Juan Camilo Bojaca
201613123

Amilkar Eslava
201611332

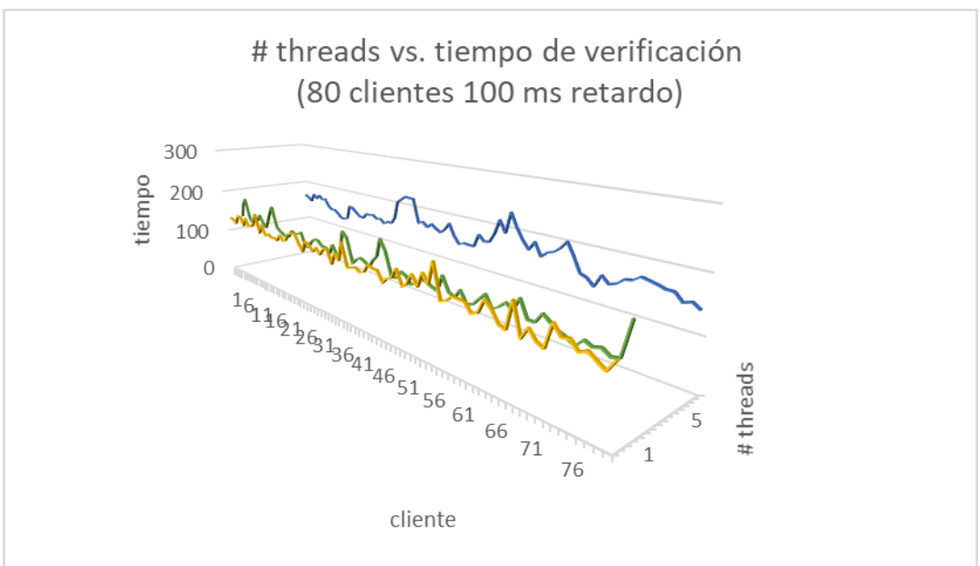
Juan Camilo Sanguino
201617597

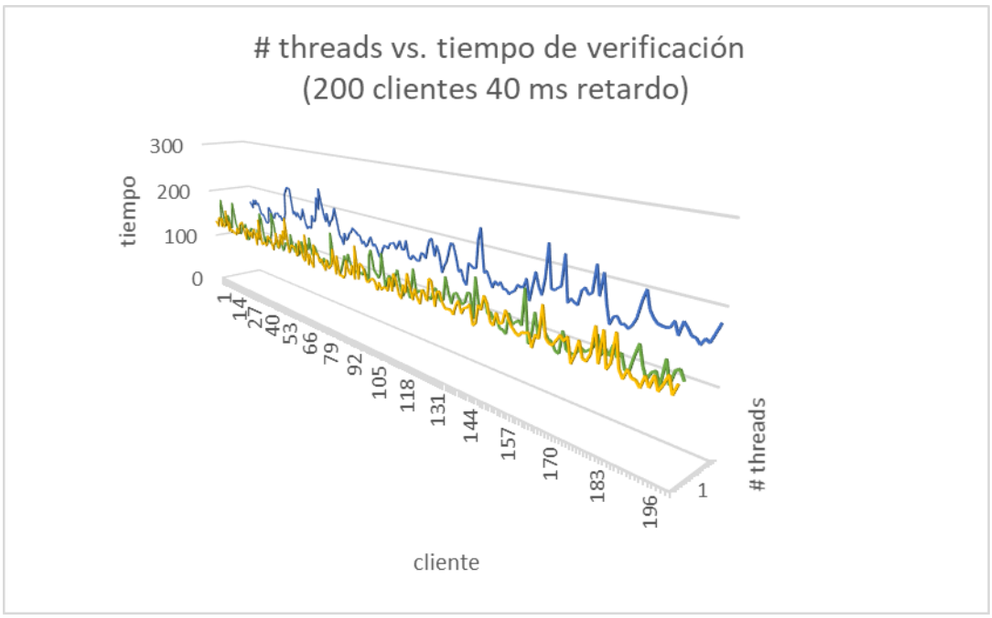
Caso 3

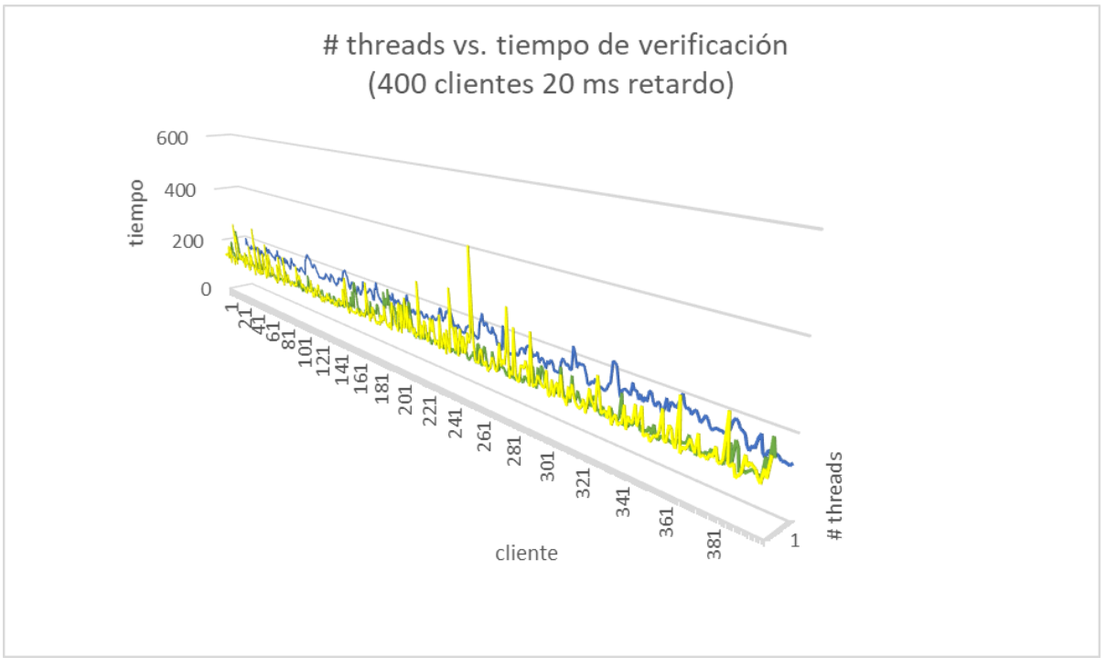
Contents

Analisis Con seguridad

Threads vs verificación





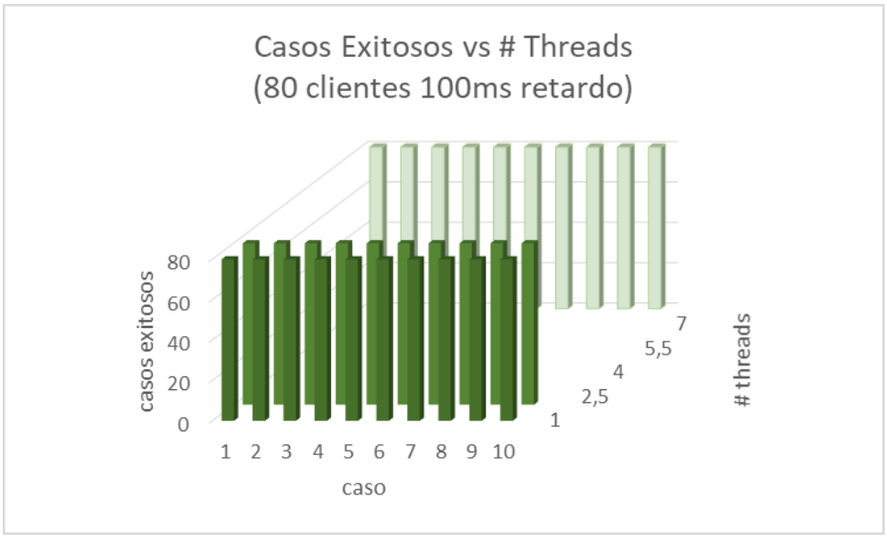


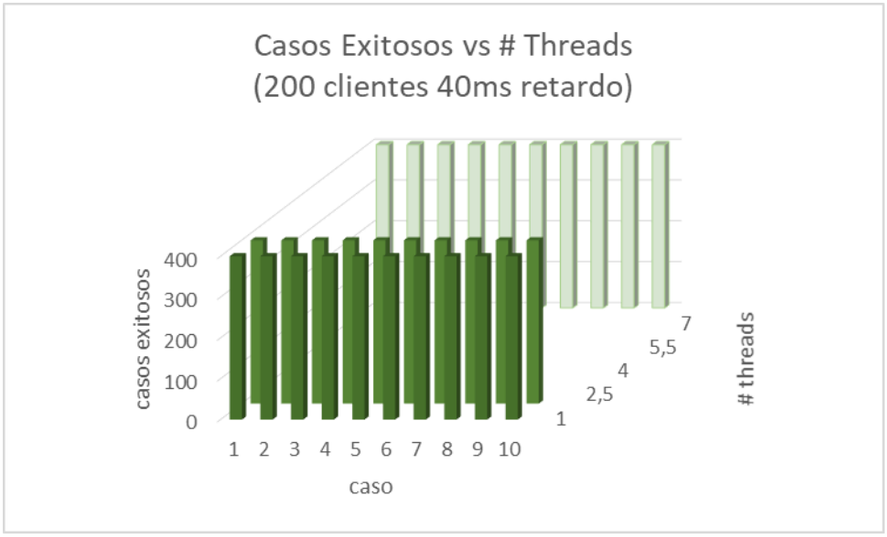
Análisis de las gráficas:

