Trabajo Práctico Especial

Protocolos de Comunicación

Revisión correspondiente a la entrega:

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

2011 - Segundo cuatrimestre

Índice

1. Descripción detallada del protocolo desarrollado

1.1. RFC

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. Tanto la clave como el valor son case
sensitive.
Un objeto puede ser:

                    simple -->   "clave" : "valor"

          compuesto --> "clave" : ["valor1", "valor2", ... , "valorn"]

O puede contener otro objeto:

        {"calve":"valor","claveObjeto":{"clave1":"valor2","clave2":"valor2"}}

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 dos objetos, "auth" (Especificado en la sección 3.1) y "type" (Especificado en la sección 3.2).
C. En caso de ser del tipo "query" debe cumplir con lo especificado en 3.3.
D. En caso de ser del tipo "assignation" debe cumplir con lo especificado en 3.4.
E. En caso de ser del tipo "delete" debe cumplir con lo especificado en 3.5.

3.1 Objeto "auth"

El objeto auth es obligatorio, debe tener el siguiente formato:

```
"auth" : ["user", "pass" ]
```

Donde "user" corresponde al nombre de usuario del administrador y pass a su contraseña.

3.2 Objeto "type"

El objeto type es obligatorio, debe ser de tipo simple y contener alguno de los siguientes valores:

- "query"
- "assignation"
- "delete"

3.2.1 Comandos de tipo "query"

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

- caccess
- multiplex
- silence
- leet
- hash
- rangeBlacklist
- loginsBlacklist
- ipBlacklist
- netBlacklist

3.2.2 Comandos de tipo "assignation"

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

3.2.2.1 Blacklist

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

- Tipo "range":
  "blacklist":["range","jid","10:15:00","18:30:00"]

- Tipo "logins":
  "blacklist":["logins","jid","5"]

- Tipo "ip":
  "blacklist":["ip","10.0.0.3"]

- Tipo "net":

```
      "blacklist":["net", "10.0.0.0/24"}
```

3.2.2.2 Acceso concurrente

El objeto caccess debe ser de formato compuesto y debe respetar el siguiente formato:

```
   "caccess":["jid", "3"]
```

3.2.2.3 Multiplexador de cuentas

El objeto multiplex debe ser de formato compuesto y debe respetar el siguiente formato:

```
   "multiplex":["jid", "10.0.0.1"]
```

3.2.2.4 Silenciar usuarios

El objeto silence debe ser de formato simple y debe respetar el siguiente formato:

```
   "silence":"jid"
```

3.2.2.5 Filtros

El objeto filter debe ser de formato compuesto y debe respetar el siguiente formato:

```
   "filter":["filtername", "jid", "state"]
```

Donde filtername puede ser leet o hash y state puede ser on u off.

3.2.3 Comandos de tipo "delete"

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

3.2.3.1 Blacklist

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

- Tipo "range":

```
   "blacklist":["range","jid"]
```

- Tipo "logins":

```
   "blacklist":["logins","jid"]
```

- Tipo "ip":

```
   "blacklist":["ip","10.0.0.3"]
```

- Tipo "net":

```
   "blacklist":["net", "10.0.0.0/24"}
```

3.2.3.2 Acceso concurrente

El objeto caccess debe ser de formato simple y debe respetar el siguiente formato:

```
   "caccess":"jid"
```

3.2.3.3 Multiplexador de cuentas

El objeto multiplex debe ser de formato simple y debe respetar el siguiente formato:

```
   "multiplex":"jid"
```

```
3.2.3.4 Silenciar usuarios

El objeto silence debe ser de formato simple y debe respetar el siguiente forma-
to:
   "silence":"jid"

3.2.3.5 Filtros

El objeto filter debe ser de formato compuesto y debe respetar el siguiente for-
mato:
   "filter":["filtername", "jid"]
Donde filtername puede ser leet o hash y state puede ser on u off.

4. Respuestas

Luego de la ejecución de un comando, la aplicación enviará una respuesta en
formato JSON. Toda respuesta va a contener el objeto status, con valor "OK" o
"ERROR" correspondiente a una respuesta satisfactoria o no, respectivamente.
Luego de cualquier comando de tipo "assignation" o "delete" 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, dependiendo de la cantidad de resultados obtenidos,
en el objeto "data".
```

