

Realizado por: Javier Sorribes

Índice.

1. INTRODUCCIÓN	4
2. NBI – DESCRIPCIÓN Y FUNCIONALIDAD	4
2.1 ACCESO AL NBI	4
2.2 FUNCIONALIDAD: HTTP, SOAP Y WSDL.....	4
2.2.1 <i>Qué es HTTP/HTTPS</i>	4
Fig. 1. Conexión HTTP entre cliente y servidor	5
2.2.2 <i>Qué es SOAP</i>	5
Fig. 2. Ejemplo de petición SOAP al NBI.....	5
2.2.3 <i>Qué es WSDL</i>	6
3. POR QUÉ NBI – VENTAJAS Y DESVENTAJAS	6
4. PRIMEROS PASOS – PETICIONES DIRECTAS CON SOAPUI.....	6
4.1 NUEVO PROYECTO Y WSDL.....	7
Fig. 3. Crear nuevo proyecto en SoapUI	7
Fig. 4. Vista de navegador con todos los servicios WSDL, y los comandos de qosService	8
4.2 NUESTRA PRIMERA PETICIÓN	8
Fig. 5. Petición getCPEbyID al NBI desde SoapUI.....	8
Fig. 6. Respuesta del servidor si falta autorización	9
Fig. 7. Incluir credenciales de autorización en el comando	9
Fig. 8. Parte de la respuesta del NBI al comando getCPEbyID.....	9
4.3 VISTA DE HTTP EN SOAPUI.....	10
5. IMPLEMENTACIÓN AVANZADA EN PYTHON	10
5.1 EMPEZANDO CON PYTHON	10
Fig. 9. Python 3 en el terminal.....	10
5.2 CONEXIÓN Y COMUNICACIÓN CON EL SERVIDOR DEL NBI	10
Fig. 10. Petición SOAP – getCPEsByManagedGroup.xml	11
Fig. 11. Petición de información de todos los CPEs en el MG 2 – nbi.py..	11
Fig. 12. Parte del resultado del programa de la Figura 11	12
5.3 TRABAJANDO CON ELEMENTOS XML.....	12
Fig. 13. Navegación a través de respuesta XML – nbi2.py	13
Fig. 14. Resultado de ejecutar el programa de la Figura 13.....	13
5.4 INTERACCIÓN DE USUARIO.....	13
Fig. 15. Modificación del MG ID a gusto del usuario – nbi3.py.....	14
Fig. 16. Ejecución de nbi3.py con distintos valores	14
5.5 DESARROLLO DE NUEVA FUNCIONALIDAD.....	14
Fig. 17. Combinación de dos comandos – nbi4.py	15
Fig. 18. Ejecución del programa de la Figura 17	15
6. APLICACIÓN WEB EN JAVASCRIPT.....	15
6.1 AJAX Y XMLHTTPRequest.....	16
Fig. 19. Método AJAX.....	16
6.2 EMPEZANDO CON JAVASCRIPT Y AJAX.....	16
Fig. 20. Documento HTML que ejecuta una función de JavaScript – hello.html	17

	Fig. 21. Script en JavaScript al que referencia hello.html – hello.js.....	17
	Fig. 22. Fallo en hello.html	18
	Fig. 23. Error recibido al abrir hello.html.....	18
	Fig. 24. Servidor web para Chrome.....	18
	Fig. 25. Hello.html tras activar el servidor web para Chrome	19
6.3	CONEXIÓN CON EL NBI	19
	Fig. 26. Petición al NBI en JavaScript – req.js.....	19
	Fig. 27. Página web con formulario para req.js – req.html	20
	Fig. 28. Error recibido al abrir req.html	20
6.3.1	<i>Problemas con CORS</i>	20
	Fig. 29. Parte del servidor proxy diseñado para respetar CORS – serve.py 21	
6.4	APLICACIÓN WEB FINAL	21
7.	CONCLUSIÓN.....	22
8.	BIBLIOGRAFÍA	22

1. INTRODUCCIÓN

En este documento se explicará cómo trabajar con el NBI para TotalNMS de la plataforma SkyEdge II-c de Gilat Satellite Networks Ltd. Con esta API, es posible comunicarse directamente con el NMS y ejecutar comandos, sin depender de la interfaz gráfica de TotalNMS.

Este manual es complementario al documento “*SkyEdge II-c NBI for TotalNMS in v4.0, Operator’s Guide*” del 21 de junio de 2016 de Gilat. En aquel se describe en profundidad la API del NBI, mientras que en éste principalmente se muestran técnicas mediante las cuales se puede utilizar la API. La lectura del manual citado es recomendada, así como la de los otros recursos señalados en la bibliografía al final de este documento.

2. NBI – DESCRIPCIÓN Y FUNCIONALIDAD

El Northbound Interface (NBI) de TotalNMS es una interfaz de comunicación directa con el NMS. A través de peticiones HTTP siguiendo el protocolo SOAP, el cliente puede ejecutar en el servidor los comandos disponibles para TotalNMS. Entre estos comandos se incluyen la creación, modificación y eliminación de CPEs y SLAs, servicios varios de monitorización, etc.

2.1 Acceso al NBI

Una vez comprobada la licencia de NBI como se describe en la página 6 de la guía del operador, se puede acceder al servidor a través del URL <http://<nms server ip>/ws>, donde “<nms server ip>” es la misma dirección IP de acceso a TotalNMS. El nombre de usuario es “admin” y la contraseña “manager”.

Desde la página del servidor, se pueden ver los distintos servicios WSDL y los comandos que se pueden ejecutar. En las secciones siguientes se puede leer más en detalle sobre estos servicios y comandos, así como sobre el protocolo SOAP.

2.2 Funcionalidad: HTTP, SOAP y WSDL

El NBI se puede utilizar mediante el protocolo SOAP 1.2 para HTTP/HTTPS. En cada petición enviada al servidor, el nombre de usuario y la contraseña expuestos en la sección 2.1 deberán ser incluidos para autenticación básica de HTTP.

2.2.1 Qué es HTTP/HTTPS

HTTP (Hyper Text Transfer Protocol) es el protocolo de transferencia de información entre clientes y servidores en la World Wide Web. El cliente envía peticiones al servidor, el cual procesa la petición y manda una respuesta de vuelta (ver Fig. 1). Entre las posibles peticiones destacan “GET” y “POST”, que piden información a un servidor. Para más información, puede visitar esta página web: <https://www.tutorialspoint.com/http/>

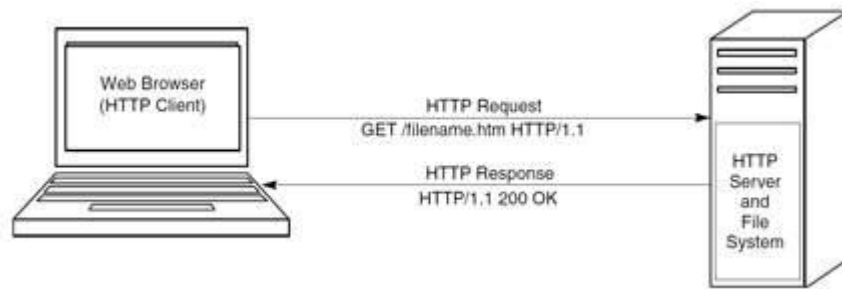


Fig. 1. Conexión HTTP entre cliente y servidor. Fuente: <http://bit.ly/2tMI7JA>

HTTPS es la versión segura de HTTP. Se basa en un sistema de encriptado, autenticación y validación por el cual el servidor debe obtener un certificado SSL o TLS para que el cliente pueda acceder a él. El NBI de TotalNMS no cuenta actualmente con dicho certificado, por lo que recomendamos que toda conexión se haga a través de HTTP.

