72.07 - Protocolos de Comunicación

Trabajo Práctico Especial

Segundo Cuatrimestre 2016

Servidor proxy para el protocolo XMPP

Autores:

Eric Nahuel Horvat - 55564

Daniel Alejandro Lobo - 51171

Martin Alexis Goffan - 55431

Resumen

Implementación de un servidor proxy para el Extensible Messaging and Presence Protocol XMPP (XMPP) [RFC6120] que puede ser utilizado por clientes XMPP existentes.

15 de Noviembre de 2016

Índice

1. Objetivo

El objetivo del trabajo fue implementar un servidor proxy para el Extensible Messaging and Presence Protocol XMPP (XMPP) [RFC6120] que puede ser utilizado por clientes XMPP existentes. El proxy provee al usuario algunos servicios extras que el servidor de origen XMPP no provee (como la manipulación del contenido del mensaje de chat).

1.1. Caracteristicas del servidor proxy implementado

- El servidor proxy soporta múltiples clientes de forma concurrente y simultánea.
- Se tuvo en cuenta en la implementación aquellos factores que puedan llegar a afectar la performance.
- El servidor proxy reporta los fallos a los User-Agents usando características del protocolo XMPP.
- El servidor proxy deja registros de los accesos en la consola que permiten entender qué requests están pasando por el proxy y su resultado.
- El sistema implementa mecanismos que recolectan métricas para entender el funcionamiento del sistema. Esto incluye: cantidad de accesos, bytes transferidos y cualquier otra métrica que hemos considerado oportuno para el entendimiento del funcionamiento dinámico del sistema.
- El sistema implementa mecanismos que permiten configurar el sistema para que un JID sea mapeado a un servidor origen diferente del configurado por defecto.
- El sistema implementa mecanismos que permitan filtrar todos los mensajes entrantes y salientes a un cierto usuario.
- Se implementaron transformaciones del texto del mensaje utilizando formato l33t.
- Se implementó la extensión SI File Transfer.
- En cuanto a la configuración referida a transformaciones y multiplexado, el sistema permite la modificación en tiempo de ejecución de forma remota.
- El servidor expone un servicio que permite monitorear el funcionamiento del sistema. Este servicio provee acceso a las estadísticas recolectadas por el sistema.

2. Protocolos desarrollados

2.1. Estadísticas del protocolo

Las estadísticas son almacenadas en un tipo de dato *AtomicLong* y pueden verse en el archivo *GLoboHData* y *ProxyData*. Gracias a que el tipo de dato es *AtomicLong* podemos solucionar problemas como la concurrencia.

Se usa el comando <*data*> para obtener métricas sobre el proxy. Las mismas se obtienen a partir de conexiones y de mensajes que pasan a través del mismo. Con ese mismo comando es posible también ver los usuarios muteados y multiplexados, entre otras cosas.

Se detalla más sobre esto en la sección *Métricas*.

