# Universidad de los Andes

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas
Infraestructura Computacional
Periodo 2022-20

# Caso 3
# Documentación

Jonathan Rivera Otero - 202022864
Juan Diego Yepes Parra - 202022391
Sebastian Enrique Casanova Ospina - 202115116

1 de noviembre de 2022
Bogotá D.C.

# Índice

# 1. Problema

A continuación se detalla el problema a resolver en este caso.

## 1.1. Enunciado

Supondremos que una compañía ofrece a sus clientes la posibilidad de hacer consultas en línea, garantizando confidencialidad e integridad de la información. Su tarea consiste en construir un programa cliente que envía solicitudes al servidor de la compañía y verifica la integridad de los resultados recibidos de acuerdo con el protocolo de comunicación establecido. A continuación, se presenta información adicional del servidor y el cliente.

- El servidor es un proceso que recibe y responde las consultas de los clientes. Para simplificar esta parte del problema y concentrarnos en los aspectos de seguridad, la consulta solo será un número y el servidor debe responder con número+1. El código fuente del servidor será entregado por los profesores del curso.
- El cliente es un proceso que envía una consulta al servidor, espera la respuesta y la presenta al usuario. Tanto servidor como cliente deben cumplir con las siguientes condiciones:
  - Las comunicaciones entre cliente-servidor deben usar *sockets* y deben estar cifradas de acuerdo con el protocolo definido.
  - Desarrolle en Java. No use librerías especiales, solo las librerías estándar (`java.security`, `javax.crypto`).
  - Generaremos por adelantado las llaves pública y privada del servidor. Tenga en cuenta que en una instalación real la llave pública puede ser leída por cualquiera, pero la llave privada solo debería estar al alcance del propietario.

### 1.2. Protocolo de ejecución

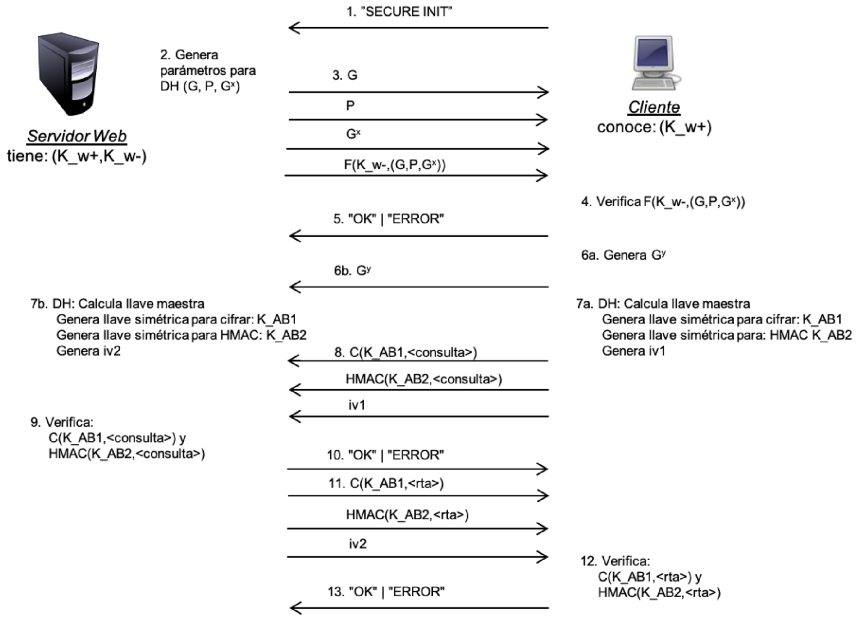



Figura 1. Protocolo de comunicación entre el servidor y el cliente.

## 2. Solución

A partir de lo anterior, proponemos la siguiente solución al problema.

### 2.1. Explicación del funcionamiento

Debido a que el código del servidor fue proporcionado, nuestra solución se centra exclusivamente en la parte del protocolo que responde al cliente.

Para la solución implementamos 2 clases `ClientMain`, que se va a encargar de ejecutar el thread o los threads de los clientes, `ClientProtocol`, que va a hacer todos los procedimientos necesarios del protocolo para el cliente, y `ClientThread`, el Thread que va a emular al cliente.

Ejecutamos la clase `ClientMain` después de que se haya ejecutado `ServidorMain` que va a poner a correr el servidor. La clase va a tener definidas las variables estáticas finales de puerto y servidor. Por medio de input pedimos al usuario que ingrese la cantidad de clientes que van a comunicarse con el servidor. A continuación por medio de un `for` loop creamos

la cantidad de objetos de la clase `ClientThread` que nos haya indicado el usuario, pasando por parámetro el número del puerto, el nombre (dirección ip) del servidor y el id que se le va a asignar a el cliente. Posteriormente inicializamos los Threads por medio del método `start()`.

```
for (int i = 0; i < number; i++) {
    ClientThread clientThread = new ClientThread(PUERTO, SERVIDOR, String.valueOf(i));
    clientThreads[i] = clientThread;
}

for (ClientThread ct:
     clientThreads) {
    System.out.println("====================================");
    ct.run();
}
```

