Pre-entrega Trabajo Práctico Especial

Protocolos de Comunicación

Andrés Mata Suarez - 50143
Jimena Pose - 49015
Pablo Ballesty - 49359

2011 - Segundo cuatrimestre

Índice

1. Documentación relevante para el desarrollo

Para el anásis del trabajo práctico se utilizan los siguientes documentos

- RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core
- RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence
- RFC 6122: Extensible Messaging and Presence Protocol (XMPP): Address Format
- XEP 0096: SI File Transfer

Posiblemente sean necesarios los siguientes

- RFC 4854: A Uniform Resource Name (URN) namespace for use in XMPP extensions
- RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2
- RFC 2222: Simple Authentication and Security Layer (SASL)

2. Protocolos a desarrollar

A continuación se presenta el RFC del protocolo *configurotocol 1.0*, diseñado para manejar la configuración del servidor proxy.

```
                        Configurotocol 1.0


Resumen
Configurotocol es un protocolo desarrollado para poder configurar en tiempo real
y de forma remota la aplicación proxy Isecu.

Estado del documento
Este protocolo forma parte de la entrega del Trabajo Práctico Especial para la
materia de Protocolos de Comunicación, de la carrera Ingeniería Informática del
Instituto Teconológico de Buenos Aires (ITBA).


1. Introducción

Configurotocol es un protocolo sin estado que permite a una entidad cliente con-
sultar y setear parámetros de configuración de la aplicación proxy Isecu.
Se basa en el formato JSON (http://www.json.org).

2. Descripción

El protocolo está diseñado para poder ejecutar comandos de configuración en tiem-
po real y de forma remota. Un comando va a estar formado por instrucciones. A
continuación se definen ambos conceptos y sus respectivos formatos.

2.1 Objeto

Un objeto es un conjunto de 2 elementos: clave y valor. Un objeto es igual a otro
si ambos contienen la misma clave. La clave es case insensitive y el valor es
key sensitive.
Un objeto puede ser:

                    simple -->   clave : "valor"

                                    ó

          compuesto --> clave : {k1:"valor1",k2:"valor2", ... , kn:"valorn"}
```

2.2 Comando

Un comando es un conjunto de objetos dentro de llaves. Los objetos están separados por comas. Que un comando sea un conjunto, implica que no contiene elementos repetidos y que el orden en que se encuentran es irrelevante.
Ejemplo de comando:

```
{nombre : "Thulsa", apellido : "Doom"}
```

3. Comandos válidos

Anteriormente se definió el formato de los comandos; en esta sección se especifican los comandos que soporta el protocolo.
Un comando es considerado válido si cumple con las siguientes reglas:

A. Respeta el formato de comando especificado en 2.2.
B. Posee al menos tres objetos, "user", "password" y "type", siendo todos ellos de formato simple.
C. El objeto "type" debe tener como valor "query" o "assignation".
D. En caso de ser del tipo "query" debe cumplir con lo especificado en 3.1.
E. En caso de ser del tipo "assignation" debe cumplir con lo especificado en 3.2.

3.1 Comandos de tipo "query"

Un comando que cumple con las reglas A, B, C y es del tipo "query" se considera válido si además posee un objeto con clave "parameter", cuyo valor sea alguno de los siguientes:

- server
- blacklist
- caccess
- multiplex
- silence
- filter

3.2 Comandos de tipo "assignation"

Un comando que cumple con las reglas A, B, C y es del tipo "assignation" se considera válido si posee al menos uno de los siguientes objetos, y son todos válidos:

3.2.1 Server

El objeto server debe ser de formato compuesto, y debe tener exactamente dos valores. El primero debe ser la dirección del servidor y el segundo el puerto.
Ejemplo:

```
{ user:"foo", password:"foo", type:"assignation",
server:{origin:"xmpp.example.com",port:"5222"}
```

3.2.2 Blacklist

El objeto blacklist debe ser de formato compuesto y de algún tipo de los siguientes:

Tipo "range":

```
 blacklist:{type:"range",user:"user",from:"from",to:"to"}
```

Tipo "logins":

```
 blacklist:{type:"logins",user:"user",qty:"qty"}
```

Tipo "net":

```
 blacklist:{type:"net",ip:"ip"}
 blacklist:{type:"net",network:"network",netmask:"netmask"}
```