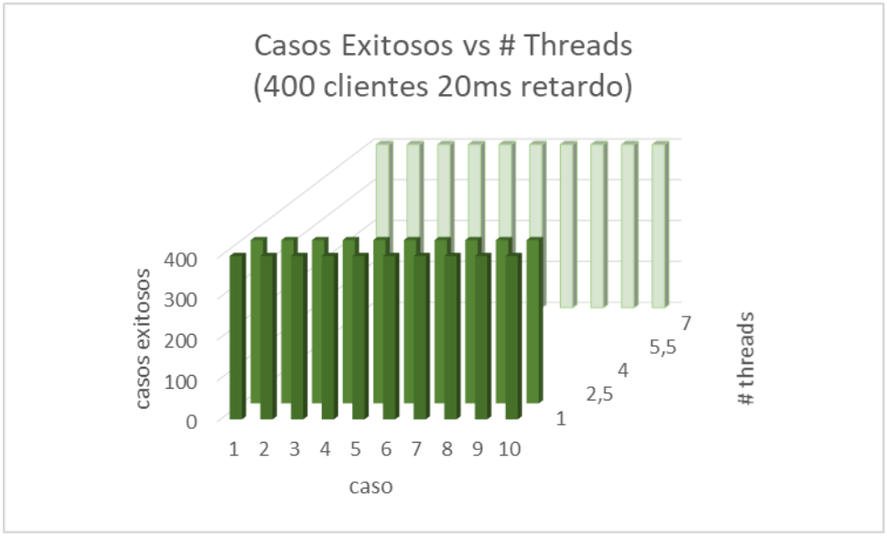
El eje de tiempo (eje y) es el tiempo que tomó en acabar de hacer la verificación en milisegundos. El eje cliente (eje x) es el número del cliente y el eje threads (eje z) es la cantidad de threads que se usaron, la linea verde es un thread, amarilla son dos threads y la azul ocho threads.

Se puede observar como la azul es la que está más arriba siempre, denotando que es la más lenta. Entre la línea amarilla y verde se puede ver que son bastante parecidas, aunque la amarilla tenía más spikes de tiempo con 400 clientes, y la verde con 80 clientes. Pero en general se comportan muy parecido siendo la amarilla un poco más lenta.

Threads vs. transacciones perdidas





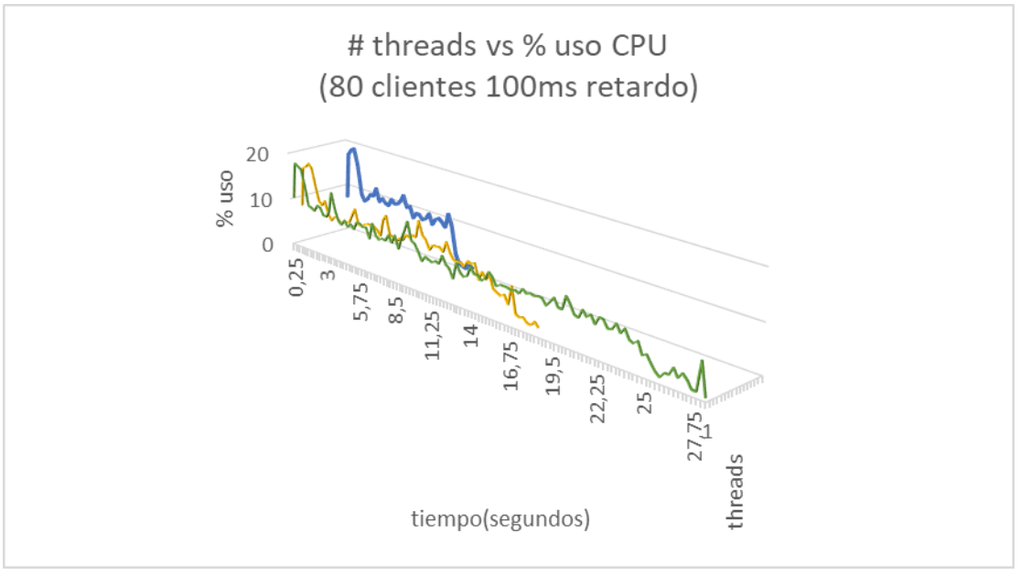


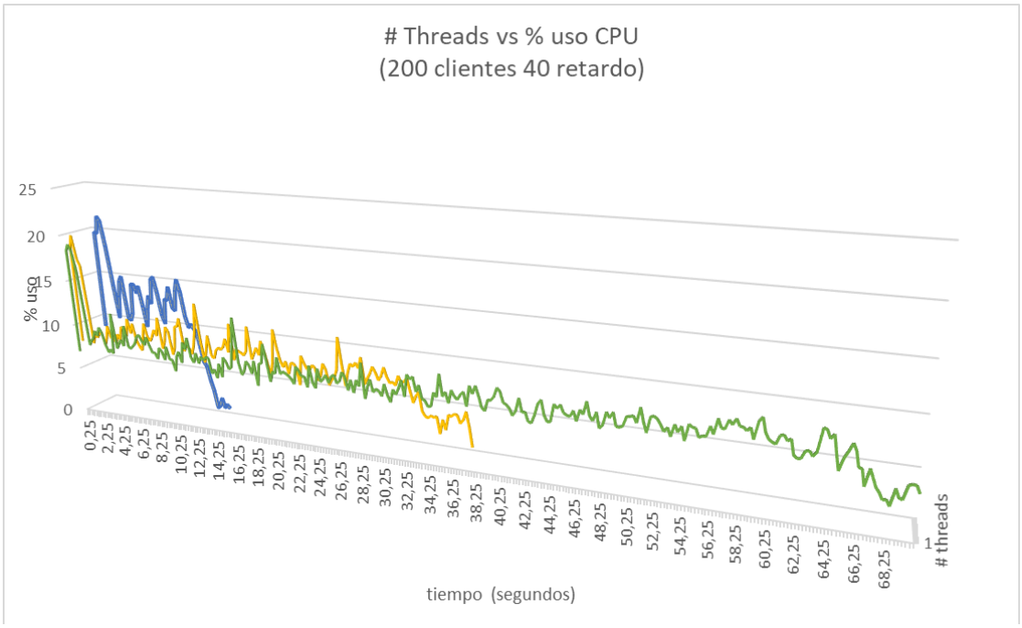
Análisis de las gráficas:

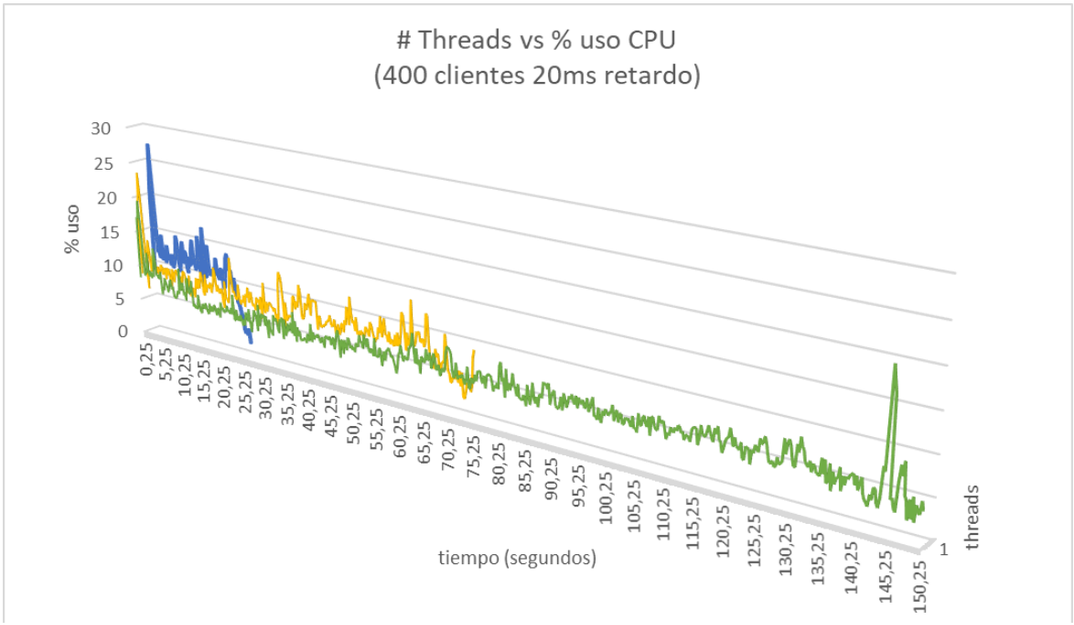
En la gráfica el eje casos exitosos (eje y) representa la cantidad de transacciones exitosas en cada una de las 10 iteraciones (eje x), el eje #threads (eje z) representa los 1, 2 y 8 threads usados.

