba que las stanzas de presencia no se generasen correctamente. La modificación de librerías de terceros siempre conlleva un riesgo, ya que una actualización posterior de la librería puede volver a traer consigo el mismo fallo. En este caso las soluciones a los fallos encontrados fueron comunicadas al equipo de desarrollo de la librería, que las incorporó a la siguiente versión.

El despliegue y puesta en producción de un servicio siempre lleva consigo dificultades adicionales, y este proyecto no es una excepción.
Como ya vimos es necesario realizar ciertas tareas de administración de sistemas para especificar los permisos acordes al esquema "*Configuración de permisos en Social Stream Presence*" del capítulo 9.
Estas tareas pueden resultar ligeramente complicadas si no se tiene experiencia.
El problema encontrado tuvo su origen en una labor de mantenimiento del servidor de producción, una rotación de logs mal configurada provocó la modificación de los permisos establecidos, impidiendo al usuario *www-data* ejecutar el script Emanagement, ya que este necesita permisos para escribir en el fichero *scripts.log* .
Esto ocasionó que los contactos de Social Stream modificados no fueran reflejados correctamente en la base de datos del servidor de presencia durante el periodo que el fallo estuvo vigente.
Tras su corrección fue necesario volver a sincronizar las bases de datos.

10.3 Lecciones aprendidas

Durante el desarrollo de este Proyecto Fin de Carrera he adquirido muchos conocimientos nuevos, algunos de tipo técnico y otros relacionados con la organización y el trabajo en equipo.

En cuanto a conocimientos técnicos he aprendido un lenguaje de programación nuevo: *Erlang,* y he profundizado en el manejo de otros lenguajes: HTML5, CSS y SASS, JavaScript incluyendo diversas librerías como *JQuery* y *Strophe,* Ruby *y* Bash*.*
En todos estos lenguajes he adquirido una soltura considerable que sin duda me será de gran utilidad en el futuro.
También he aprendido mucho acerca del entorno de desarrollo web Ruby on Rails, si bien el principio de "*Convención sobre configuración"* hace más abrupta la curva de aprendizaje, una vez superada la barrera inicial permite un desarrollo más ágil.
Como entorno de trabajo para Ruby on Rails he utilizado *Aptana RadRails*.
Igualmente he adquirido experiencia en el uso del sistema de control de versiones *Git* así como de la plataforma de desarrollo colaborativo *GitHub*.
En la elaboración de este proyecto también he adquirido importantes conocimientos técnicos sobre tecnologias como Comet, BOSH o WebSockets y de protocolos como XMPP.
He ganado amplia experiencia en el diseño de servicios de presencia y he aprendido a configurar diferentes elementos de la arquitectura como servidores Apache y Nginx, pero principalmente me he formado en el manejo del servidor de presencia ejabberd.
Como consecuencia del desarrollo del acceso seguro al API REST, he puesto en práctica numerosos conocimientos sobre seguridad aprendidos a lo largo de la carrera y he ampliado dichos conocimientos en cuanto al diseño de interfaces web seguras.

También he aprendido la importancia de priorizar los esfuerzos, aprender de los errores cometidos y de afrontar los diversos problemas que van surgiendo a lo largo del proyecto con paciencia y tranquilidad.
Otro aspecto importante es definir claramente los objetivos antes de lanzarse a programar, consultando a las partes involucradas y tomando el tiempo necesario para definir las especificaciones.
El proyecto me ha permitido desarrollar mis habilidades de trabajo en equipo participando en las reuniones del proyecto Social Stream, donde se realizaban tareas de análisis de requisitos, diseño de componentes y se comentaban diversos aspectos de la implementación.

Este proyecto me ha servido como colofón a la formación recibida durante toda la carrera y como experiencia pre-profesional al permitirme integrarme dentro de un grupo de trabajo enmarcado en un proyecto que se encuentra en producción, participando directamente en los diversos aspectos técnicos y organizativos.