Esto nos va a llevar a `clientThread`, donde creamos el socket, el escritor (`PrintWriter`) y el lector(`BufferedReader`). Procedemos a crear una instancia de `ClientProtocol` pasando por parámetro el `id` y posteriormente se ejecuta el método `protocol()` de dicha clase, ingresando por parámetro el lector y el escritor.

```
try {
    socket = new Socket(server, port);
    escritor = new PrintWriter(socket.getOutputStream(), true);
    lector = new BufferedReader(new InputStreamReader(socket.getInputStream()));
} catch (IOException e){
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
}

ClientProtocol clientProtocol = new ClientProtocol(id);
try {
    clientProtocol.protocol(lector,escritor);
} catch (Exception e) {
    throw new RuntimeException(e);
}
```

En la clase `ClientProtocol` empezamos por mandar el mensaje “SECURE INIT” para iniciar el protocolo. Se llama al método `println()` del escritor para enviar el mensaje. A continuación se utiliza el lector, por medio del método `readline()` para recibir los mensajes enviados por el servidor en respuesta, que serán los valores G, P y Gx para calcular la llave maestra por medio de Diffie-Hellman, y la firma creada a partir de los valores G,P y Gx como mensaje y la llave privada del servidor como llave.

```
 pOut.println("SECURE INIT");

if ((stringG = pIn.readLine()) != null) System.out.println("Client: " + id + " found G: " +
    stringG);
if ((stringP = pIn.readLine()) != null) System.out.println("Client: " + id + " found P: " +
    stringP);
```

```
5 if ((stringGx = pIn.readLine()) != null) System.out.println("Client: " + id + " found Gx: " +
      stringGx);
6 if ((firma = pIn.readLine()) != null) System.out.println("Client: " + id + " found signature: "
      + firma);
```

A continuación hacemos la verificación de la firma con el método `checkSignature()` de la clase `SecurityFunctions`, colocando como parámetros la llave publica del servidor que se nos proporciona y los valores G,P Y Gx que recibimos, concatenados. El resultado del método `checkSignature()` va a arrojar un booleano que nos dirá si la firma enviada se ha verificado exitosamente. De ser falso el resultado se enviara el mensaje ERROR y se acabara el proceso en los threads de cliente y servidor. De lo contrario se continuara el protocolo.

```
1 assert firma != null;
2 boolean auth = f.checkSignature(publicaServidor, str2byte(firma), msj);
```

Continuando con el protocolo, se enviará un mensaje OK al servidor para que también prosiga con el protocolo. A continuación, generaremos nuestro valor X necesario para calcular la llave maestra con Diffie-Hellman de manera aleatoria. Calcularemos Gy por medio del método `modPow()` de Java y lo enviaremos al servidor. Posteriormente utilizaremos Gx y P dados por el servidor, y nuestro X generado para calcular la llave maestra, de nuevo utilizando el método `modPow()` de Java.

```
1  System.out.println("Client " + id + " found signature successful, OK message sent.");
2  pOut.println("OK");
3
4  SecureRandom r = new SecureRandom();
5  int integerX = Math.abs(r.nextInt());
6  BigInteger x = BigInteger.valueOf(integerX);
7  BigInteger gy = G2X(G, x, P);
8
9  BigInteger llave_maestra = calcular_llave_maestra(Gx,x,P);
10 String str_llave = llave_maestra.toString();
```

A continuación, generaremos la llave simétrica para cifrar y el HMAC para hacer la autenticación, ambos se generarán con la llave maestra llamando csk1 y csk2 de la clase `SecurityFunctions`. Estos métodos harán digest para generar los hash correspondientes. Se genera después el vector de inicialización, de manera aleatoria.

```
1 SecretKey sk_srv = f.csk1(str_llave);
2 SecretKey sk_mac = f.csk2(str_llave);
3 byte[] iv1 = generateIvBytes();
```

Lo próximo que hará el programa será cifrar la consulta, para esto se llamara al método `senc()` de la clase `SecurityFunctions`, pasando como parámetro el mensaje de consulta (que será el id que se le asigno al Thread del cliente), la lleva simétrica generada y el vector de inicialización. Se generara también el código de autenticación HMAC con el mensaje de consulta y la llave de HMAC generada, posteriormente se enviara el servidor la consulta cifrada, el código de autenticación y el vector de inicialización.

```
String consulta = id;
String str_iv1 = byte2str(iv1);
IvParameterSpec ivSpec1 = new IvParameterSpec(iv1);
byte[] byteC = f.senc(consulta.getBytes(), sk_srv, ivSpec1, "Client: "+ id);
String c = byte2str(byteC);
byte[] byteHMAC = f.hmac(consulta.getBytes(), sk_mac);
String hmac = byte2str(byteHMAC);

pOut.println(c);
System.out.println("Client " + id + " C: " + c);
pOut.println(hmac);
System.out.println("Client " + id + "HMAC: " + hmac);
pOut.println(str_iv1);
System.out.println("Client " + id + " iv1: " + str_iv1);
```