2.2. Documentación ABNF del protocolo

```
1   ALPHA          = %x41-5A /   %x61-7A ; A-Z /  a-z
2   DIGIT          = %x30-39 ; 0-9
3   CHAR           = %x01-7F ; any 7-bit US-ASCII char, excludes NUL
4   VCHAR          = %x21-7E ; visible (printing) characters
5   HTAB           = %x09 ; horizontal tab
6   SP             = %x20 ; space
7   WSP            =  SP / HTAB ; white space
8   CR             = %x0D ; carriage return
9   LF             = %x0A ; linefeed
10  CRLF           = CRLF ; Internet standard newline
11  LWSP           = *(WSP / CRLF WSP) ; linear white space (past newline)
12  DEC-VAL        = "d" 1*DIGIT [ 1*( "." 1*DIGIT/("-"1*DIGIT)]
13  USER_INPUT     = n*(ALPHA / DIGIT / VCHAR / SP / CHAR)
14  INFO_MSG       = n*(ALPHA / DIGIT / VCHAR / SP / CHAR)
15
16  OPTION =
17  (read_bytes / written_bytes / connections / sent_messages /
        received_messages / muted_users / multiplex /
        average_sent_message / average_received_message /
        median_sent_message / median_received_message /
        outside_messages_muted / inside_messages_muted)
18
19  L33T               = <l33t>("on"/"off")</l33t>
20  CODE               = <code>3DIGIT</code>
21  GLOBOH             = <globoh>
22  XML_INIT_SERVER    = <?xml version="1.0"?>
23  HELLO_SERVER       = <hello>
24  SERVER_OK_SINGLE   = <ok/>
25  BOOLEAN_OPTION     = (true / false)
26  GOODBYE            = <goodbye/>
```

```
27
28      AUTH              = <auth>USER_INPUT:USER_INPUT</auth>
29      OPTION_TAB        = <option>OPTION</option>
30      DESCRIPTION       = <description>INFO_MSG</description>
31      VALUE             = <value>USER_INPUT</value>
32
33      DATA              = <data>OPTION</data>
34      DATA_WITH_USER    = <data user="USER_INPUT">USER_INPUT</data>
35      SERVER_OK_COMPLEX = <ok>OPTION_TAB DESCRIPTION VALUE</ok>
36
37      MSG_DESCRIPTION   =
            <message_description>INFO_MSG</message_description>
38      ERROR             = <error>CODE MESSAGE_DESCRIPTION</error>
39      SUCCESS           = <success>CODE MESSAGE_DESCRIPTION</success>
40
41      MUTE              = <mute set="BOOLEAN_OPTION">USER_INPUT</mute>
42      SERVER_MULTIPLEX  = <server_multiplex
            user="USER_INPUT">USER_INPUT</server_multiplex>
43      MULTIPLEX         = <multiplex
            user="USER_INPUT">USER_INPUT</multiplex>
```

3. Aplicaciones desarrolladas

3.1. Soporte de STARTTLS, TLS, SSL

El proyecto no cuenta con soporte de STARTTLS, TLS, SSL. Asume autenticacion en texto plano (PLAIN).

3.2. Concurrencia

El proxy se desarrolló utilizando sockets no bloqueantes, particularmente Java NIO. Se eligió esta opción ya que los sockets no bloqueantes son mas performantes en servidores concurrentes.

3.3. Fallos

El protocolo cuenta con los siguientes mensajes para reportar fallos:

- 400 - "Not well formed XML"
- 401 - "Unexpected command"
- 402 - "Bad command format"
- 403 - "Bad command option"
- 404 - "Unrecognized command"
- 405 - "Incorrect username or password"
- 911 - "Unknown error"

Se puede ver más en el archivo *GLoboHError*.

En la sección *Ejemplos de prueba* se pueden ver ejemplos de estos errores.

3.4. Registros de acceso

Este servidor proxy cuenta con registros de acceso. Esto significa que en cada oportunidad que un usuario se conecta al mismo, se registra que el acceso.

Este número puede ser consultado como cualquier otra métrica del servidor.

3.5. Métricas

Este servidor proxy cuenta con un mecanismo para medir las métricas del mismo y permiten tener un mejor entiendimiento del funcionamiento del sistema.

Las métricas que aporta este mecanismo son las listadas a continuación:

- Mensajes enviados: *sent_messages*
- Mensajes recibidos: *received_messages*
- Bytes leídos: *read_bytes*
- Bytes transferidos: *written_bytes*
- Números de conexiones establecidas: *connections*
- Cantidad de usuarios muteados: *muted_users*
- Cantidad de usuarios multiplexados: *multiplex*
- Cantidad promedio de mensajes enviados: *average_sent_message*
- Cantidad promedio de mensajes recibidos: *average_received_message*
- Cantidad Mediana de mensajes enviados: *median_sent_message*
- Cantidad Mediana de mensajes recibidos: *median_received_message*
- Cantidad de mensajes externos muteados: *outside_messages_muted*
- Cantidad de mensajes muteados del cliente: *inside_messages_muted*

Cabe destacar que las métricas obtenidas por el servidor proxy son volátiles. Esto quiere decir que cuando se reinicia el servidor, se pierden.