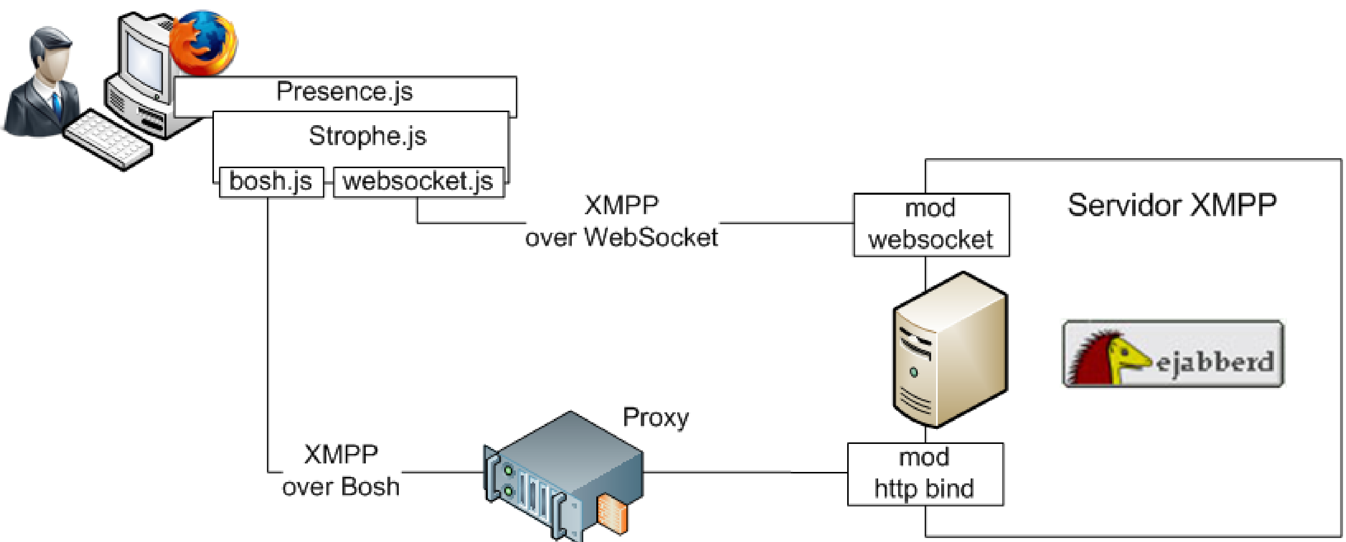
11. Trabajo Futuro

Trabajo Futuro

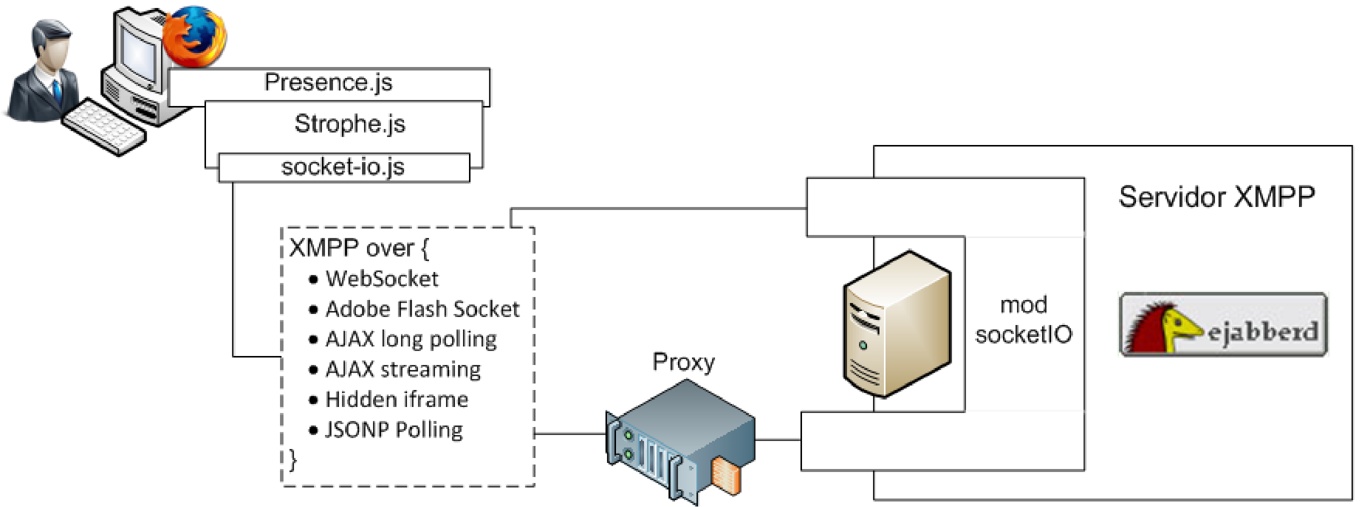
En esta sección se van a comentar algunas posibles mejoras y funcionalidades que sería interesante implementar en versiones futuras de la gema Social Stream Presence.