Se puede observar como todas las transacciones fueron exitosas en las diez iteraciones con los tres posibles números de threads en los tres casos probados.

Threads vs. porcentaje de uso de la CPU





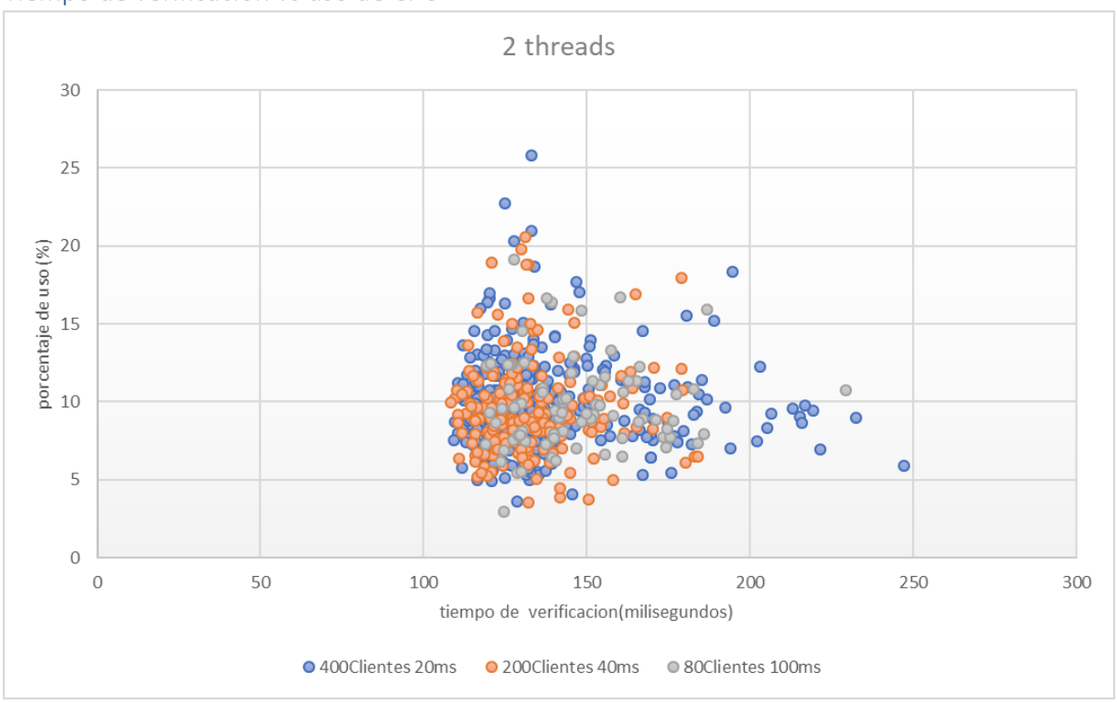


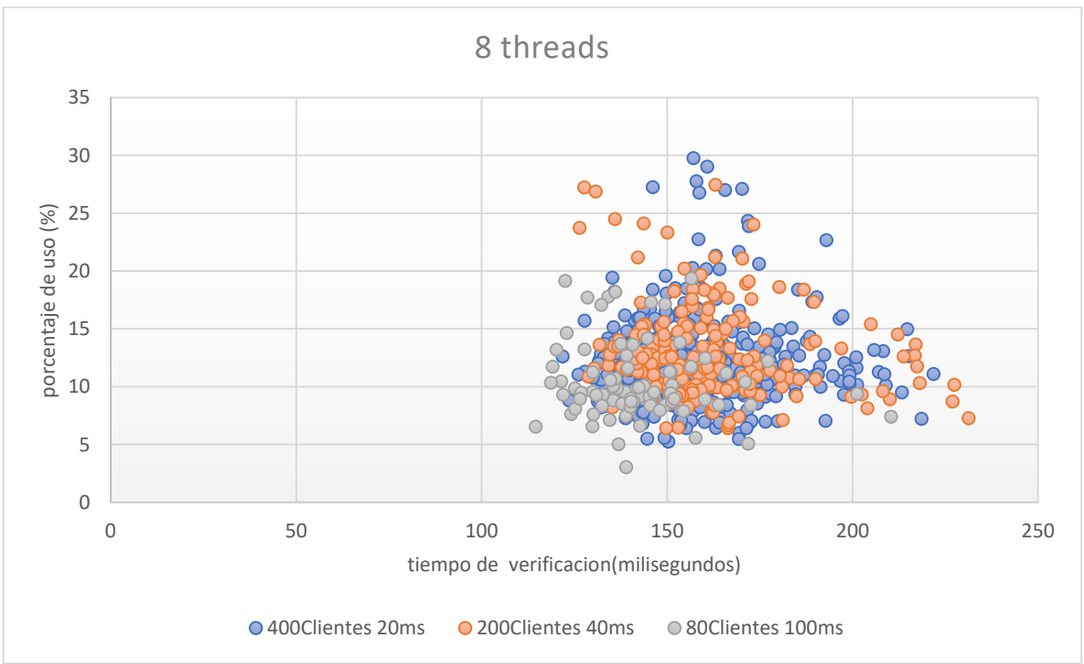
Análisis de las gráficas:

El eje de tiempo (eje y) es el porcentaje de uso del cpu que usó para acabar la transacción. El eje tiempo (eje x) es el tiempo en segundos y el eje threads (eje z) es la cantidad de threads que se usaron, la línea verde es un thread, amarilla son dos threads y la azul ocho threads.

Se puede observar como la línea azul es la que más cpu consume al principio de la ejecución, seguido por el amarillo y luego el verde. A medida que pasa el tiempo, el uso de cpu del azul cae precipitadamente, el verde se mantiene casi constante, y el amarillo, aunque no cae tan rápido como el azul, tampoco va constante como el verde. El tiempo que se demoran en acabar las transacciones también sigue este orden: primero azul, luego amarillo y por último verde. Por ello se puede ver lo eficiente que es usar los ocho threads vs un thread, siendo esta notoria en las tres gráficas.

Tiempo de verificación vs uso de CPU





Para obtener esta graficas se usó el % de uso cpu en el servidor una vez finalizado el protocolo de cada cliente obtenido los datos con scrip2.py

Análisis de las gráficas:

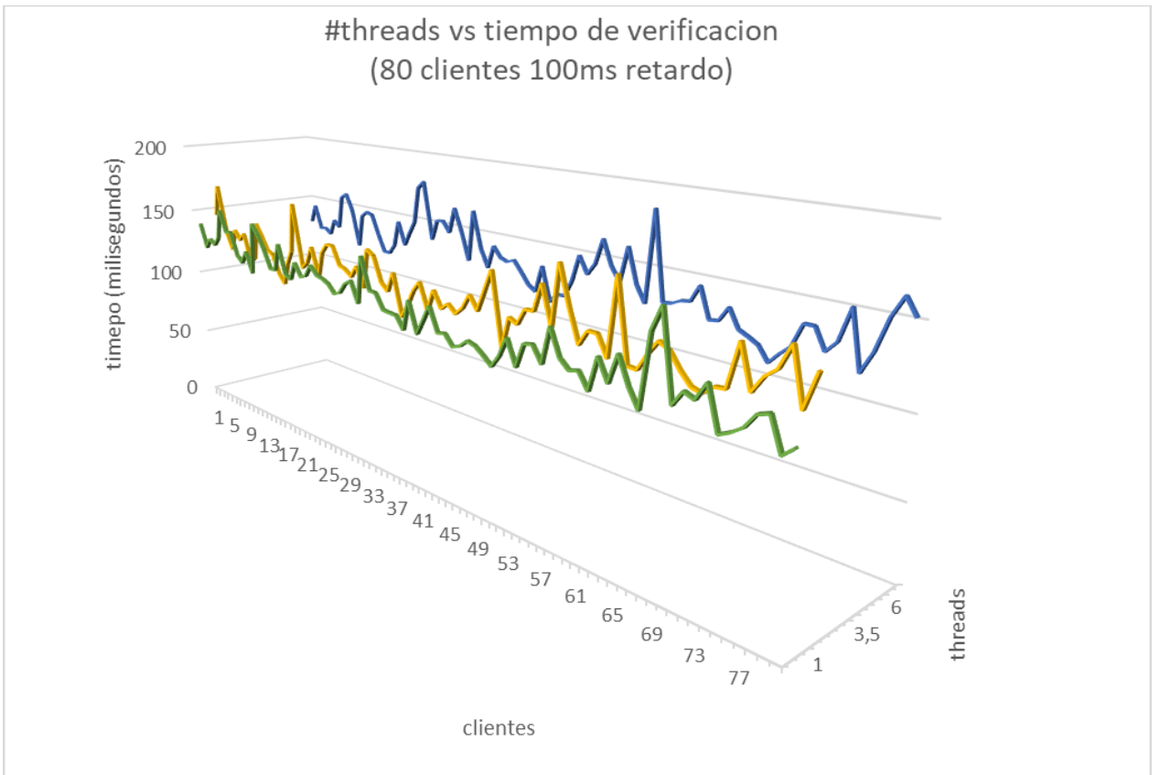
a mayor cantidad de Threads se observa un aumento en el tiempo de verificación al igual que en el uso de cpu. Un hecho a notar es la falta de puntos con alto uso de cpu y tiempo de verificación tendiendo a evitarlas dos características al tiempo. Finalmente, a mayor número de clientes más tendencia a irse a extremos tanto en tiempo de verificación como uso de la cpu se obtiene. Esto, sin embargo, posee una explicación los datos donde hay mayos uso de cpu son datos del inicio de la ejecución donde aún no se exige demasiado al servidor por lo que los tiempos de verificación son cortos. Entre mayor sea el número de clientes más memoria usara en el inicio de le ejecución. Por parte de mayor tiempo de verificación estos debido la sobrecarga más sin embargo no es equiparable al coste para el cpu de atender las solicitudes iniciales.

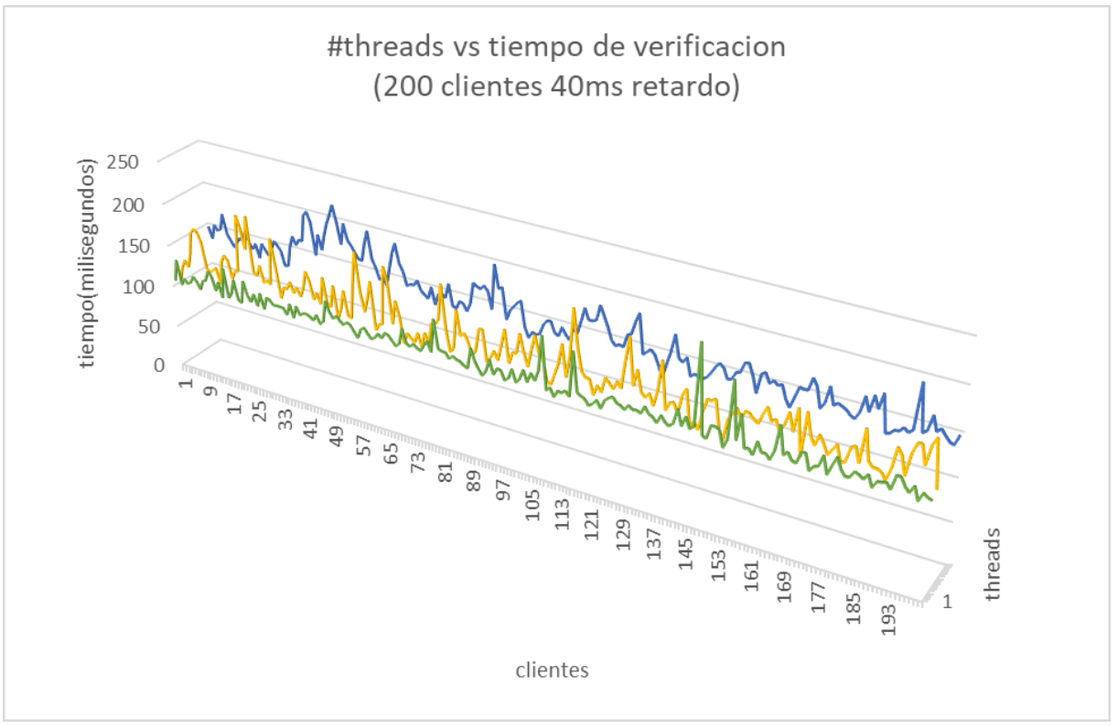
Análisis sin seguridad

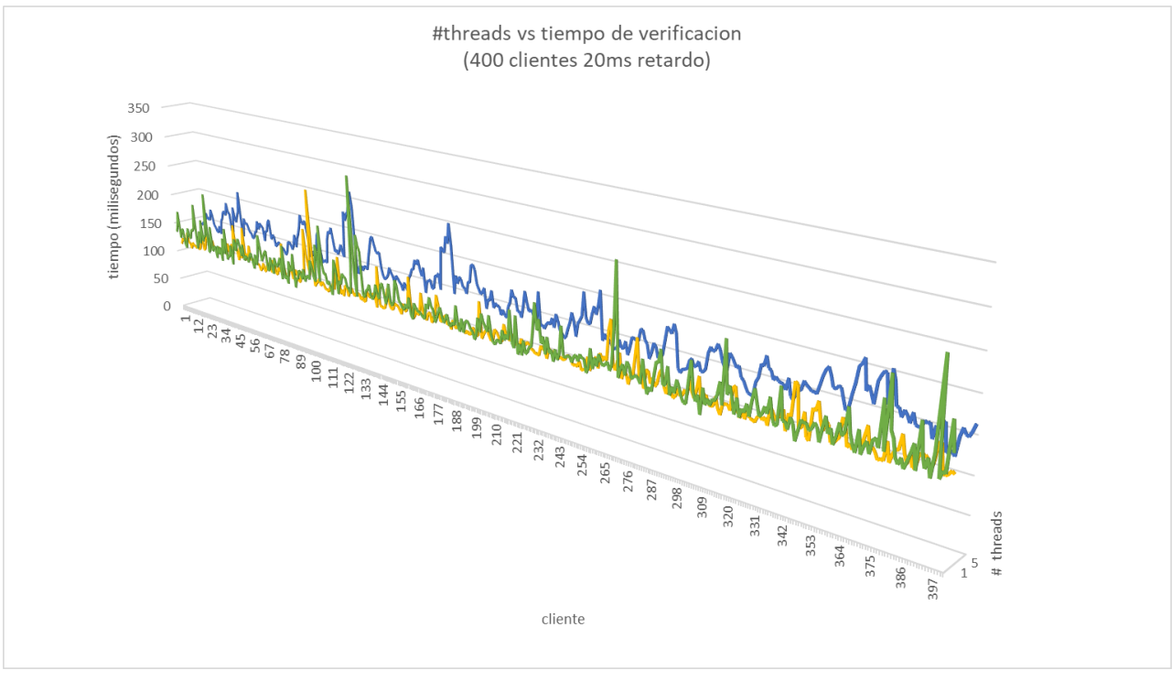
¿Cuál es el resultado esperado sobre el comportamiento de una aplicación que implemente funciones de seguridad vs. una aplicación que no implementa funciones de seguridad?

En términos de tiempo esperamos que la aplicación que no implementa seguridad sea más rápida que una aplicación con seguridad. Esto debido a que la aplicación con seguridad tiene que hacer más procesospara la misma iteración. Y en términos de uso de cpu esperamos que la aplicación sin seguridad use menos porcentaje de cpu ya que tiene que hacer menos procesos.