3.2.3 Acceso concurrente

```
El objeto caccess debe ser de formato compuesto y debe respetar el siguiente
formato:
 caccess:{user:"user",qty:"qty"}

3.2.4 Multiplexador de cuentas
El objeto multiplex debe ser de formato compuesto y debe respetar el siguiente
formato:
  multiplex:{jid:"jid", serverto:"serverto"]

3.2.5 Silenciar usuarios
El objeto silence debe ser de formato simple y debe respetar el siguiente forma-
to:
 silence:"user"

3.2.6 Filtros
El objeto filter debe ser de formato compuesto y debe respetar el siguiente for-
mato:
 filter{name:"filtername",status:"on/off"}

4. Respuestas
Luego de la ejecución de un comando la aplicación enviará una respuesta.
Toda respuesta va a contener el objeto status, con valor "OK" o "ERROR" corres-
pondiente a una respuesta satisfactoria o no, respectivamente. Luego de cualquier
comando de tipo "assignation" se va a responder con una respuesta conformada
sólo con un objeto status. En caso de que el comando haya sido de tipo "query"
los resultados de esa query se enviarán en formato compuesto o simple, depen-
diendo de la cantidad de resultados obtenidos, en el objeto "data".
```

3. Potenciales problemas y dificultades

3.1. Comunicación encriptada

Como se especifica en el RFC 6120, cuando dos entidades quieren comunicarse, primero negocian qué método de encriptación utilizarán, es decir, el server manda sus *features* y el cliente avisa cuál utilizará. Luego, comienzan nuevamente la sesión, utilizando el método escogido.

Para el servidor proxy, esto es sin dudas un problema: luego de que ambas entidades escojan un protocolo de encriptación (los dos que se mencionan son TLS y SASL) los paquetes comienzan a viajar encriptados y por ende, se vuelve dificultosa la implementación de algunos de los requerimientos que se piden (en especial los de transformaciones de datos, como es el filtro de l33t).

Para evitar la llegada de paquetes asegurados al proxy, se decide que una vez negociado el método a utilizar por las dos entidades, el server proxy abrirá dos endpoints de conexión, uno para cada extremo del intercambio, de manera tal que sepa encriptar y desencriptar los mensajes de cada parte de la conexión. Ejemplificando, si quiere reenviar todo lo recibido del cliente al servidor, previamente deberá desencriptarlo y luego volver a encriptarlo.

3.2. Transparencia

El proxy XMPP a desarrollar debería actuar de manera completamente invisible frente a cada par de entidades (cliente y servidor) conectadas. En otras palabras, el flujo de información que envía una parte debería arribar sin ningún tipo de alteración al otro extremo de la conexión, siempre y cuando no estén activas ninguna de las funciones de transformación ofrecidas por la aplicación. Por su parte, el resto de los requerimientos funcionales no deberían modificar en lo más mínimo el *stream* XML enviado entre pares.

3.3. Concurrencia

Uno de los problemas que deberá enfrentar el server proxy es una cantidad grande de conexiones, es decir, atender a grandes cantidades de clientes. Mas allá, que este problema se puede paliar planteando una arquitectura con varios servidores proxies, e ir balanceando la concurrencia, la idea es que además cada servidor proxy maneje de manera performante la concurrencia, y pueda servir la mayor cantidad de clientes. Para esto, dentro de lo posible se intentará utilizar N selectores cada uno corriendo en un *Thread*, siendo N la cantidad de procesadores que tenga el host donde se corra la aplicación, y se van a ir asignando sockets a cada selector mediante un algoritmo Round-Robin.

Se ponen *N Threads* igual a la cantidad de procesadores, para que, en lo posible corran en paralelo realmente, y no perder tiempo de *scheduling*.

3.4. Manejo de grandes *streams* de información

Los *streams* de información que se consideren de gran tamaño deben ser almacenados temporalmente en disco. Obviar esta política implicaría riesgo de agotación de la memoria y, en consecuencia, la pérdida de la estabilidad deseada en la aplicación. Debido a esto, es necesario implementar un módulo eficiente con la función de mantener ciertos *streams* en disco e ir llevándolos a memoria a medida que sea necesario. La aplicación no debería dejar de funcionar (o comenzar a funcionar erráticamente) debido al tamaño de los *streams* que se manejen.

De nuevo, se espera que el manejo de recursos sea eficiente: no es deseable la creación excesiva de archivos en disco así como tampoco lo es la no eliminación de archivos que ya no están en uso por la aplicación.

3.5. Controles y aplicaciones externas

En caso de que algún control efectuado por la aplicación de proxy (por ej.: control de accesos) tenga efecto sobre cierta entidad en una conexión, el mensaje de error devuelto deberá tener la forma de un mensaje de error XMPP. Una aplicación externa que utilize el proxy no debería tener motivos para sospechar que un error del tipo mencionado no proviene del otro *endpoint* de conexión.