11.1 Migración a WebSockets

En el estado del arte hablamos con detalle de la tecnología de los WebSockets, llegando a la conclusión de que a pesar de ser actualmente una tecnología poco madura con bajo soporte por parte de los navegadores, estaba destinada a convertirse en el estándar para la comunicación web del futuro.
También hablamos del módulo creado por *Superfeedr* que añade al servidor ejabberd soporte para WebSockets y del posible soporte futuro de Socket.IO en ejabberd.
Strophe.js incorpora soporte para BOSH de forma nativa, sin embargo, se ha empezado a desarrollar una versión de Strophe.js que incorpora conectividad mediante WebSockets (empleando BOSH como *fallback*) y mediante Socket.IO.
Esta versión desarrollada por *Superfeedr* puede encontrarse en el repositorio https://github.com/superfeedr/strophejs/tree/protocol-ed .
En base a estos componentes, se proponen dos arquitecturas para proporcionar el servicio mediante WebSockets y Socket.IO.
La utilización de Socket.IO está supeditada al posible soporte futuro de dicha librería en ejabberd (el módulo *mod socketIO* representado no existe en la actualidad), por otro lado la arquitectura basada en WebSockets podría implementarse a día de hoy, sin embargo como ya comentamos, no todos los navegadores tienen el soporte adecuado.



Conectividad mediante WebSockets



Conectividad mediante Socket.IO

Sin duda alguna, cuando la tecnología WebSockets esté más madura, la opción de WebSockets + BOSH como mecanismo de fallback se antoja como la más indicada, sin embargo mientras esta se encuentre en desarrollo pueden surgir dificultades para establecer la comunicación entre cliente y servidor, o para determinar cuándo utilizar el mecanismo de fallback cuando el soporte de la tecnología sea parcial.

11.2 Realización de Tests

Toda aplicación ha de tener baterías de test para comprobar que funciona correctamente, y para comprobar que el funcionamiento sigue siendo correcto ante futuras modificaciones.

Durante el proyecto no se ha escrito ningún test, el motivo principal es que la escritura de test conlleva una gran cantidad de tiempo, algo especialmente crítico cuando se es la única persona en el desarrollo, además cuando se realizan cambios en el código o refactorizaciones hay que volver a modificar los test consumiendo todavía más tiempo, por tanto hubiera resultado extremadamente difícil mantener los tests actualizados durante el desarrollo del proyecto debido a los números cambios en las especificaciones.

Por otro lado, el servicio desarrollado consta de multitud de componentes (cliente XMPP, scripts, módulos del servidor XMPP,…) implementados en diferentes lenguajes que interaccionan entre ellos, por lo que la realización de test se convierte en un proceso todavía más laborioso. Además, siempre resulta recomendable que los test sean escritos por una persona o grupo de personas ajenas a la aplicación.

No obstante, se ha probado a fondo el funcionamiento del servicio para comprobar su correcto funcionamiento, además, el tener la aplicación en producción resulta de gran ayuda ya que los usuarios notifican los problemas y errores que se van encontrando, y envían sugerencias para mejorar la aplicación.