1.2. Consideraciones

Se decidió basar el formato del protocolo en JSON, ya que este es fácil de leer y escribir para los usuarios y es fácil de parsear y generar para las máquinas. Por otro lado, facilita mucho el parseo del archivo de configuración, ya que el mismo también se guarda en formato JSON. Por lo tanto, al momento de configurar el proxy, la generación e interpretación de comandos no debería ser ningún problema, tanto para un usuario como para una máquina.
Es importante destacar que todos los cambios exitosos de configuración son persistidos. Por lo que ante cualquier inconveniente, en caso de que el proxy deba reiniciarse, este inicia con la última configuración persistida
Con el objetivo de facilitar la configuración del proxy, se decidió no solo ofrecer comandos para modificar la misma, si no también comandos para consultarla.
Debido a que el protocolo es sin estado, se debe enviar una autenticación en cada comando. Se decidió desarrollar un protocolo sin estados ya que de esta manera es mucho más fácil el manejo de comandos para una máquina.
En cuanto a la estructura, se puede encontrar el diagrama UML en la figura 4

2. Problemas encontrados durante el diseño y la implementación

A continuación se detallan las decisiones que se han tomado durante el desarrollo del trabajo, para enfrentar los problemas que consideramos críticos.

2.1. Concurrencia

2.1.1. Problema

El servidor proxy debe poder atender una cantidad considerable de sesiones concurrentes, sin dejar a ningún cliente bloqueado sin servicio.

2.1.2. Decisión

Una de las primeras opciones que se planteó fue la de utilizar *SimpleThreads*, la cual consistía en manejar cada socket con un Thread por separado. Esta opción fue descartada al instante, ya que se mantendrían gran cantidad de Threads corriendo, sin estar procesando nada, y pocos procesando la información que fuere enviada por el cliente, lo que resultaría poco performante. Además, se sumaría el tiempo de *scheduling* que crecería en igual o mayor ritmo que las conexiones.

Una vez, descartada la opción anterior, decidimos utilizar Non-blocking Input Output (**Java NIO**). En la pre-entrega del trabajo se propuso utilizar un selector por Thread, y tener n Threads, siendo n la cantidad de núcleos

del procesador donde esté corriendo la aplicación. De esta manera, no se perdería performance por *scheduling*, pero llegado el momento de implementarlo, decidimos utilizar un solo Thread (salvo el caso de archivos que se explicará mas adelante), ya que la implementación de lo propuesto resultaba muy compleja, y acarrería problemas al momento de comunicar dos sockets, o mantener estados conjuntos.

2.2. Parser y modelo no bloqueante

2.2.1. Problema

Habiendo tomado la decisión de seguir un modelo de atención no bloqueante, esto derivó en que cualquier procesamiento que se haga ante un evento del selector, debía ser no bloqueante, ya que de lo contrario, seguiríamos teníendo el problema de la sección anterior.
Los parsers de XML mas conocidos como Xerces, SAX, DOM son tradicionalmente bloqueantes, lo cual no calzó con el modelo que se había planteado.

2.2.2. Decisión

Comenzamos investigando acerca de parsers de XML no bloqueantes, encontramos información sobre *StAX* y *StAX2* (Streaming API for XML) ambos métodos de parsing de stream de XML. En donde la versión 2, utilizaba la métodología de *Pull Parsing* la cual nos permitía ir appendeando *chunks* de datos. La única implementación de StAX2 completa era *WoodStox*, pero esta arrojaba excepciones fatales cuando se appendeaban *chunks* de datos con porciones de XML inválido, cosa que era esperable que pueda suceder. Luego de que *WoodStox* fue descartado encontramos que el mismo desarrollador de la librería, tenía en versión Beta, el parser XML **aaltoXML**, el cual poseía escaza/nula documentación, pero que arrojaba el evento EVENT_INCOMPLETE cuando se appendeaban datos de XML inválido. Luego de revisar los testeos de unidad en el source code de la librería, reconocimos como era su funcionamiento, y decidimos utilizarla.