Se puede ver más en el archivo *GLoboHStreamHandler*.

3.6. Multiplexador de cuentas

El presente servidor XMPP permite ser configurado para que un JID sea mapeado a un servidor origen diferente del configurado por default.

Es importante destacar que la multiplexación afecta al momento de conectarse. Cualquier cambio que ocurra mientras el usuario esté conectadado, no se verá reflejado hasta que se desconecte y se establezca nuevamente la conexión.

Se puede ver más al respecto en el archivo *GLoboHData*.

3.7. Silenciar usuarios

El sistema implementa un mecanismo que permite filtrar todos los mensajes entrantes y salientes a un determinado usuario en forma elegante.

Paralelamente, este mecanismo notifica al usuario de que fue silenciado.

Se puede ver la implementación en la carpeta *XMPP* > *handler* donde está la comunicación de cliente a servidor y de servidor a cliente, por separado. En la implementación de cliente a servidor se silencia al usuario logueado. En cambio, en la implementación de servidor a cliente, se silencia a las personas que te le hablan a un determinado usuario.

3.8. Transformación de mensajes

Este servidor XMPP permite hace posible comunicarse mediante el lenguaje *eleet* o *leetspeak*. Se puede modificar el contenido de un mensaje realizando las siguientes sustituciones:

- a por 4 (cuatro)
- e por 3 (tres)
- i por 1 (uno)
- o por 0 (cero)
- c por < (menor)

Para habiltar esta *feature*, se puede enviar <*leet*>*on*|*off*</*leet*> con una de las dos opciones que aparecen entre los tags.

Más detalles sobre la implementación, se puede ver en el archivo *MessageElement*.

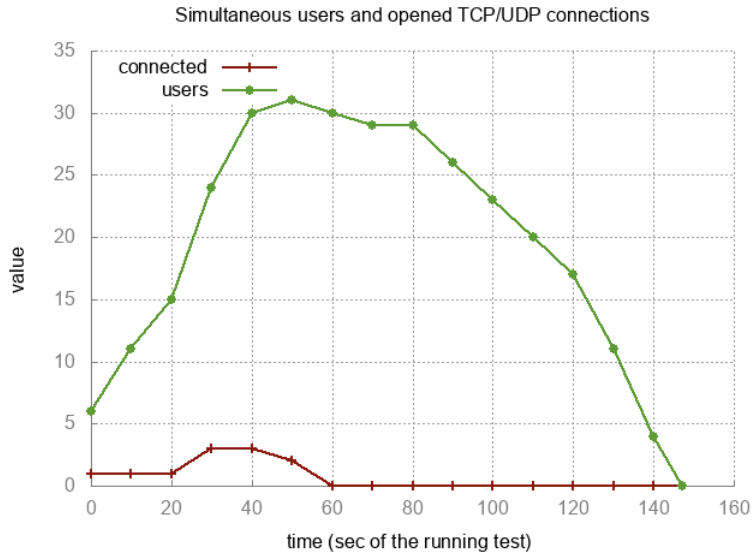
3.9. Pruebas de performance

Se llevó a cabo un testing de stress utilizando una herramienta open source que sirve para hacer *load testing* en diferentes protocolos, entre ellos, servidores XMPP. La misma se llama Tsung.

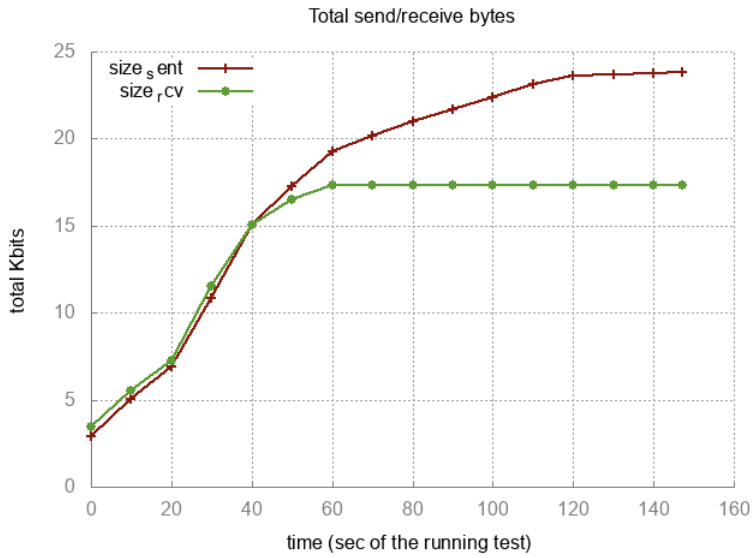
En la carpeta *performance_results* se pueden ver los resultados del testing de stress que se llevó a cabo.