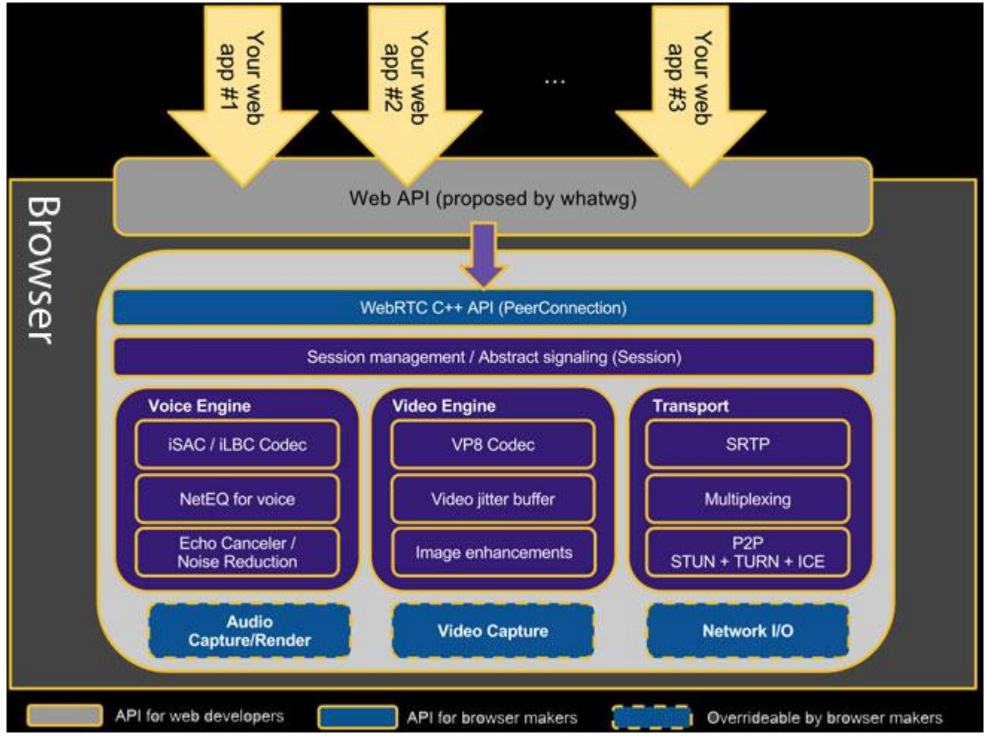
Sin embargo, resultaría recomendable de cara al futuro realizar baterías de pruebas como mínimo para la gema de Ruby on Rails y para el cliente JavaScript, empleando por ejemplo el entorno de pruebas de *Closure* o *JSUnit*.

11.3 Salas de chat privadas

La versión actual de Social Stream Presence solo cuenta con salas de chat abiertas. Una funcionalidad interesante sería la posibilidad de crear *salas sólo para miembros* para los grupos privados, de modo que solo pudieran acceder a dichas salas de chat los usuarios pertenecientes al grupo.

11.4 Video chat mediante WebRTC

WebRTC es un entorno de trabajo que facilita la tarea de construir aplicaciones de comunicación en tiempo real basadas en el navegador utilizando únicamente HTML5 y APIs JavaScript. Incluye los bloques fundamentales para construir comunicaciones de alta calidad en la web tales como componentes de red, audio y video empleados en aplicaciones de chat de voz y video.



Arquitectura WebRTC

Por tanto, sería posible ofrecer el servicio de video chat mediante WebRTC eliminando la dependencia con servicios de terceros (OpenTok), la comunicación se llevaría a cabo directamente entre los clientes, sin servidor intermedio. Podría usarse *Jingle* para la negociación y gestión de las sesiones multimedia mediante XMPP.
El principal obstáculo de esta mejora reside en el escaso soporte actual de WebRTC por parte de los navegadores.

11.5 Mejora del servicio de juegos multijugador

11.5.1 Salas de juego

Una de las limitaciones de la versión actual del módulo Game era que solo permitía partidas entre dos jugadores, sin duda de cara a la próxima versión resultaría muy interesante proporcionar partidas multijugador mediante salas de juego.
Para la gestión de las salas de juego sería necesario emplear el servicio MUC para crear salas de chat temporales, sin embargo, sería conveniente ampliar la funcionalidad del servidor de presencia para gestionar correctamente estas salas.
Los usuarios podrían tomar diferentes roles dentro de las salas de juego: jugador, observador, árbitro, así como diferentes estados dentro de ellas: ausente, preparado para jugar, jugando, observando, etc.
De igual modo las salas podrían también tener un estado asociado: partida en curso o abierta, de modo que, por ejemplo, el servidor podría prohibir la entrada a una sala de juego mientras haya una partida en curso.
Una posible mejora sobre este servicio básico, sería la inclusión de un chatbot controlado por el servidor (empleando la librería xmpp4r) que estuviera siempre presente en la sala y ejerciese de árbitro, este usuario no humano se limitaría a observar la partida, y en caso de observar alguna acción ilegal, expulsaría al usuario infractor. También podría realizar otro tipo de acciones tales como informar al servidor de los resultados de las partidas para elaborar rankings de puntuaciones, o guardar el estado de las partidas cuando se produzca la desconexión de un usuario para restaurarla posteriormente.

11.5.2 Sincronización de acciones

Ya vimos anteriormente que era necesario dotar a los juegos multijugador online en tiempo real de algún mecanismo de sincronización.
Por tanto una mejora consistiría en dotar al *Game Core* de algún mecanismo de sincronización que abstrajese al desarrollador de esta complejidad.