2.2.2 Qué es SOAP

SOAP (Simple Object Access Protocol) es una forma de comunicación y transmisión de datos entre aplicaciones a través de HTTP. Los mensajes se intercambian en formato XML mediante peticiones “POST”. A continuación, se puede observar una petición SOAP que pide al NBI toda la información sobre un CPE dado su ID:

```
POST http://172.19.254.3/ws/cpeService HTTP/1.1
Connection: close
Accept-Encoding: gzip,deflate
Content-Type: text/xml; charset=UTF-8
SOAPAction: ""
Authorization: Basic YWRtaW46bWFuYWdlcg==
Content-Length: 394
Host: 172.19.254.3
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:com="com.gilat.ngnms.server.services.ws.cfg.face">
  <soapenv:Header/>
  <soapenv:Body>
    <com:getCPEbyID>
      <id>
        <managedGroupId>4</managedGroupId>
        <subscriberId>40005</subscriberId>
      </id>
    </com:getCPEbyID>
  </soapenv:Body>
</soapenv:Envelope>
```

Fig. 2. Ejemplo de petición SOAP al NBI

En este ejemplo, se pueden ver las distintas partes del mensaje enviado al servidor. La primera línea corresponde a la cabecera, donde se especifica que la petición usa el método “POST”, el servidor y la versión de HTTP. Las siguientes líneas hasta el espacio son “headers”, que expresan atributos de la petición. Finalmente, también se incluye un cuerpo o “body” del mensaje en formato XML.

El documento XML enviado en un protocolo SOAP se conforma de un elemento principal: el “Envelope”. Dentro de él puede haber a su vez otros tres elementos: un “Header” (metadatos), un “Body” (con el comando a ejecutar en el NBI) y un “Fault” (en caso de fallo o error, normalmente en las respuestas del servidor). Si se sigue este formato, una aplicación SOAP como es el servidor de NBI es capaz de procesar la petición, ejecutar el comando apropiado y responder con otro mensaje XML en formato SOAP.

2.2.3 Qué es WSDL

WSDL (Web Services Description Language) describe un servicio web en XML, con cuatro tipos de elementos fundamentales: “types”, “message”, “portType” y “binding”.

En el servidor NBI del URL citado en la sección 2.1 de este manual, se puede también acceder a los documentos WSDL para los cuatro servicios o “endpoints” que ofrece el NBI: /cpeService, /elementsInformationService, /multiCastService y /qosService. Clicando en el link correspondiente a cada uno y marcado como “WSDL”, se abre documento WSDL para ese servicio. Este documento se usará más adelante como descripción del servicio SOAP del NBI y así determinar qué mensajes enviar para cada comando.

3. POR QUÉ NBI – VENTAJAS Y DESVENTAJAS

Hay numerosas razones por las que utilizar el NBI de TotalNMS en lugar de su interfaz gráfica puede resultar beneficioso, entre las que destacan:

1. Mayor flexibilidad para proporcionar una mejor experiencia al usuario.
2. Capacidad de combinar funciones, creando nueva funcionalidad avanzada.
3. Control preciso del NMS.
4. Facilidad para automatizar la funcionalidad.
5. Integración sencilla y mecánica dentro de aplicaciones, mediante un sistema de peticiones y respuestas entre cliente y servidor.

Por todos estos motivos, es útil conocer el NBI. Sin embargo, también existen ciertas desventajas en utilizar el NBI y el protocolo SOAP, como por ejemplo:

1. Restricción del NBI al protocolo SOAP como método único.
2. Ligera incomodidad a la hora de inspeccionar un elemento XML en formato SOAP para acceder a la parte relevante.
3. Acceso completo al servidor, con los peligros que esto conlleva.

No obstante, si se tienen en cuenta todas estas características del NBI de TotalNMS, se puede explotar todo su potencial y desarrollar aplicaciones muy útiles y eficientes.

4. PRIMEROS PASOS – PETICIONES DIRECTAS CON SOAPUI

En este capítulo se muestra cómo interactuar de forma sencilla con el servidor del NBI, mandando las peticiones XML directamente desde una interfaz SOAP. Como ejemplo práctico para este tutorial, tomaremos la interfaz SoapUI; la cual se puede descargar siguiendo el siguiente enlace (la versión OpenSource es gratuita y cuenta con todo lo que necesitamos): <https://www.soapui.org/downloads/soapui.html>. Aprenderemos a generar

las peticiones SOAP a partir de los documentos WSDL, así como a utilizar éstas para interactuar con el servidor.

4.1 Nuevo proyecto y WSDL

Una vez instalado SoapUI, crearemos un nuevo proyecto SOAP mediante la opción “File→New SOAP Project”. Entonces se abrirá una ventana de configuración. Aquí incluiremos uno de los cuatro WSDLs mencionados en la sección 2.2.3 de este manual, y a los que tenemos acceso mediante el servidor NBI. Nosotros hemos elegido empezar con “/cpeService”, pero más adelante añadiremos los demás:

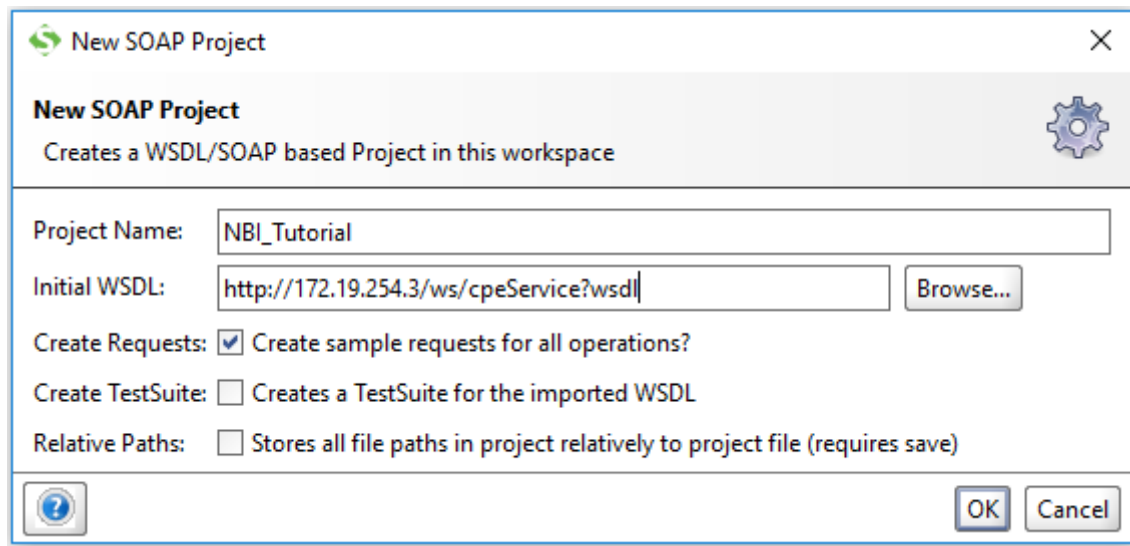


Fig. 3. Crear nuevo proyecto en SoapUI

Tome nota de dos detalles de la Figura 3:

1. Marcamos la casilla “Create Requests” para crear todas las plantillas posibles de las peticiones. Esto será muy útil no sólo en SoapUI, sino también para otras aplicaciones con las que trabajaremos después.
2. Utilizamos el URL del servidor seguido de “/cpeService?wsdl” para nuestro WSDL inicial. Esto cargará la documentación para crear las plantillas de las peticiones. Si tuviese algún problema o error por el que este URL no funcionase, otra solución sería copiar y pegar ese documento WSDL (que puede abrir en un navegador) en un archivo local, y luego utilizar la opción “Browse...” para seleccionarlo.

Ahora, en el navegador de la izquierda podrá ver el nuevo proyecto, con una subcarpeta llamada “CPEServiceSoapBinding”. Dentro de esta carpeta se encuentran todos los comandos del servicio correspondiente. Para añadir los demás servicios y comandos, seleccionaremos la opción de menú “Project→Add WSDL” y proporcionaremos el URL (o dirección local si hemos descargado el archivo WSDL) para cada servicio. De nuevo, asegúrese de marcar la casilla para crear plantillas de peticiones. El proyecto debería quedar como se muestra en la siguiente imagen.

