Implementación del algoritmo

Por Melanie Pinto dos Anjos

Java fue elegido el lenguaje de programación en el que se escribiría el trabajo desde el primer boceto del programa. Tras realizar un breve trabajo de investigación[i] sobre el algoritmo, en seguida me dispuse a hacer un primer diseño en este lenguaje dado que es el lenguaje con el que más he trabajado y con el que más cómoda me siento. Tanto las pruebas iniciales como las diferentes versiones del programa hasta lograr la definitiva fueron desarrolladas en Eclipse como IDE.

Primera versión del programa

La primera implementación del algoritmo constó sólo de tres clases:

- **Inicializa.java** donde, en cada ejecución, se generan aleatoriamente[ii] un número primo y un generador que cumplen las condiciones que dicta el algoritmo de Diffie-Hellman.
- **Usuario.java** donde se crea un usuario que tiene los siguientes **métodos** y **variables de instancia**:
    - **Primo**: Variable con el único objetivo de almacenar el número primo inicializado en esta ejecución.
    - **Generador**: Variable con el único objetivo de almacenar el generador inicializado en esta ejecución.
    - **Secreto**: Inicialmente *null*. En esta variable se almacenará el secreto generado aleatoriamente por este usuario en esta ejecución.
    - **Clave** (privada): Inicialmente *null*. En esta variable se almacenará la clave privada generada mediante el algoritmo en esta ejecución.
    - **Clave compartida**: Inicialmente *null*. En esta variable se almacenará la clave compartida final con el único objetivo de compararla con la del otro usuario durante la realización de pruebas.
    - **asignaClavesPublicas**: Este método recibe por parámetros el *primo* y el *generador* y los asigna en las correspondientes variables de instancia del usuario.
    - **calculaSecreto**: Este método simula la acción que realizaría el usuario de escoger un número *secreto*, generando aleatoriamente un número menor que el *primo*.
    - **calculaClave**: Este método calcula la *clave* privada del usuario según el algoritmo de Diffie-Hellman. Calcula la potencia del *generador* elevado al *secreto* y calcula el módulo de esta potencia con el *primo*.
    - **secretoComun**: Este método recibe por parámetro la *clave* privada de otro usuario y calcula la *clave compartida* mediante el algoritmo de Diffie-Hellman, esto es, calcula la potencia de la *clave* recibida elevada a su *secreto* propio y calcula el módulo de esta potencia con el *primo*.
- **Main.java** donde se comprueba que los métodos funcionan correctamente. En esta primera versión del programa se llama al método main de la clase **Inicializa.java** para generar unos nuevas claves públicas (*primo* y *generador*) en cada ejecución y se

muestran éstas por consola. A continuación se generan dos usuarios (en este ejemplo se llaman **Alberto** y **Beatriz**) llamando dos veces al constructor de la clase Usuario.java. Se asignan las claves públicas a ambos usuarios y cada uno de ellos calcula su *secreto* y su *clave* privada. Después se simula el intercambio de claves en dos variables llamadas **sharedAB** y **sharedBA** que, como sus nombres indican, calculan la *clave compartida* siendo **Alberto** quien recibe la clave de **Beatriz** y siendo **Beatriz** quien recibe la clave de **Alberto**. Para comprobar que todo está en orden hay unas últimas líneas de código con un condicional que imprime por consola si la *clave compartida* generada en ambos usuarios es o no la misma.

```
import java.math.*;

public class Main {

    //Intercambio de claves Diffe-Hellman entre dos clientes
    //Compresión y Seguridad
    public static void main(String[] args) {

        //INICIALIZAMOS EL SISTEMA Y GENERAMOS LAS DOS CLAVES PÚBLICAS
        //GUARDAMOS AMBOS VALORES PARA UTILIZARLOS POSTERIORMENTE
        Inicializa init = new Inicializa();
        BigInteger primo = init.getPrimo();
        BigInteger generador = init.getGenerador();

        System.out.print("*** Claves públicas *** \nPrimo: "+primo+"\nGenerador: "+generador+"\n");

        //DECLARAMOS E INICIALIZAMOS LOS DOS CLIENTES QUE USARÁN EL SISTEMA
        //LES ASIGNAMOS LAS CLAVES PUBLICAS GENERADAS ANTERIORMENTE
        Usuario a = new Usuario("Alberto");
        a.asignaClavesPublicas(primo, generador);
        a.calculaSecreto();
        a.calculaClave();

        Usuario b = new Usuario("Beatriz");
        b.asignaClavesPublicas(primo, generador);
        b.calculaSecreto();
        b.calculaClave();

        //AMBOS CLIENTES CALCULAN LA CLAVE COMPARTIDA
        //Esto deberia ser privado, no deberia poder acceder.
        BigInteger sharedAB = a.secretoComun(b.getClave());
        BigInteger sharedBA = b.secretoComun(a.getClave());

        if(sharedAB.compareTo(sharedBA)==0){
            System.out.println("La clave compartida es igual.");
        }
        else{
            System.out.println("La clave compartida es distinta.");
        }

    }

}
```

