Trabajo Práctico Especial

Protocolos de Comunicación

Grupo 4

Integrantes:

Francisco Bartolomé	54308
Natalia Navas	53559
Juan Moreno	54182

ÍNDICE

Introducción:

A continuación se describe la implementación de un Proxy Server para el Extensible Messaging and Presence Protocol Xmpp.

Se hizo uso de la herramienta Gradle para la construcción de dependencias y se utilizaron las siguientes librerías adicionales:

- Aalto XML Parser[1], en las clases donde se interpretaba contenido XML.
- Apache Commons[2] para interpretar archivos de configuración XML.
- Commons Beanutils[3] una dependencia de apache commons.
- Apache Logging Log4j[4] como sistema de logueo.
- Slf4j[5] para el encapsulamiento del sistema de logueo.
- Google Guava[6] para la verificación de precondiciones, como por ejemplo verificación de parámetros.
- Args4j[7] para el parseo de argumentos de aplicación.

El proyecto está divido de la siguiente forma:

- config: Cuenta con los archivos de configuración del proyecto.

[1] https://github.com/FasterXML/aalto-xml
[2] group: 'org.apache.commons', name: 'commons-configuration2', version: '2.1'
[3] group: 'commons-beanutils', name: 'commons-beanutils', version: '1.9.3'
[4] group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
[5] group: 'org.slf4j', name: 'slf4j-api', version: '1.7.21'
[6] group: 'com.google.guava', name: 'guava', version: '19.0'
[7] group: 'args4j', name: 'args4j', version: '2.33'

- dispatcher: Cuenta con archivos que permiten una encapsulación del Selector del servidor y contiene métodos para suscribir y desubscribir canales al mismo.
- io: Cuenta con la clase closeables que ofrece un método para cerrar una conexión.
- net: Cuenta con una clase de definición de una net address, utilizada a la hora de multiplexación de usuarios.
- protocol: Cuenta con las herramientas de ambos protocolos, los parsers, negotiators, etc.
- proxy: Cuenta con la clase de una conexión y el handler de la misma, donde tiene que hacer para leer, escribir, etc.

Descripción de los protocolos:

Protocolo NTTP

El protocolo de desarrollado permite enviar solicitudes las cuales serán analizadas por el servidor, ejecutadas y respondidas. Tanto el servidor como el cliente podrán analizar las respectivas solicitudes y respuestas de forma algorítmica. Las solicitudes, en forma de comandos, están orientadas a configuraciones de aspectos y variables que del servidor proxy que afectarán la forma en que el servidor opera en tanto a los mensajes XMPP, además de proveer métricas referidas al funcionamiento del sistema.

El protocolo NTTP está inspirado tanto en el protocolo HTTP como en línea de comandos, esto se verá posteriormente en el formato de las solicitudes de los clientes y las respuestas del servidor.

NTTP es case insensitive, por lo tanto comandos como “help”, “HELP” y “hElP” resultan equivalentes.

Además, el protocolo provee un sistema de respuestas multilínea, clasificación de respuestas y diferentes tipos de autenticación.

Para comunicarse con el servicio, se debe establecer una conexión TCP con el puerto 1081.

Solicitudes

Las solicitudes del cliente tienen la siguiente estructura:

<command> [--head [value]] (args)*\n

Es decir, el nombre del comando es obligatorio, y luego puede tener un head antecedido por "--", el cual puede tener un valor, y finalmente el comando podrá tener cero o más argumentos. Tanto el comando, como el head, su valor y sus argumentos, están separados por un espacio (' '), y al finalizar el comando, se requiere un '\n'. En cada ejemplo mostrado se omitirá el uso del '\n', por cuestiones de sencillez.

Tipos de comandos:

-hello

El comando es necesario para iniciar la conversación con el servidor. Necesita como head "version" seguida como valor la versión a utilizar. La versión tiene el siguiente formato: "<major>.<minor>", dónde major y minor son números enteros los cuales son incrementales, es decir, 1.2 es menor que 1.12. En el caso de que minor sea 0, se admite obviar dicho número además del "." entre major y minor. Nuestro servidor soporta hasta la versión 1.1.