En la carpeta *performance_results* > *comparison* se puede ver, en gráficos, la comparación de correr dos veces el testing de stress.

En el gráfico siguiente se pueden ver los resultados de conexiones simultáneas que soporta y conexiones TCP/UDP abiertas.



En el gráfico siguiente se pueden ver la cantidad total de bytes recibidos y enviados.



Las pruebas realizadas nos sirvieron para comprobar que el rendimiento del proxy es aceptable al contar con una gran cantidad de conexiones concurrentes.

4. Problemas encontrados

4.1. Durante el proceso de diseño

Se discutió y analizó extensamente el diseño antes de comenzar a implementarlo. Sin embargo, un problema encontrado a lo largo del mismo fue poder parsear los XML eficientemente. Es decir, considerando los distintos casos en que un *tag XML* puede estar abierto o cerrado.

Inicialmente parseabamos el *tag XML* tal cual los recibíamos y luego parseabamos el tag que cerraba. Sin embargo, obviamente el problema ocurrió cuando en algunos escenarios la información que el usuario mandaba era mucho más grande que el tamaño estático que contenía nuestro buffer.

Se terminó decidiendo implementar buffers dinámicos.

4.2. Durante la implementación de protocolos

Luego de que las características escenciales de los protocolos fueron implementadas, nos dimos cuenta que no funcionaba tal cual lo habíamos planeado inicialmente. Encontrar las causas de los errores, fue lo que más tiempo nos llevó pero fue lo que nos permitió iterar y alcanzar el objetivo deseado.

4.3. Durante la implementación de aplicaciones

Dado que el proyecto tiene una estructura compleja, nos resultó fácil pensar e implementar funcionalidades al inicio del proyecto. Sin embargo, esto no fue así hacia el final del proyecto, ya que en algunas oportunidades perdíamos la perspectiva del problema que estabamos atacando y muchas funcionalidades terminaban solapandose entre sí. Quizás más experiencia en el desarrollo de estas aplicaciones, haga que un futuro desarrollo o extensión de este proyecto sea más maduro para evitar estas dificultades.

Encontramos que hacer tests era mucho más complicado de lo que habíamos pensado inicialmente ya que la compleja funcionalidad del proyecto hace que los tests no sean tan sencillos de implementar.

De todas maneras, vale destacar que pudimos hacer testeo manual localmente y en un servidor XMPP instalado en AWS.

5. Ejemplos de prueba

5.1. Ejemplos de funcionalidades

Ejemplo 1: Debería loguearse antes de habilitar el modo l33t.

```
<globoh>
<?xml version="1.0">
<hello>
<l33t>on</l33t>
<error>
    <code>401</code>
    <message_description>Unexpected command</message_description>
</error>
```

Ejemplo 2: Elegante autenticación de un cliente.

```
<globoh>
<?xml version="1.0">
<hello>
<auth>glh:0000</auth>
<ok/>
```

Ejemplo 3: Uso de mute.

```
<mute value="on">e@example.com</mute>
<ok/>
```

Ejemplo 4: Multiplexación de un usuario

```
<server_multiplex user="w@example.com">other.com</server_multiplex>
<ok/>
```