11.5.3 Negociación

Consistiría en un proceso previo para comprobar diversas características del cliente como la versión del navegador, el soporte de HTML5, la velocidad de conexión, etc.

11.5.4 Importación de juegos JavaScript externos

En el módulo *Game* un juego queda completamente definido por un identificador y un fichero JavaScript que implementa toda la lógica y gráficos del juego.
Por tanto, sería posible importar juegos externos mediante la descarga del código JavaScript. Esto permitiría, por ejemplo, que cada red social pudiera exportar a otras sus juegos, teniendo un catálogo global mucho más amplio.

12. Bibliografía

Libros

Agile Web Development with Rails *4ª ed.*
Sam Ruby, David Heinemeier Hansson and Dave Thomas - The Pragmatic Programmers

HTML5 and CSS3: Develop with Tomorrow's Standards Today
Brian P. Hogan - The Pragmatic Programmers

JavaScript Patterns: Build Better Applications with Coding and Design Patterns
Stoyan Stefanov - O'Reilly

Professional XMPP Programming with JavaScript and jQuery*.*
Jack Moffitt - Wrox

Webs

Wikipedia: *http://www.wikipedia.org*

Stack Overflow: *http://stackoverflow.com*

Ruby on Rails Documentation: *http://api.rubyonrails.org*

Getting Started with Erlang User's Guide: *http://www.erlang.org*

ejabberd Community Site: *http://www.ejabberd.im*

ejabberd Installation and Operation Guide:

https://git.process-one.net/ejabberd/mainline/blobs/raw/v2.1.10/doc/guide.html

Strophe.js: *http://strophe.im/strophejs*

XMPP4R: *http://home.gna.org/xmpp4r*

Social Stream Documentation: *http://rubydoc.info/gems/social_stream/frames*

OpenTok: *http://www.tokbox.com*

Jack Moffitt's blog: *http://metajack.wordpress.com*

Anders Conbere's blog: *http://anders.conbere.org*

RFC 3920 *"XMPP: Core"*: *http://xmpp.org/rfcs/rfc3920.html*

RFC 3921 *"XMPP: Instant Messaging and Presence"*: *http://xmpp.org/rfcs/rfc3921.html*

XEP-0045 *"Multi-User Chat"*: *http://xmpp.org/extensions/xep-0045.html*

13. Planos

Planos

Emanagement		
Función	Parámetros	Descripción
addBuddyToRoster	userJid buddyJid buddyNick buddyGroup subscription_type	Añade a {*buddyJid*} al roster de {*userJid*} con el nombre {*buddyNick*}, el grupo {*buddyGroup*} y el estado de subscripción {*subscription_type*}.
removeBuddyFromRoster	userJid buddyJid	Elimina a {*buddyJid*} del roster de {*userJid*}.
setBidirectionalBuddys	userAJid userBJid userANick userBNick groupForA groupForB	Refleja un contacto bidireccional en la base datos, entre el usuario {*userAJid*} con nombre {*userANick*} que será agregado al grupo {*groupForA*} y el usuario {*userBJid*} con nombre {*userBNick*} y cuyo grupo será {*groupForB*}.
unsetBidirectionalBuddys	userJid oldFriendJid oldFriendNick groupForOldFriend	Refleja el paso de un contacto bidireccional a unidireccional en la base de datos. Refleja que el contacto {*oldFriendJid*} ha roto su enlace con {*userJid*}.
checkBidirecctionalBuddys	userAJid userBJid	Comprueba si el contacto entre {*userAJid*} y {*userBJid*} está reflejado en la base de datos como bidireccional.
getRoster	userJid	Devuelve el roster de {*userJid*}.
removeRoster	userJid	Elimina el roster de {*userJid*}.
removeAllRostersByDomain	domain	Elimina todos los rosters de un dominio.
removeAllRosters	-	Elimina todos los rosters.
getAllUserJidsWithRoster ByDomain	domain	Devuelve todos los usuarios que tienen algún contacto en su roster en un dominio determinado.
printAllRostersByDomain	domain	Imprime todos los rosters de un dominio.
printAllRosters	-	Imprime todos los rosters.
printAllBidirecctionalBuddys ByDomain	domain	Imprime todos los contactos bidireccionales reflejados en la base de datos en un dominio determinado.
checkUserJid	userJid	Devuelve *true* si {*userJid*} tiene algún contacto en su roster.
sendPresence	userJid show	Envía una stanza de presencia con el estado {*show*} a nombre de {*userJid*}.
setPresence	userJid	Envía una stanza de presencia de tipo *available* a nombre de {*userJid*}.
unsetPresence	userJid	Envía una stanza de presencia de tipo *unavailable* a nombre de {*userJid*}.
sendMessageToUser	fromJid toJid msg	Envía el mensaje {*msg*} al usuario {*toJid*} a nombre de {*fromJid*}.
getUserResource	userJid	Devuelve el recurso de {*userJid*}.
isEjabberdNodeStarted	-	Devuelve *true* si ejabberd está arrancado.
broadcast	admin userJids msg	Envía el mensaje {*msg*} a todos los usuarios contenidos en el array {*userJids*} a nombre de {*admin*}. Si {*userJids*} vale *all* se lo envía a todos.
broadcastToConnectedUsers	admin userJids msg	Envía el mensaje {*msg*} a todos los usuarios conectados que estén contenidos en el array {*userJids*} a nombre de {*admin*}. Si {*userJids*} vale *all* se lo envía a todos.