4. Ambiente de desarrollo y testing

La aplicación será desarrollada bajo entorno Unix/Linux.

Para la realización de testeos de unidad, se utilizará el framework **JUnit** (`http://www.junit.org/`).

Para testeos de carga (*stress-testing*), se utilizará la aplicación **JMeter** (`http://jakarta.apache.org/jmeter/`).

Como servidor de XMPP se utilizará **ejabberd** (`http://www.ejabberd.im/`); el mismo puede instalarse en Ubuntu directamente desde los repositorios ejecutando desde consola:

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

Como cliente se utilizará **Pidgin** (`http://www.pidgin.im/`); el mismo puede instalarse en Ubuntu directamente de los respositorios ejecutando desde consola:

```
sudo apt-get install pidgin
```

5. Casos de prueba a realizar

Para la ejecución de los casos de prueba, se va a contar con los siguientes usuarios:

- **foo** con contraseña **123123123**
- **bar** con contraseña **123123123**
- **baz** con contraseña **123123123**
- **foobar** con contraseña **123123123**

5.1. Testeos de requerimientos funcionales

Funcionalidad	Por rango de horarios
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para exigir que el usuario **foo** sólo pueda acceder de 6:00 a 18:00 horas. Iniciar sesión con dicho usuario en dicho rango horario.
Resultado esperado	El usuario **foo** inicia sesión sin problemas.

Funcionalidad	Por rango de horarios
Tipo de test	Negativo
Descripción	Utilizar el protocolo *configurotocol* para exigir que el usuario **foo** sólo pueda acceder de 6:00 a 18:00 horas. Iniciar sesión con dicho usuario fuera de ese rango horario.
Resultado esperado	El usuario **foo** no tiene permitido el inicio de sesión en dicho rango horario. Se recibe un error acorde al protocolo.

Funcionalidad	Por rango de horarios
Tipo de test	Negativo
Descripción	Utilizar el protocolo *configurotocol* para exigir que el usuario **foo** sólo pueda acceder de 19:00 a 06:00 horas. Iniciar sesión con dicho usuario dentro de ese rango horario.
Resultado esperado	El usuario **foo** no tiene permitido el inicio de sesión en dicho rango horario. Se recibe un error acorde al protocolo. Que el configurotocol permita este seteo.

Funcionalidad	Por cantidad de logins exitosos por usuario y día
Tipo de test	Positivo
Descripción	Asegurarse de que el usuario **bar** no haya iniciado sesión anteriormente en el día. Utilizar el protocolo *configurotocol* para exigir que el usuario **bar** sólo pueda acceder al sistema un máximo de 1 (una) vez por día. Iniciar sesión en el sistema como dicho usuario.
Resultado esperado	El usuario **bar** inicia sesión sin problemas.

Funcionalidad	Por cantidad de logins exitosos por usuario y día
Tipo de test	Negativo
Descripción	Asegurarse de que el usuario **bar** no haya iniciado sesión anteriormente en el día. Utilizar el protocolo *configurotocol* para exigir que el usuario **bar** sólo pueda acceder al sistema un máximo de 1 (una) vez por día. Iniciar sesión en el sistema como dicho usuario. Cerrar sesión. Iniciar sesión una vez más.
Resultado esperado	El usuario **bar** ya cumplió su cuota diaria de accesos. El segundo login no es aceptado. Se devuelve mensaje de error acorde al protocolo.

Funcionalidad	Por lista negra (dirección IP)
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para impedir conexiones entrantes de la dirección 192.168.1.50. Iniciar sesión como **baz** desde la dirección 192.168.1.51.
Resultado esperado	El usuario **baz** inicia sesión sin problemas.

Funcionalidad	Por lista negra (dirección IP)
Tipo de test	Negativo
Descripción	Utilizar el protocolo *configurotocol* para impedir conexiones entrantes de la dirección 192.168.1.50. Iniciar sesión como **baz** desde la dirección 192.168.1.50.
Resultado esperado	La dirección 192.168.1.50 se encuentra en la lista negra. No se permite el inicio de sesión. Se devuelve mensaje de error acorde al protocolo.

Funcionalidad	Por lista negra (redes IP)
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para impedir conexiones entrantes del rango de direcciones 192.168.1.0/25. Iniciar sesión como **foobar** desde la dirección 192.168.1.130. Iniciar sesión como **baz** desde la dirección 192.168.1.254.
Resultado esperado	Ambos usuarios inician sesión sin problemas.