Threads vs verificación





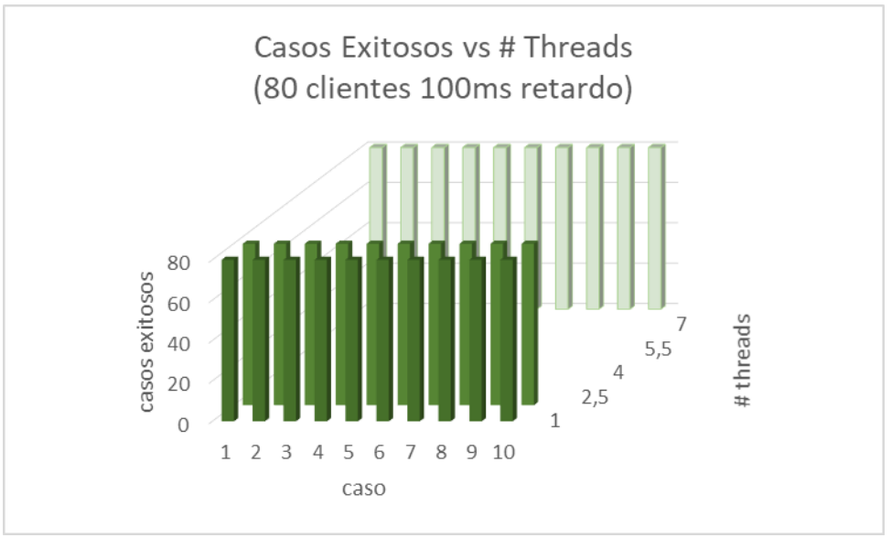


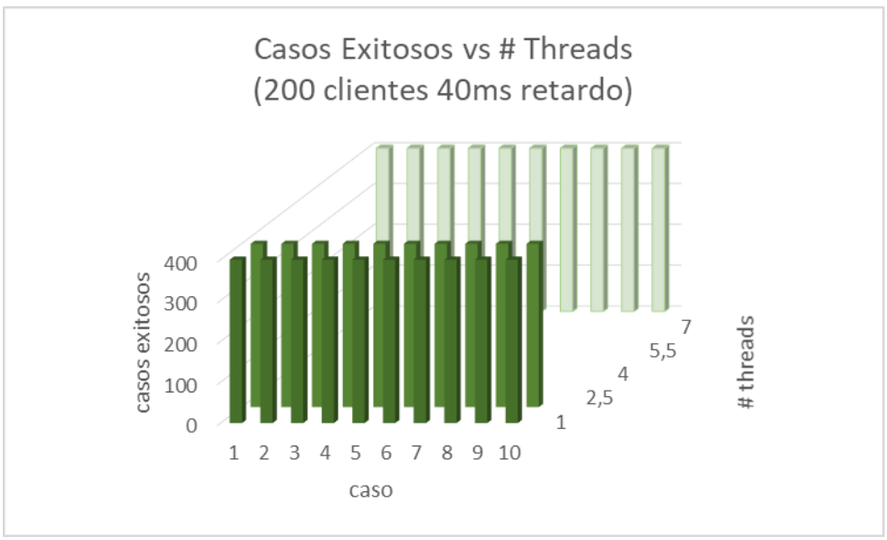
Análisis de las gráficas:

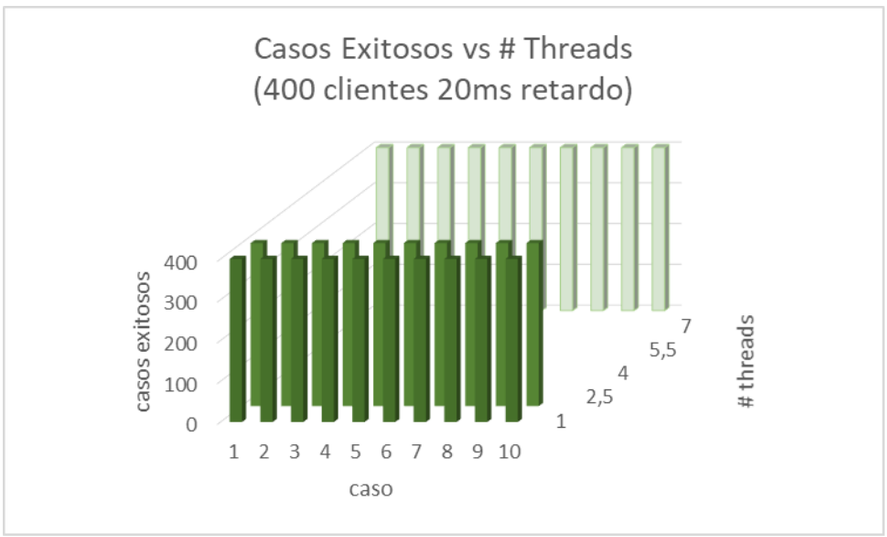
El eje de tiempo (eje y) es el tiempo que tomó en acabar de hacer la verificación en milisegundos. El eje cliente (eje x) es el número del cliente y el eje threads (eje z) es la cantidad de threads que se usaron, la linea verde es un thread, amarilla son dos threads y la azul ocho threads.

Se puede observar un comportamiento similar al de la aplicación con seguridad. La linea azul es la que más tiempo toma para la verificación, la line amarilla y verde estan bastante cerca, aunque a diferencia del servidor con seguridad, con el caso de 80 y 200 clientes se puede ver más claro que la amarilla es más lenta que la verde

Threads vs. transacciones perdidas





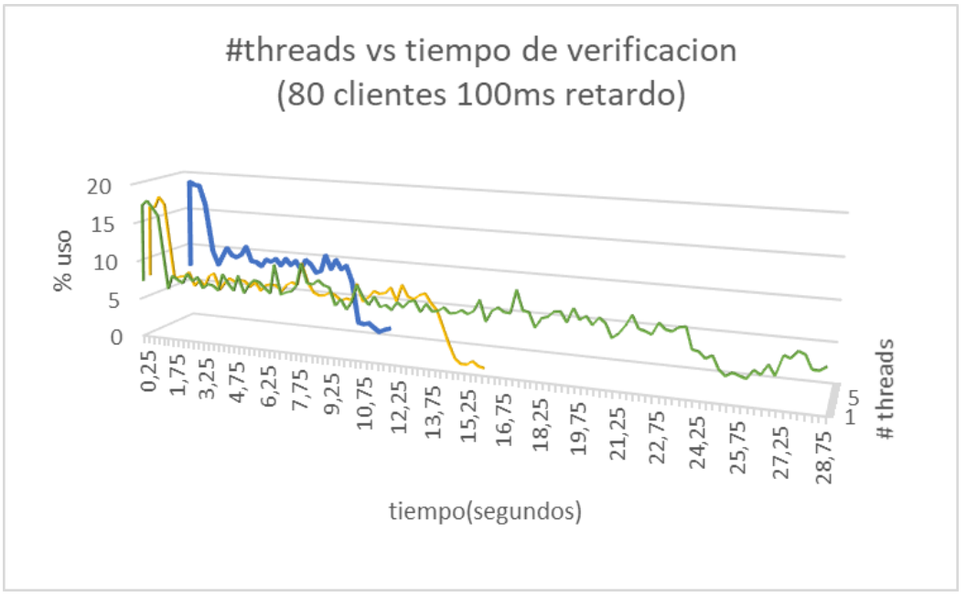


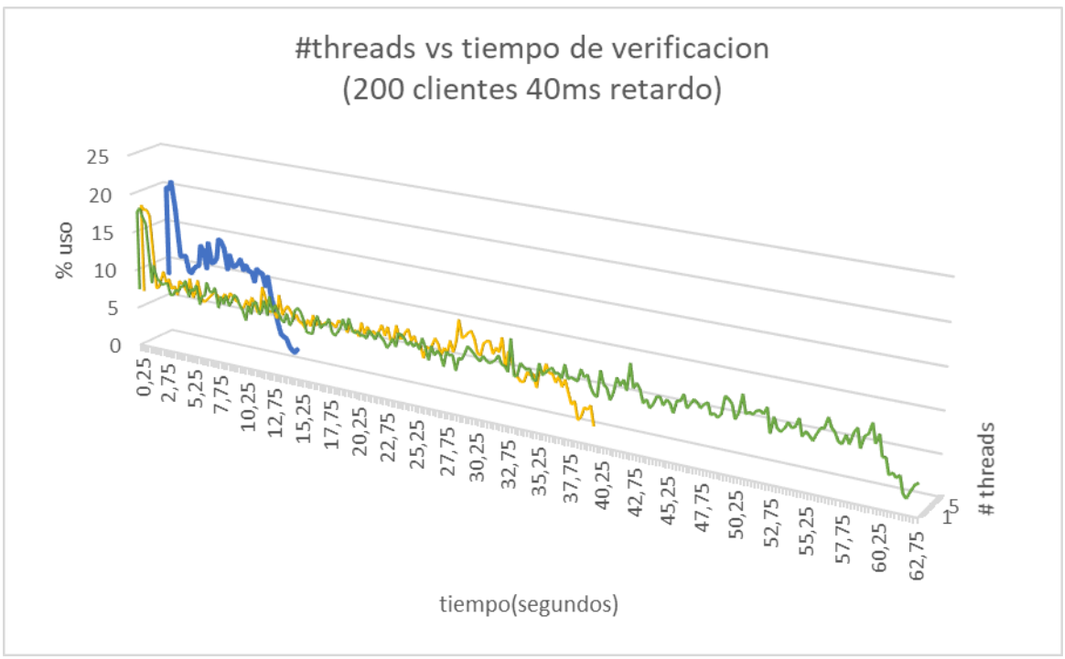
Análisis de las gráficas:

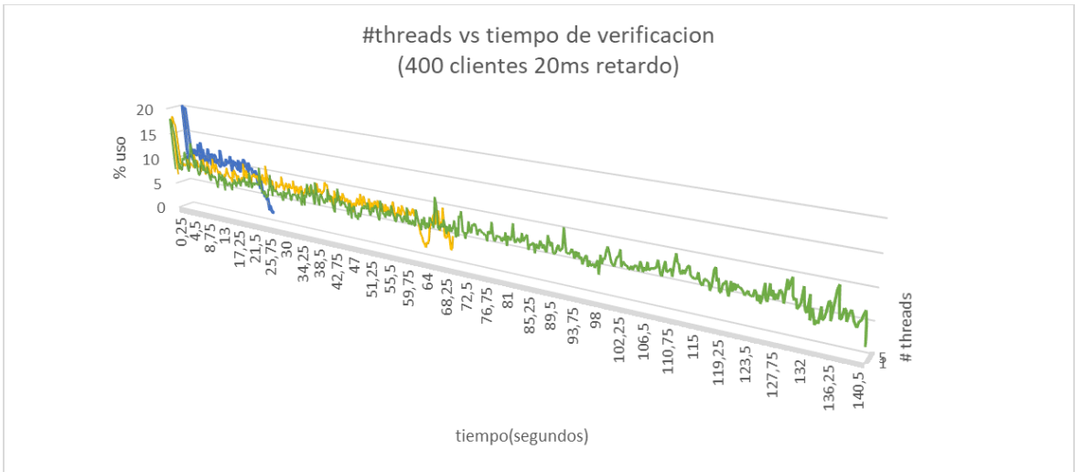
En la gráfica el eje casos exitosos (eje y) representa la cantidad de transacciones exitosas en cada una de las 10 iteraciones (eje x), el eje #threads (eje z) representa los 1, 2 y 8 threads usados.

Se puede observar como todas las transacciones fueron exitosas en las diez iteraciones con los tres posibles números de threads en los tres casos probados.

Threads vs. porcentaje de uso de la CPU







Análisis de las gráficas:

El eje de tiempo (eje y) es el porcentaje de uso del cpu que usó para acabar la transacción. El eje tiempo (eje x) es el tiempo en segundos y el eje threads (eje z) es la cantidad de threads que se usaron, la línea verde es un thread, amarilla son dos threads y la azul ocho threads.

Se puede observar un comportamiento similar al de la aplicación con seguridad. Se mantiene la línea azul como la que más cpu consume al principio de la ejecución, seguido por el amarillo y luego el verde. A medida que pasa el tiempo, el uso de cpu del azul cae precipitadamente, el verde se

mantiene casi constante, y el amarillo, aunque no cae tan rápido como el azul, tampoco va constante como el verde. El tiempo que se demoran en acabar las transacciones también sigue este orden: primero azul, luego amarillo y por último verde. Por ello se puede ver lo eficiente que es usar los ocho threads vs un thread, siendo esta notoria en las tres gráficas.

Conclusiones del análisis:

Esperábamos que la aplicación con seguridad fuera notoriamente más lenta que la aplicación con seguridad. En promedio el tiempo que toma hacer la verificación es de:

Con seguridad

	80 clientes	200 clientes	400 clientes
1 thread	132,8316588	131,8886578	151,0336429
2 threads	186,9257127	133,4672142	140,5130708
8 threads	143,042722	163,8361983	157,241018

Sin seguridad

	80 clientes	200 clientes	400 clientes
1 thread	121,0361987	106,6669995	133,3745701
2 threads	131,0714135	140,3784817	122,2566478
8 threads	136,0412643	153,9780024	155,2066206

Por lo que podemos concluir que la diferencia de hacer la verificación entre seguridad vs sin seguridad es notoria.

Identificacion de la plataforma

atributo	Medida
Arquitectura	64 bits
Numero de núcleos	2
velocidad del procesador	2.20 GHz
Tamaño de la memoria RAM	6 GB
Espacio de memoria asignado a la JVM	2 GB

Modificaciones

A continuación, se explican los cambios sobre el código de servidores y clientes para medir los indicadores solicitados. A sabiendas de que no hay diferencia entre las modificaciones hechas para el cliente con y sin seguridad el código expuesto corresponde a este primero. El mismo caso ocurre para el servidor.

Cliente

El primer cambio que se hizo en el cliente por cuestiones de orden fue mudar todo el código de la carpeta src principal al paquete logic donde se ***eliminó el Main de la clase cliente***. En este punto se cambió el constructor para permitir que la configuración ***entrara como parámetro*** y evitar que fuera sacada de los archivos para cada cliente. Igualmente, con el fin de medir el tiempo de verificación y de consulta se optó por almacenar este mismo en archivos distintos recibiendo el printStream de los archivos de salida y almacenándolos en ***nuevos atributos***.

Para medir el tiempo de verificación se agregaron dos líneas de código en el protocolo una primera medida de tiempo ***luego de enviar la el ok correspondiente a la verificación del certificado.*** Y posteriormente el cálculo del tiempo total en milisegundos hasta recibir el ok de la ***verificación de la llave simétrica***. Para el ultimo se separó el código original ya que la entrada pasaba a directamente a un método de validación de respuesta.

Para medir el tiempo de consulta se agregaron dos líneas de código una primera medida de tiempo ***una vez enviada la consulta*** sin aun enviar el hmac. Y el cálculo de tiempo de la operación la línea siguiente a la llegada de ***la respuesta por parte del servidor***.

antes	después

`public class Cliente {`
`  private Config;`
`  private KeyPair;`
`…`
	`public class Cliente {`
	`  private Config;`
	`  private KeyPair keyPair;`
	`  private PrintStream Verificacion;`
	`  private PrintStream Consulta;`
	`  private PrintStream general;`
	`…`

`public class Cliente {`
`…`
`  public Cliente() throws Exception {`
`  Security.addProvider(new BouncyCastleProvider());`
`  System.out.println("iniciando cliente");`
`  config = new Config();`
`  KeyPairGenerator generator = KeyPairGenerator.getInstance(config.getAlgotimoAsimetrico());`
`  generator.initialize(config.getAsimetricoSize());`
`  keyPair = generator.generateKeyPair();`
`  }`
`…`
	`public class Cliente {`
	`…`
	`  public Cliente(Config config, PrintStream verificacion, PrintStream consulta, PrintStream general)`
	`  throws NoSuchAlgorithmException {`
	`  Verificacion = verificacion;`
	`  Consulta = consulta;`
	`  this.general = general;`
	`  Security.addProvider(new BouncyCastleProvider());`
	`  this.config = config;`
	`  KeyPairGenerator generator = KeyPairGenerator.getInstance(config.getAlgotimoAsimetrico());`
	`  generator.initialize(config.getAsimetricoSize());`
	`  keyPair = generator.generateKeyPair();`
	`  }`
	`…`

`public class Cliente {`
`…`
`  public static void main(String[] args) {`
`  try {`
`  Cliente = new Cliente();`
`  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`
`  System.out.println("escribir un int");`
`  int i = br.read();`
`  System.out.println(cliente.protocolo(i));`
`  } catch (Exception e) {`
`  e.printStackTrace();`
`  }`
`  }`
`…`	

`public class Cliente {`
`  …`
`  public String protocolo(int codigo) throws Exception {`
`  …`
`  canal.send(Const.ok);`
`  // Etapa 4`
`  byte[] llaveEncriptada = DatatypeConverter.parseHexBinary(canal.recive().toString());`
`  Cipher = Cipher.getInstance(config.getAlgotimoAsimetrico());`
`  cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());`
`  byte[] llave = cipher.doFinal(llaveEncriptada);`
`  cipher = Cipher.getInstance(config.getAlgotimoAsimetrico());`
`  cipher.init(Cipher.ENCRYPT_MODE, llaveServidor);`
`  llaveEncriptada = cipher.doFinal(llave);`
`  canal.send(DatatypeConverter.printHexBinary(llaveEncriptada));`
`  respErronea(Const.ok.getValue(), canal.recive());`
`  …`
`  }`
`…`
	`public class Cliente {`
	`  …`
	`  public String protocolo(int codigo) throws Exception {`
	`  …`
	`  canal.send(Const.ok);`
	`  long ini = System.nanoTime();`
	`  // Etapa 4`
	`  byte[] llaveEncriptada = DatatypeConverter.parseHexBinary(canal.recive().toString());`
	`  Cipher = Cipher.getInstance(config.getAlgotimoAsimetrico());`
	`  cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());`
	`  byte[] llave = cipher.doFinal(llaveEncriptada);`
	`  cipher = Cipher.getInstance(config.getAlgotimoAsimetrico());`
	`  cipher.init(Cipher.ENCRYPT_MODE, llaveServidor);`
	`  llaveEncriptada = cipher.doFinal(llave);`
	`  canal.send(DatatypeConverter.printHexBinary(llaveEncriptada));`
	`  CharSequence v = canal.recive();`
	`  Verificacion.println(System.nanoTime() - ini);`
	`  respErronea(Const.ok.getValue(), v);`
	`…`
	`  }`
	`…`