getConnectedJidsByDomain	domain	Devuelve los JIDs de todos los usuarios conectados a un dominio determinado.
getConnectedJids	-	Devuelve los JIDs de todos los usuarios conectados.
kickUserJid	userJid	Cierra la sesión de {*userJid*}.
createPersistentRoom	roomName domain	Crea una sala de chat persistente en el dominio {*domain*} con el nombre {*roomName*}.
createRoom	roomName domain	Crea una sala de chat en el dominio {*domain*} con el nombre {*roomName*}.
destroyRoom	roomName domain	Destruye la sala de chat del dominio {*domain*} con nombre {*roomName*}.
destroyAllRoomsByDomain	domain	Destruye todas las salas de chat de un dominio determinado.
destroyAllRooms	-	Destruye todas las salas de chat.
getAllJidsOfRoom	roomName domain	Devuelve el JID de todos los ocupantes de la sala de chat del dominio {*domain*} con el nombre {*roomName*}.
getAffiliationsOfRoom	roomName domain	Devuelve las afiliaciones de todos los ocupantes de la sala de chat del dominio {*domain*} con el nombre {*roomName*}.
setJidAffiliationOfRoom	roomName domain jid affiliation	Establece la afiliación {*affiliation*} a {*jid*} en la sala de chat del dominio {*domain*} con el nombre {*roomName*}.
printAllRoomsByDomain	domain	Imprime todas las salas de chat de un dominio.
printAllRooms	-	Imprime todas las salas de chat.
help	-	Imprime el panel de ayuda.

API REST		
URI Base: /xmpp/		
Función	Parámetros	Descripción
setConnection	name [userSlug]	Notifica la conexión de un usuario.
unsetConnection	name [userSlug]	Notifica la desconexión de un usuario.
setPresence	name [userSlug] status [userStatus]	Notifica el cambio de estado de un usuario.
unsetPresence	name [userSlug]	Notifica el envío de una stanza de presencia de tipo *unavailable*.
resetConnection	-	Marca a todos los usuarios como desconectados.
synchronizePresence	name [Array con los slugs de los usuarios conectados]	Sincroniza la información de presencia en el servidor Web.
updateSettings	settings_section [“chat”] enable_chat [true/false]	Actualiza la configuración del chat de un usuario.
requestVideoChat	► - ◄ Datos en XML.	Solicita un identificador de sesión y dos tokens para iniciar una sesión de videoconferencia.
requestGames	► - ◄ Datos en XML.	Solicita una lista con los juegos disponibles.