Posteriormente recibiremos la respuesta del servidor a nuestros mensajes, si esta arroja un mensaje diferente a OK, quiere decir que el servidor a devuelto error en nuestra consulta por lo que se acabara el proceso en ambos threads. De lo contrario continuaremos el protocolo. Para continuar el programa va leer los mensajes enviados por el servidor que serán la respuesta de la consulta cifrada, el código de autenticación para la anterior y el vector de inicialización usado para cifrar la respuesta. A continuación descifraremos la respuesta llamando al método `sdec()`, asginando los valores por parámetro del mensaje cifrado, nuestra llave de cifrado simétrico generada y el vector de inicialización enviado por el servidor. Finalmente vereficaremos la integridad del mensaje descifrado , para esto llamaremos al método `checkInt()` con el mensaje descifrado, la llave de HMAC que generamos y el código de autenticación que recibimos del servidor.

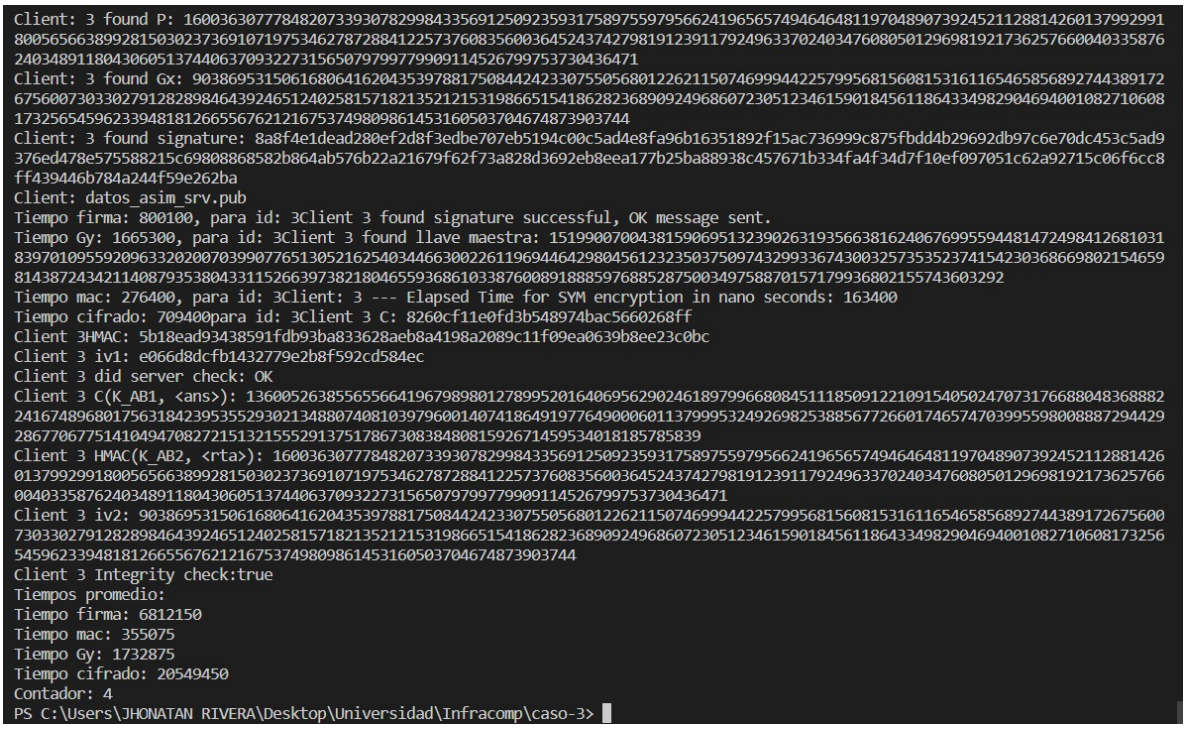
```
/* 11. We receive C(K_AB1, <rta>) and HMAC(K_AB2, <rta>)*/
    if ((stringCiphered = pIn.readLine()) != null) System.out.println("Client " + id + " C(K_AB1
    , <ans>): " + stringG);
    if ((stringHMAC = pIn.readLine()) != null) System.out.println("Client " + id + " HMAC(K_AB2,
     <rta>): " + stringP);
    if ((StringIv2 = pIn.readLine()) != null) System.out.println("Client " + id + " iv2: " +
    stringGx);

    assert StringIv2 != null;
    byte[] iv2 = str2byte(StringIv2);
    IvParameterSpec ivSpec2 = new IvParameterSpec(iv2);

    /* 12. We verify C(K_AB1, <rta>) and HMAC(K_AB2, <rta>)*/
    assert stringCiphered != null;
    byte[] descifrado = f.sdec(str2byte(stringCiphered), sk_srv, ivSpec2);
    assert stringHMAC != null;
    boolean verificar = f.checkInt(descifrado, sk_mac, str2byte(stringHMAC));
    System.out.println("Client " + id + " Integrity check:" + verificar);
```