2.3. Tamaño de buffers

2.3.1. Problema

Antes de haber decidido utilizar *aaltoXML*, teníamos el problema de tener que fijar un tamaño de buffer para lectura del cliente que nos asegure que lleguen las *stanzas* completas. Como el RFC del protocolo no explicita un tamaño máximo de *stanza*, sino que da la opción de enviar mensajes de error si la stanza enviada, es mayor a lo esperado por el servidor, y que cada implementación fije su límite. Nuestra aplicación como Proxy, no podía setearse un valor estipulado.

2.3.2. Decisión

AaltoXML solucionó este problema, entonces se dejó la posibilidad de configurar los tamaños de buffers de lectura, para que el usuario del proxy pueda seleccionar el que convenga según la situación.

2.4. Modificación y edición de mensajes XMPP

2.4.1. Problema

Una vez parseados sin problemas los mensajes, el proxy necesitaba en varios casos editar el contenido de los mismos, para realizar distintas acciones (filtros, hash de archivos, etc). En una primera ocasión se decidió tener un *StringBuffer* marcando con punteros, las distintas secciones, pero esto resultó ser un problema ya que notamos que las ediciones de los mensajes iban a ser muchas.

2.4.2. Descisión

Para lograr la edición de distintas partes de los mensajes del protocolo, se decidió mapear los tipos de mensajes mediante una estructura recursiva. De esta manera se pudieron realizar las ediciones que se necesitaban, y quedó un método cómodo y escalable para realizarlo.

2.5. Transferencia de archivos

2.5.1. Problema

El requerimiento de calcular/validar el hash de los archivos que se envían resultó totalmente un problema, ya que, entre los dos participantes de la transmisión arreglan para comunicarse entre ellos fuera de banda. De esta manera, el

proxy para interceptar esa comunicación y recibir el archivo debía modificar los mensajes de ese *handshaking* para la comunicación se haga hacia él, recibir el archivo, calcular el hash, para luego enviarlo al destinatario original.

2.5.2. Decisión

Por la complejidad del requerimiento, la transferencia del archivo, se realiza en un Thread aparte de toda la aplicación, manteniendo los Threads en un CachedThreadPool, para que, en medida de lo posible, poder reutilizar la Threads que se crean. Se admite que esta decisión, va en contra del modelo no bloqueante que se tomó para todo el resto de la aplicación, aunque como el envío de archivos no tiene la misma frecuencia que los mensajes de XMPP que se transmiten constantemente, no creemos que esta decisión afecte mucho a la performance de la aplicación en general.

3. Limitaciones de la aplicación

3.1. Encriptación y mecanismos de autenticación

La aplicación no permite métodos de encriptación basados en TLS ni relacionados. Es decir, el funcionamiento del Proxy bajo esas circunstancias es impredecible. Por otra parte, debido a la variedad de mecanismos de autenticación SASL que existen para el protocolo, y dada la cuidadosa implementación que cada uno requiere, se volvería un trabajo denso y sin mucho sentido brindar soporte para todos ellos. Por lo tanto, se optó por monopolizar el uso de SASL/DIGEST-MD5 (se encuentra dentro de los *mandatory-to-implement* de los RFC del protocolo) como método predilecto. El proxy analizará las features enviadas por el servidor y eliminará aquellos mecanismos que no sean el mencionado. En caso no quedar ninguno, se devolverá al usuario un elemento ¡failure¿de tipo ¡temporary-auth-failure¿indicando los motivos evidentes.

4. Posibles extensiones

5. Conclusiones

6. Ejemplos de testeo

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**

6.1. 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.

6.2. 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.

7. Guía de instalación

```
 Dependencias:

 - Para la compilación del proyecto es necesario que el sistema cuente con Maven2. El cual
en el caso de Ubuntu puede instalarse
con el siguiente comando:

  > sudo apt-get install maven2

 - Una vez que el sistema cuenta con Maven, se procede a generar el ejecutable con el siguiente
 comando (desde la carpeta raíz del proyecto):

  > mvn clean package
```

8. Instrucciones para la configuración

Para configurar la aplicación, el proxy cuenta con un archivo de parámetros iniciales llamado *init.properties*, el cual puede encontrarse en la raíz.