Si no se logró concretar el hello, no se podrá realizar ningún otro tipo de solicitud y el servidor responderá con el código adecuado.

Cuando se concreta un hello, todas las variables de la sesión serán reiniciadas, por lo tanto si el usuario ya estaba autorizado y realiza un hello luego, entonces éste dejará de estar en sesión.

Ejemplos:

(válido) hello --version 1.1
(válido) hello --version 1
(inválido) hello --version 1,1
(inválido) hello --version 1.1.1
(inválido) hello --version
(inválido) hello version 1.1
(inválido) hello

-help

Sin parámetros ni head: muestra una lista de los posibles comandos. Puede recibir como parámetro el nombre de otro comando, en cuyo caso mostrará una breve descripción de dicho comando.

Ejemplos:

(válido) help
(válido) help hello
(válido) help auth
(inválido) help hello auth

-auth

Este comando es necesario para autorizarse.

Si se utiliza únicamente con el head "methods", se mostrará una lista de los métodos disponibles de autenticación. Nuestro servidor solo soporta el método "simple".

Con el head "request" seguido de su respectivo valor (auth --request <method>), se establece el método por el cual el usuario se autenticará, donde method es el método de autenticación. En caso de que el método no sea soportado, el servidor enviará una respuesta adecuada al cliente informando que dicho método no es soportado.

Una vez que se logre establecer el método de autenticación, se puede utilizar el comando junto con dos parámetros: usuario y contraseña (auth user pass). Si el usuario

y contraseña son correctos, entonces se concretará la autorización, en caso contrario el servidor informará al cliente con una respuesta adecuada.

Si el cliente no está autorizado, sólo tendrá permiso de utilizar los comandos auth, hello, quit y help, en caso de no estarlo el servidor informará al cliente con una adecuada respuesta.

Para lograr autenticación en nuestro servidor, usar usuario admin y contraseña admin (auth admin admin).

Ejemplos:

(válido) auth --methods
(válido) auth --request simple
(válido) auth admin admin
(inválido) auth --methods --request simple
(inválido) auth simple
(inválido) auth admin

-quit

Cierra la conexión entre el cliente y el servidor.

-silence

Este comando requiere de un parámetro "user" (silence user), dónde user será el JID de su cuenta XMPP. Luego de realizar ésta solicitud, el usuario que corresponda al respectivo JID no podrá enviar stanzas <message> XMPP a través de nuestro servidor proxy, entrando a un estado de "silenciado".

Ejemplos:

(válido) silence usuario1
(inválido) silence

-unsilence

Este comando requiere de un parámetro "user" (unsilence user), dónde user será el JID de su cuenta XMPP. Luego de realizar ésta solicitud, el usuario que corresponda al respectivo JID podrá enviar stanzas <message> XMPP a través de nuestro servidor proxy. En caso de que el usuario no esté previamente "silenciado", entonces esta solicitud no tendrá ningún efecto. En caso de que el usuario esté en el estado de "silenciado" previo a realizar la solicitud, entonces el usuario saldrá de dicho estado.

Ejemplos:

(válido) unsilence usuario1
(inválido) unsilence

-transformation

Este comando requiere de un parámetro, el cual sólo podrá ser "true" o "false".

En el caso que que el parámetro sea "true", entonces habilita las transformaciones l33t dentro del servidor proxy. De esta forma el contenido de la etiqueta <body> dentro de todos los stanza <message> de los mensajes que pasen por el servidor proxy serán modificados utilizando el formato l33t.

En el caso que que el parámetro sea "false", entonces deshabilita las transformaciones l33t dentro del servidor proxy. De esta forma el contenido de la etiqueta <body> dentro de todos los stanza <message> de los mensajes que pasen por el servidor proxy no serán modificados.

Ejemplos:

(válido) transformation true
(válido) transformation false
(inválido) transformation

-metrics

Este comando muestra las métricas recolectadas desde que se inicia el servidor proxy. Las métricas incluidas son: bytes transferidos, cantidad de accesos y cantidad de clientes que se conectaron exitosamente.

Si se utiliza sin ningún parámetro, entonces se mostrarán todas las métricas disponibles.