Ejemplo 5: muestra los usuarios multiplexados

```
<data>multiplex</data>
<ok>
    <option>multiplex</option>
    <description>List of multiplexed users: (Format:
        user->host)</description>
    <value>
        user1->host:7890
        user2->host:4567
        w@example.com->other.com/64.71.34.22:5222
    </value>
</ok>
```

Ejemplo 6: muestra los usuarios muteados

```
<data>muted_users</data>
```

```
    <ok>
        <option>muted_users</option>
        <description>List of muted users:</description>
        <value>
            muted2@localhost.com
            e@example.com
            muted@localhost.com
        </value>
    </ok>
```

Ejemplo 7: uso de l33t

```
<l33t>on</l33t>
<ok/>
```

5.2. Ejemplos de errores

Ejemplo 8: Se envía un XML que el servidor no acepta

```
<data/>
<error>
    <code>400</code>
    <message_description>Not well formed XML</message_description>
</error>
```

Ejemplo 9: El cliente ya está logueado (no puede volver a loguearse)

```
<auth>anakin:skywalker</auth>
<error>
    <code>401</code>
    <message_description>Unexpected command</message_description>
</error>
```

Ejemplo 10: read_bytes no espera un usuario.

```
<data user="qe@example.com">read_bytes</data>
<error>
    <code>402</code>
    <message_description>Bad command format</message_description>
</error>
```

Ejemplo 11: Se pide una métrica (not_valid) que el servidor no mide.

```
<data>not_valid</data>
<error>
    <code>403</code>
    <message_description>Bad command option</message_description>
</error>
```

Ejemplo 12: Incorrecto uso del comando *multiplex*

```
<multiplex user="w@example.com">other.com</multiplex>
<error>
    <code>404</code>
    <message_description>Unrecognized command</message_description>
</error>
```

Ejemplo 13: Autenticación incorrecta

```
<globoh>
<?xml version="1.0">
<hello>
<auth>i:dunno</auth>
<error>
    <code>405</code>
    <message_description>Incorrect username or
        password</message_description>
</error>
```

6. Instrucciones para la configuración

Detallamos las instrucciones para la configuración del proyecto.

Ver el archivo de configuración del proxy el cuál es *globoh.proxy.properties* donde se pueden modificar las propiedades que se muestran en el siguiente archivo de ejemplo.

```
1   xmpp.server.name = flowics.com.ar
2   server.default.host = localhost
3   server.default.port = 5222
4   l33t.default = false
5   users.muted = muted@localhost, muted2@localhost
6   users.multiplexed = user1->host:7890, user2->host:4567
7   globoh.username = glh
8   globoh.password = 0000
9   buffer.size.bytes = 1024
10  timeout.value.ms = 4000
```

Para poder ver en la consola logs útiles para debuggear, es necesario usar la aplicación en modo *DEBUG*. Para ello, es necesario agregar como parámetros de ejecución, sin espacios, esto: *-Dorg.slf4j.simpleLogger.defaultLogLevel=debug*

7. Guía de instalación

Se recomienda ver primero la sección *Instrucciones para la configuración* donde se setean correctamente el server, el host default y el nombre del servidor XMPP.

Compilar el proyecto con *Maven*, se recomienda usar un IDE que integre *Maven*.

Para correr el proyecto, ejectuar los siguientes comandos en una terminal, estando ubicado en la raíz del proyecto.:

- mvn clean package
- java -cp target/GLoboH-jar-with-dependencies.jar ar.edu.itba.pdc.Main

Es importante destacar que este proyecto usa el Java SE Development Kit 8. No se podrá compilar si el SDK utilizado no es este.

8. Ejemplos de monitoreo

A continuación se muestran ejemplos de monitoreo del proxy.