Los valores que deben setease son:

proxy Ip donde bindear el servicio de proxy.

proxyPort Puerto donde bindear el servicio de proxy.

config Ip donde bindear el servicio de configuración.

configPort Puerto donde bindear el servicio de configuración.

origin Ip del origin server.

originPort Puerto del servicio XMPP del origin server.

bufferSize Tamaño de buffers de lectura a utilizar.

fileTransferBufferSize Tamaño de buffer utilizado para el envío de archivos.

adminUsername Usuario de administración.

adminPassword Contraseña del usuario de administración.

9. Ejemplos de configuración

A continuación se presenta una posible configuración de la aplicación

```
 #Interfaz y puerto del servicio proxy
proxy=0.0.0.0
proxyPort=9999

#Interfaz y puerto del servicio de configuración
config=127.0.0.1
configPort=9998

#Origin server default
```

```
origin=127.0.0.1
originPort=5222

#Buffer size
bufferSize=256

#File transfer buffer size
fileTrasferBufferSize=100

#Admin username and password
adminUsername=admin
adminPassword=admin
```

10. Diseño del proyecto

10.1. Diagramas UML

10.1.1. Handlers

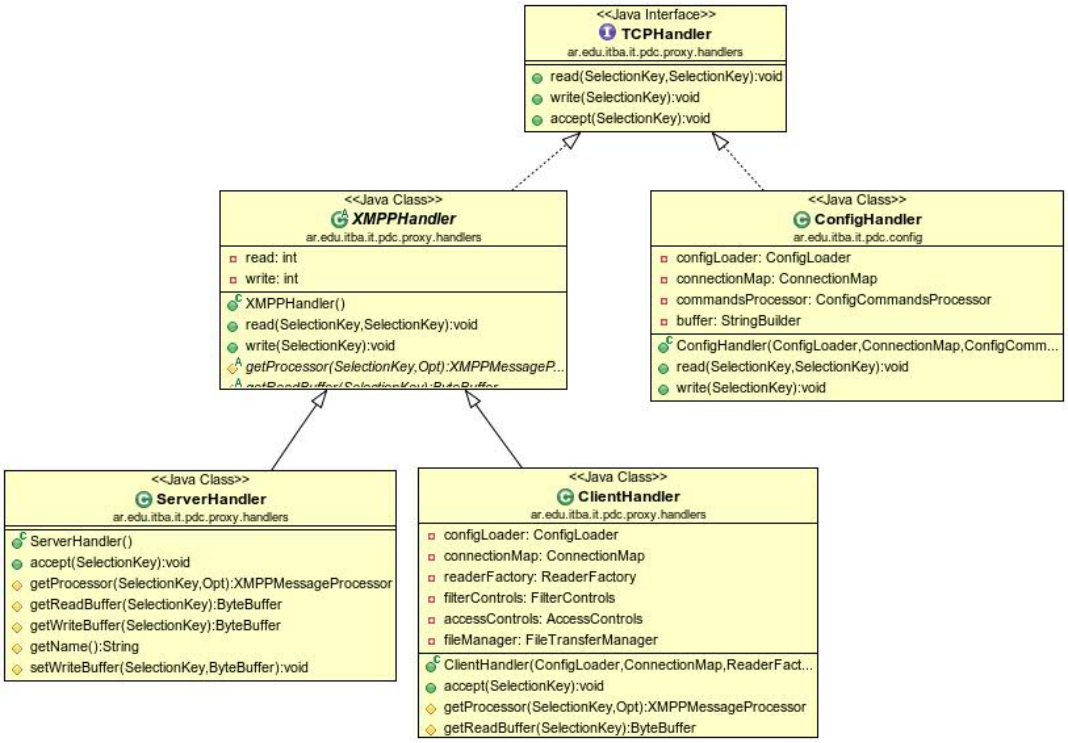

Figura 1: Diagrama UML de los handlers

10.1.2. Elements

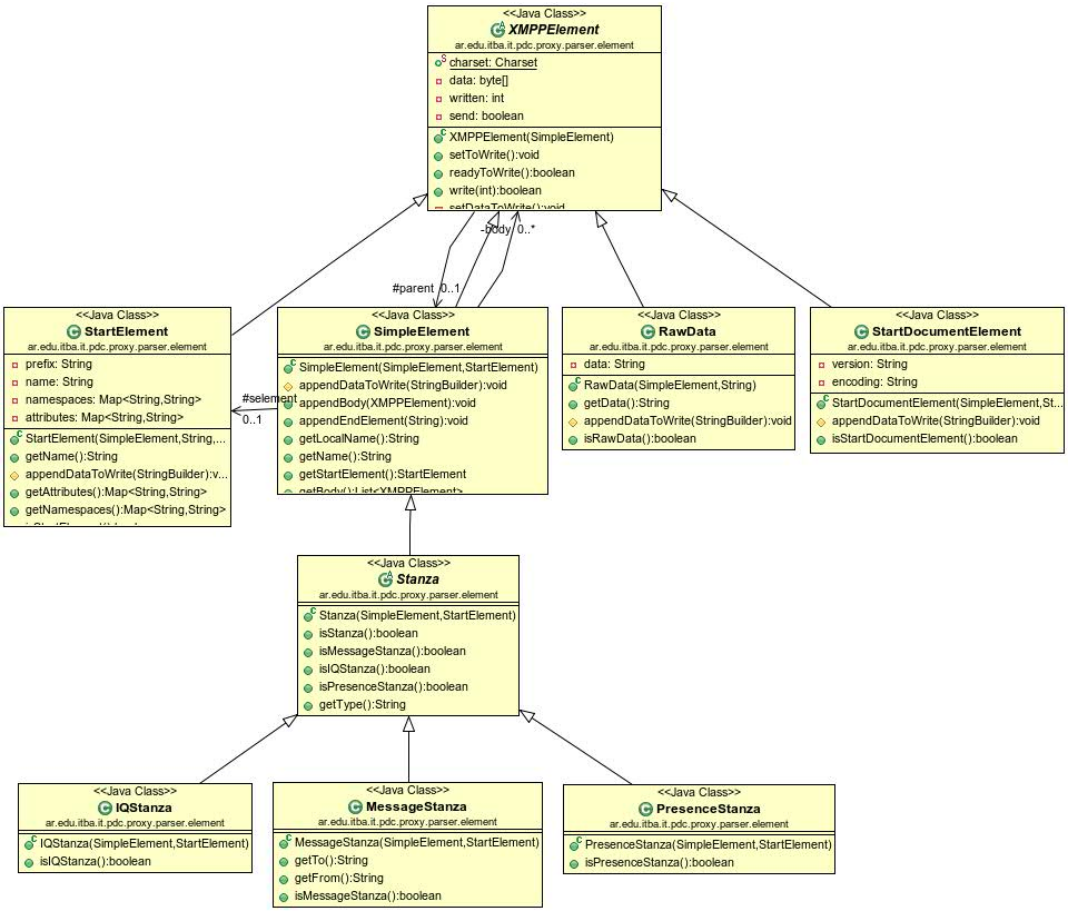

Figura 2: Diagrama UML de los elementos XMPP

10.1.3. Processor

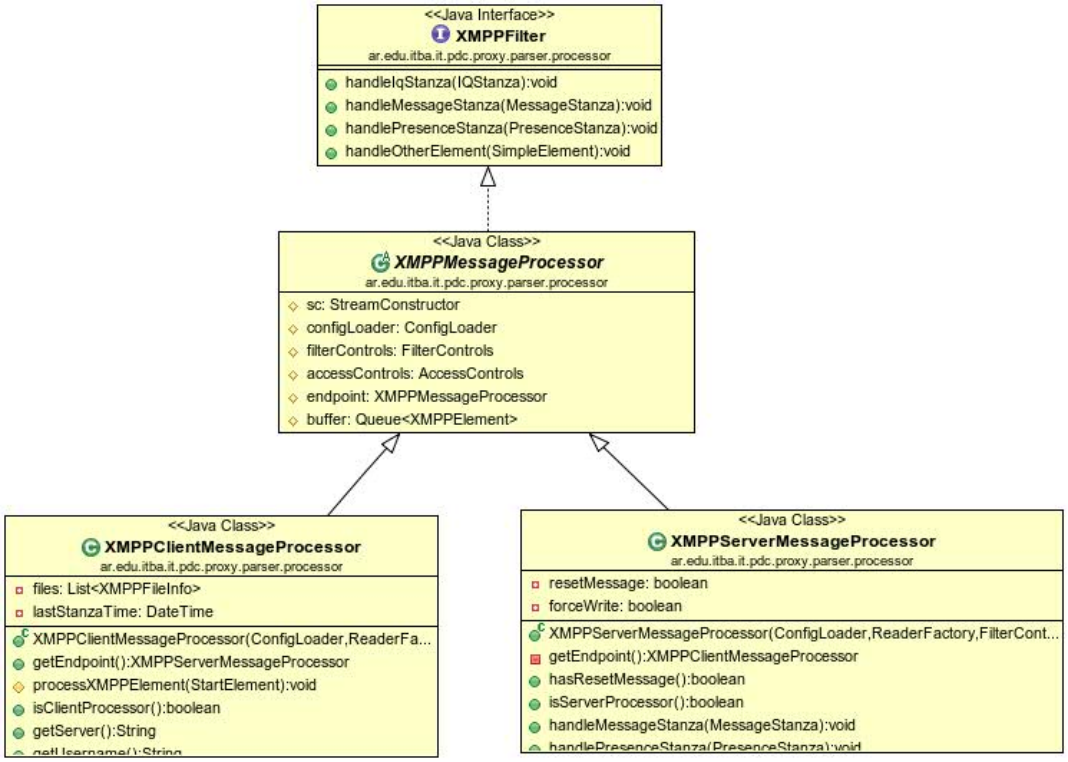

Figura 3: Diagrama UML del mecanismo para procesar los streams

10.1.4. Configuración

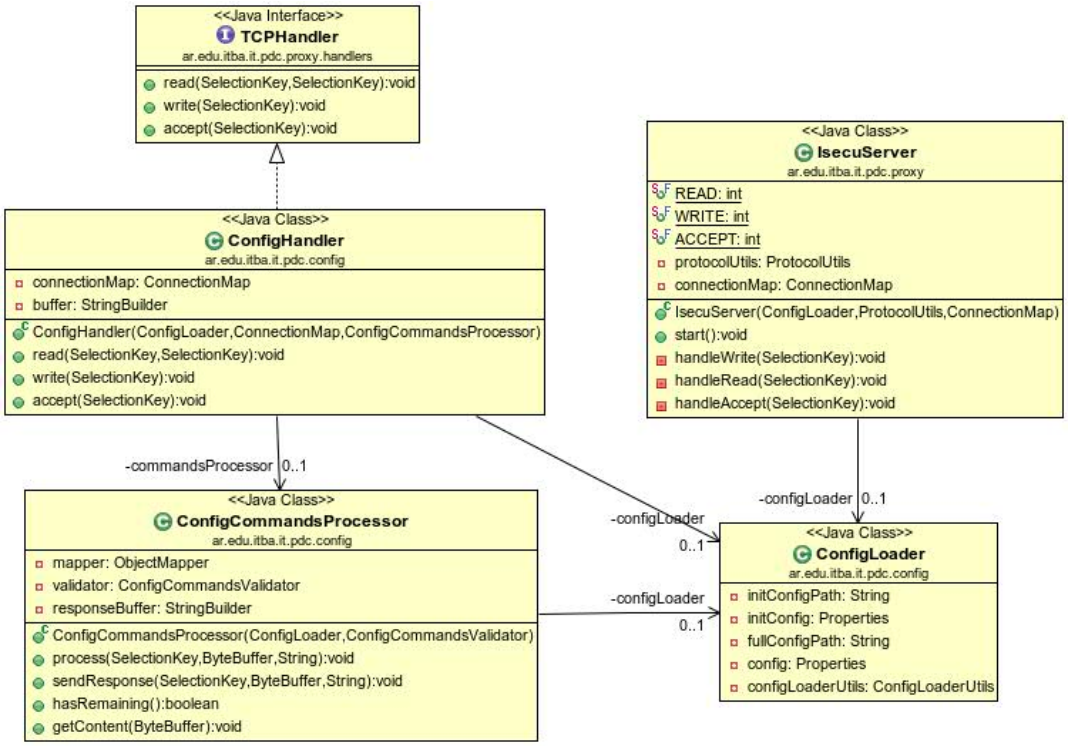

Figura 4: Diagrama UML del protocolo de configuración