Se puede especificar por un parámetro "metric" (métrica metric) una métrica en particular. El parámetro metric puede ser "bytes", "accesses" o “accepted”.

La métrica bytes devuelve la cantidad de bytes que fueron transferidos a través del proxy server, la métrica accesses devuelve la cantidad de clientes que intentaron conectarse y empezaron con el proceso de negociación y finalmente la métrica accepted devuelve la cantidad de clientes que se conectaron exitosamente.

Ejemplos:

(válido) metrics
(válido) metrics bytes
(válido) metrics accesses
(válido) metrics accepted
(inválido) metrics accesses bytes
(inválido) metrics accesses bytes accepted

-state

Muestra el estado de las variables relevantes, las cuales son el nombre del usuario del cliente conectado y si las transformaciones están habilitadas. Este comando no recibe ningún tipo de head ni parámetros.

-getSilenced

Muestra el JID de todos los usuarios que están silenciados. Este comando no recibe ningún tipo de head ni parámetros.

-multiplex

Recibe tres parámetros: JID, ip y puerto. Mapea el JID a un servidor origen que corre en la ip y puerto determinados.

Ejemplos:

(válido) multiplex user 10.1.34.208 1080

(inválido) multiplex

getMultiplex

Muestra el JID de los usuarios multiplexados, junto con sus respectivos ip y puertos. No recibe parametros.

Respuestas

Las respuestas del servidor tienen la siguiente estructura:

```
[+n] (.|?|!|X) code message\n
[lines\n]{n}
```

La respuesta del servidor se divide en líneas separadas por un '\n'. En cada ejemplo mostrado se omitirá el uso del '\n', por cuestiones de sencillez.

Si la respuesta es multilínea, el inicio de la primer línea deberá comenzar con el símbolo “+” inmediatamente seguido del número de las siguientes líneas, es decir, sin contar la primera. Si la respuesta no es multilínea, entonces la respuesta no deberá tener dicho indicador.

Ejemplo:

```
-C   hello --version 1.1
-S   . 10 "Welcome"
-C   help
-S   +12 . 0 "OK"
     hello
     help
     auth
     quit
     silence
     unsilence
     transformation
     metrics
     state
     getSilenced
     multiplex
```

```
getMultiplex
```

Luego la primer línea de la respuesta deberá tener un símbolo indicando el tipo de respuesta (si la respuesta no es multilínea, este símbolo será el inicial de la respuesta, si la respuesta es multilínea, entonces dicho símbolo estará separado del indicador de multilínea por un espacio: ' ').

Existen cuatro tipos de símbolos. Un punto ('.') indica que la solicitud es correcta y no hay problemas para responder.

Ejemplo:

```
-C    hello --version 1.1
-S    . 10 "Welcome"
```

Un signo de pregunta ('?') indica el servidor no entiende la solicitud enviada por el cliente.

Ejemplo:

```
-C    aaaaaaaa
-S    ? 0 "What?"
```

Un signo de exclamación ('!') indica que si bien el servidor entiende la solicitud enviada por el cliente, dicha solicitud tiene argumentos inválidos o la solicitud no llega en el momento adecuado (por ejemplo: el cliente quiere silenciar un usuario pero no está autorizado), es decir el servidor rechazó la solicitud del cliente.

Ejemplo:

```
-C    help help help
-S    ! 0 "Wrong arguments"
-C    metrics
-S    ! 7 "You need to authenticate"
```

Una cruz (‘X’) indica un error en el servidor.

Luego del símbolo indicador del tipo de respuesta, sigue código que indica de forma más específica el tipo de respuesta. El código se separa del símbolo por un espacio (‘ ‘). El código es un número entero positivo, incluyendo al cero.

Finalizando la primer línea, y separada por un espacio del número de código, sigue un mensaje detallando el tipo de respuesta. Dicho mensaje es delimitado por comillas dobles (‘"‘).

Luego del mensaje se finaliza la primer línea con un ‘\n’.

Luego de la primer línea (en el caso de que la respuesta sea multilínea) seguirán la demás líneas. Cada línea finaliza con un ‘\n’, y la cantidad de líneas debe ser equivalente al número establecido al principio de la primer línea.