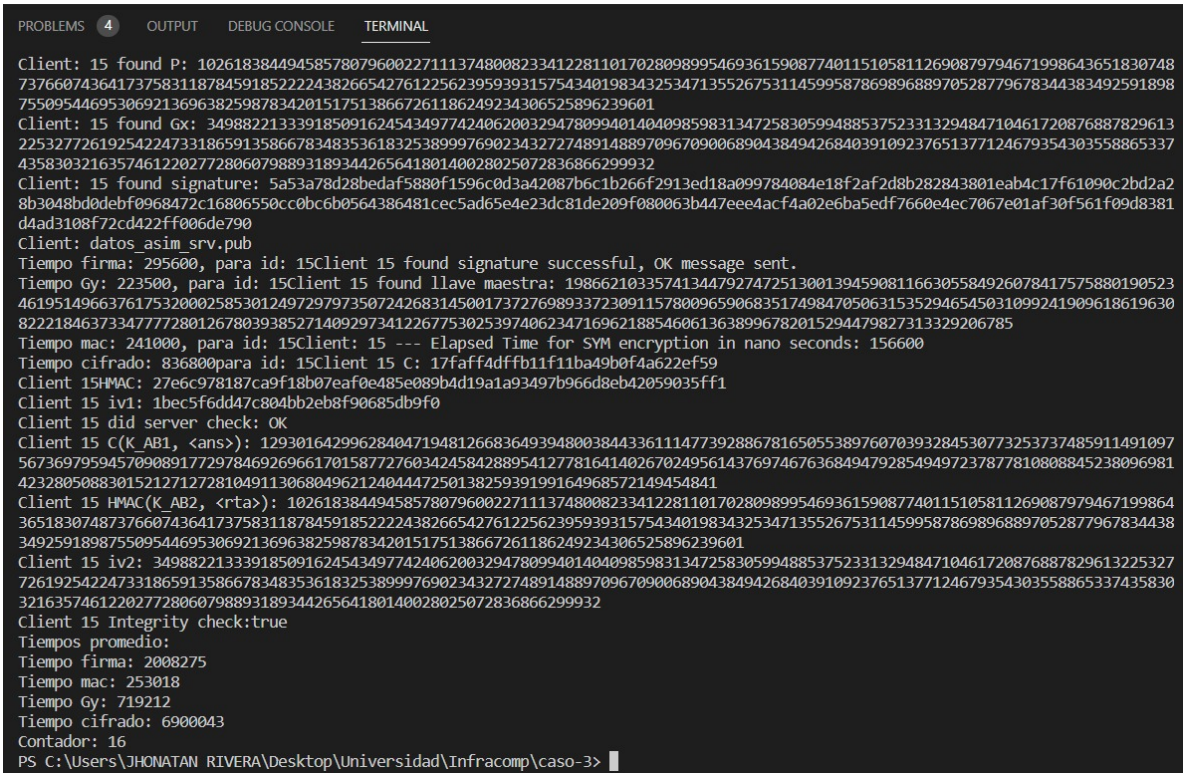
Si el método `checkInt()` devuelve true acabaremos el protocolo con un mensaje OK indicando que el protocolo se concluyo de manera exitosa. De lo contrario, quiere decir que fallo

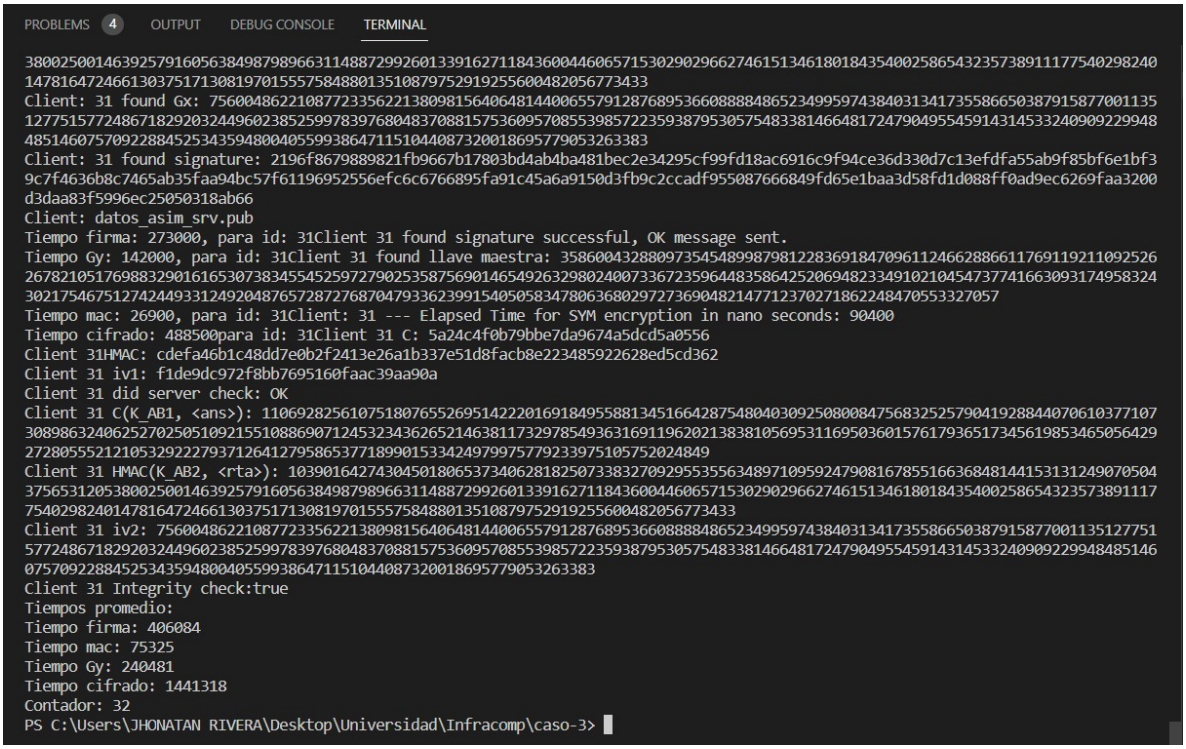
la verificación de la integridad, por lo que enviaremos un mensaje ERROR indicando que concluimos el protocolo de manera no exitosa.

```
if(verificar){
    pOut.println("OK");
}
else {
    pOut.println("ERROR");
}
```