Funcionalidad	Por lista negra (redes IP)
Tipo de test	Negativo
Descripción	Utilizar el protocolo *configurotocol* para impedir conexiones entrantes del rango de direcciones 192.168.1.0/25. Iniciar sesión como **foobar** desde la dirección 192.168.1.126. Iniciar sesión como **bar** desde la dirección 192.168.1.10.
Resultado esperado	No se permite ninguno de los dos inicios de sesión. Se devuelven mensajes acordes para cada instancia de la aplicación.

Funcionalidad	Por cantidad de sesiones concurrentes
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para restringir la cantidad máxima de sesiones concurrentes del usuario **baz** a 3. Iniciar sesión como dicho usuario desde 2 clientes distintos.
Resultado esperado	Se permite el inicio de sesión en cada instancia del programa.

Funcionalidad	Por cantidad de sesiones concurrentes
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para restringir la cantidad máxima de sesiones concurrentes del usuario **baz** a 3. Iniciar sesión como dicho usuario desde 3 clientes distintos. Utilizar un nuevo cliente e iniciar sesión nuevamente.
Resultado esperado	Se permite el inicio de sesión en el último cliente, y el primer cliente que se utilizó se desconecta recibiendo un error acorde al protocolo.

Funcionalidad	Por silenciamiento de usuarios
Tipo de test	Positivo
Descripción	Utilizar el protocolo *configurotocol* para asegurarse de que el usuario **foobar** no se encuentre silenciado. Iniciar sesión como dicho usuario y emitir mensajes hacia el usuario **foo** que estará conectado. Luego el usuario **foo** emite mensajes hacia **foobar**.
Resultado esperado	Los usuarios se comunican sin problemas.

Funcionalidad	Por silenciamiento de usuarios
Tipo de test	Negativo
Descripción	Utilizar el protocolo *configurotocol* para silenciar al usuario **foobar**. Iniciar sesión como dicho usuario y emitir mensajes hacia el usuario **foo** que estará conectado. Luego **foo** envía mensajes hacia **foobar** y hacia **bar**.
Resultado esperado	**foo** y **foobar** no pueden comunicarse, y **bar** recibe los mensajes de **foo**.

Funcionalidad	Filtros(L33t)
Tipo de test	Positivo
Descripción	Mediante *configurotocol* activar el filtro de **L33t**. Emitir mensajes desde **foo** hacia **bar** que posean vocales.
Resultado esperado	**bar** recibe los mensajes en formato **L33t**.

Funcionalidad	Filtros(L33t)
Tipo de test	Positivo
Descripción	Mediante *configurotocol* activar el filtro de **L33t**. Emitir mensajes desde **foo** hacia **bar** que posean vocales.
Resultado esperado	**bar** recibe los mensajes en formato **L33t**.

Funcionalidad	Filtros(Hash)
Tipo de test	Positivo
Descripción	Mediante *configurotocol* activar el filtro de **Hash**. Enviar un archivo desde **foo** hacia **bar**. Asegurarse que se envía el hash correspondiente.
Resultado esperado	**bar** recibe el archivo sin problemas, con el hash generado.

Funcionalidad	Filtros(Hash)
Tipo de test	Positivo
Descripción	Mediante *configurotocol* activar el filtro de **Hash**. Enviar un archivo desde **foo** hacia **bar**. Quitar el hash del mensaje.
Resultado esperado	**bar** recibe el archivo sin problemas, con el hash generado.

Funcionalidad	Filtros(Hash)
Tipo de test	Negativo
Descripción	Mediante *configurotocol* activar el filtro de **Hash**. Enviar un archivo desde **foo** hacia **bar**. Alterar el hash.
Resultado esperado	**bar** no recibe el archivo.

5.2. Testeos de requerimientos no funcionales

Requerimiento no funcional	Envío de archivos
Tipo de test	Positivo
Descripción	Desde el usuario **foo** enviar un archivo a **bar** de un tamaño de 30Mb.
Resultado esperado	**bar** recibe el archivo sin problemas.

Requerimiento no funcional	Concurrencia
Tipo de test	Positivo
Descripción	Configurar JMeter para que realice n conexiones al servidor proxy y emita mensajes. Ir aumentando n.
Resultado esperado	Para n aceptable (analizaremos durante el desarrollo cuanto sería un n aceptable) no se rechazan conexiones y pueden emitirse los mensajes.