Ejemplo de una conversación entre cliente y servidor:

```
-C    help
-S    ! 10 "Hello first"
-C    hello
-S    +1 ! 0 "Wrong arguments"
      Initiates dialog with the user. Needs the head values
--version along with the version.
-C    hello --version 2
-S    ! 9 "Version not supported"
```

```
-C    hello --version 1.1
-S    . 10 "Welcome"
-C    help
-S    +12 . 0 "OK"
      hello
      help
      auth
      quit
      silence
      unsilence
      transformation
      metrics
      state
      getSilenced
      multiplex
      getMultiplex
-C    state
-S    ! 7 "You need to authenticate"
-C    auth --request plain
-S    ! 1 "Authentication method not supported"
-C    auth --methods
-S    +1 . 4 "Authentication methods"
      simple
-C    auth --request simple
-S    . 1 "Go Ahead"
-C    auth user pass
-S    ! 4 "Incorrect user or password"
-C    auth admin admin
-S    . 2 "Logged in"
-C    state
-S    +2 . 0 "OK"
      Transformation: enabled
      User: admin
-C    transformation false
-S    . 8 "Transformation Disabled"
```

```
-C    state
-S    +2 . 0 "OK"
      Transformation: disabled
      User: admin
-C    silence user1
-S    . 5 "User silenced"
-C    help getSilenced
-S    +2 . 0 "OK"
      getSilenced:
      Shows the silenced users.
-C    getSilenced
-S    +1 . 0 "OK"
      root

-C    quit
-S    . 11 "Bye bye"
```

ABNF

La estructura de las solicitudes y respuestas están documentadas detalladamente en el siguiente ABNF.

```
; Formato de las solicitudes del cliente

request =
HELLO/HELP/AUTH/QUIT/SILENCE/UNSILENCE/TRANSFORMATION/METRICS/STATE/GETSILENCED
/MULTIPLEX/GETMULTIPLEX

number = 1*DIGIT
text = 1*(DIGIT/ALPHA)
textwithoutcrlf = *CHAR    ;excluir el CRLF
methods = "simple"
boolean = "true"/"false"
metrics = "bytes"/"accesses"/"accepted"
symbol = "."/"!"/"?"/"X"

HELLO = "hello" SP "--version" SP ((number "." number)/number) CRLF
HELP = "help" (CRLF/SP
("hello"/"help"/"auth"/"quit"/"silence"/"unsilence"/"transformation"/"metrics"/
"state"/"getsilenced"/"multiplex") CRLF)
AUTH = "auth" SP (("--methods")/("--request" SP methods)/(text SP text)) CRLF
QUIT = "quit" CRLF
SILENCE = "silence" SP text CRLF
UNSILENCE = "unsilence" SP text CRLF
TRANSFORMATION = "transformation" SP boolean CRLF
METRICS = "metrics" (CRLF/(SP metrics CRLF))
STATE = "state" CRLF
GETSILENCED = "getsilenced" CRLF
MULTIPLEX = "multiplex" SP text SP text SP number CRLF
GETMULTIPLEX = "getmultiplex" CRLF



;Formato de las respuestas del servidor
```

```
response = 0*1("+" number + SP) symbol SP number SP DQUOTE textwithoutcrlf
DQUOTE CRLF 0*line  ;la cantidad de "line" debe equivaler al "number"
especificado previamente
line = textwithoutcrlf CRLF
```

Protocolo XMPP

El protocolo Xmpp manipula la información entrante de distintas maneras dependiendo de quién es el receptor, el emisor y en qué estado de la conexión se encuentra. Contamos con cuatro clases principales:

- XmppClientNegociator: se ocupa del cliente hasta que solicita una autorización y permite saber el nombre de usuario.

- XmppServerNegociator: se ocupa de enviar al servidor las solicitudes previamente manipuladas por XmppClientNegociator.

- XmppParser: se ocupa de interpretar los mensajes del cliente y efectuar acciones correspondiente a lo solicitado en este trabajo práctico.