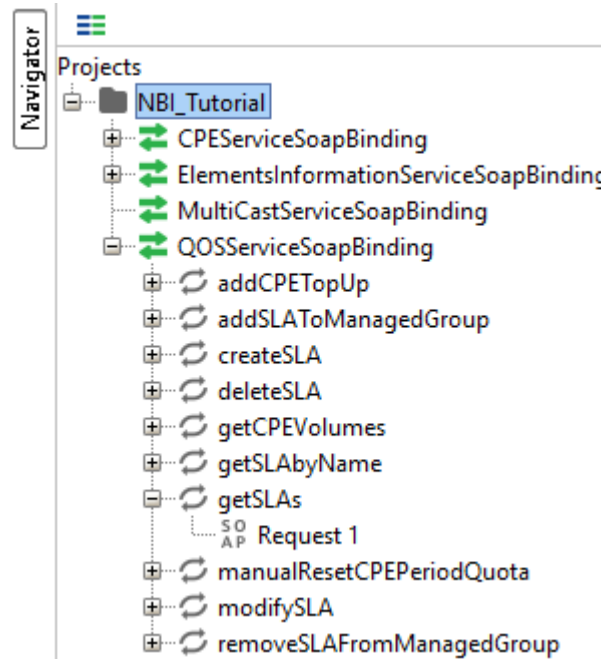


Fig. 4. Vista de navegador con todos los servicios WSDL, y los comandos de qosService

4.2 Nuestra primera petición

Para mandar una petición al servidor del NBI, abriremos el menú del comando correspondiente presionando en el “+” y haremos click en “Request 1”. Esto abrirá una nueva ventana de editor con la plantilla XML de ese comando. En nuestro ejemplo, hemos seleccionado el comando “getCPEbyID” como se puede ver en la Figura 5:

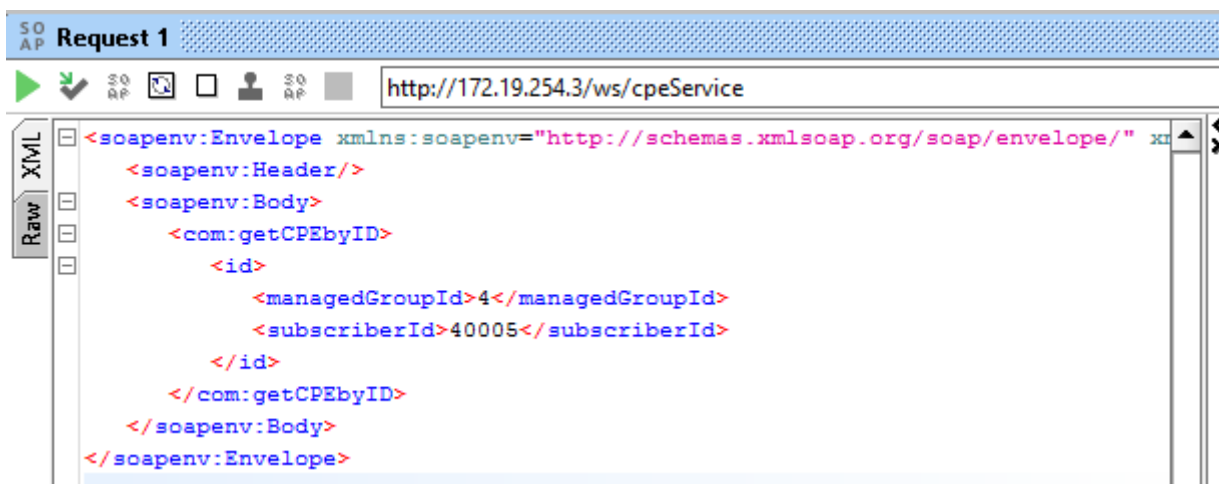
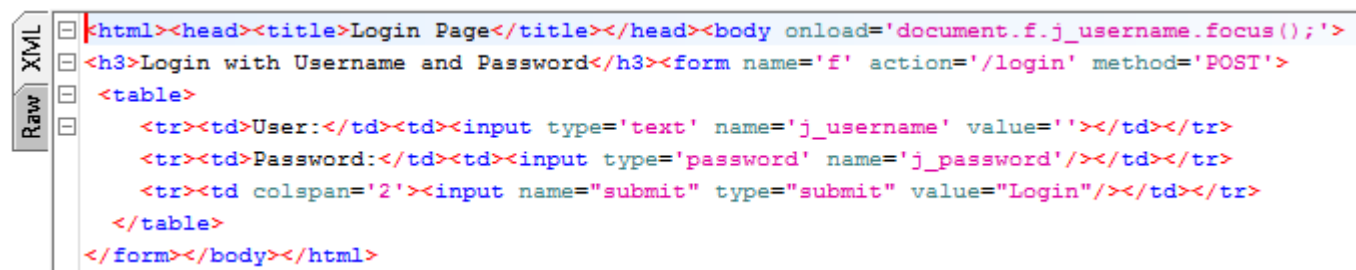


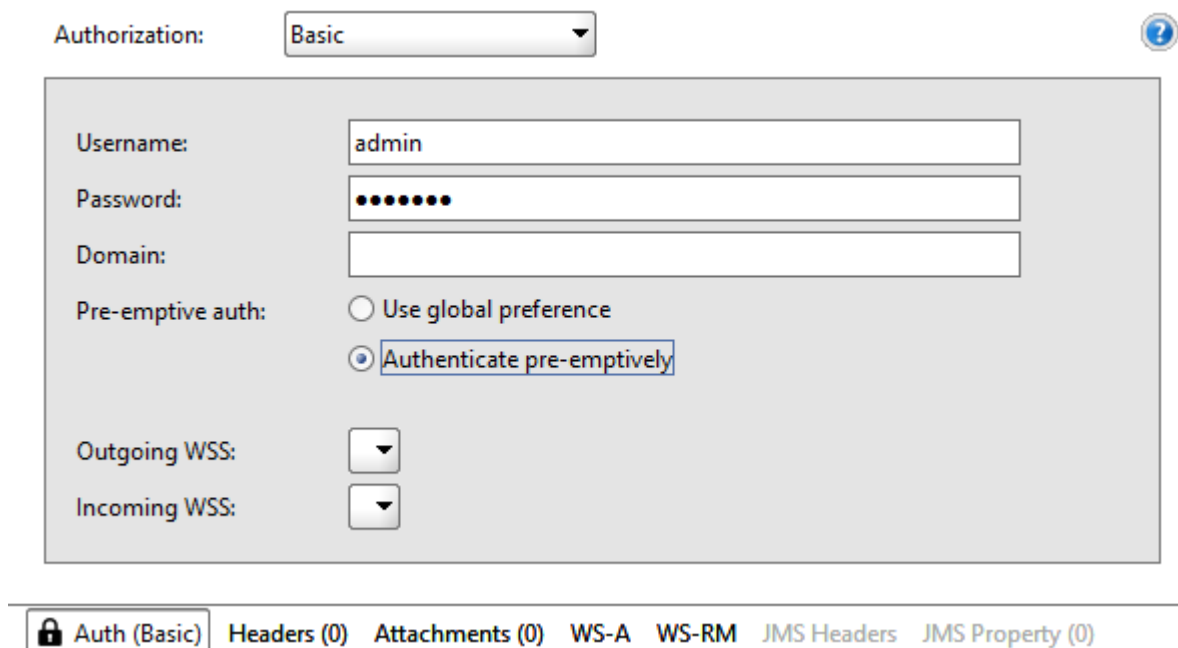
Fig. 5. Petición getCPEbyID al NBI desde SoapUI

Sin embargo, al tratar de ejecutar esta función presionando el botón verde de play, el mensaje que recibimos nos indica que debemos adjuntar un nombre de usuario y una contraseña (Figura 6). Para incluir esta información de autenticación, debemos añadirla en cada comando que deseemos ejecutar abriendo la ventana “Auth” que se encuentra debajo del editor, seleccionarla autorización básica y proporcionar las credenciales mencionadas en la sección 2.1 como se muestra en la Figura 7.



```
<html><head><title>Login Page</title></head><body onload='document.f.j_username.focus();'>
<h3>Login with Username and Password</h3><form name='f' action='/login' method='POST'>
<table>
<tr><td>User:</td><td><input type='text' name='j_username' value=''></td></tr>
<tr><td>Password:</td><td><input type='password' name='j_password'></td></tr>
<tr><td colspan='2'><input name='submit' type='submit' value='Login'></td></tr>
</table>
</form></body></html>
```

Fig. 6. Respuesta del servidor si falta autorización



Authorization: Basic

Username: admin

Password:

Domain:

Pre-emptive auth: ☐ Use global preference ☒ Authenticate pre-emptively

Outgoing WSS:

Incoming WSS:

Auth (Basic) Headers (0) Attachments (0) WS-A WS-RM JMS Headers JMS Property (0)

Fig. 7. Incluir credenciales de autorización en el comando

Con las credenciales incluidas, el comando se puede ejecutar y se recibirá la respuesta pertinente del servidor:



```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<ns2:getCPEbyIDResponse xmlns:ns2="com.gilat.ngnms.server.services.ws.cfg.fa">
<return>
<cpeId>
<managedGroupId>4</managedGroupId>
<subscriberId>40005</subscriberId>
</cpeId>
<getMacFromCPE>DISABLE</getMacFromCPE>
<macAddress>00A0AC1C88C2</macAddress>
<clusterCode>091A</clusterCode>
<platform>Gemini</platform>
<ipStack>IPV4_ONLY</ipStack>
</return>
</ns2:getCPEbyIDResponse>
</soap:Body>
</soap:Envelope>
```

Fig. 8. Parte de la respuesta del NBI al comando getCPEbyID

4.3 Vista de HTTP en SoapUI

SoapUI puede ser de gran ayuda para ver qué mensajes XML se intercambian el cliente y el servidor, y para generar automáticamente las plantillas para cada petición. Además, también podemos visualizar los mensajes completos, es decir, los mensajes HTTP. Cambiando a vista “raw” en lugar de “XML”, veremos el comando POST junto con los “headers” y “body” de los que hablamos en la sección 2.2.2. La Figura 2 de esa sección muestra esta funcionalidad de SoapUI que nos permite entender mejor qué está ocurriendo durante la comunicación con el servidor y que puede ser útil para aplicaciones sofisticadas que desarrollaremos más adelante.

5. IMPLEMENTACIÓN AVANZADA EN PYTHON

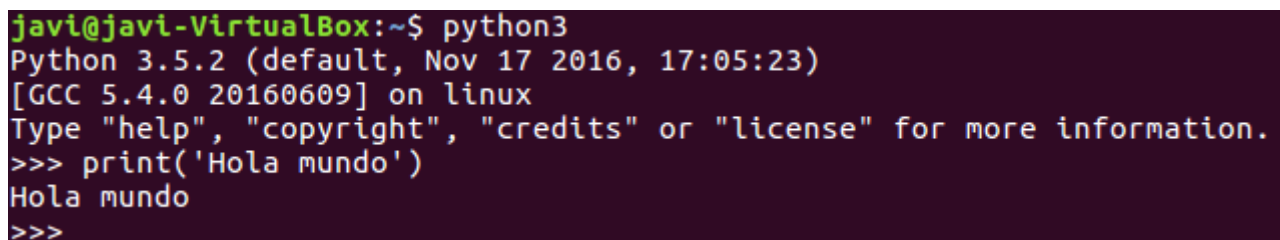
En este capítulo, extenderemos nuestros conocimientos del NBI para permitir al usuario ver información sobre todos los CPEs de un Managed Group. Para ello, desarrollaremos una aplicación en Python 3 que se comunicará con el servidor. Esto se podría hacer en otros lenguajes de programación como Java, pero hemos elegido Python por la facilidad de su aprendizaje y su valor como medio de scripting.

Todos los archivos mencionados en este capítulo se pueden ver en el repositorio de GitHub https://github.com/javisorribes/TotalNMS_NBI. La mayoría de los programas se encuentran en el directorio “TutorialPY” y los documentos XML en “XMLReqs”.

5.1 Empezando con Python

Es bastante sencillo empezar a programar en Python. Sólo hay que mantener en mente que hay dos versiones principales, la 2 y la 3. Nosotros usaremos Python 3 porque es más moderno y un número creciente de programadores se están cambiando a esta versión. Para descargar e instalar Python 3, siga el siguiente enlace y seleccione la versión más reciente: <https://www.python.org/downloads/>

Una vez instalado, debería ser capaz de escribir un archivo de extensión .py y ejecutarlo con el comando “python3 <script>.py”. También se puede ejecutar el comando “python3” para abrir un interpretador de Python en el mismo terminal, como se ve en la Figura 9:



```
javi@javi-VirtualBox:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hola mundo')
Hola mundo
>>>
```

Fig. 9. Python 3 en el terminal

Si está interesado en aprender Python 3, existen muchos buenos tutoriales en Internet. Sin embargo, no es necesario tener un alto nivel de Python para nuestro proyecto. Son suficientes unos conocimientos básicos de programación.

5.2 Conexión y comunicación con el servidor del NBI

Para conectar con el servidor a través de HTTP, podríamos usar numerosos módulos de Python con la funcionalidad requerida, como “urllib” o “http.client”. Sin embargo, nosotros

nos decantamos por “requests” (<http://docs.python-requests.org/en/master/>) por su sencillez y eficiencia para nuestros objetivos. Para instalar este módulo, debería bastar con ejecutar el comando “pip3 install requests” en el terminal (puede que sea sólo “pip” sin el “3” dependiendo de su equipo). Con este módulo, podemos establecer conexión con el servidor y enviar y recibir mensajes.

La Figura 11 muestra un simple programa que envía el mensaje escrito en el documento XML de la Figura 10 al servidor del NBI y responde con otro mensaje XML.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:com="com.gilat.ngnms.server.services.ws.cfg.face">
  <soapenv:Header/>
  <soapenv:Body>
    <com:getCPEsByManagedGroup>
      <managedGroupId>2</managedGroupId>
      <lastIndex>0</lastIndex>
    </com:getCPEsByManagedGroup>
  </soapenv:Body>
</soapenv:Envelope>
```

Fig. 10. Petición SOAP – *getCPEsByManagedGroup.xml*

```
1 import requests
2
3 host = 'http://172.19.254.3/ws'
4
5 with requests.Session() as s:
6     s.auth=('admin','manager')
7     #s.verify = False # only needed if https with no SSL certificate
8     s.headers.update({'Content-Type': 'text/xml; charset=UTF-8',
9                       'SOAPAction': '', 'Connection': 'keep-alive'})
10
11     with open('../XMLReqs/getCPEsByManagedGroup.xml','r') as f:
12         body = f.read()
13
14     r = s.post(host+'/cpeService', data=body)
15     print(r) # response object
16     print('- '*60)
17     print(r.text) # response body
```

Fig. 11. Petición de información de todos los CPEs en el MG 2 – *nbi.py*

En la primera línea del programa de la Figura 11, importamos el módulo “requests”. En la línea 3 marcamos el servidor del NBI como host, aunque especificaremos el servicio concreto a utilizar (en este caso “/cpeService”) más adelante al enviar la petición para mayor flexibilidad. En la línea 5 abrimos una sesión (que se cierra automáticamente al finalizar), y a continuación establecemos las credenciales de autorización y los “headers” apropiados. La línea 7 sólo es necesaria si intentamos acceder al servidor NBI a través de una conexión HTTPS, puesto que actualmente no cuenta con un certificado SSL (ver sección 2.2.1). En las líneas 11 y 12 guardamos los contenidos del archivo XML de la Figura 10. Entonces, mandamos un comando “POST” de HTTP al servidor con el mensaje SOAP en la línea 14, y la respuesta se guarda en la variable “r”. Finalmente, imprimimos la respuesta en sí (que muestra el código de estado) y el “body” de la respuesta (que contiene el mensaje SOAP en XML):