XmppServerOrder		
Función	Parámetros	Descripción
setRosterForBidirectionalTie	userASid userBSid userANick userBNick groupForA groupForB	Refleja en la base de datos del servidor de presencia la creación de un contacto bidireccional.
unsetRosterForBidirectionalTie	userSid oldfriendSid oldfriendNick oldfriendGroup	Refleja en la base de datos del servidor de presencia el paso de un contacto bidireccional a unidireccional.
addBuddyToRoster	userSid buddySid buddyNick buddyGroup subscription_type	Refleja en la base de datos del servidor de presencia la creación de un contacto unidireccional.
removeBuddyFromRoster	userSid buddySid	Refleja en la base de datos del servidor de presencia la eliminación de un contacto unidireccional.
createPersistentRoom	roomName domain	Crea una sala de chat persistente.
destroyRoom	roomName domain	Destruye una sala de chat.
synchronizePresence	domain	Sincroniza la información de presencia obteniendo los contactos conectados del servidor de presencia.
resetPresence	-	Marca a todos los usuarios como desconectados.
synchronizeRosters	domain	Sincroniza las listas de contactos en base a los usuarios.
synchronizeRooms	domain	Sincroniza las salas de chat en base a los grupos.
copyFolderToXmppServer	oPath dPath	Copia un directorio o fichero al servidor XMPP.
executeCommands	[commandArray]	Ejecuta un comando en el servidor XMPP y devuelve la respuesta.
generateRSAKeys	keysPath	Genera pares de claves RSA pública/privada para cada servidor.
addWebDomain	domain url	Añade la configuración necesaria en el servidor de presencia para dar servicio a un nuevo dominio web.
removeWebDomain	domain	Elimina un dominio web.
updateWebDomain	domain url	Actualiza la información de un dominio web existente.

14. Pliego de condiciones

Pliego de condiciones

Existen una serie de aspectos impuestos por el entorno en que se ha desarrollado este Proyecto Fin de Carrera que han condicionado de una u otra forma el camino a seguir durante el desarrollo.
Fundamentalmente, como ya se ha explicado a lo largo de la memoria, el proyecto desarrollado debe funcionar como un plugin de la plataforma *Social Stream*, aumentando su funcionalidad proporcionando servicios de presencia y mensajería instantánea pero sin interferir ni modificar su funcionamiento original.

Este hecho ha impuesto una serie de condiciones:

- Dado que Social Stream es un motor de Ruby on Rails, el plugin o gema también debe ser desarrollado mediante el entorno de desarrollo web Ruby on Rails.
  Esto supone también la utilización del lenguaje de programación Ruby en la aplicación Web.
- El grupo de trabajo del proyecto Social Stream emplea fundamentalmente el sistema operativo Ubuntu en sus máquinas de desarrollo, además, los servidores que tiene el departamento con la aplicación en producción también emplean este sistema operativo.
  Por lo tanto una condición importante es que la instalación y despliegue del servicio debía ser viable bajo el sistema operativo Ubuntu.
- Como ya vimos al tratar el tema de la *recarga mediante ajax*, debido a que la gema no puede interferir en el funcionamiento estándar de Social Stream no es posible aplicar soluciones intrusivas con la aplicación base.
- Tanto la funcionalidad como la interfaz del cliente XMPP basado en el navegador debe integrarse perfectamente con Social Stream.
- La gema Social Stream Presence está orientada a desarrolladores, de forma que estos pueden configurarla acorde a sus necesidades.
  Esto implica que es necesario proporcionar diferentes comportamientos en función de la configuración establecida: activación o desactivación de la gema, modo local o remoto, acceso básico o seguro al API REST, activación de funcionalidades extra en el cliente XMPP, etc.
- Como el objetivo final es que cualquier desarrollador pueda instalar y configurar la gema en su aplicación Web, es necesaria la elaboración de una documentación oficial extensa y detallada para guiar al desarrollador en este proceso.

15. Presupuesto

Presupuesto

El presupuesto de este proyecto se desglosa, según la naturaleza de los costes implicados, en tres partes: costes materiales, costes de mano de obra y otros costes. En un último apartado se muestra el presupuesto total.

15.1 Costes materiales

Para la realización del PFC ha sido necesario utilizar ciertos recursos materiales. No obstante, los costes que la mayoría de estos recursos implican, no son totalmente imputables al desarrollo específico del proyecto, ya que su período de vida es superior al del propio proyecto. Es por ello, que pueden ser utilizados para proyectos futuros. Como consecuencia de esto, para poder valorar el equipamiento material que ha sido utilizado en el proyecto, se han tenido que considerar diferentes porcentajes en función de la utilización del equipo en funciones atribuibles directamente a este proyecto, y respecto al tiempo de vida total que se estima para dicho equipamiento.