## 2.2. Pantallazos de ejecución




PROBLEMS 4
OUTPUT
DEBUG CONSOLE
TERMINAL
Client: 15 found P: 10261838449458578079600227111374800823341228110170280989954693615908774011510581126908797946719986436518307487376607436417375831187845918522224382665427612256239593931575434019834325347135526753114599587869896889705287796783443834925918987550954469530692136963825987834201517513866726118624923430652589623960 1
Client: 15 found Gx: 34988221333918509162454349774240620032947809940140409859831347258305994885375233132948471046172087688782961322532772619254224733186591358667834835361832538999769023432727489148897096709006890438494268403910923765137712467935430355886533743583032163574612202772806079889318934426564180140028025072836866299932
Client: 15 found signature: 5a53a78d28bedaf5880f1596c0d3a42087b6c1b266f2913ed18a099784084e18f2af2d8b282843801eab4c17f61090c2bd2a28b3048bd0debf0968472c16806550cc0bc6b0564386481cec5ad65e4e23dc81de209f080063b447eee4acf4a02e6ba5edf7660e4ec7067e01af30f561f09d8381d4ad3108f72cd422ff006de790
Client: datos_asim_srv.pub
Tiempo firma: 295600, para id: 15Client 15 found signature successful, OK message sent.
Tiempo Gy: 223500, para id: 15Client 15 found llave maestra: 19866210335741344792747251300139459081166305584926078417575880190523461951496637617532000258530124972979735072426831450017372769893372309115780096590683517498470506315352946545031099241909618619630822218463733477772801267803938527140929734122677530253974062347169621885460613638996782015294479827313329206785
Tiempo mac: 241000, para id: 15Client: 15 --- Elapsed Time for SYM encryption in nano seconds: 156600
Tiempo cifrado: 836800para id: 15Client 15 C: 17faff4dffb11f11ba49b0f4a622ef59
Client 15HMAC: 27e6c978187ca9f18b07eaf0e485e089b4d19a1a93497b966d8eb42059035ff1
Client 15 iv1: 1bec5f6dd47c804bb2eb8f90685db9f0
Client 15 did server check: OK
Client 15 C(K_AB1, <ans>): 1293016429962840471948126683649394800384433611147739288678165055389760703932845307732537374859114910975673697959457090891772978469269661701587727603424584288954127781641402670249561437697467636849479285494972378778108088452380969814232805088301521271272810491130680496212404447250138259391991649685721494548 41
Client 15 HMAC(K_AB2, <rta>): 10261838449458578079600227111374800823341228110170280989954693615908774011510581126908797946719986436518307487376607436417375831187845918522224382665427612256239593931575434019834325347135526753114599587869896889705287796783443834925918987550954469530692136963825987834201517513866726118624923430652589623960 1
Client 15 iv2: 34988221333918509162454349774240620032947809940140409859831347258305994885375233132948471046172087688782961322532772619254224733186591358667834835361832538999769023432727489148897096709006890438494268403910923765137712467935430355886533743583032163574612202772806079889318934426564180140028025072836866299932
Client 15 Integrity check:true
Tiempos promedio:
Tiempo firma: 2008275
Tiempo mac: 253018
Tiempo Gy: 719212
Tiempo cifrado: 6900043
Contador: 16
PS C:\Users\JHONATAN RIVERA\Desktop\Universidad\Infracomp\caso-3>




# 3. Resultados

## 3.1. Solución de preguntas propuestas

- En el protocolo descrito el cliente conoce la llave pública del servidor (`K_w+`). ¿Cuál es la manera común de enviar estas llaves para comunicaciones con servidores web?