`public class Cliente {`
`  …`
`  public String protocolo(int codigo) throws Exception {`
`  …`
`  canal.send(Const.ok);`
`  // Etapa5`
`  SecretKey SimmetricKey = new SecretKeySpec(llave, 0, llave.length, config.getAlgotimoSimetrico());`
`  cipher = Cipher.getInstance(config.getAlgotimoSimetrico());`
`  cipher.init(Cipher.ENCRYPT_MODE, SimmetricKey);`
`  byte[] mensajeEncriptado = cipher.doFinal(consulta.getBytes());`
`  canal.send(DatatypeConverter.printHexBinary(mensajeEncriptado));`
`  Mac = Mac.getInstance("Hmac" + config.getHmacName());`
`  mac.init(SimmetricKey);`
`  byte[] codigoAutentificacion = mac.doFinal(consulta.getBytes());`
`  canal.send(DatatypeConverter.printHexBinary(codigoAutentificacion));`
`  String[] ans = canal.recive().toString().split(Const.sep.getValue());`
`  respErronea(Const.ok.getValue(), ans[0]);`
`  return ans[1];`
`  …`
`  }`
`…`
	`public class Cliente {`
	`  …`
	`  public String protocolo(int codigo) throws Exception {`
	`  …`
	`  canal.send(Const.ok);`
	`  // Etapa5`
	`  SecretKey SimmetricKey = new SecretKeySpec(llave, 0, llave.length, config.getAlgotimoSimetrico());`
	`  cipher = Cipher.getInstance(config.getAlgotimoSimetrico());`
	`  cipher.init(Cipher.ENCRYPT_MODE, SimmetricKey);`
	`  byte[] mensajeEncriptado = cipher.doFinal(consulta.getBytes());`
	`  canal.send(DatatypeConverter.printHexBinary(mensajeEncriptado));`
	`  ini = System.nanoTime();`
	`  Mac = Mac.getInstance("Hmac" + config.getHmacName());`
	`  mac.init(SimmetricKey);`
	`  byte[] codigoAutentificacion = mac.doFinal(consulta.getBytes());`
	`  canal.send(DatatypeConverter.printHexBinary(codigoAutentificacion));`
	`  String[] ans = canal.recive().toString().split(Const.sep.getValue());`
	`  Consulta.println(System.nanoTime() - ini);`
	`  respErronea(Const.ok.getValue(), ans[0]);`
	`  …`
	`}`
	`…`

Además de las modificaciones ya descritas para lograr la carga necesaria se usó Gload para crear un generador de carga automático.

En primer lugar, se creó la clase caseDescription con el único objetivo de determinar un caso de prueba especifico y sus propiedades:

```
package carga;

public class CaseDescription {
	private int numberOfTask;
	private int gapBetweenTask;
	private int numberOfTest;
	private String Name;

	public CaseDescription(int numberOfTask, int gapBetweenTask, int numberOfTest,
String name) {
```

```
                this.numberOfTask = numberOfTask;
                this.gapBetweenTask = gapBetweenTask;
                this.numberOfTest = numberOfTest;
                Name = name;
        }

}
```

Luego se prosiguió con la creación de clientTask en esta clase se crean clientes y se les pide realizar el protocolo con algún número aleatorio, sin embargo, los más importante es que lleva registro de los clientes que terminan de manera exitosa y los que no. Esto lo hace almacenándolos en variables estáticas de la clase. Para asegurar la integridad de las variables se usa un monitor sobre las mismas.

```
package carga;

import …

public class ClientTask extends Task {
        public static Object obj = new Object();
        public static int bien = 0;
        public static int mal = 0;

        …

        public static int numberOfClients() {
                synchronized (obj) {
                        return mal + bien;
                }
        }

        public static void clear() {
                synchronized (obj) {
                        mal = 0;
                        bien = 0;
                }
        }

        @Override
        public void fail() {
                synchronized (obj) {
                        mal++;
                }
        }

        @Override
        public void success() {
                synchronized (obj) {
                        bien++;
                }
        }

        @Override
        public void execute() {
                try {
                        Cliente cliente = new Cliente(config, fileVerificacion,
fileConsulta, System.out);
                        cliente.protocolo((int) (Math.random() * (int) 100));

                        success();
                } catch (Exception e) {
                        fail();
                }
        }
}
```

Por último, se tiene la clase principal de carga:

lo primero a notar es que se ejecuta el protocolo para un cliente inicial, esto debido a que experimentalmente se notó que este primero demora un tiempo considerablemente mayor que cualquier otro.

```
package carga;

import …

public class Generator {
```

```
        private LoadGenerator generator;
        private Config config = new Config();
        private int numberOfTask;
        private int gapBetweenTask;
        private int numberOfTest;

        public Generator(List<CaseDescription> cases) throws Exception {
                firstClient();
```

Posteriormente se procede a generar la carpeta donde se almacenarán los reportes y se ejecutan los casos de prueba.

```
                DateFormat format = new SimpleDateFormat("yyyy-MM-dd_hh-mm");
                String path = "./Reports/" + format.format(new Date());
                (new File(path)).mkdirs();

                for (CaseDescription des : cases) {
                        numberOfTask = des.getNumberOfTask();
                        numberOfTest = des.getNumberOfTest();
                        gapBetweenTask = des.getGapBetweenTask();

                        String pathCase = path + "/" + des.getName();
                        (new File(pathCase)).mkdirs();
                        runCase(pathCase);
                }
        }
```

En cada ejecución de cado de prueba se generan tres archivos el de tiempo de consulta, el de tiempo de verificación y el de transacciones perdidas. Para cada uno de estos se corren con los mismos parámetros una cierta cantidad de test preestablecida (10 en el caso del taller).

Posteriormente se espera 4 segundos esto con el fin de dar tiempo al garbage colector en el servidor de pasar y liberar la memoria.

```
        private void runCase(String basePath) throws FileNotFoundException,
IOException, InterruptedException {
                String verification = basePath + "/verificacion.txt";
                String consulta = basePath + "/consula.txt";
                String otros = basePath + "/otros.txt";

                try (PrintStream ver = new PrintStream(getFile(verification));
                                PrintStream con = new PrintStream(getFile(consulta));
                                PrintStream otr = new PrintStream(getFile(otros))) {

                        Task work = createTask(config, ver, con);
                        generator = new LoadGenerator(basePath, numberOfTask, work,
gapBetweenTask);

                        for (int i = 0; i < numberOfTest; i++) {
                                System.out.println("-----------CASO " + (i + 1) + "-------
----");
                                otr.println("-----------CASO " + (i + 1) + "-----------");
                                otr.println(System.currentTimeMillis());
                                con.println("-----------CASO " + (i + 1) + "-----------");
                                ver.println("-----------CASO " + (i + 1) + "-----------");
                                runTest(ver, con, otr);
                                System.out.println("----------------------------");
                                Thread.sleep(4000);
                        }

                        con.flush();
                        otr.flush();
                        ver.flush();
                }
        }
```

Cada vez que se corre un test se llama a generate y se realiza espera activa hasta que todos los clientes hayan finalizado.

```
        private void runTest(PrintStream ver, PrintStream con, PrintStream otr) throws
FileNotFoundException, IOException {
                ClientTask.clear();

                generator.generate();

                while (ClientTask.numberOfClients() < numberOfTask) {
                        Thread.yield();
                        try {
```

```
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
                }
                otr.println(ClientTask.bien + ";" + ClientTask.mal);
        }
```

Finalmente se tiene el main donde se configuran los casos y se da inicio a la ejecución

```
        public static void main(String[] args) {
                try {
                        List<CaseDescription> set = new LinkedList<>();
                        set.add(new CaseDescription(20, 100, 10, "20_100"));
                        set.add(new CaseDescription(10, 100, 10, "10_100"));
                        set.add(new CaseDescription(5, 100, 10, "5_100"));
                        new Generator(set);

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
```

Servidor

Para medir el uso del cpu optamos por un log que imprima el mismo con un determinado periodo, a la vez que imprime las entradas y salidas de los clientes. Esto sobre la generación de archivos independientes por caso, ya que en la situación de que un cliente falle al generar archivos independientes se corre con el riesgo de que generar reportes que mezclen información de más de un caso.