- XmppForwarder: se ocupa de enviar al cliente todos los mensajes del servidor

Para el parseo de mensajes utilizamos una librería de parseo de xml llamada aalto[8]. El mismo permite un parseo no bloqueante de ByteBuffers. Se ocupa de interpretar el contenido del buffer y se ocupa de informar el estado en el que se encuentra: START_DOCUMENT, START_ELEMENT, CHARACTERS, END_ELEMENT, entre otros. El estado EVENT_INCOMPLETE informa que se necesita por lo menos otro ByteBuffer para llegar a uno de los estados previamente mencionados. Dentro de cada estado la librería dispone de métodos para obtener la información que pudo interpretar.

[8] https://github.com/FasterXML/aalto-xml

Todos estos handlers XMPP implementan a la interfaz ProtocolHandler que cuenta con las siguientes funciones:

```
public abstract class ProtocolHandler {
   protected Connection connection;

   public void setConnection(Connection connection) {
       this.connection = checkNotNull(connection);
   }

   public abstract void afterConnect();

   public abstract void afterRead(final ByteBuffer buffer);

   public abstract void afterWrite();

   public abstract void beforeClose();
}
```

Esta interfaz permite interactuar a la conexión y al protocolo. Cuando la conexión lee datos sobre un ByteBuffer, envía esa información al protocolo mediante este interfaz (utilizando el método 'afterRead(final ByteBuffer buffer)'). Lo mismo sucede cuando se escribe, se termina de efectuar una conexión y cuando se finaliza la misma.

Hay dos negociadores que son los que se llaman para inicializar una conexión entre el cliente y el servidor. Después de efectuarse la conexión con el cliente, se utiliza el handler XmppClientNegociator para intereactuar con el cliente y hacer de servidor. Esto se hizo para poder obtener el nombre de usuario necesario para saber a qué servidor se tiene que conectar. Durante estas etapas es necesario verificar los mensajes para el correcto funcionamiento.

A continuación se detallan los negociadores previamente mencionados:

Negociación con el cliente (XmppClientNegotiator):

Al comienzo de la conexión se verifica que la declaración del XML (en caso de existir) se corresponda con la versión y encoding soportados (versión 1.0 y encoding UTF_8). Si hay error se le envían al usuario los siguientes errores: INVALID_XML[9] en el caso de version incorrecta y UNSOPPORTED_ENCODING[10]. En el caso donde no se envia ningun processing instruction al inicio se continúa con la negociación ya que la rfc aclara que no es obligatorio.

Luego cuando le llegue un tag stream:stream verifica que contenga el namespace y el prefijo. Si no envía INVALID_NAMESPACE[11] o BAD_NAMESPACE_PREFIX[12] respectivamente. Al haber verificado esto construye en un ByteBuffer de retorno la respuesta que se le enviará al cliente. Para hacer esto se fija que contenga el atributo 'To' y lo escribe en un atributo 'From' y de la misma manera intercambia en el caso de un 'From' lo convierte en un 'To'. En el caso de que no se haya encontrado el atributo 'To' se le devuelve un error de tipo HOST_UNKNOWN[13] al cliente. También verifica que le llegue una versión válida, en nuestro caso soportamos hasta la versión 1.0, cuando la versión no coincide se le envía al cliente un error de tipo UNSOPPORTED_VERSION[14]. Al tag de respuesta se le agrega un id, que para que sea único[15] utilizamos una variable estática de tipo long. Una vez hecho este tag se agrega un tag de tipo stream:features, donde le indicamos al cliente cuales son las capacidades con las que vamos a contar, para la encriptación del mensaje solo contamos con el mecanismo PLAIN. Esto se le envía al cliente y luego se espera su respuesta.