A continuación se resumen los costes materiales y el uso realizado de los mismos:

1. Un ordenador de sobremesa con procesador Intel Core 2 Duo a 2 GHz, disco duro de 160 GB y 2 GB de memoria RAM, utilizado como estación de trabajo del ingeniero encargado del proyecto, y en el que se han realizado las tareas de documentación, desarrollo e implementación, así como parte de las pruebas realizadas.
2. Otro ordenador de sobremesa igual al anterior, empleado únicamente para la realización de pruebas.
3. Todo el software empleado: sistema operativo *Ubuntu*, servidor de presencia *ejabberd*, entorno de trabajo *Aptana RadRails*,…, es gratuito y se distribuye bajo licencias de software libre, por lo que no existen costes de licencias imputables al proyecto.

Materiales	Coste Total	Uso	Coste Imputable
Ordenador de sobremesa Estación de trabajo	1.400,00 €	30%	420,00 €
Ordenador de sobremesa Banco de pruebas	1.400,00 €	30%	420,00 €
TOTAL			**840,00 €**

Costes Materiales

15.2 Costes de mano de obra

A continuación se desglosan las actividades realizadas a lo largo del desarrollo del proyecto, teniendo en cuenta que había un único desarrollador.

Tarea	Duración (Horas)
Estudio del entorno de trabajo Ruby on Rails	150
Estudio de HTML5	10
Estudio de CSS y SASS	10
Estudio de JavaScript, JQuery y de patrones de diseño JavaScript	50
Estudio de la librería Strophe.js y sus plugins	10
Estudio de Erlang	30
Estudio de Bash	5
Investigación sobre las diferentes tecnologías y arquitecturas para proporcionar servicios de presencia y mensajería instantánea en redes sociales	40
Estudio del protocolo XMPP	25
Estudio del servidor de presencia ejabberd	25
Diseño de la arquitectura	50
Implementación de los mecanismos de autenticación	35
Desarrollo del módulo de ejabberd SSpresence	30
Desarrollo de Emanagement	50
Implementación del API REST	25
Implementación de los módulos y clases internas de Social Stream Presence: XmppServerOrder, BuddyManager y GroupManager	70
Implementación del cliente XMPP basado en el navegador	180
Elaboración de los scripts de instalación y tareas de sincronización	25
Pruebas de integración	40
Configuración, despliegues y tareas de administración de sistemas	20
Elaboración de documentación: guías, tutoriales y memoria final	150
TOTAL	**1.030**

Tareas del proyecto Social Stream Presence

Teniendo en cuenta el sueldo medio por hora de un ingeniero de Telecomunicación, establecido por el Colegio Oficial de Ingenieros de Telecomunicación (COIT) en **18,35€**, y asumiendo que no se pagan impuestos ni se cotiza a la Seguridad Social por la contratación del mismo, el coste total asociado a la mano de obra será:

Horas de trabajo	Coste medio hora de trabajo	Coste total mano de obra
1.030	18,35 €	18.900,5 €

Costes de mano de obra

15.3 Otros costes

15.3.1 Costes generales

Bajo este concepto se engloban los costes derivados del consumo eléctrico (estimado en unos 1.000 kWh a un coste medio de 12,5 c€/kWh), el gasto en material consumible (papel, tinta,…) y el gasto en servicios de comunicaciones:

Concepto	Coste
Consumo eléctrico total	125,00 €
Otros gastos	70,00 €
TOTAL	**195,00 €**

Costes generales

15.3.2 Beneficio industrial

Se aplica un incremento del 10% al presupuesto de ejecución en concepto de beneficio industrial.

15.3.3 Impuestos

Se aplica un incremento del 18% al presupuesto de ejecución en concepto de I.V.A.

15.4 Presupuesto general

A continuación se presenta el presupuesto general, calculado a partir de los costes parciales obtenidos anteriormente:

Concepto	Coste
Costes materiales	840,00 €
Costes mano de obra	18.900,5 €
Presupuesto de ejecución material	**19.740,5**
Costes generales	195 €
Presupuesto de ejecución	**19.935,5 €**
Beneficio industrial (10%)	1.993,55 €
Presupuesto de ejecución + beneficio industrial	**21.929,05 €**
I.V.A. (18%)	3.947,23 €
TOTAL	**25.876,3 €**

Presupuesto general

El presupuesto del presente proyecto asciende a: **VEINTICINCO MIL OCHOCIENTOS SETENTA Y SEIS EUROS CON TREINTA CÉNTIMOS DE EURO**.

Madrid, ____ de ____________________ de 2012

Aldo Gordillo Méndez