Ejemplo 1: cantidad de mensajes enviados

```
<data user="e@example.com">sent_messages</data>
<ok>
    <option>sent_messages</option>
    <description>Number of messages sent by e@example.com through
        proxy</description>
    <value>0</value>
</ok>
```

Ejemplo 2: cantidad de bytes leídos

```
<data>read_bytes</data>
<ok>
    <option>read_bytes</option>
    <description>Number of read bytes</description>
    <value>17320</value>
</ok>
```

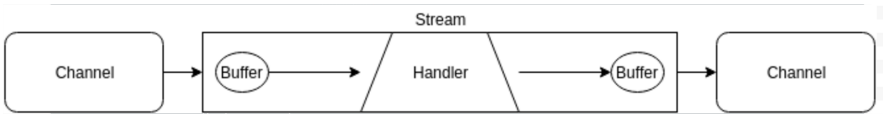
9. Sobre el diseño del proyecto

En cuanto al diseño del proyecto en general, las ideas principales sobre las que funciona este servidor proxy son las mencionadas en esta sección.

En primer medida se pensó en tener un manager de protocolos. Luego pudimos establecer una implementación de esa interfaz.

Esto nos permitió definir luego lo que es un protocolo y usar esto como punto de partida para escalar el diseño e implementación de las otras capas de protocolos.

Además, implementamos una conversación TCP. Esto escala luego en *XML* y *XMPP*. Una conversación es la que se puede ver en el stream representado en el siguiente gráfico.



Para los streams, hay distintos tipos de handlers que varían en cuanto al *request* que reciben y la *response* que retornan.

10. Librerías utilizadas

Para un mejor seguimiento de estas, se recomienda ver el archivo *pom.xml* en la raíz del proyecto.

De todas maneras, se listan a continuación:

- SLF4J: para generar los logs.
- Aalto XML: para parsing no bloqueante de streams XML.
- Apache Commons Lang 3: para codificacion y decodificación. *Base64*.

11. Limitaciones de la aplicación

La implementación de este servidor proxy tiene ciertas limitaciones en su funcionamiento. Las que consideramos más importantes son detalladas a continuación.

La limitación más importante, sobre todo por ser un servidor proxy XMPP es que no se soportan mecanismos seguros de autenticacion de usuarios. Esto es, la autenticación es mediante texto plano. Una posible mejora para futuras versiones es mencionada en la sección siguiente a esta, *Posibles extensiones*.

Otra limitación que consideramos muy importante es que nuestra aplicación no puede utilizar multiples hilos de ejecución, es *single-thread*. Obviamente el performance podría ser mejor si fuera una aplicación *multi-thread*. Algo importante para destacar es que cada *thread*, o hilo de ejecución, cuenta con un selector. Esto permite mejorar el performance con muchos usuarios.

Otra pequeña limitación es que los logs no se guardan en un archivo. Al reiniciarse el servidor, los logs se pierden y no queda registro de lo ocurrido en algún momento del pasado. Se comenta más sobre esto en la próxima sección.

12. Posibles extensiones

Explicamos las posibles extensiones que podría tener este proyecto.

Una gran mejora sería poder utilizar algún tipo de autenticación que no sea de texto plano.

Idealmente el servidor debería contar con una implementación del framework de autorización OAuth 2.0. Esto lo hace, por ejemplo, Ejabberd el cual es el servidor XMPP más robusto del mercado. Simplemente tener esta *feature* permitiría a este proyecto poder ser integrado a aplicaciones Web o que hagan uso del protocolo HTTP. Además, usando tokens OAuth se puede esconder la contraseña del cliente XMPP mismo, entre

otras ventajas.

Otra posible mejora sería implementar un servidor que haga *multi-threading*, esto es, que tenga varios hilos de ejecución. Esto mejoraría ampliamente la performance y posible escalabilidad del proyecto en sí.

Una simple y gran mejora sería guardar los logs del servidor, incluyendo un archivo aparte que también incluya los que se producen en el modo *DEBUG*, en dos archivos por separado y por fecha en que comenzó cada instancia del servidor. Separando así cada vez que se reinició el servidor.

13. Conclusiones

Gracias a que la consigna del trabajo practico contempló, tanto factores de funcionalidad como también de performance y además el uso de librerías externas, pudimos tener una mejor noción del protocolo XMPP y del desarrollo de aplicaciones que para su funcionamiento necesitan un amplio conocimiento y manejo adecuado de conceptos como la concurrencia y el performance.