[9] https://tools.ietf.org/html/rfc6120#section-4.9.3.11
[10] https://tools.ietf.org/html/rfc6120#section-4.9.3.22
[11] https://tools.ietf.org/html/rfc6120#section-4.9.3.10
[12] https://tools.ietf.org/html/rfc6120#section-4.9.3.2
[13] https://tools.ietf.org/html/rfc6120#section-4.9.3.6
[14] https://tools.ietf.org/html/rfc6120#section-4.9.3.25
[15] https://tools.ietf.org/html/rfc6120#section-4.7.3

Una vez enviado esto el usuario debería responder con un tag de tipo auth con el usuario encriptado en base64 (Plain). Verificamos que este sea el caso, que no tenga un prefijo, que tenga el namespace correcto y que cuente con el mecanismo plain. Sino se envian los errores BAD_NAMESPACE_PREFIX, INVALID_NAMESPACE e INVALID_MECHANISM[16] respectivamente. Luego se interpretan los caracteres y cuando se llega a un tag de cierre de tipo tag, se guarda el usuario que luego usará para conectar al servidor correspondiente. Si no se puede interpretar el mensaje porque el nombre está con un encoding inválido se envia el error de INCORRECT_ENCODING.

Cuando esto termina se intenta conectar con el servidor para responderle exactamente lo que responde el servidor (devolviendo un error en el caso de no poder hacerlo).

Negociación con el Servidor (XmppServerNegotiator):

Al momento de conectarse con el servidor en la función afterConnect solicita escribir lo primero que se le envía el servidor, que sería la declaración del XML y el stream:stream inicial, incluyendo el atributo 'to' con el valor que le envía el XmppClientNegotiator y el namespace correspondiente.

Luego se lee, y se verifica en el inicio del contenido el encoding y la versión de la misma manera que en el negociador con el cliente. Se verifica que lo primero que llegue sea un stream:stream, que reciba las stream:features y que contenga el mecanismo PLAIN. En el caso que no cumpla con algunas de estas condiciones devolvemos un error al cliente del tipo INTERNAL_SERVER_ERROR[17]. En el caso que si se verifiquen las condiciones se le envía al servidor un tag del tipo auth con el usuario en base64, con el usuario que recibimos del XmppClientNegotiator.

[16] https://tools.ietf.org/html/rfc6120#section-6.5.7
[17] https://tools.ietf.org/html/rfc6120#section-4.9.3.8

Al enviar esto la respuesta que se recbie del servidor se le envia directo al cliente, sea un mensaje de éxito o de fallo.

Cuando se llega a este estado se procede solicitar lectura del lado del cliente y se maneja ese lado de la conexión con su handler XmppParser (que no va a estar habilitado para lectura hasta que termine la negociación con el servidor) y se crea un nuevo handler XmppForwarder que se encarga de hacer forwarding de los mensajes que recibe desde el servidor hacia el cliente.

Etapa de Parseo (XmppParser):

Una vez que ya se realiza la negociación exitosamente se deja de actuar como un servidor y un cliente y se pasa a comportar como un proxy.

El parser lo que hace es ir generando en un Byte Buffer el xml recibido y solo lo modifica en el caso que esté activado el modo de transformación, en cuyo caso se reemplazan por los caracteres correspondientes al texto contenido dentro de un tag <body> dentro de un tag <message>. Para este proceso se tuvieron que tener en cuenta los caracteres que se tenían que ser escapados, para que no sean mal interpretados como por ejemplo un tag de cierre de xml.

El parser no envia mensajes necesariamente completos, envía los caracteres que recibe y los tags que están formados correctamente. Lo que no se logre enviar por estar incompleto se guarda en el parser y en el momento que lee más contenido, lo sigue generando con lo que tenia guardado y el contenido nuevo.

En este punto solo se devuelve error en el caso que el xml este mal formado. Si este es el caso se solicita escribir un mensaje de error tanto en el cliente como en el servidor, se cierra la conexión y se solicita el cierre de esta conexión al servidor.

Problemas Encontrados Durante el diseño y la implementación:

Los 2 mayores problemas encontrados durante la etapa de diseño fue la del Multiplexador de cuentas ya que el diseño inicial no lo tuvo en cuenta y se tuvo que volver a efectuar un diseño acorde. El otro problema fue causado por el Silenciador de cuentas ya que las stanzas provenientes del cliente siempre se enviaban al servidor pero si el usuario se encuentra silenciado se debe enviar un mensaje de error al cliente y no al servidor y se tuvo que tener en cuenta el caso de estar escribiendo dicho error y que el servidor al mismo tiempo solicite al handler escribir sobre el cliente.