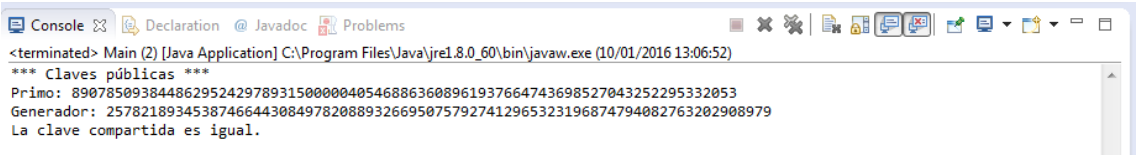
Inclusión de la librería BigInteger

La primera versión del algoritmo fue implementada con números enteros (pequeños y más rápidos a la hora de hacer operaciones) para lograr la máxima eficiencia mediante la sencillez. Una vez fue comprobado que el algoritmo funcionaba correctamente me dispuse a adaptarlo a números de 256 bits mediante la librería BigInteger. Para esto, tuve que realizar otro breve trabajo de investigación[iii] dado que nunca había trabajado con esta librería con anterioridad. Fue, sin embargo, bastante sencillo.

Sólo fue necesario modificar el tipo de las variables de instancia de las diferentes clases y modificar algunos métodos de la clase Usuario.java de modo que funcionaran con los métodos nativos de este nuevo tipo de datos. Adjunto, a continuación, una captura de pantalla de los nuevos métodos de la clase **Usuario.java** ya preparados para trabajar con números grandes.

```java
public void asignaClavesPublicas(BigInteger p, BigInteger g){
    primo = p;
    generador = g;
}

public BigInteger calculaSecreto(){
    secreto = new BigInteger(numBits, new Random());

    while(primo.compareTo(secreto)<0){ //Mientras primo sea menor que s
        secreto = new BigInteger(numBits, new Random());
    }

    return secreto;
}


/** << ALGORITMO >>
 * Alice and Bob agree to use a modulus p = 23 and base g = 5
 * (which is a primitive root modulo 23).
 * Alice chooses a secret integer a = 6,
 * then sends Bob A = g^a mod p
 * A = 5^6 mod 23 = 8
 * A = g^a mod p **/

//modPow(BigInteger exponent, BigInteger m)
//Returns a BigInteger whose value is (thisexponent mod m)
public BigInteger calculaClave(){
    clave = generador.modPow(secreto, primo);
    return clave;
}

public BigInteger secretoComun(BigInteger clave){
    sharedkey  = clave.modPow(secreto, primo);
    return sharedkey;
}
```

Tras comprobar que el algoritmo seguía funcionando correctamente después de estos cambios, el programa quedaba listo para comenzar a trabajar en la conexión entre dos ordenadores para realizar el intercambio de claves entre ellos y dejar de lado las simulaciones en un único equipo.



Adaptación a la conexión de red (parte I)

Tal vez haya sido la parte más complicada de implementar. Hasta ahora el programa era bastante sencillo ya que constaba únicamente de tres clases. Para adaptarlo a la conexión de red tuvo que ser complementado con ocho clases nuevas, sumando un total de once clases.

Las clases nuevas son las siguientes:

- Para la ejecución del programa
    - Host.java
    - Client.java
- Para las conexiones entre los ordenadores
    - Servidores
        - Servidor1.java
        - Servidor2.java
        - Servidor3.java
    - Clientes
        - Cliente1.java

- Cliente2.java
- Cliente3.java

Para que el programa sea ejecutado de forma paralela en ambos ordenadores son necesarias dos instancias distintas dado que no ambos usuarios ejecutan el mismo código.

La clase **Host.java** llama al método main de la clase **Inicializa.java** y genera en cada ejecución el par de claves públicas correspondientes al primo y generador. Asigna estas claves a las correspondientes variables de instancia declaradas con este fin y procede a realizar la primera conexión con el ordenador que está ejecutando **Cliente.java**.

Todas las conexiones se realizan de la misma forma. Uno de los ordenadores actúa como servidor y el otro como cliente. En el caso particular de la primera conexión, **Host.java** actúa como cliente llamando al método main de la clase **Cliente1.java** y el ordenador que está ejecutando paralelamente la clase **Client.java** actúa como servidor en esta primera conexión, llamando simultáneamente al método main de la clase **Servidor1.java**.

La clase **Cliente1.java** crea un canal de comunicación con la IP del usuario que ejecuta el **Servidor1.java** y un puerto que es común a ambos. En este caso en particular, la función de la conexión no es otra que hacer llegar a la clase **Client.java** el primo y el generador que ha inicializado **Host.java** en su ejecución.

Inmediatamente a continuación, cuando ambos usuarios tienen ya almacenadas las claves públicas proceden a calcular sus correspondientes claves privadas para intercambiarlas. La función de las dos conexiones restantes entre Host.java y Client.java es intercambiar las claves privadas entre ambos usuarios de modo que ambos puedan calcular la clave compartida. Por las pruebas anteriores del algoritmo se asume que funciona correctamente y ambos usuarios obtienen la misma clave.

Adaptación a la conexión de red (parte II)

Para finalizar, la cuarta y última versión del programa permite que las IPs y los puertos de las distintas conexiones sean parámetros que reciben las clases **Host.java** y **Client.java**, un requerimiento para trabajar con la interfaz gráfica. Para que funcione esta última versión del algoritmo se requiere que todas las variables estén almacenadas en un array de Strings que se pasará por parámetro a todas las llamadas a las distintas clases Cliente y Servidor.

El programa ya está listo.

---

[i] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
[ii] http://docs.oracle.com/javase/7/docs/api/java/util/Random.html
[iii] http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html