  Si bien la llave publica puede ser conocida por cualquier persona sin que afecte la seguridad de la encriptación, es responsabilidad del servidor de la llave determinar un método seguro para enviarla al cliente, ya que alguien podría suplantar la identidad del servidor y enviar una llave diferente al cliente para poder tener acceso al contenido que esta vaya a intercambiar con el servidor. Usualmente para el envío seguro de llaves públicas por parte de un servidor a un cliente se utiliza `SSH`. `SSH` (Secure Shell) es un protocolo de red para transferir información de manera segura en una red posiblemente no segura.

- El protocolo Diffie-Hellman garantiza “Forward Secrecy”, explique en qué consiste esta garantía.

  Esta propiedad proporcionada por el protocolo Diffie-Hellman consiste en garantizar que el eventual descubrimiento de las claves secretas utilizadas en una iteración/ins-

tancia del protocolo no va a comprometer la seguridad de otras iteraciones/instancias/usos del protocolo, ya que las claves usadas no se pueden utilizar para descifrar otras claves. Esto se da por que el protocolo siempre va a arrojar nuevas llaves cada vez que se llame, por lo que tener el conocimiento de una llave usada no va a repercutir en ningún uso próximo o anterior del protocolo.

## 3.2. Tiempos de ejecución

Teniendo en cuenta el enunciado, para cada escenario propuesto (4, 16 y 32 clientes delegados) se muestran los tiempos de ejecución y sus respectivas gráficas, para la ejecución de los clientes delegados

| Cantidad de clientes | 4 | 16 | 32 |
|---|---|---|---|
| Tiempo cifrado | 11345525 | 3393668.75 | 2141137 |
| Tiempo generación MAC | 154525 | 112831 | 74534 |
| Tiempo verificación firma | 4193500 | 1764581 | 667925 |
| Tiempo cálculo Gy | 731275 | 350293 | 257559 |

Tabla 1: Tiempos promedio de cada cliente

| Cliente | Cifrado | MAC | Ver.Firma | Gy |
|---|---|---|---|---|
| 1 | 44083300 | 228000 | 15417100 | 947300 |
| 2 | 320200 | 140900 | 657100 | 911300 |
| 3 | 435600 | 95600 | 287300 | 802900 |
| 4 | 543000 | 153600 | 412500 | 263600 |

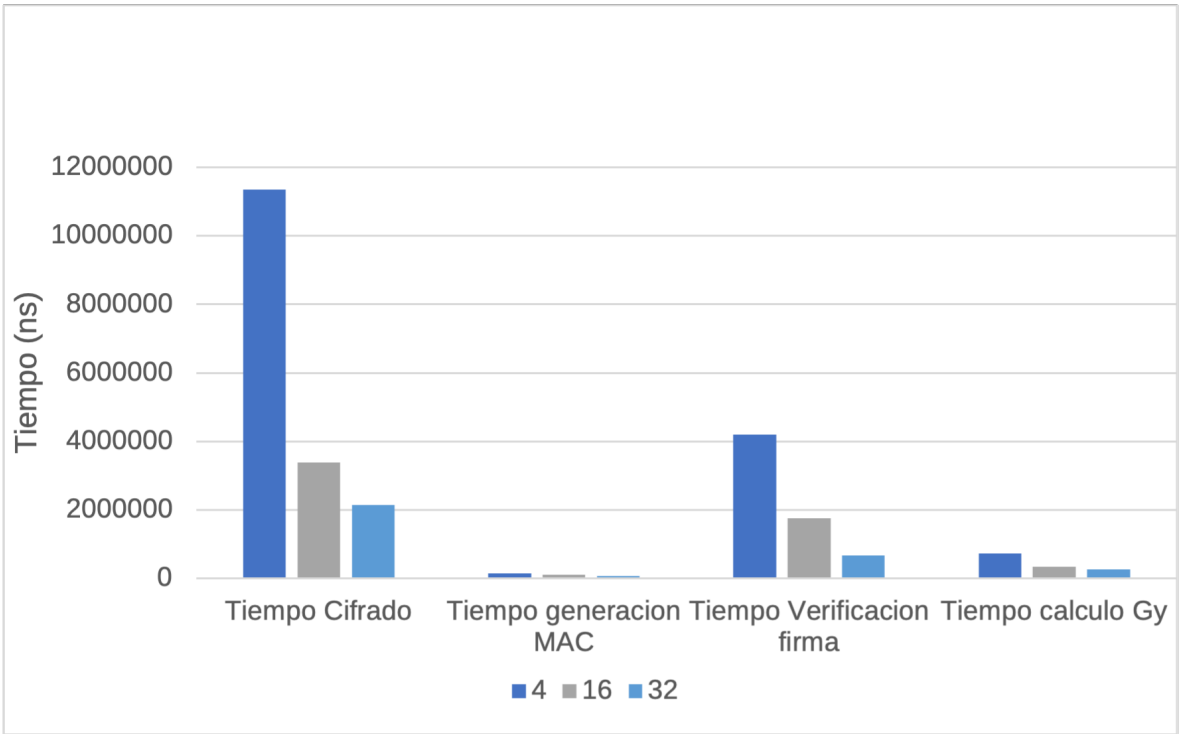
Tabla 2: Tiempos para 4 clientes

| Cliente | Cifrado | MAC | Ver.Firma | Gy |
|---|---|---|---|---|
| 1 | 47726900 | 238300 | 23683700 | 1187500 |
| 2 | 529000 | 92400 | 654300 | 1012700 |
| 3 | 389400 | 103700 | 297800 | 631200 |
| 4 | 627300 | 113600 | 358000 | 568500 |
| 5 | 383900 | 109800 | 199700 | 167200 |
| 6 | 413000 | 113700 | 212700 | 329400 |
| 7 | 581000 | 88400 | 361400 | 203400 |
| 8 | 356600 | 85600 | 188000 | 196000 |
| 9 | 356900 | 78200 | 189500 | 136000 |
| 10 | 560700 | 90700 | 333500 | 156300 |
| 11 | 421500 | 93100 | 347800 | 252100 |
| 12 | 452900 | 233500 | 241400 | 205100 |
| 13 | 408900 | 69400 | 418000 | 164400 |
| 14 | 379100 | 109700 | 215800 | 124800 |
| 15 | 391300 | 87700 | 364400 | 129400 |
| 16 | 320300 | 97500 | 167300 | 140700 |

Tabla 3: Tiempos para 16 clientes

| Cliente | Cifrado | MAC | Ver.Firma | Gy |
|---|---|---|---|---|
| 1 | 55048500 | 258000 | 13754700 | 942100 |
| 2 | 327500 | 116500 | 726500 | 1016100 |
| 3 | 511800 | 127600 | 450900 | 714000 |
| 4 | 435900 | 144500 | 252800 | 689500 |
| 5 | 336600 | 140600 | 225600 | 207000 |
| 6 | 531000 | 111800 | 305200 | 942400 |
| 7 | 393600 | 72600 | 192700 | 137100 |
| 8 | 407400 | 77500 | 188600 | 167600 |
| 9 | 606900 | 117600 | 419400 | 170100 |
| 10 | 337200 | 72900 | 184900 | 150900 |
| 11 | 403000 | 71200 | 342000 | 247200 |
| 12 | 584800 | 80100 | 248600 | 175700 |
| 13 | 442700 | 138500 | 321100 | 160600 |
| 14 | 342600 | 107500 | 254900 | 163400 |
| 15 | 390000 | 128300 | 185700 | 123400 |
| 16 | 495000 | 100600 | 171900 | 104900 |
| 17 | 458900 | 119800 | 175300 | 118500 |
| 18 | 408700 | 23300 | 237000 | 140500 |
| 19 | 490800 | 18400 | 172800 | 147900 |
| 20 | 360800 | 22600 | 201800 | 107200 |
| 21 | 356600 | 24700 | 235600 | 302700 |
| 22 | 453200 | 18000 | 164500 | 112000 |
| 23 | 380500 | 21400 | 161100 | 114900 |
| 24 | 509600 | 92000 | 197300 | 115000 |
| 25 | 363000 | 23000 | 171200 | 107600 |
| 26 | 414300 | 20600 | 240900 | 100900 |
| 27 | 440400 | 24200 | 274800 | 111600 |
| 28 | 459900 | 20600 | 158400 | 110200 |
| 29 | 507900 | 27900 | 204500 | 117700 |
| 30 | 511100 | 24800 | 220300 | 158900 |
| 31 | 328600 | 17000 | 159300 | 115100 |
| 32 | 477600 | 21000 | 173300 | 149200 |

Tabla 4: Tiempos para 32 clientes




Gráfica 1: Tiempos de ejecución promedio

### 3.3. Comentarios sobre tiempos de ejecucion

Observando los resultados medidos en el punto previo se pueden plantear varias conclusiones: En primer lugar, notamos que los tiempos promedio de cada uno de los 4 procesos analizados se organizan de la siguiente manera (del menos demorado al mas demorado):

1. Generación código de autenticación (MAC)
2. Cálculo Gy
3. Verificación de la firma
4. Cifrado de consulta

El tiempo para generar el código de autenticación es el menor (y por un margen bastante amplio), debido a que utiliza digest (generación de hash) para realizar la operación, la generación de códigos de hash es un proceso bastante rápido, considerablemente mas rápido que la mayoría de algoritmos de cifrado, ya sean simétricos o asimétricos. El calculo de Gy también es considerablemente rápido comparado a los procesos de verificación de firma y cifrado de consulta, considerando que para la generación de este valor la maquina

solo tiene que hacer 2 operaciones: elevar una base (potenciar) y sacar un modulo, y si bien estas operaciones son pesadas en el procesador comparados a una suma, multiplicación, xor, etc, el hecho de que solo se realicen una vez, hace del proceso algo simple. Los tiempos de calculo ya suben considerablemente para los procesos de cifrado y el de verificación de firma. Esto se debe a que ya son algoritmos mas complejos que realizan un mayor numero de operaciones, por su parte para el cifrado de la consulta se utiliza un algoritmo simétrico de AES con CBC para un mensaje de 256 bits con Padding, para la verificación de la firma se utiliza un algoritmo asimetrico RSA. Los tiempos de la verificación de la firma son menores a los del cifrado de la consulta, si bien los algoritmos simétricos suelen ser mas rápidos que los algoritmos asimétricos los resultados se explican por la longitud del mensaje a cifrar. Para el cifrado de la consulta se coge el mensaje original y se le añade un padding para que su longitud total sea 256 bit, lo que lo hace un proceso especialmente demorado ya que supera la longitud de valores con los que puede operar el procesador (64 bits), lo que provoca que se tengan que hacer varios procesos adicionales para realizar operaciones sobre el mensaje a cifrar. Por otro lado, la verificación de la firma, si bien utiliza el algoritmo mas demorado de todos los que se utilizaron en el caso, tiene un mensaje muy corto que descifrar (los valores concatenados de G,P y Gx que utilizo el servidor para el protocolo Diffie-Hellman).

Otro aspecto importante a destacar con los resultados obtenidos en la toma de datos es el hecho de que todos los tiempos calculados en cada uno de los 4 procesos disminuye conforme se aumenta la cantidad de clientes concurrentes. Si bien este suceso puede parecer contra-intuitivo:

¿Más clientes $\Rightarrow$ más trabajo para el procesador $\Rightarrow$ más demora en los cálculos?

Generamos la siguiente hipótesis para explicar los resultados: Si bien es cierto que en teoría el calculo de los procesos se debería retrasar una vez el procesador este ocupado con muchos clientes, en la situación planteada los cálculos necesarios no son lo suficientemente demandantes del procesador con el que se realizaron, por lo que nunca va a generar un retraso en estos, a no ser que la cantidad de clientes aumente a un numero bastante grande. Por otra parte la disminución de tiempos se puede explicar por el hecho de que el procesador va a estar mas centrado en realizar los procesos analizados conforme aumenta el numero de clientes, el hecho de que el programa demande mas poder del procesador hará que este se centre específicamente en estos procesos (probablemente el cache del procesador tenga casi que exclusivamente datos relevantes de los procesos). En la toma de datos evidenciamos que para todos los procesos sin importar la cantidad de clientes concurrentes, el primer calculo de cada proceso (del primer cliente que lo haga) es el mas demorado por un margen bastante amplio (puede llegar a ser 100 veces mas demorado), esto junto a el hecho de que generalmente los tiempos tienden a la baja para el siguiente cliente que lo ejecute (ver tabla de tiempos ampliada en el Excel anexo), son una gran prueba de la hipótesis planteada.

### 3.4. Análisis de la máquina

Velocidad del procesador: 2.6 Ghz 1Hz = 1 ciclo por segundo 1Ghz= 1.000.000.000 de ciclos por segundo 2.6 Ghz= 2.600.000.000 de ciclos por segundo

#### 3.4.1. Para Cifrado de consulta

Algoritmo usado: AES-256bit
Según la fuente consultada (link) para el algoritmo usado con una clave de 256 bytes, le toma al procesaror aproximadamente 6 ciclos de reloj por byte realizar el cifrado.

6 ciclos por byte *256 bytes= 1536 ciclos de reloj para cifrar
2.600.000.000 de ciclos por segundo/1536 ciclos por cifrado = 1.692.708 cifrados por segundo

#### 3.4.2. Para Generar código de autenticación

Algoritmo usada: SHA-512
Según la fuente consultada (link) para el algoritmo usado, le toma al procesador 9 ciclos por byte para realizar el hash.
9 ciclos por byte *512 bytes = 4608 ciclos para generar 2.600.000.000 de ciclos por segundo/4608 ciclos por generado = 564.236 códigos de autenticación generados por segundo

#### 3.4.3. Para verificación firma

Algoritmos usado: SHA-256
Según la fuente consultada (link) para el algoritmo usado, le toma al procesador 9 ciclos por byte para realizar el descifrado //

9 ciclos por byte *256 bytes = 2304 ciclos para generar 2.600.000.000 de ciclos por segundo/2304 ciclos por generado = 1.128.118 verificaciones de firma por segundo

## 4. Instalación y repositorio

Para acceder al repositorio del código fuente diríjase al siguiente link.

Para correr el programa, clone el repositorio y desde IntelliJ abra los archivos `./src/ClientMain.java` y `./src/ServerMain.java` que contienen métodos `main()`. Ejecutarlos, primero el servidor y luego el cliente, dando click en Run.