A la hora de parsear el contenido recibido en el parser XMPP, inicialmente se delimitaban los mensajes para luego analizarlos. De esta forma había que ir guardando todos los fragmentos de un mensaje hasta que esté completo, excluyendo el tag stream:stream en donde no esperábamos a que cierre. Esto generó muchas complicaciones respecto al tamaño del ByteBuffer donde guardabamos el mensaje, ya que no se podía saber que tan grande podría ser. Finalmente se optó por prescindir de la delimitación. De esta manera se analiza el contenido del mensaje a medida que va llegando, y se devuelven las etiquetas completas. Así dejó de ser necesario ir guardando el mensaje, ya que se devuelven los bytes entrantes pero analizados y/o modificados. Esto logra mayor eficiencia al analizar el “body” de un “message”, ya que éste podría ser de un gran tamaño, y ahora en vez de esperar a que éste llegue por completo, se envía a medida que va llegando.

Limitaciones de la Aplicación:

Debido a que nuestros ByteBuffers tienen un tamaño fijo establecido, y el hecho de que guardamos toda una etiqueta (ya sea una que abre o que cierra) para luego retornarla, surge la limitación que una etiqueta no puede superar cierto tamaño. Si bien los caracteres que se encuentran entre un elemento que cierra y otro que abre pueden tener cualquier tamaño (ya que, eso es devuelto inmediatamente de ser recibido por el parser), un elemento que abre o que cierra no podrá superar el máximo tamaño del ByteBuffer ya que no podrá ser almacenado.

Dado el protocolo xmpp y los tags que maneja es improbable que llegue un tag que se excede del tamaño del ByteBuffer reservado (también teniendo en cuenta la máxima longitud que puede tener un JID), pero dado el caso donde llegue un tag que exceda dicho tamaño, puede causar un Buffer Overflow exception.

Otra limitación, es el hecho de que las configuraciones hechas por medio de la interacción del protocolo NTTP sean volátiles. Por este motivo, dichas configuraciones se reinician cada vez que se reinicia el servidor.

Tambien se puede considerar una limitación al hecho que solo se soporta UTF_8, durante el momento de negociar con el cliente y con el servidor se ofrece solo este método de encoding.

Posibles Extensiones:

Una extensión que podría llegar a hacerse en el futuro sería la de poder cerrar el servidor proxy a través del protocolo nttp, algo que en este momento el protocolo no soporta. Si se contase con esta funcionalidad se podrían guardar en el archivo config.xml los últimos cambios que se realizaron sobre el protocolo nttp, como por ejemplo los usuarios que se silenciaron, y de esta manera no tener que siempre ingresar nuevamente la información previa.

Podrían incluirse nuevas métricas para recolectar otra información relevante acerca de las diversas conexiones.

Otra extensión podría ser la inclusión de soporte para más métodos de encoding (como ser UTF_16).

También se pueden agregar métodos de autenticación XMPP ya que solo se soporta el método PLAIN en este trabajo.

Conclusión:

Si bien el protocolo XMPP es simple de entender, surgen varios factores que le dan a este proyecto una mayor complejidad. Por ejemplo la necesidad de manejar mensajes de gran tamaño, la negociación con el cliente y el servidor o la utilización de una nueva herramienta como Java NIO.

La necesidad de actuar como cliente con el servidor y de servidor con el cliente fue una exigencia que nos hizo comprender ambos lados de la comunicación entre ambas partes. Para ésto debimos aprovechar en todo lo posible las herramientas que provee el protocolo XMPP. También el hecho de tener que representar ciertos comportamientos no definidos por XMPP (como silenciar usuarios) nos obligó a interpretar como dicho protocolo interpreta sus mensajes y aplicarlos efectivamente.

Además el diseño de un protocolo propio nos dió una perspectiva de cómo se debe analizar las solicitudes y respuestas de forma algorítmica, y de la rigidez de la forma en la que se deben crear los mensajes para que el análisis de los mismos sea posible.

Ejemplos de Prueba

Se utilizó para probar la herramienta JMeter, pudimos probar conectar 100 usuarios exitosamente.