Para realizar lo anterior se procedió con modificar la clase delegado. Lo primero fue agregar un printStream a el archivo donde se escribe el log y agregarlo en el constructor. Posteriormente se agregó una impresión al principio y final del método run.

antes

```
public class Delegado implements Runnable {
…
        // Atributos
        private Socket sc = null;
        private String dlg;
        private byte[] mybyte;
…
}
```

```
public class Delegado implements Runnable {
…
        Delegado(Socket csP, int idP) {
                sc = csP;
                dlg = new String("delegado " + idP + ": ");
                try {
                        mybyte = new byte[520];
                        mybyte = Coordinador.certSer.getEncoded();
                } catch (Exception e) {
                        System.out.println("Error creando encoded
del certificado para el thread" + dlg);
                        e.printStackTrace();
                }
        }
…
}
```

```
public class Delegado implements Runnable {
…
        public void run() {
                String linea;
                System.out.println(dlg + "Empezando atencion.");
                try {
…
                        sc.close();
                        System.out.println(dlg + "Termino
exitosamente.");

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
…
}
```

después

```
public class Delegado implements Runnable {
…
        // Atributos
        private Socket sc = null;
        private String dlg;
        private byte[] mybyte;
        private PrintStream file;
…
}
```

```
public class Delegado implements Runnable {
…
        Delegado(Socket csP, int idP, PrintStream file) {
                this.file = file;
                sc = csP;
                dlg = new String("delegado " + idP + ": ");
                try {
                        mybyte = new byte[520];
                        mybyte = Coordinador.certSer.getEncoded();
                } catch (Exception e) {
                        System.out.println("Error creando encoded del
certificado para el thread" + dlg);
                        e.printStackTrace();
                }
        }
…
}
```

```
public class Delegado implements Runnable {
…
        public void run() {
                String linea;
                System.out.println(dlg + "Empezando atencion.");
                file.println("->" + dlg);
                try {
…
                        sc.close();
                        System.out.println(dlg + "Termino
exitosamente.");
                        file.println("<-" + dlg);

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
…
}
```

También se creó la clase para imprimir el uso del cpu en un archivo, a intervalos equidistantes.

```
package logic;

import …

public class CpuLog extends Thread {
        private Boolean fin = false;
        private int delay;
        private PrintStream file;

        public CpuLog(int delay, PrintStream file) {
                this.file = file;
                this.delay = delay;
        }

        @Override
        public void run() {
                try {
                        synchronized (fin) {
```

```
                        while (!fin) {
                                file.println(getSystemCpuLoad());
                                sleep(delay);

                        }
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
    }

    public synchronized void fin() {
        fin = true;
    }

    public static double getSystemCpuLoad() throws Exception {
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName name =
ObjectName.getInstance("java.lang:type=OperatingSystem");
        AttributeList list = mbs.getAttributes(name, new String[] {
"SystemCpuLoad" });
        if (list.isEmpty())
                return Double.NaN;
        Attribute att = (Attribute) list.get(0);
        Double value = (Double) att.getValue();
        // usually takes a couple of seconds before we get real values
        if (value == -1.0)
                return Double.NaN;
        // returns a percentage value with 1 decimal point precision
        return ((int) (value * 1000) / 10.0);
    }

}
```

Para concluir se modificó el coordinador, para crear el printstream e iniciar el log del uso del cpu. Y para agregar el pool de threads.

antes

```
public class Coordinador {

        private static ServerSocket ss;
        private static final String MAESTRO = "MAESTRO: ";
        static java.security.cert.X509Certificate certSer; /* acceso default */
        static KeyPair keyPairServidor; /* acceso default */

        /**
         * @param args
         */
        public static void main(String[] args) throws Exception{
                // TODO Auto-generated method stub

                System.out.println(MAESTRO + "Establezca puerto de
conexion:");
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                int ip = Integer.parseInt(br.readLine());
                System.out.println(MAESTRO + "Empezando servidor maestro en
puerto " + ip);
                // Adiciona la libreria como un proveedor de seguridad.
                // Necesario para crear certificados.
                Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider());
                keyPairServidor = Seg.grsa();
                certSer = Seg.gc(keyPairServidor);

                int idThread = 0;
                // Crea el socket que escucha en el puerto seleccionado.
                ss = new ServerSocket(ip);
                System.out.println(MAESTRO + "Socket creado.");

                while (true) {
                        try {
                                Socket sc = ss.accept();
                                System.out.println(MAESTRO +
"Cliente " + idThread + " aceptado.");
                                Delegado3 d3 = new
Delegado3(sc,idThread);
                                idThread++;
                                d3.start();
                        } catch (IOException e) {
                                System.out.println(MAESTRO +
"Error creando el socket cliente.");
                                e.printStackTrace();
                        }
                }
        }
}
```

después

```
public class Coordinador {

        private static ServerSocket ss;
        private static final String MAESTRO = "MAESTRO: ";
        static java.security.cert.X509Certificate certSer; /* acceso default */
        static KeyPair keyPairServidor; /* acceso default */

        /**
         * @param args
         */
        public static void main(String[] args) throws Exception {
                int numeroThreads = 4;
                int delay = 50;

                System.out.println(MAESTRO + "Establezca puerto de
conexion:");
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr);
                int ip = Integer.parseInt(br.readLine());
                System.out.println(MAESTRO + "Empezando servidor maestro en
puerto " + ip);

                // Adiciona la libreria como un proveedor de seguridad.
                // Necesario para crear certificados.
                Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider());
                keyPairServidor = Seg.grsa();
                certSer = Seg.gc(keyPairServidor);

                int idThread = 0;
                // Crea el socket que escucha en el puerto seleccionado.
                ss = new ServerSocket(ip);
                System.out.println(MAESTRO + "Socket creado.");
                ExecutorService executor =
Executors.newFixedThreadPool(numeroThreads);

                DateFormat format = new SimpleDateFormat("yyyy-MM-dd_hh-
mm");
                String path = "./Reports/" + format.format(new Date()) +
".txt";

                try (PrintStream pr = new PrintStream(getFile(path))) {
                        CpuLog cpu = new CpuLog(delay, pr);
                        cpu.start();
                        while (true) {
                                try {
                                        Socket sc =
ss.accept();

        System.out.println(MAESTRO + "Cliente " + idThread + " aceptado.");
                                        Delegado d3 = new
Delegado(sc, idThread, pr);
                                        idThread++;
                                        executor.execute(d3);
                                        pr.flush();
                                } catch (IOException e) {

        System.out.println(MAESTRO + "Error creando el socket cliente.");
                                        e.printStackTrace();
                                }
                        }
                }
        }

        public static File getFile(String path) throws IOException {
                File file = new File(path);
                if (!file.exists())
                        file.createNewFile();
                return file;
        }
}
```

En el caso específico de nuestras ejecuciones no se presentaron clientes que fallaran por lo que para procesar el archivo proveniente del servidor se decidió usar un script

```
def load(casos, path):
    with open(path, 'r') as f:
        subs, i = [], 0
        for x in casos:
            i += 1
            subs.append(["test {:d}".format(i)])
            i %= 10


        cliente, cont, indice, estado = 1, 0, 0, 0
        for x in f.readlines():
            if estado == 0:
                if x == ("->delegado {:d}: \n".format(cliente)):
                    cliente += casos[indice] - 1
                    indice += 1
                    estado = 1
            elif estado == 1:
                if x.startswith('<') or x.startswith('-'):
                    if x == ("<-delegado {:d}: \n".format(cliente)):
                        estado = 2
                else:
                    subs[indice - 1].append(x[:-1].replace('.', ','))
            else:
                if cont < 40:
                    subs[indice - 1].append(x[:-1].replace('.', ','))
                else:
                    cliente += 1
                    estado, cont = 0, 0
                cont += 1
    return subs




def print(casos, subs, path):
    with open(path, 'w') as w:
        w.write(';'.join(list(map(lambda i: str(i), casos))) + "\n")
        sale, line = False, 0
        while not sale:
            t, sale = [], True
            for i in range(len(casos)):
                if line < len(subs[i]):
                    t.append(subs[i][line])
                    sale &= False
                else:
                    t.append(' ')
            w.write(';'.join(t) + "\n")
            line += 1


x = 10
casos = [400]*x + [200]*x + [80]*x
mat = load(casos, "seguridad/1ThreadServidor.txt")
print(casos, mat, 'seguridad/1ThreadServidor.csv')
```