```
javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ python3 nbi.py
<Response [200]>
-----
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body
><ns2:getCPEsByManagedGroupResponse xmlns:ns2="com.gilat.ngnms.server.services.w
s.cfg.face" xmlns:ns3="urn:com.gilat.ngnms.server.ws.dto.cfg" xmlns:ns4="urn:com
.gilat.ngnms.server.ws.faults.dto"><return><totalNumber>3</totalNumber><lastInde
x>3696714</lastIndex><hasMore>false</hasMore><cpes><ns3:CPE><cpeId><managedGroup
Id>2</managedGroupId><subscriberId>2</subscriberId></cpeId><getMacFromCPE>ENABLE
</getMacFromCPE><macAddress>00A0AC1C88E0</macAddress><clusterCode>091A</clusterC
ode><platform>Gemini</platform><ipStack>IPV4_ONLY</ipStack><slaName>Referencia <
/slaName><rtncClassifierName>RTN Referencia_cClass</rtncClassifierName><description
>GILAT-CG-SH7-03</description><authorization>FULL_ACCESS</authorization><dataAut
horization>AUTHORIZED</dataAuthorization><locationCode></locationCode><ipAddress
ing>ROUTER</ipAddressing><provStatus>FULL_ACCESS</provStatus><operationalState>0
nline</operationalState><autoGenerateResetQuota>ENABLE</autoGenerateResetQuota><
resetQuotaDay>3</resetQuotaDay><resetQuotaTime>2</resetQuotaTime><creationDate>1
406620802</creationDate><lastModification>1494935126</lastModification><serviceA
ctivationRFAudit><status>MEASUREMENT_PASSED</status><coPol>735</coPol><crossPol>
0</crossPol><delta>735</delta><obEsNo>163</obEsNo><ibThreshold>135</ibThreshold>
<obThreshold>140</obThreshold><startTime>1431604813</startTime><endTime>14316048
14</endTime><attenuatorP1dB>0</attenuatorP1dB></serviceActivationRFAudit><otherR
```

Fig. 12. Parte del resultado del programa de la Figura 11

Sin embargo, como se puede ver en la Figura 12, la respuesta SOAP en XML es larga y complicada de leer. Por ello, conviene conocer bien el formato tanto de la petición como de la respuesta para el comando que deseemos ejecutar, mediante inspección de la guía del NBI de Gilat (Documento 1 de la Bibliografía) y con ayuda de SoapUI. Así, podremos navegar por el elemento XML para trabajar con las partes de interés de forma eficiente, como explicaremos en la próxima sección.

5.3 Trabajando con elementos XML

Un documento XML se puede leer como un objeto de estructura de árbol. En SOAP, la raíz es el elemento principal y se llama “soapenv:Envelope” en las peticiones y “soap:Envelope” en las respuestas, como se puede ver claramente en SoapUI. A partir de la raíz, podemos ir navegando por los elementos y acceder a la información que nos interese.

En Python, un módulo que podemos utilizar para esto es “xml.etree”; y en particular la clase “ElementTree” (<https://docs.python.org/3/library/xml.etree.elementtree.html>). Ésta nos permite navegar por un elemento XML como si éste fuera una lista de los elementos que engloba. Existen dos formas de acceder a los elementos dentro de otro elemento: con índices de posición y con los métodos find(match, namespaces=None) y findall(match, namespaces=None). La Figura 13 muestra una extensión al programa anterior, en el cual accedemos a datos de interés dentro de la respuesta del servidor, usando “ElementTree”.

```

1 import requests
2 import xml.etree.ElementTree as ET
3
4 host = 'http://172.19.254.3/ws'
5 ns = {'ns3': 'urn:com.gilat.ngnms.server.ws.dto.cfg'} #namespaces for find()
6
7 with requests.Session() as s:
8     s.auth=('admin','manager')
9     s.headers.update({'Content-Type': 'text/xml;charset=UTF-8',
10                      'SOAPAction': '', 'Connection': 'keep-alive'})
11
12     with open('../XMLReqs/getCPEsByManagedGroup.xml','r') as f:
13         body = f.read()
14
15     r = s.post(host+'/cpeService', data=body)
16     print(r) #response object
17     print('-'*60)
18
19     resp = ET.fromstring(r.text) #creates XML element
20     print(resp) #ElementTree object
21     print('-'*60)
22     numcpes = resp[0][0].find('return/totalNumber')
23     print(numcpes.tag, numcpes.text, sep=': ')
24     print('-'*60)
25     cpelist = resp[0][0][0].find('cpes')
26     for i in range(len(cpelist)):
27         print("CPE #{0}'s physical port's supported vlanId: ".format(i+1),
28               cpelist[i].find('physPorts/ns3:PhysicalPort/supportedVRs/vlanID',ns).text)

```

Fig. 13. Navegación a través de respuesta XML – nbi2.py

A partir de la línea 19 de la Figura 13 se puede observar cómo hemos utilizado “ElementTree” (importado como “ET”) para navegar por el documento XML tanto mediante índices como el método find(). Fíjese que si queremos usar namespaces, debemos declararlos (línea 5) e incluirlos en la llamada a find() (línea 28). Recomendamos que ejecute este comando en SoapUI y analice la respuesta recibida, puesto que así podrá visualizar el elemento XML y sus partes. La próxima imagen muestra el resultado de la ejecución de este programa:

```

javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi2.py
<Response [200]>
-----
<Element '{http://schemas.xmlsoap.org/soap/envelope/}Envelope' at 0x7fbe8a89bdb8>
-----
totalNumber: 3
-----
CPE #1's physical port's supported vlanId: 200
CPE #2's physical port's supported vlanId: 24
CPE #3's physical port's supported vlanId: 200

```

Fig. 14. Resultado de ejecutar el programa de la Figura 13

5.4 Interacción de usuario

No sólo podemos acceder a la información de un elemento XML, sino que también podemos modificarla. De esta forma, podemos preguntar al usuario qué le gustaría hacer, como por ejemplo qué Managed Group quiere ver, y mandar la petición correspondiente de forma dinámica.

Hemos modificado el programa nbi2.py para que tome un número del usuario y lo use como Managed Group ID para ver cuántos CPEs hay en ese Managed Group. En la

Figura 15 se muestra la parte nueva que hemos introducido entre la lectura del documento XML y el envío de la petición “POST”.

```

14     mgid = input('Enter Managed Group ID: ')
15     assert(mgid.isdigit())
16     xmlbody = ET.fromstring(body)
17     xmlbody[1][0][0].text = mgid #managedGroupId tag gets new value
18     body = ET.tostring(xmlbody)

```

Fig. 15. Modificación del MG ID a gusto del usuario – nbi3.py

```

javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi3.py
Enter Managed Group ID: 2
totalNumber: 3
javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi3.py
Enter Managed Group ID: 3
totalNumber: 513
javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi3.py
Enter Managed Group ID: 4
totalNumber: 1
javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi3.py
Enter Managed Group ID: 6
totalNumber: 51

```

Fig. 16. Ejecución de nbi3.py con distintos valores

También se debe tener en cuenta que el usuario puede introducir datos equivocados o corruptos. Por ello, se deben realizar validaciones como la aserción de la línea 15 de la Figura 15 o el manejo de errores y excepciones mediante bloques “try/except”.

5.5 Desarrollo de nueva funcionalidad

Como hemos visto, podemos utilizar documentos XML como objetos y modificarlos. De esta forma, seremos capaces de combinar comandos del NBI, extraer la información de nuestro interés y crear nuevas funciones.

Por ejemplo, aunque hay un comando para ver los volúmenes de consumo de un CPE (“getCPEVolumes”), no lo hay para ver todos los volúmenes de los CPEs de un mismo Managed Group. Combinando “getCPEsByManagedGroup” con “getCPEVolumes”, podemos desarrollar esta nueva funcionalidad, como se puede apreciar en las siguientes Figuras 17 y 18. El proceso es el mismo, y lo único a tener en cuenta es el servicio WSDL al que pertenece cada comando.

Este es el poder del NBI. Combinando comandos apropiadamente, podemos obtener la información que deseemos en el formato que deseemos, y así proporcionar una experiencia más personalizada a cada cliente. Además, gracias al alto nivel de control sobre la funcionalidad, podemos lograr esto de forma dinámica y eficiente.

```

1 import requests
2 import xml.etree.ElementTree as ET
3
4 host = 'http://172.19.254.3/ws'
5
6 with requests.Session() as s:
7     s.auth=('admin','manager')
8     s.headers.update({'Content-Type': 'text/xml;charset=UTF-8',
9                       'SOAPAction': '', 'Connection': 'keep-alive'})
10
11     with open('../XMLReqs/getCPEsByManagedGroup.xml','r') as f:
12         body = f.read()
13     with open('../XMLReqs/getCPEVolumes.xml','r') as f2:
14         body2 = f2.read()
15
16     mgid = input('Enter Managed Group ID: ')
17     assert(mgid.isdigit())
18     xmlbody = ET.fromstring(body)
19     xmlbody[1][0][0].text = mgid #managedGroupId tag gets new value
20
21     r = s.post(host+'/cpeService', data=ET.tostring(xmlbody))
22     resp = ET.fromstring(r.text)
23     cpes = resp[0][0].find('return/cpes')
24
25     print("CPEs' Consumed Volume for MG", mgid)
26     xmlbody2 = ET.fromstring(body2)
27     xmlbody2[1][0][0][0].text = mgid #MG Id same for all CPEs
28     for cpe in cpes:
29         subsid = cpe[0][1].text
30         xmlbody2[1][0][0][1].text = subsid #subscriber Id set
31         r2 = s.post(host+'/qosService', data=ET.tostring(xmlbody2))
32         resp2 = ET.fromstring(r2.text)
33         print('CPE #{}:'.format(subsid),
34               resp2[0][0].find('return/consumed/cmbQuota').text)

```

Fig. 17. Combinación de dos comandos – nbi4.py

```

javi@javi-VirtualBox:~/Desktop/TotalNMS_NBI/TutorialPY$ py3 nbi4.py
Enter Managed Group ID: 2
CPEs' Consumed Volume for MG 2
CPE #2: 5035462
CPE #21: 0
CPE #40: 15662003

```

Fig. 18. Ejecución del programa de la Figura 17

6. APLICACIÓN WEB EN JAVASCRIPT

Lo que hemos visto hasta ahora puede ser muy útil especialmente a la hora de desarrollar nuevas funciones, pero la mayoría de usuarios accederán al servicio mediante una aplicación web en lugar del terminal. Aunque es posible implementar aplicaciones web en Python (con módulos como Flask, por ejemplo), la inmensa mayoría de páginas y servicios web de hoy en día están basadas en tecnologías JavaScript. Por lo tanto, es interesante estudiar cómo aplicar en JavaScript lo aprendido hasta ahora para así poder desarrollar páginas web a través de las que los clientes accedan a los servicios del NBI.

Debemos tener en cuenta que JavaScript cuenta con multitud de tecnologías y metodologías para hacer lo que queremos (AJAX, JQuery, NodeJS...). Por lo tanto, tras consideraciones de efectividad y sencillez, hemos elegido AJAX y en particular XMLHttpRequest para nuestros objetivos. Cabe destacar, no obstante, que hay otras buenas opciones como EasySoap de NodeJS, por ejemplo.

En este tutorial usaremos HTML, JavaScript y algo de Python. Es recomendable tener unos conocimientos básicos de todos ellos, para lo que recomendamos los tutoriales de W3Schools y TutorialsPoint. Aun así, la lectura de este manual debería ser asequible sin haber realizado ningún tutorial especializado de estos lenguajes de programación.

6.1 AJAX y XMLHttpRequest

AJAX (Asynchronous JavaScript And XML), es una técnica de JavaScript que permite comunicarse con un servidor cuando la página ya ha cargado y sin redirigir a otra página. El proceso consiste en crear un objeto XMLHttpRequest, el cual puede mandar y recibir mensajes HTTP asincrónamente, y preparar la página para que, dado un evento, utilice dicho objeto para obtener información de un servidor y alterar algo en la misma página. Los mensajes enviados pueden estar en formato XML, JSON, de texto, etc. La siguiente imagen muestra este proceso:

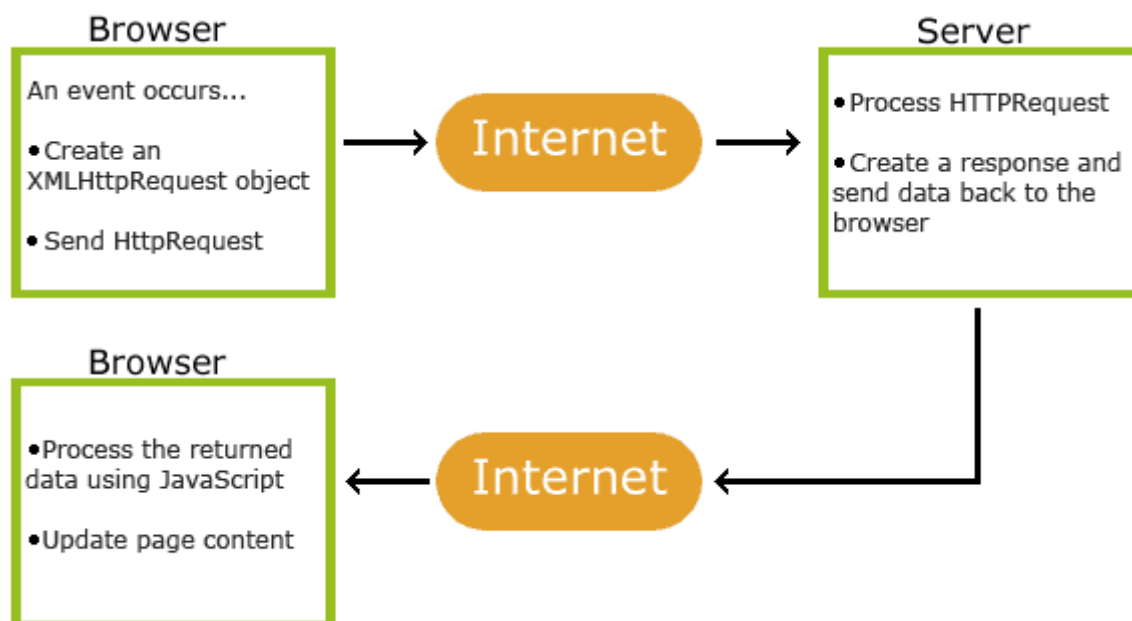


Fig. 19. Método AJAX. Fuente: <https://www.w3schools.com/xml/ajax.gif>

Gracias esta tecnología, podremos preparar mensajes XML a utilizar en nuestra aplicación SOAP, y trabajar con las respuestas del servidor. A continuación veremos cómo hacerlo, pero para mayor detalle recomendamos revisar los Documentos 10, 11 y 12 de la Bibliografía. Conviene analizar la API de XMLHttpRequest (Documento 12 de la Bibliografía) y familiarizarse con sus propiedades y métodos.

6.2 Empezando con JavaScript y AJAX

JavaScript es un lenguaje de programación que, en términos globales, sirve para dar funcionalidad a una página web escrita en HTML. Ambos lenguajes están incluidos en

cualquier navegador web moderno y a priori no requieren ningún tipo de instalación ni configuración. Como se puede apreciar en la siguiente imagen, un documento HTML sirve para estructurar una página, y puede importar scripts de JavaScript (línea 10) de referencia. También se puede ejecutar una función de JavaScript (en este caso “readFile()”) al ocurrir cierto evento, como que el usuario haga click en un botón (línea 9).

```

1 <!DOCTYPE html>
2 <html lang="en-US">
3 <head>
4   <title>Read Hello File</title>
5   <meta charset="utf-8">
6 </head>
7 <body>
8   <p id="p">Nothing</p>
9   <button onclick="readFile('./hello.txt',write2p)">Say Hello!</button>
10  <script src="hello.js"></script>
11 </body>
12 </html>

```

Fig. 20. Documento HTML que ejecuta una función de JavaScript – hello.html

```

1 /* Reads file and calls writer with the plain text received */
2 function readFile(filename, writer=dummy) {
3   var file = new XMLHttpRequest();
4   file.onreadystatechange = function() {
5     if( file.readyState == 4 && file.status == 200 ){
6       writer(file.responseText);
7     }
8   };
9   file.open('GET', filename);
10  file.overrideMimeType('text/plain');
11  file.send();
12 }
13
14 /* Doesn't do much, it's there just in case */
15 function dummy(...args) {
16   console.log('Dummy function called with the following arguments: ');
17   for( var i=0; i<args.length; i++) {
18     console.log((i+1) + ': ' + args[i]);
19   }
20 }
21
22 function write2p(text) {
23   document.getElementById('p').innerHTML = text;
24 }

```

Fig. 21. Script en JavaScript al que referencia hello.html – hello.js

Como se ve en las Figuras 20 y 21, “hello.html” usa la función “readFile()” de “hello.js” (ambos archivos se encuentran en el directorio “TutorialJS” del repositorio GitHub) para leer un archivo local (“hello.txt”) y a continuación escribir ese texto en el elemento de id “p” en el documento HTML.

En mayor detalle, la función “readFile()” crea un nuevo objeto XMLHttpRequest (línea 3) y especifica lo que va a hacer una vez que la respuesta sea recibida (línea 5) en el bloque entre las líneas 4 y 8. Finalmente, envía una petición “GET” al servidor o, en este caso, el archivo local, para leer el texto. Es importante mencionar que esta petición es asíncrona,

así que el script no esperará a recibir la respuesta, sino que seguirá con otras operaciones y ejecutará el comando de la línea 6 una vez la reciba.

Para ejecutar todo esto, basta con abrir el documento HTML en un navegador de Internet cualquiera. Sin embargo, si hacemos esto es muy probable que no funcione (Figura 22). Si abrimos la consola en la vista de desarrollador (Ctrl+Shift+I en la mayoría de navegadores), veremos un error como el de la Figura 23.

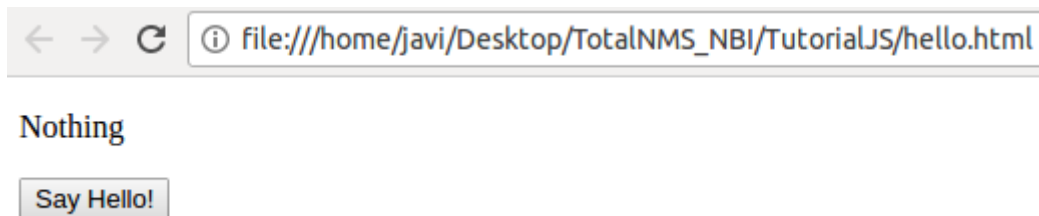


Fig. 22. Fallo en hello.html



Fig. 23. Error recibido al abrir hello.html

Este error se debe a que el navegador y nuestro equipo local provienen de orígenes distintos. Debido al protocolo de seguridad CORS (vea sección 6.3.1), no podemos realizar este tipo de operaciones entre servidores de distinto origen. Puede haber varias soluciones para esto, como deshabilitar los proxies, pero la que nosotros encontramos más conveniente es "Web Server for Chrome" (Figura 24 de la derecha). Éste permite que nuestro equipo local actúe como un servidor web. En este caso, al abrir el URL <http://127.0.0.1:8887> en Chrome, veremos los contenidos de la carpeta local "/TotalNMS_NBI". Desde ahí, podremos abrir "hello.html", el cual funcionará de la forma deseada (Figura 25) al estar accediendo al archivo de texto dentro del mismo servidor, que ahora es en realidad nuestro equipo local. Esto sólo es necesario Para descargar esta app de Google Chrome, puede hacerlo siguiendo el siguiente link: <http://bit.ly/1OCQaj9>

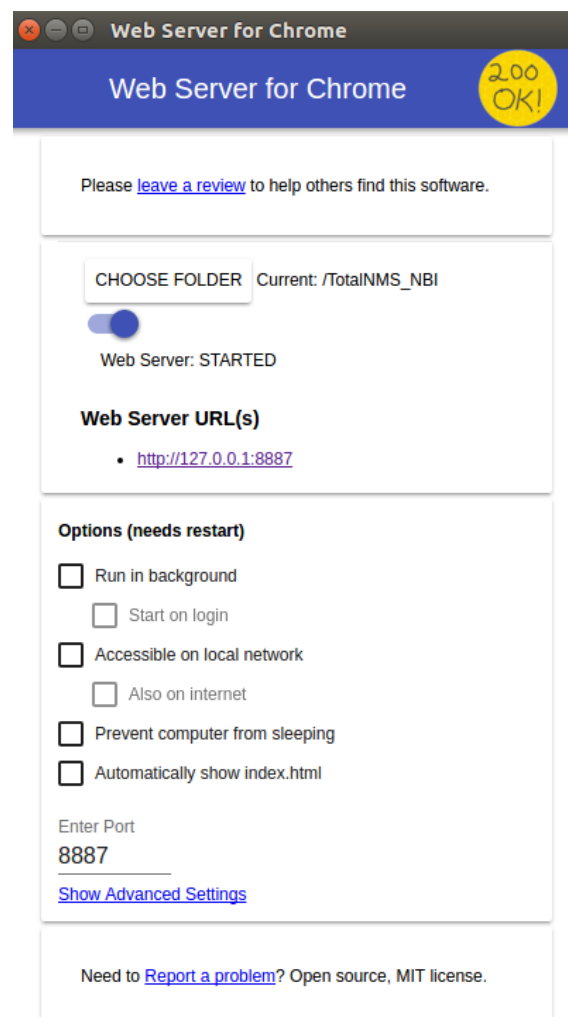


Fig. 24. Servidor web para Chrome



Fig. 25. *Hello.html* tras activar el servidor web para Chrome

6.3 Conexión con el NBI

Ahora, podemos utilizar lo visto anteriormente para leer un documento XML del directorio “XMLReqs” y mandar una petición al servidor NBI. La siguiente imagen muestra “req.js” del directorio “TutorialJS”, que acepta un MG Id y pide al servidor del NBI el número de CPEs en ese grupo.

```

1 var host = 'http://172.19.254.3/ws';
2 var username = 'admin';
3 var password = 'manager';
4
5 function getCPEsByManagedGroup(mgid) {
6   document.getElementById('numCPEs').innerHTML = 'Loading...'
7   var xhttp = new XMLHttpRequest();
8   xhttp.onreadystatechange = function() {
9     if (this.readyState == 4 && this.status == 200) {
10      xmlDoc = xhttp.responseXML; //response body as XML element
11      ret = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
12      document.getElementById('numCPEs').innerHTML
13        = ret.getElementsByTagName('totalNumber')[0].childNodes[0].nodeValue;
14    }
15  };
16  xhttp.open('POST', host+'/cpeService', true, username, password);
17  xhttp.setRequestHeader('Content-Type', 'text/xml; charset=UTF-8');
18  xhttp.setRequestHeader('SOAPAction', '');
19  //xhttp.setRequestHeader('Connection', 'keep-alive'); //not required
20  var req;
21  readXMLFile('../XMLReqs/getCPEsByManagedGroup.xml', function(xml) {
22    req = xml;
23  }, false); // async=false so it waits and puts the xml into req
24  getFirstChild(getFirstChild(getChild(req.documentElement, 1))).childNodes[0].nodeValue
25    = mgid; //input MG Id
26  reqtxt = new XMLSerializer().serializeToString(req.documentElement);
27  xhttp.send(reqtxt);
28 }

```

Fig. 26. *Petición al NBI en JavaScript – req.js*

En la Figura 26 podemos apreciar cómo se manda una prepara “POST” al NBI en la línea 16. En las líneas 21, 22 y 23 leemos un documento en formato XML con el “body” de la petición mediante una función similar a la descrita en la sección anterior y que se encuentra en “helpers.js” (también en el repositorio GitHub). Fíjese que “readXMLFile()” toma un tercer parámetro que sirve para determinar si se debe esperar a recibir la respuesta del servidor o no. En este caso, sí que se debe esperar (línea 23), para asegurarnos de que el “body” enviado al NBI contiene el XML del documento. A continuación, en las líneas 24 y 25 el MG Id se cambia a el que ha sido dado a nuestra función. Aquí, usamos otras funciones que también están definidas en “helpers.js” y que sirven para asegurarse de que los espacios en blanco en el documento sean ignorados como nodos del elemento XML. En la línea 26 se convierte el XML a formato “string” y en la línea 27 se manda la petición. Finalmente, una vez recibida la respuesta del NBI, se

ejecutan los comandos del bloque entre las líneas 9 y 14, escribiendo el número de CPEs en ese MG en el documento HTML descrito por “req.html” (Figura 27).

```

1 <!DOCTYPE html>
2 <html lang="en-US">
3 <head>
4   <title>NBI Request</title>
5   <meta charset="utf-8">
6 </head>
7 <body>
8   <form name="cpesbymgid" action="javascript:getCPesByManagedGroup(document.getElementById('managedGroupId').value)">
9     <label for="managedGroupId">Managed Group ID:</label><br/>
10    <input id="managedGroupId" type="number" value="2" required autofocus autocomplete="off"
11      onkeydown="document.getElementById('numCPes').innerHTML=' '>
12    <input type="submit" value="Search">
13  </form><br/>
14
15  <p>Total number of CPEs found:</p>
16  <div id="numCPes" style="border-style:groove; display: inline-block; min-width: 4.5em; text-align:center"></div>
17
18  <script src="../helpers.js"></script>
19  <script src="req.js"></script>
20 </body>
21 </html>

```

Fig. 27. Página web con formulario para req.js – req.html

Como se puede apreciar en la anterior imagen, “req.html” contiene un formulario que, al ser completado, realiza la acción de ejecutar la función definida en “req.js”.

Sin embargo, al abrir esta página web (a través del servidor web para Chrome de la sección anterior), surge otro error (Figura 28). Éste se debe a que el servidor del NBI no respeta todos los requerimientos del protocolo de seguridad CORS, el cual procederemos a analizar a continuación.

✖ XMLHttpRequest cannot load <http://172.19.254.3/ws/cpeService>. req.html:1
Response for preflight is invalid (redirect)

Fig. 28. Error recibido al abrir req.html

6.3.1 Problemas con CORS

CORS (Cross-Origin Resource Sharing), es un mecanismo por el cual se controla la seguridad de la comunicación entre cliente y servidor de orígenes distintos. Antes de enviar ciertas peticiones a un servidor (entre las que se incluyen la que nosotros tratamos de hacer), el navegador manda una petición “OPTIONS” en nombre del cliente preguntando al servidor si la petición que quiere mandar el cliente está permitida. El servidor debe responder con cierta información que indica al navegador que se puede continuar con la petición. Para más detalle sobre esto, recomendamos visitar el siguiente link: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

El error que mostramos en la Figura 28 lo recibimos debido a que el servidor del NBI no responde a “OPTIONS” con la información necesaria, por lo que el navegador cree que nuestra petición no está permitida y corta la comunicación. Para solucionar esto, hemos escrito un servidor proxy que nos puede servir de intermediario en la comunicación con el NBI. Éste se puede ver en “serve.py” dentro del directorio “JS_PYServer”, y es un script en Python3 que sencillamente añade la información requerida a la respuesta del servidor del NBI, mediante el módulo “http.server”.


```

35 def do_POST(self):
36     length = int(self.headers['Content-Length'])
37     body = self.rfile.read(length).decode('utf-8')
38     auth = self.headers['Authorization']
39     if auth:
40         if ' ' in auth: #there is a descriptor before the credentials
41             auth = auth[auth.index(' ')+1:] #like 'Basic ', for example
42             username, password = str(b64decode(auth),'utf-8').split(':') #decode from ascii
43         else: #credentials were not given
44             print('--Authorization required to access NBI')
45             username, password = None, None
46     response = self.post_NBI(body, username, password) #post request sent to NBI
47     self.send_response(response.status_code)
48     for h,v in response.headers.items():
49         if h.lower() not in ['date', 'server']: #automatically provided by our server
50             self.send_header(h,v)
51     self.send_header('Access-Control-Allow-Origin', self.headers['Origin'])
52     self.send_header('Access-Control-Allow-Credentials', 'true')
53     self.end_headers()
54     self.wfile.write((response.text).encode("utf-8"))
55
56 def do_OPTIONS(self):
57     self.send_response(200)
58     self.send_header('Content-type', 'text/plain')
59     self.send_header('Access-Control-Allow-Origin', self.headers['Origin']) #or '*' for all origins
60     self.send_header('Access-Control-Allow-Methods', 'POST, GET, OPTIONS')
61     self.send_header('Access-Control-Allow-Headers', self.headers['Access-Control-Request-Headers'])
62     #self.send_header('Access-Control-Max-Age', '86400') #OPTIONS req cached for 24h
63     self.send_header('Access-Control-Allow-Credentials', 'true')
64     self.end_headers()
65
66 def post_NBI(self, body, username='', password=''):
67     host = 'http://172.19.254.3/ws'
68     nbiswds = ['cpeService', 'qosService', 'elementsInformationService', 'multiCastService']
69     with requests.Session() as s:
70         s.auth=(username,password)
71         s.headers.update(self.headers)
72         for service in nbiswds: #try all wsds until right one
73             response = s.post(host+'/'+service, data=body)
74             if response.status_code!=500 or ET.fromstring(response.text)[0][0].find('detail') is not None:
75                 break #only continue if command not in service (status_code 500 and XML response has no detail in fault)
76         else:
77             print('--NBI command not found') #could maybe return xml message instead
78     return response

```

Fig. 29. Parte del servidor proxy diseñado para respetar CORS – serve.py

Como se puede apreciar en la Figura 29, el proxy diseñado responde a “OPTIONS” con los “headers” necesarios así como con un código de estatus 200. Para las peticiones “POST” hace lo propio y además ejecuta la función “post_NBI”. Ésta prueba a ejecutar el comando con todos los servicios del NBI hasta que reciba una respuesta apropiada, incluyendo la información de autenticación que debe ser enviada como “header” por la aplicación.

6.4 Aplicación web final

Gracias a este proxy que hemos diseñado, “req.js” podrá ahora comunicarse con el NBI. Para ello, sólo debemos cambiar el “host” a la IP del proxy (en nuestro caso 127.0.0.1:9000) y enviar las credenciales como un “header” individual, en lugar de incluidas en el método “open()” de la línea 16 de la Figura 26.

Para hacer funcionar la funcionalidad que hemos desarrollado, ahora sólo tendremos que ejecutar el servidor proxy mediante el comando “python3 serve.py” y abrir “index.html” del directorio “JS_PYServer” a través del servidor web para Chrome (“index.html” es idéntico a “req.html” de la sección 6.3). En esta página podrá introducir un Managed Group Id para ver cuántos CPEs hay en él. De esto se encarga “req.js” del directorio “JS_PYServer”, al cual se le ha añadido un poco de código para hacerlo más robusto, pero que no explicaremos en mayor profundidad aquí.

Animamos al lector a que habrá estos archivos y haga un esfuerzo por familiarizarse con el código, puesto que estos archivos se pueden ver como plantillas para desarrollar más

funciones de forma mecánica. Además, también sería beneficioso ejecutar los programas y probar con distintas peticiones, prestando especial atención al proceso HTTP que podrá seguir en la vista de desarrollador del navegador.

Finalmente, cabe recalcar que, igual que como hicimos en Python, también puede combinar funciones en JavaScript para desarrollar nueva funcionalidad.

7. CONCLUSIÓN

A lo largo de este documento hemos podido ver cómo varias tecnologías y técnicas pueden ser utilizadas para acceder y trabajar directamente con el servidor del NBI para el TotalNMS de Gilat. Gracias a todo ello, se puede desarrollar funcionalidad más avanzada, dinámica y eficiente, y tener mayor control sobre la experiencia del cliente.

8. BIBLIOGRAFÍA

1. Gilat Satellite Networks Ltd. SkyEdge II-c NBI for TotalNMS in v4.0, Operator's Guide. Revisión 1.2, Junio 2016. Documento número DC-103782(C)
2. *TutorialsPoint*. HTTP Tutorial: <https://www.tutorialspoint.com/http/>
3. *TutorialsPoint*. SOAP Tutorial: <https://www.tutorialspoint.com/soap/>
4. *W3Schools*. XML SOAP Tutorial: https://www.w3schools.com/xml/xml_soap.asp
5. *W3Schools*. XML WSDL Tutorial: https://www.w3schools.com/xml/xml_wsd.asp
6. *SoapUI*. Getting started with SoapUI: <https://www.soapui.org/getting-started.html>
7. *Requests*: HTTP for Humans: <http://docs.python-requests.org/en/master/>
8. *Python Software Foundation*. Python 3 Documentation: <https://docs.python.org/3/>
9. *W3Schools*. AJAX Tutorial: https://www.w3schools.com/xml/ajax_intro.asp
10. *MDN*. AJAX Docs: <https://developer.mozilla.org/en-US/docs/AJAX>
11. *MDN*. XMLHttpRequest API:
<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>
12. *MDN*. HTTP Access Control (CORS):
https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS
13. *Wikipedia*. Cross-origin resource sharing:
https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
14. *Repositorio GitHub del manual*: https://github.com/JaviSorribes/TotalNMS_NBI