Aplicación segura cliente-servidor con estado, XML y JSON

Descripción

En esta práctica se pide el diseño e implementación de una aplicación cliente-servidor con estado para proporcionar un servicio calculadora. El protocolo se implementa sobre sockets TCP y en una arquitectura multi-hilo, los mensajes van codificados en XML o en JSON. Se parte de la base de la práctica anterior y en esta versión se añade la opción de solicitar una conexión segura con el servidor basada en SSL y tanto con autenticación de cliente como de servidor.

Diseño

Sobre el diseño de la segunda práctica, mantenemos la misma estructura general, ya que lo único que se necesita añadir es la opción de crear una conexión SSL entre dos sockets seguros. De esta forma, en la clase *Server* es donde creamos tanto el socket no seguro como el seguro y con ayuda de dos hilos (*NonSecureServerThread, SecureServerThread*) esperamos peticiones en ambos de manera simultánea. De forma similar, en el cliente ahora preguntamos al lanzar la aplicación si se desea establecer una conexión segura.

Para poder hacer todo esto, hemos comentado que necesitamos crear un socket seguro en los dos extremos de la conexión, que se hará de manera prácticamente idéntica. Para ello, precisamos de certificados tanto para la autenticación de cliente como para la de servidor, esto lo haremos con ayuda de una CA llamada *practica3CA*. Esta Autoridad de Certificación será la encargada de generar y firmar los certificados del servidor y de los clientes. La generación tanto del certificado autofirmado de la CA como del resto de certificados se ha hecho con ayuda de la herramienta *OpenSSL*, posteriormente se ha realizado la conversión de todo esto al formato JKS que usa Java con la herramienta gráfica *KeyStore Explorer*.

La forma en la que se ha creado el contexto SSL en Java para fabricar un socket seguro ha sido por medio del uso de dos almacenes de certificados, el *KeyStore* y el *TrustedStore*. En el primero es donde almacenamos el certificado y la clave del usuario del socket, y en el segundo es donde se guarda el certificado de la CA, que se comprueba para decidir si se confía en la otra parte.

Protocolo

El protocolo consta de un único mensaje que sirve tanto para la petición como para la respuesta, esto es debido a que cada parte lo interpreta de una forma o de otra. El esquema que sigue este mensaje es el siguiente. En primer lugar, se encuentra la longitud del mensaje seguida de un final de línea (/n). A continuación, aparece el formato en el que se está mandando el mensaje (xml o json). Finalmente, y separado de lo anterior por el carácter “~”, se sitúa el mensaje propiamente dicho. De esta manera, el formato del mensaje quedaría:

LongitudMensaje

TipoMensaje~Mensaje

Sabiendo esto, queda ver cómo están formados los mensajes JSON y XML, que como se puede ver en los esquemas inferiores, simplemente representan un objeto *Calculator*.

```
{
   "user":"String",
   "operand1":"String",
   "operator":"String",
   "operand2":"String",
   "result":"String"
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<calculator>
   <user>String</user>
   <operand1>String</operand1>
   <operator>String</operator>
   <operand2>String</operand2>
   <result>String</result>
</calculator>
```

Ejemplos de uso

Como ya se mostró en las prácticas anteriores el funcionamiento de la concurrencia y de la persistencia, además del formato de los mensajes, en este ejemplo nos centraremos en cómo crea el cliente una conexión segura.

A continuación, se ve como el cliente Jorge inicia una conexión con el servidor de forma segura.

```
Do you want a secure connection? [y/n]
y
Enter user name:
Jorge
For the next query, XML or JSON
json
Enter the query or EXIT to end the connection:
1+1
Result: 2.0
For the next query, XML or JSON
xml
Enter the query or EXIT to end the connection:
ans+2
Result: 4.0
```

Ilustración 1: ejemplo de uso en el cliente Jorge

Como sabemos, todo el proceso del establecimiento de la conexión segura es transparente al usuario. De esta forma, lo analizaremos con ayuda de las trazas generadas por la aplicación en el apartado siguiente.

Trazas

Con ayuda de Wireshark, obtenemos el siguiente intercambio de mensajes durante la ejecución del ejemplo anterior:

```
TLSv1.2 Client Hello
TLSv1.2 Server Hello, Certificate, Server Key Exchange, Server Hello Done
TLSv1.2 Client Key Exchange
TLSv1.2 Change Cipher Spec
TLSv1.2 Encrypted Handshake Message
TLSv1.2 Change Cipher Spec
TLSv1.2 Encrypted Handshake Message
TLSv1.2 Application Data
TLSv1.2 Application Data
TLSv1.2 Application Data
TLSv1.2 Application Data
```

Ilustración 2: intercambio de mensajes del ejemplo

Observamos cómo se realiza el *Handshake* entre los dos extremos de la conexión, con su correspondiente intercambio de certificados y negociación de la suite de cifrado que utilizarán. Una vez finalizado, vemos que el tráfico intercambiado (petición-respuesta-petición-respuesta) ya va cifrado y somos incapaces de interpretarlo. Si analizamos con mayor detalle el intercambio, podemos encontrar el mensaje donde se presenta el certificado del servidor:

```
Certificates (916 bytes)
   Certificate Length: 913
 ⌄ Certificate: 3082038d30820275a0030201020201023300d06092a864886... (id-at-commonName=server,id-at-organizationName=practica3CA
   › signedCertificate
   › algorithmIdentifier (sha256WithRSAEncryption)
     Padding: 0
     encrypted: 73380d54ca26d58c2304158a0f7a1724b8b4dbe2ed7836cc...
```

Ilustración 3: detalle del certificado del servidor