Además utilizamos otro plugin existente para el servidor utilizado, OpenFire que crea usuarios[18]. Utilizamos este plugin para crear 100 usuarios y hacer que se envían mensajes entre ellos y tambien lo logro hacer exitosamente.

[18] User Creation, Jive Software, V 1.3.0

Guia de instalación:

Para la construcción de este proyecto se utiliza la herramienta Gradle[19].

Desde la terminal correr el comando correspondiente a la acción que se quiere efectuar:

- Para compilar:
  `gradle build`
- Para ejecutar:
  `gradle run`

En caso de no disponer de gradle se pueden utilizar el wrapper incluido en el proyecto (remplazar 'gradle' por './gradlew' en los comandos anteriores).

Para abrir el proyecto desde Intellij's Idea se debe seleccionar import project y navegar a la raíz del directorio y seleccionar 'import project from externand mode' y seleccionar 'Gradle'. O bien correr desde la terminal 'gradle run'.

[19] https://gradle.org

Instrucción para la configuración:

```
$> natto [<address>] [<port>] [--config-file (-c) <path>]
[--psp-port <port>] [--xmpp-port <port>]
       Where:
<address>                  : Sets the default XMPP server
hostname
<port>                     : Sets the default XMPP server port
       --config-file (-c) <path> : Sets the proxy's config file
(default: config.xml)
       --psp-port <port>          : Sets the proxy's PSP
listening port number
       --xmpp-port <port>         : Sets the proxy's XMPP
listening port number
```

Ejemplos de Configuración y monitoreo:

La clase XmppData tiene cargada la información a la que accede el protocolo NTTP. Se carga durante la ejecución del servidor proxy desde dicho protocolo e inicialmente se carga información predefinida en un archivo config.xml. Se hace esto en el paquete config.

Contamos con un paquete config que cuenta con tres clases de configuración. Por un lado tenemos la clase Defaults donde se establecen muchos valores por defecto utilizados a lo largo del proyecto:

```
public final class Defaults {
   public static final int SERVER_PORT = 5222;
   public static final String SERVER_ADDRESS = "localhost";
   public static final int XMPP_PORT = 1080;
   public static final int PSP_PORT = 1081;
   public static final boolean TRANSFORMATION_ENABLED = true;
   public static final boolean SILENCE_ENABLED = false;
   public static final String CONFIG_PATH = "config.xml";
}
```

Algunas de estas variables tales como TRANSFORMATION_ENABLED no se usan en el caso de cambiar la opción desde el protocolo nttp.

La recolección de las métricas se guarda en la clase XmppData, donde se recolecta la cantidad de bytes transferidos a través del proxy server, la cantidad de usuarios que intentan comenzar con la negociación, y la cantidad de usuarios que se

conectan exitosamente. La recolección de esta información es volátil, es decir que una vez que se reinicia el servidor la información se pierde y se empieza a recolectar nuevamente.

Para acceder a estas métricas se puede a través de la consola con la utilización del protocolo nttp.

En el paquete de configuración también se puede encontrar la clase Config, que se encarga de cargar la información encontrada en el archivo config.xml en la clase XmppData. El archivo config.xml tiene valores de usuarios silenciados, multiplexados, etc.

Por último se cuenta con la clase Arguments, que recibe los argumentos que se pasan por consola a la hora de ejecutar el programa, donde se puede elegir entre otras cosas, que archivo de configuración usar, que puerto utilizar para correr nttp y qué puerto utilizar para correr xmpp.

A continuación se muestra el archivo de configuración utilizado por defecto:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
    <psp>
        <port>1081</port>
        <passwords>
            <password user="admin">admin</password>
        </passwords>
    </psp>
    <xmpp>
        <port>1080</port>
        <transformation enabled="true"/>
        <silenced enabled="true">
            <user name="root"/>
```

```
        </silenced>
        <servers>
            <default>
                <address>localhost</address>
                <ip>5222</ip>
            </default>
            <user name="root">
                <address>localhost</address>
                <port>5222</port>
            </user>
        </servers>
    </xmpp>
</configuration>
```

Estructura del Proyecto: