LABORATORIO 2

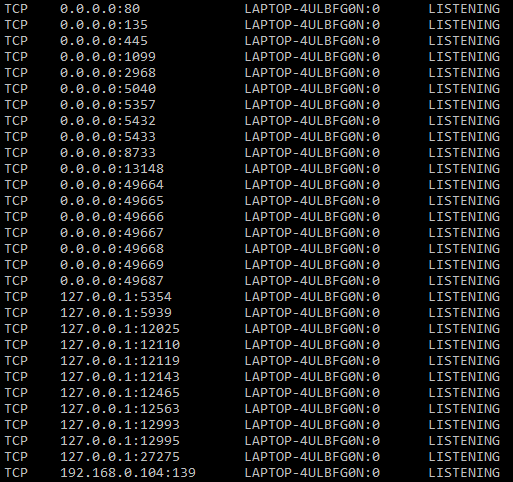
SISTEMAS DISTRIBUIDOS 2020

Barja, María Marta

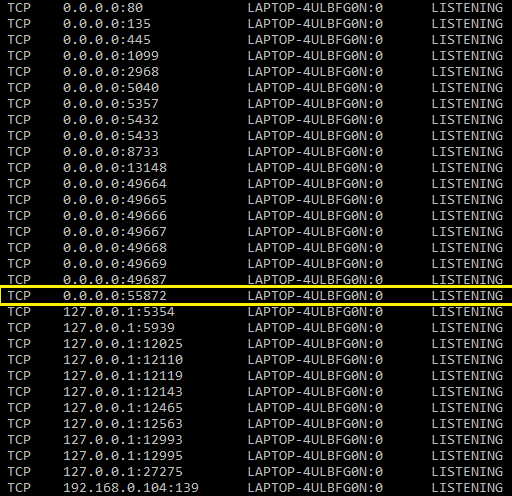
1. Se implementaron las interfaces remotas InterfaceRemota que define los métodos suma y resta, e InterfaceRemotaMD que define multiplicación y división. Luego se implementaron las clases ObjetoRemoto y ObjetoRemotoMD que extienden de UnicastRemoteObject e implementan las interfaces remotas definidas. Luego en el servidor se instancian los 2 objetos remotos y se publican en la rmiregistry.

En el cliente se obtienen las referencias de los objetos remotos y se guardan en 2 variables. Luego se implementaron los 4 métodos solicitados (suma, resta, multiplicación y división) llamando a los métodos remotos de los 2 objetos. Por último, en el método Main se solicita al cliente que ingrese una operación y luego 2 números, según la operación ingresada ejecutará el método correspondiente.

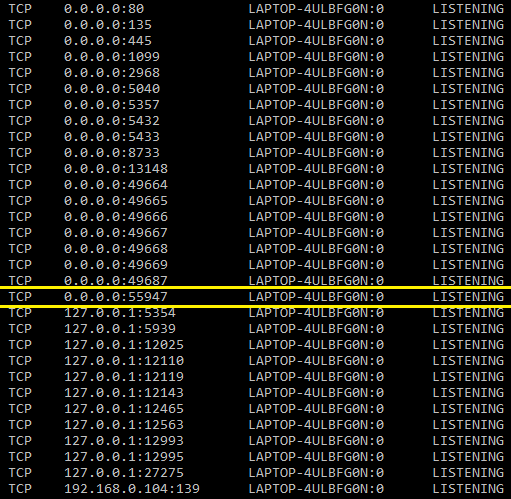
1. Se ejecutó el comando netstat -a | find "LISTENING" antes de lanzar y se obtuvo el siguiente resultado:



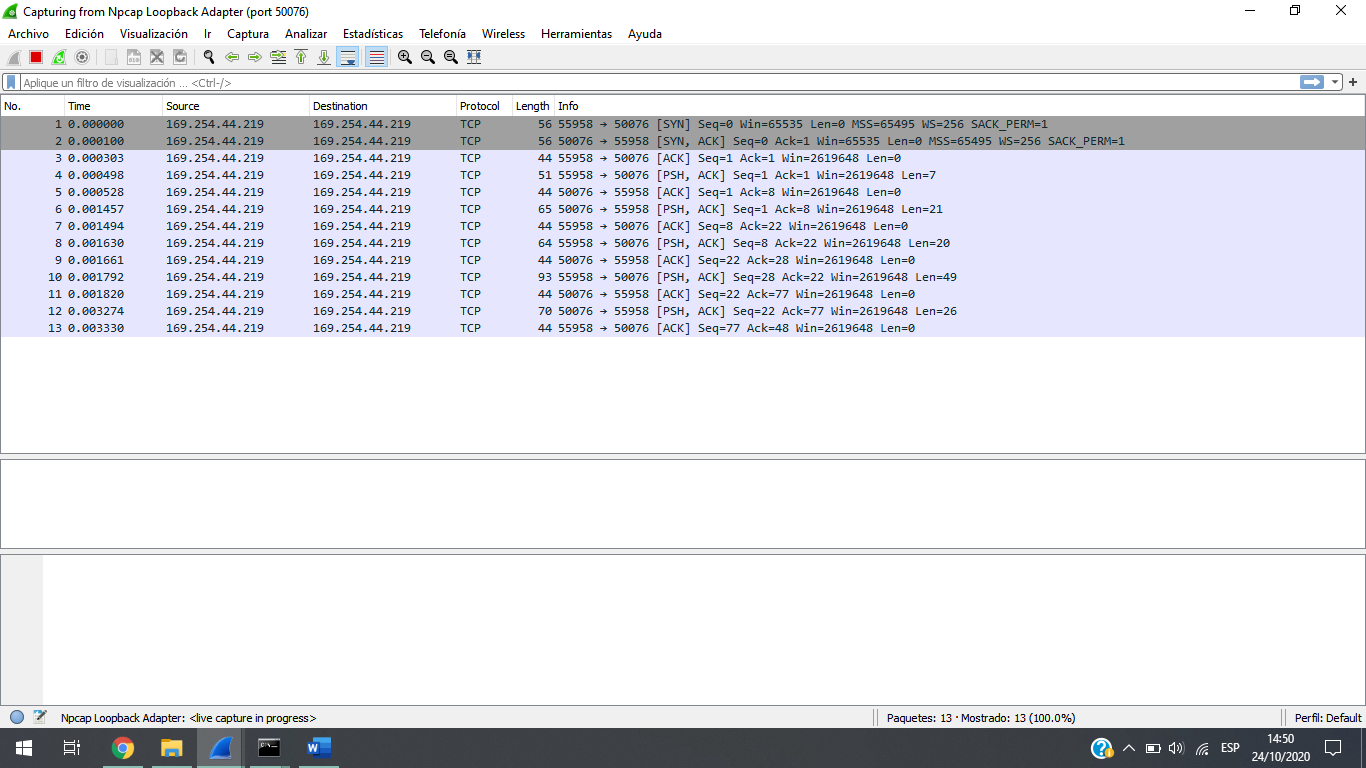
Luego de lanzar el servidor se ejecutó nuevamente el comando y se observa que el puerto del servidor remoto es el 55872



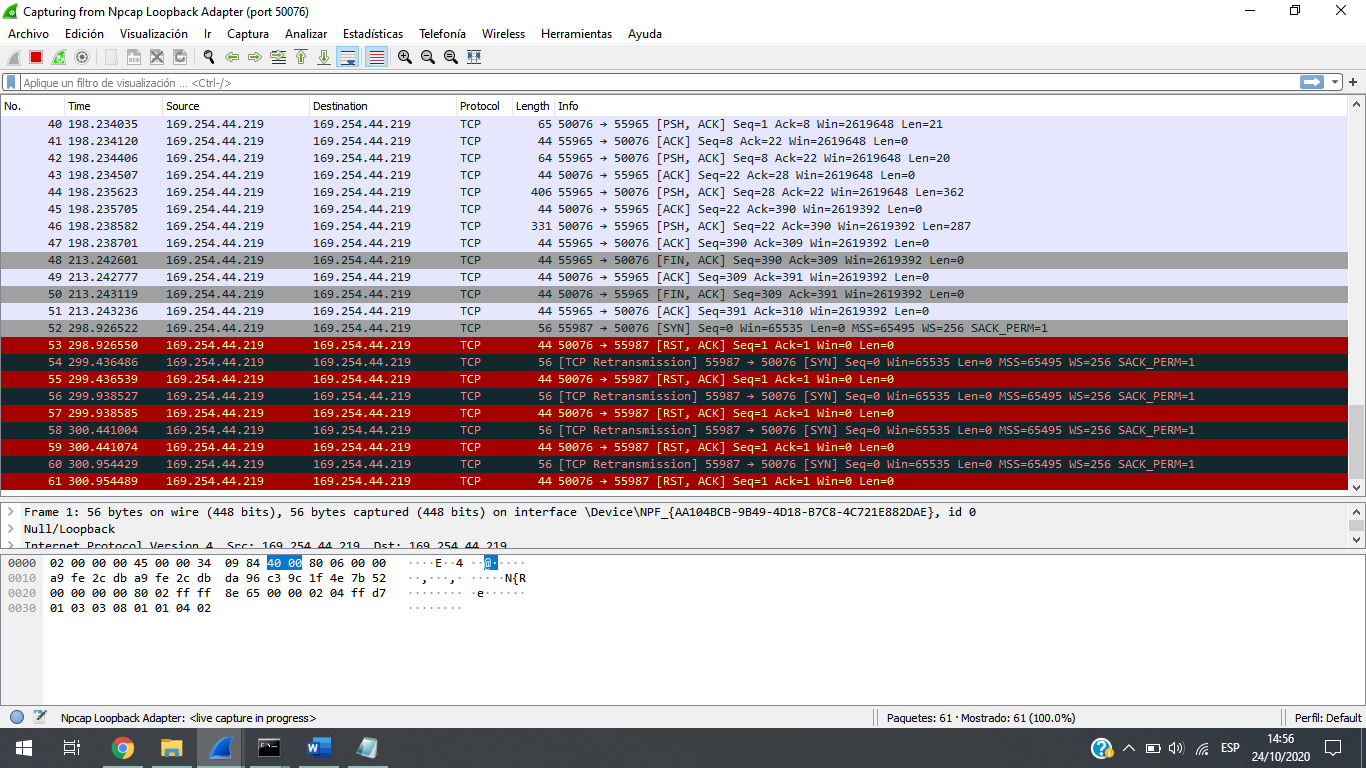
Luego se detuvo el servidor y se lanzo nuevamente para verificar si el puerto en el que escucha cambio. Se ejecuto el comando nuevamente y se observa que el puerto del servidor es el 55947



1. Como se puede observar en las tablas del punto anterior, en el cuarto lugar se encuentra el RMIRegistry que atiende en el puerto 1099.
2. Se aplica un filtro sobre el puerto 50076 que es el puerto del servidor, y al realizar una suma desde el cliente se identificaron los siguientes mensajes:



Luego se detuvo la ejecución del servidor y se intento realizar otra suma desde el cliente. Estos fueron los paquetes identificados.



1. Como se observa en la imagen anterior, los mensajes se valen de TCP
2. Servidores concurrentes
3. Para el correcto funcionamiento de un servidor concurrente, se deben poder gestionar por separado todas las peticiones de los clientes, por cada vez que se conecta un nuevo cliente al servidor, y además el servidor debe seguir esperando las peticiones de otros clientes. Para resolver esto normalmente se abre un socket y se inicia un bucle infinito en el cual, cada vez que se recibe una conexión entrante de un cliente a través del socket, se instancia un hilo que se encarga de procesar las peticiones de ese cliente. Luego el servidor continúa escuchando y repite este procedimiento por cada cliente que inicia una conexión.

Además, se deben establecer bloqueos o semáforos para garantizar la consistencia cuando varios clientes acceden a recursos compartidos.

Java RMI se encarga automáticamente de desplegar los threads requeridos para dotar de concurrencia a un servicio implementado usando esta tecnología. Y respecto del segundo problema mencionado, Java nos permite la sincronización de eventos sobre métodos o bloques. Esto se verificó en el punto 3 c.

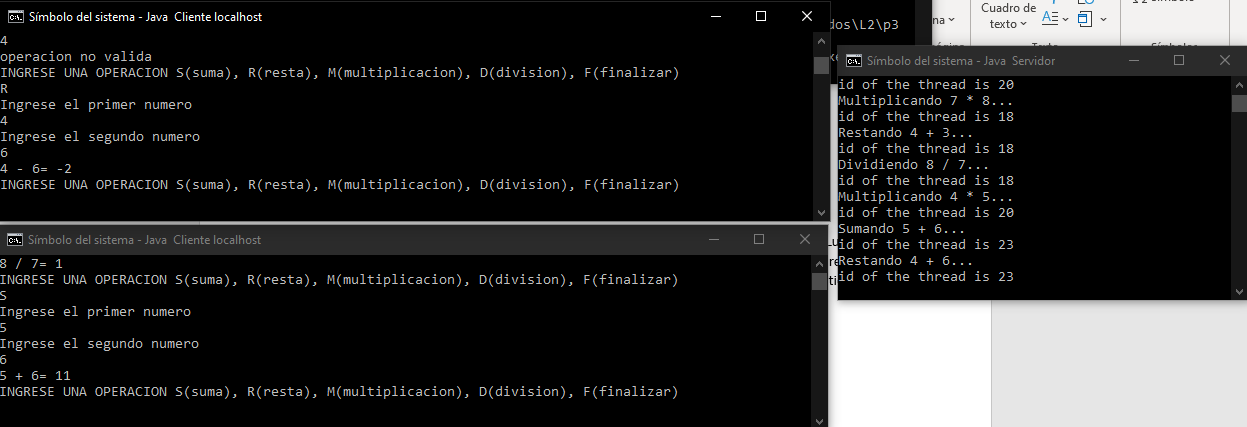
1. Tanto RPC como RMI ofrecen interfaces que definen la funcionalidad remota para el empleo de los clientes (interfaz de servicio en RPC e interfaz remota para RMI), podría decirse que cuentan con el mismo nivel de abstracción ya que la implementación de la funcionalidad remota es transparente para los clientes en ambos casos. Podría agregarse también que hay una mayor abstracción con RMI, debido a la explotación de objetos, referencias, herencia, polimorfismo y excepciones, ya que la tecnología está integrada en el lenguaje.
2. Similitudes entre RPC y RMI:

* Ambos soportan programación con interfases, con los beneficios que surgen de este enfoque
* Ambos se construyen sobre protocolos de petición/respuesta
* Ambos ofrecen un nivel similar de transparencia, las llamadas locales y remotas emplean la misma sintaxis pero la interfaz remota típicamente expone la naturaleza distribuida de la llamada subyacente, por ejemplo soportando excepciones remotas

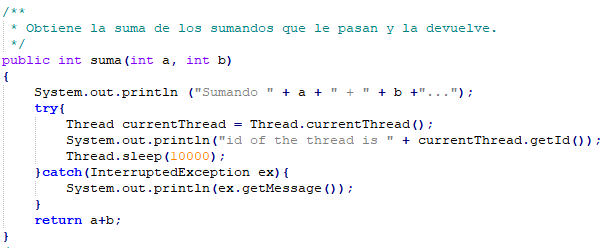
Diferencias entre RPC y RMI:

* RPC solo admite programación estructurada mientras que RMI admite programación orientada a objetos.
* Los parámetros pasados ​​a la llamada a procedimientos remotos consisten en estructuras de datos ordinarias. Por otro lado, los parámetros pasados ​​al método remoto consisten en objetos.

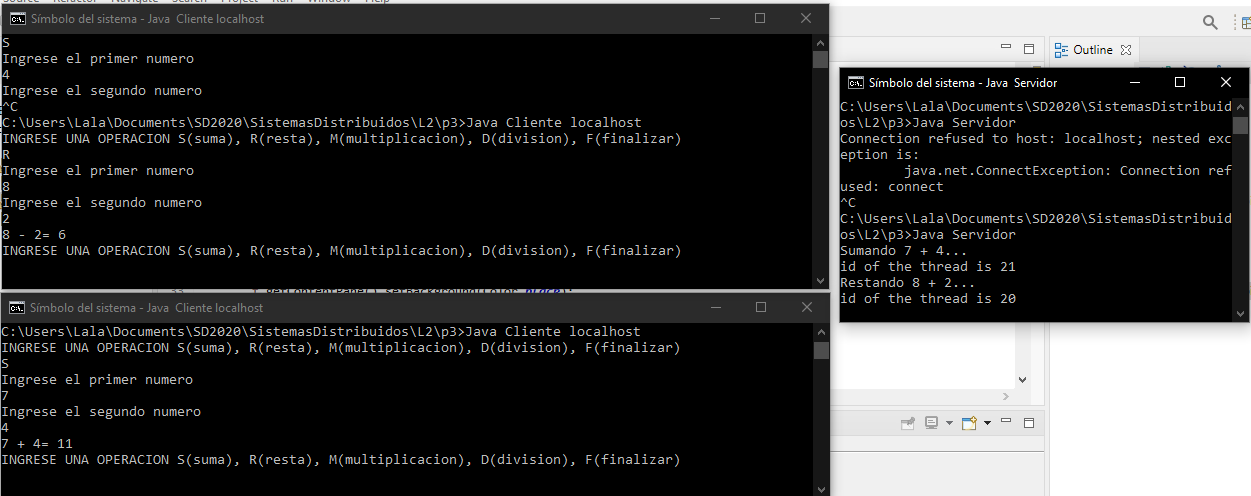
1. Se agregó un print del id del hilo a los métodos de los objetos concurrentes. Luego se lanzaron 2 clientes y se observó que los id de los hilos que ejecutaron los métodos remotos pueden ser diferentes para un mismo cliente. Por ello se concluye que Java RMI tiene un thread por requerimiento.



3.

1. Se agrego un delay de 10 segundos a la operación suma. Luego se lanzaron 2 clientes, en uno se realizó una suma y en el otro una resta, y se verificó que la resta se realiza automáticamente.

En la consola donde se ejecuta el servidor se observa que se imprimen ambos mensajes previos a la ejecución de los métodos.



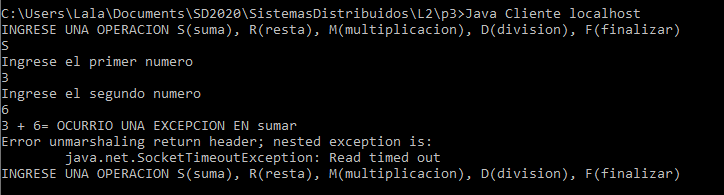
1. En el cliente, se establecieron las siguientes propiedades

System.setProperty("sun.rmi.transport.tcp.responseTimeout", "4000");

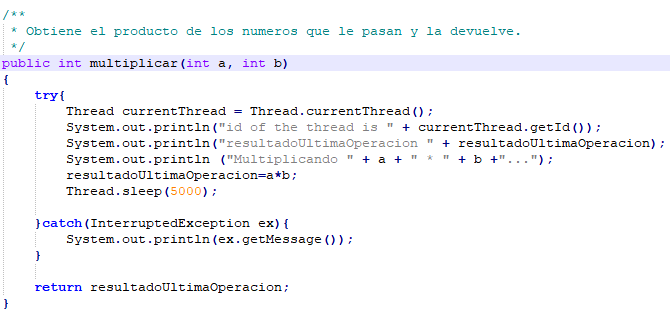
System.setProperty("sun.rmi.transport.connectionTimeout", "4000");

System.setProperty("sun.rmi.transport.tcp.readTimeout", "4000");

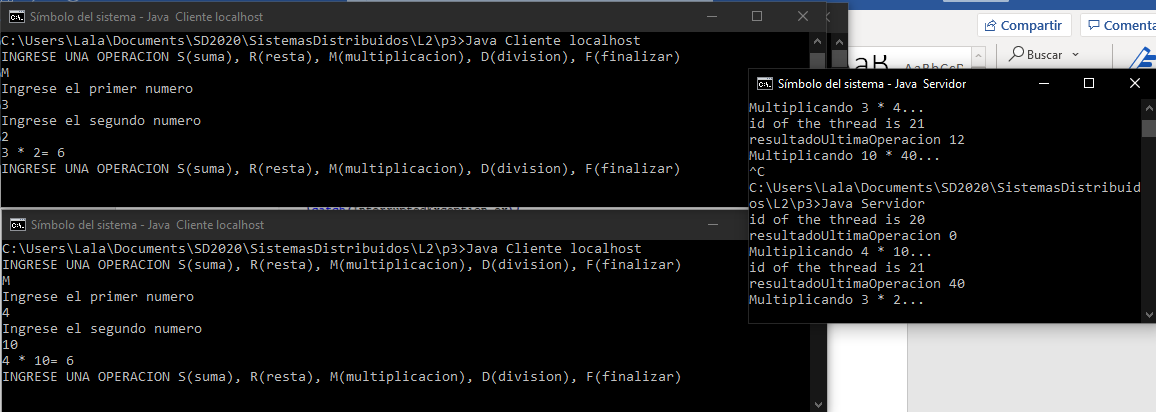
Al ejecutar una operación de suma, teniendo en cuenta que tenían definido un delay de 10 segundos, se observa este mensaje:



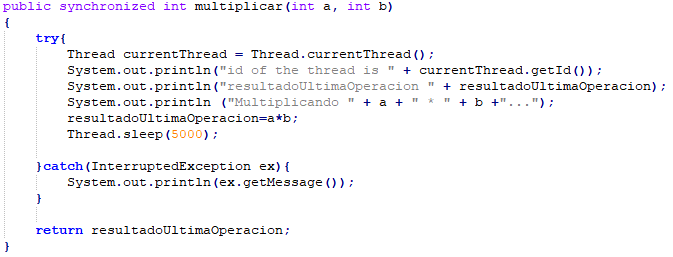
1. Se agregó una variable en el objeto remoto de multiplicación y división a la que se le asigna el resultado de la multiplicación, luego un delay de 5 segundos y finalmente se devuelve el valor de esa variable.

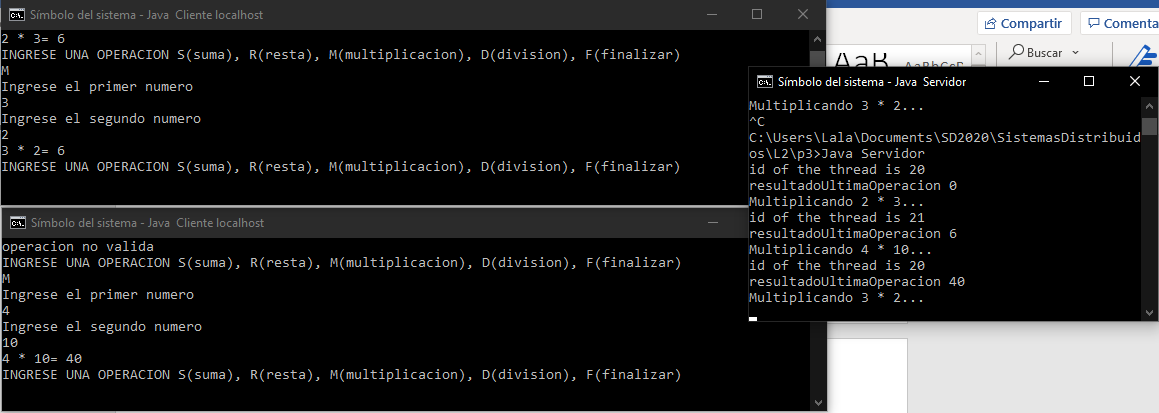


Se lanzaron 2 multiplicaciones en 2 clientes distintos y se observa que el valor entregado en la primer multiplicación (4\*10) es incorrecto, corresponde a la segunda multiplicación. Por ende, se verifica que las variables de estado de los objetos si pueden ser contaminadas.



Luego, se definió el método multiplicador como ***synchonized*** para bloquear su ejecución entre los distintos hilos y se volvió a realizar la misma prueba que arrojó resultados correctos.



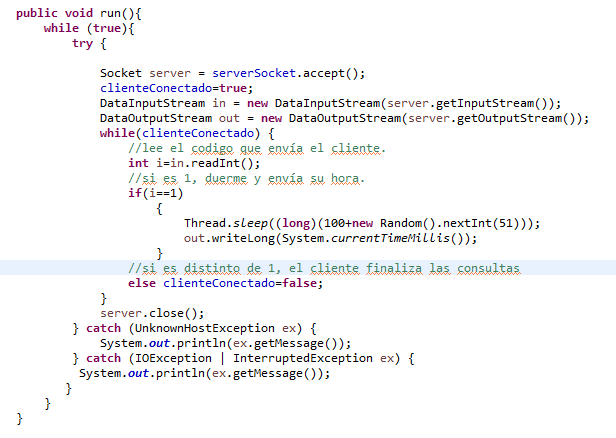


1. Para este ejercicio se implementaron las siguientes clases:

* Servidor.java

Esta clase extiende de Thread. Su constructor recibirá un número de puerto y creará un socket. En su método main instancia un objeto de Servidor con el número de puerto(1333), y lo inicia.

En el método run, dentro de un bucle infinito, que acepta y procesa las solicitudes de los clientes. Cuando un cliente envía una solicitud, setea una variable clienteActivo en verdadero, para que atienda todas las solicitudes del cliente hasta que el mismo le indique que se desconecta. El cliente le envía un entero con un código, 1 para solicitar la hora y 2 para indicarle que finalizó su consulta. Luego, dentro del bloque que esta procesando las solicitudes de hora del cliente, se agregó un sleep con un número random para simular diferentes retardos del servidor, ya que al estar cliente y servidor en la misma maquina los retardos eran siempre iguales y casi nulos. Para resumir, cuando se conecta un cliente, el servidor leerá el código que le envía el mismo, si es 1, hace un sleep y luego envía la hora. Así sucesivamente hasta que el cliente envíe un código distinto de 1.



* MuestraRelojServidor.java

Esta clase es para guardar las muestras que se obtienen de cada consulta que se realiza al servidor, cada instancia tendrá la hora del cliente, el retardo y la hora del servidor.

* Reloj.java

Esta clase representa a la interfaz del reloj, se crea en su constructor, y luego tiene un método actualizar hora que recibe un valor en milisegundos y actualiza la hora en pantalla.

* Cliente.java

Esta clase tiene definidas 2 clases estáticas: RelojDigital, que será el hilo que represente el reloj del cliente, y además instancia la interfaz de reloj y la actualiza en cada incremento de la hora. Y la clase ActualizarHora que se encarga de obtener las muestras del servidor, obtener la de menor retardo y analizar en base a esta muestra y la hora del cliente, si es necesario agrandar o achicar la deriva.

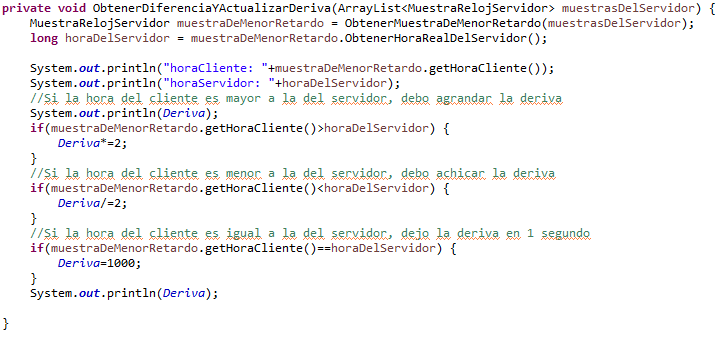


Como se observa en la imagen de la clase RelojDigital, la deriva inicial la recibe por parámetros, se verá mas adelante que el usuario debe definir este valor al iniciar el cliente. Además, se inicia el timer con la hora actual al que se lo retrasa 10 minutos para ver mas claramente la funcionalidad.



En la imagen se ve el método actualizar de la clase actualizarHora, mas adelante se verá que el usuario debe indicar cada cuántos milisegundos desea que se ejecute. Este método se conecta con el servidor, y envía un código 1 para indicarle que le debe enviar la hora, se registra el timestamp inicial antes de la solicitud, y el timestamp final cuando recibe la respuesta, a partir de estas 2 variables obtiene el retardo. Luego instancia un objeto de la clase MuestraRelojServidor con la hora actual del cliente (tomada del timer de RelojCliente), el retardo que calculó, y la hora del servidor, este objeto muestra lo añade a un arreglo de muestras. Este proceso lo realiza 5 veces, luego le envía un código 2 al servidor y finaliza la conexión.

Luego llama al método ObtenerDiferenciaYActualizarDeriva con las 5 muestras del servidor obtenidas.

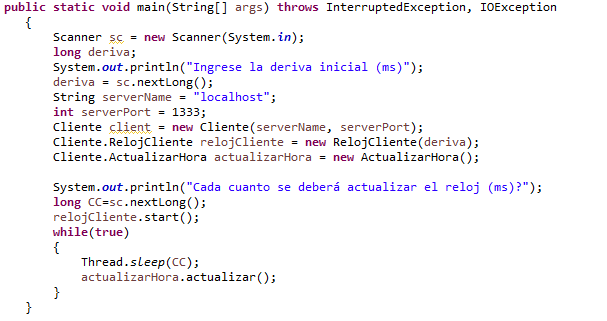


En este método lo primero que se debe hacer es definir qué muestra utilizará para comparar con la hora local, la muestra de menor retardo se obtiene a partir del método ObtenerMuestraDeMenorRetardo que recorre el arreglo de muestras comparándolas y devuelve la muestra que tenga el valor de retardo menor. Luego, para obtener la hora real del servidor que tiene la muestra, se llama al método HoraRealDelServidor() definido dentro de la clase MuestraRelojServidor, es decir que el objeto muestra obtiene su propia hora real y la devuelve.



Finalmente se compara la hora del cliente con la hora real del servidor, y en base a esta comparación la deriva se duplica o se reduce a la mitad.

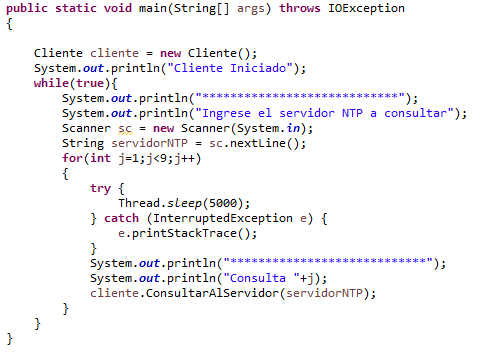
Se observó que dividirla o multiplicarla por dos es muy extremo, sería mas adecuado tal vez reducirla o ampliarla en un 30% o algún valor mas cercano a la deriva actual. Se definió de esta manera para ver más clara la ejecución.



En la imagen del método main del cliente, se ve como se solicitan los parámetros de deriva e intervalo de actualización.

1. Para este ejercicio se utilizaron las clases SNTPClient y NTPMessage. Se implementó una clase Tabla que representa la interfaz de la tabla. Además, recibe los valores de retardo, offset y fecha, los procesa y los añade como filas.

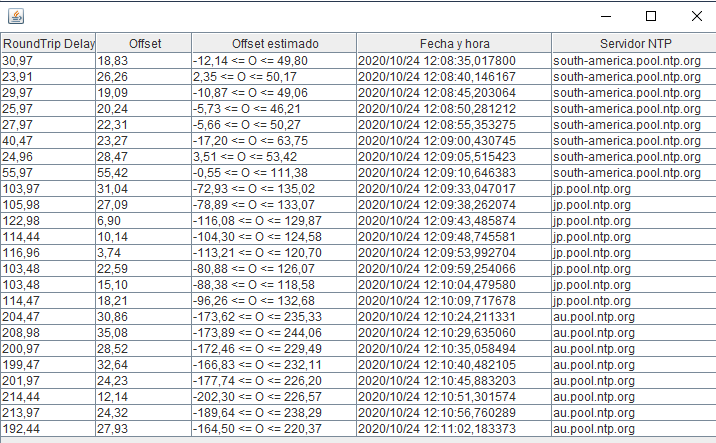
También se implementó una clase Cliente, que tiene una un objeto tabla que instancia en su constructor. Luego en el método main se llama al constructor y se solicita al usuario ingresar el servidor NTP que se desea consultar, se realizan 8 llamadas al método que consulta al servidor NTP ingresado. Se agrego un delay de 5 segundos entre consultas al servidor para que se vea como se va llenando la tabla.



El método ConsultarAlServidor es igual al método main de la clase SNTPClient.java con la única diferencia que cuando finaliza la consulta, llama al método agregarNuevaFila de la clase Tabla con los valores de roundTripDelay/2, localClockOffset, y la fecha y hora del mensaje.

Se realizaron pruebas con los servidores south-america.pool.ntp.org, jp.pool.ntp.org (servidor NTP de Japón) y au.pool.ntp.org (servidor NTP de Australia). Se observo que los retardos obtenidos con el servidor de América del sur son menores a los obtenidos respecto del servidor de Japón, y estos últimos a su vez son menores (casi la mitad) de los retardos obtenidos al consultar al servidor de Australia.

Todos los valores de offset y retardo que se muestran en la tabla están expresados en milisegundos.



1. Protocolo de 2 fases:

Fase 1: cada participante vota para que la transacción sea consumada o abortada. Una vez que un participante ha votado por la consumación de la transacción, no se le permite que la aborte. Por tanto, antes de que un participante vote para que se consume la transacción, debe asegurarse de que será capaz de llevar a cabo su parte del protocolo de consumación, incluso si falla y es reemplazado en su transcurso. Se dice que un participante en una transacción en estado preparado para una transacción si será finalmente capaz de consumarla. Para estar seguro de esto, cada participante guarda un dispositivo de almacenamiento permanente todos los objetos que haya alterado durante la transacción, junto con su estado (preparado).

Fase 2: todo participante en la transacción lleva a cabo la decisión conjunta. Si alguno de los participantes vota por abortar, entonces la decisión ha de ser de abortar la transacción. Si todos los participantes votan consumar, entonces la decisión es de consumar la transacción.

Protocolo de 2 fases para transacciones anidadas:

En esta aproximación, el protocolo de consumación en dos fases se convierte en un protocolo anidado multi-nivel. El coordinador de la transacción de nivel superior se comunica con los coordinadores de las subtransacciones para las que es su madre inmediata. Envía mensajes consultando si puedenComprometer a cada una de estas últimas, las cuales, a su vez, los pasan a los coordinadores de sus transacciones hijas (y así sucesivamente en el árbol). Cada participante recoge las respuestas de sus descendientes antes de responder sus madres.

Los argumentos necesarios del mensaje puedeComprometer son el id de la transacción de nivel superior que será usado cuando se preparen los datos. El segundo argumento es el id de la transacción del participante que hace la llamada a puedeConsumar. El participante que recibe la llamada busca en su lista de transacciones cualquier transacción o subtransacción consumada provisionalmente que concuerde con el id de transacción del segundo argumento.

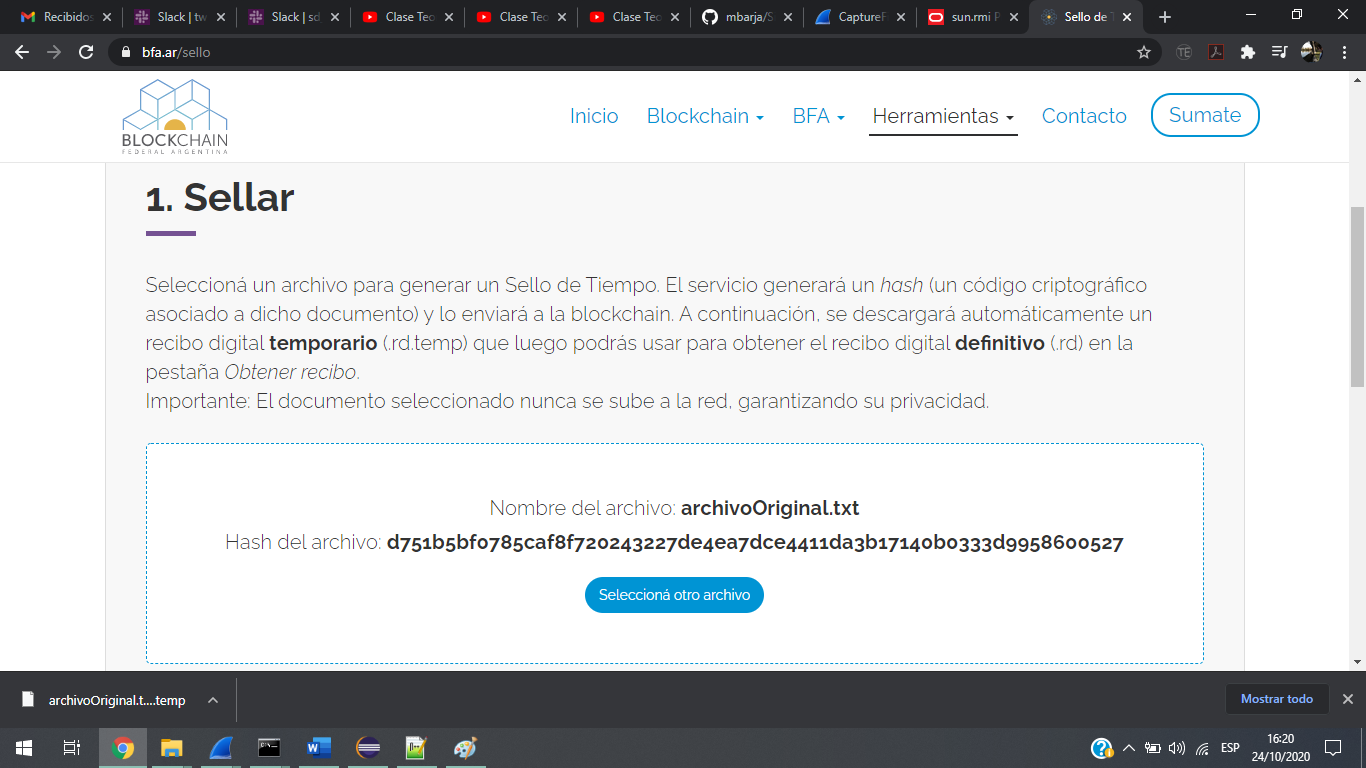
Si un participante encuentra cualquier subtransacción que se ajuste al segundo argumento, prepara los objetos y responde con un voto SI. Si no consigue encontrar alguna, entonces debe haberse caído desde que realizó la subtransacción y le responde con un voto NO.

El protocolo de consumación en 2 fases para el caso de transacciones anidadas puede ocasionar que se retrasen el coordinador o un participante en los mismos pasos que la versión no anidada. Aparece un cuarto paso en el cual la subtransacción puede retrasarse. En el caso de las subtransacciones consumadas provisionalmente y que son hijas de subtransacciones abortadas, puede que no sean informadas del resultado de la votación de la transacción. Para abordar estas situaciones, cualquier subtransacción que no haya recibido un mensaje puedeConsumar indagará pasado un periodo de tiempo limite. Para hacer posibles tales preguntas, los coordinadores de las subtransacciones abortadas han de sobrevivir un cierto período de tiempo. Si una subtransacción huérfana no puede contactar con su madre, al final, abortara.

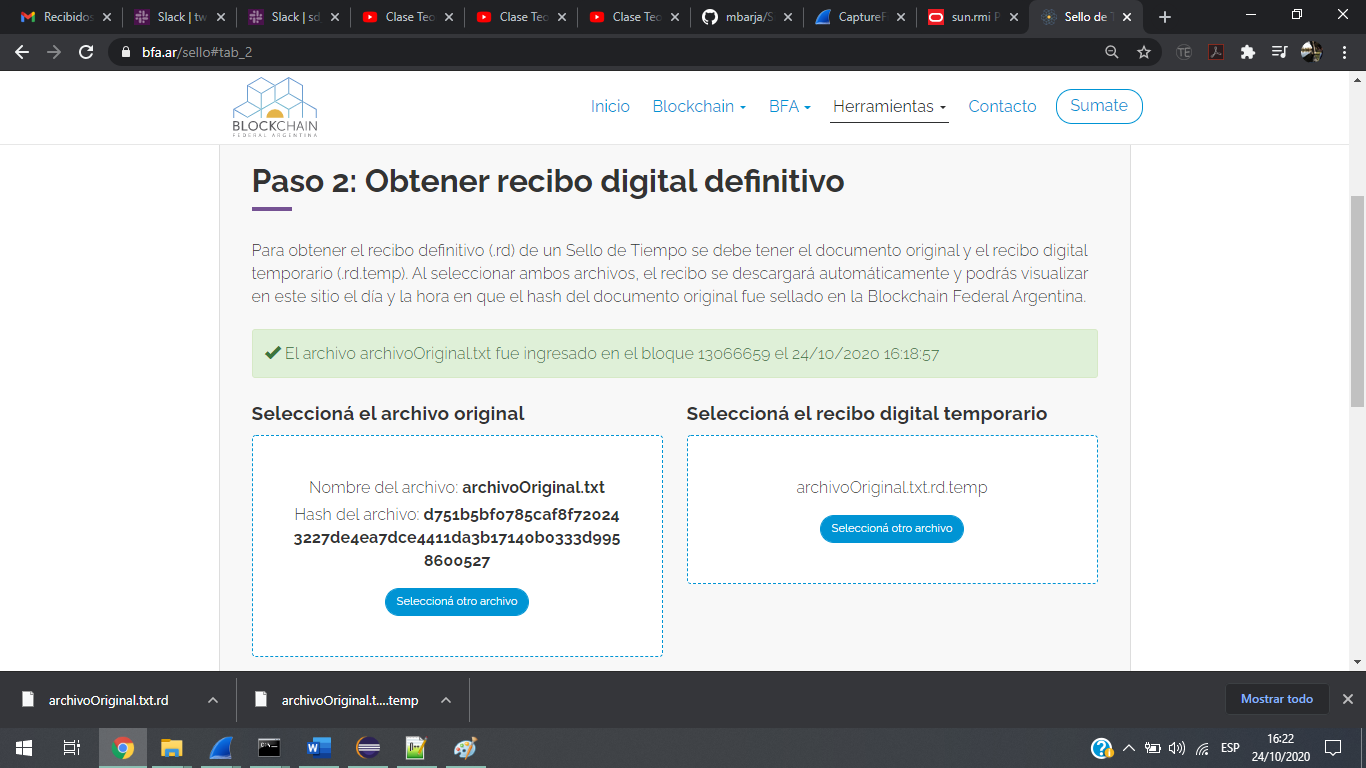
1. Se creo un archivo .txt

Bfa.ar

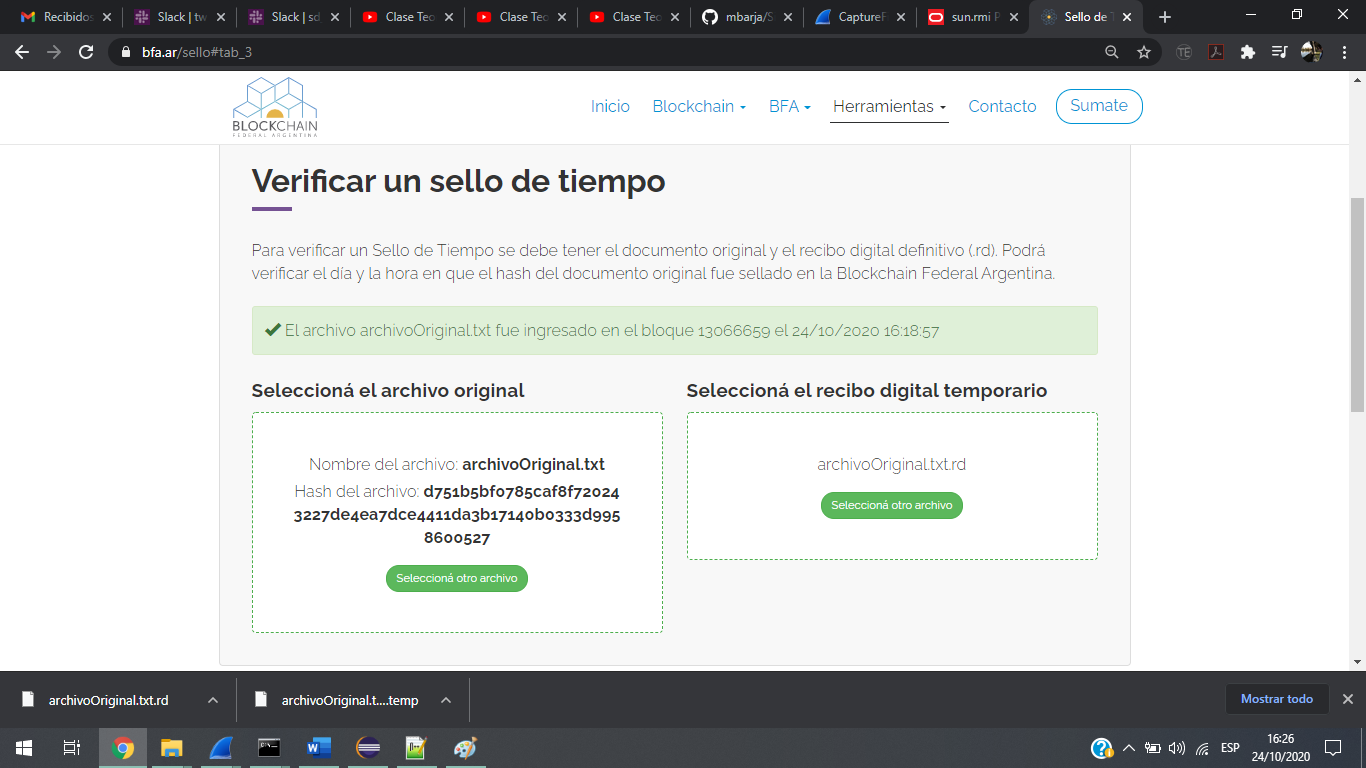
Se importa el archivo archivoOriginal.txt, y automáticamente se descargó el recibo digital archivoOriginal.txt.rd.temp y se generó el hash.



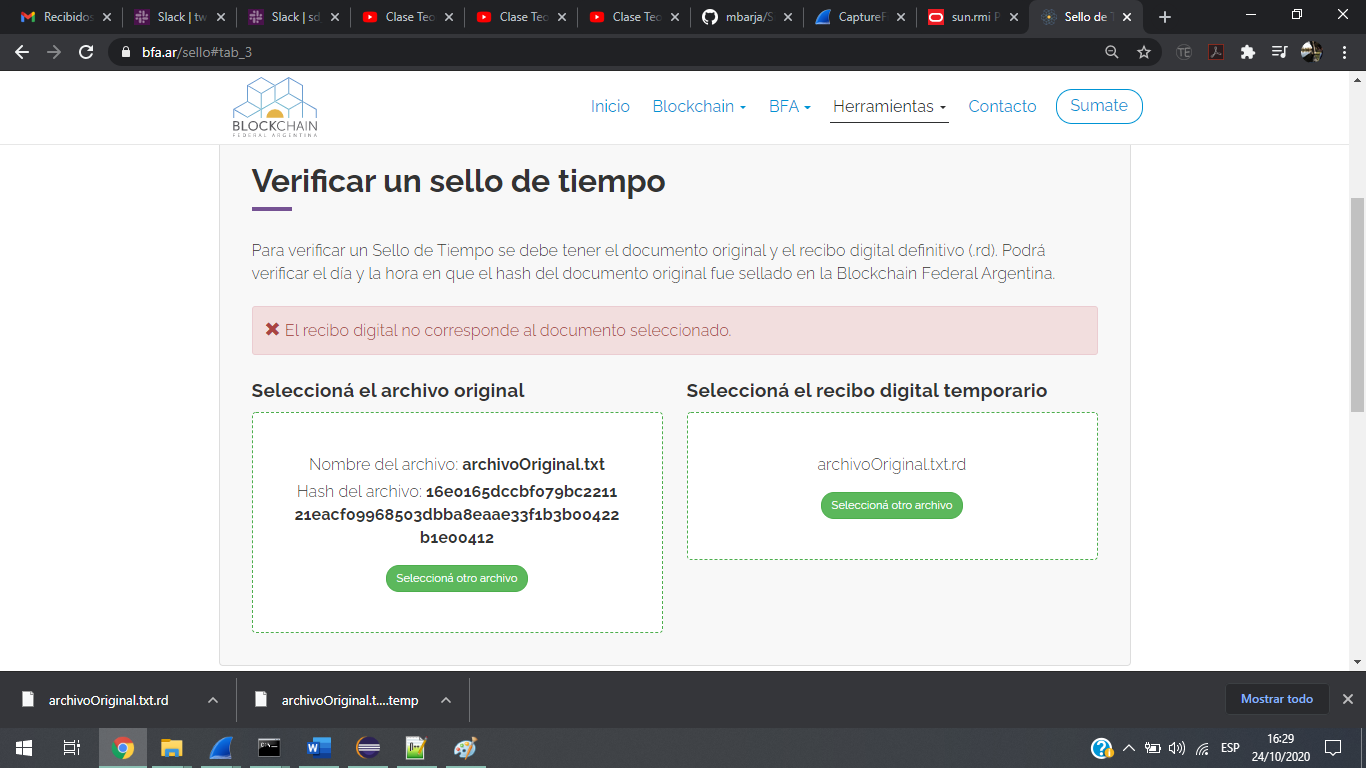
Luego en la pestaña de Obtener recibo, se importaron ambos archivos y se descarga automáticamente el archivo archivoOriginal.txt.rd



Finalmente, en la pestaña verificar, se ingreso el archivo original y el .rd y se verifica el sello de tiempo del archivo.



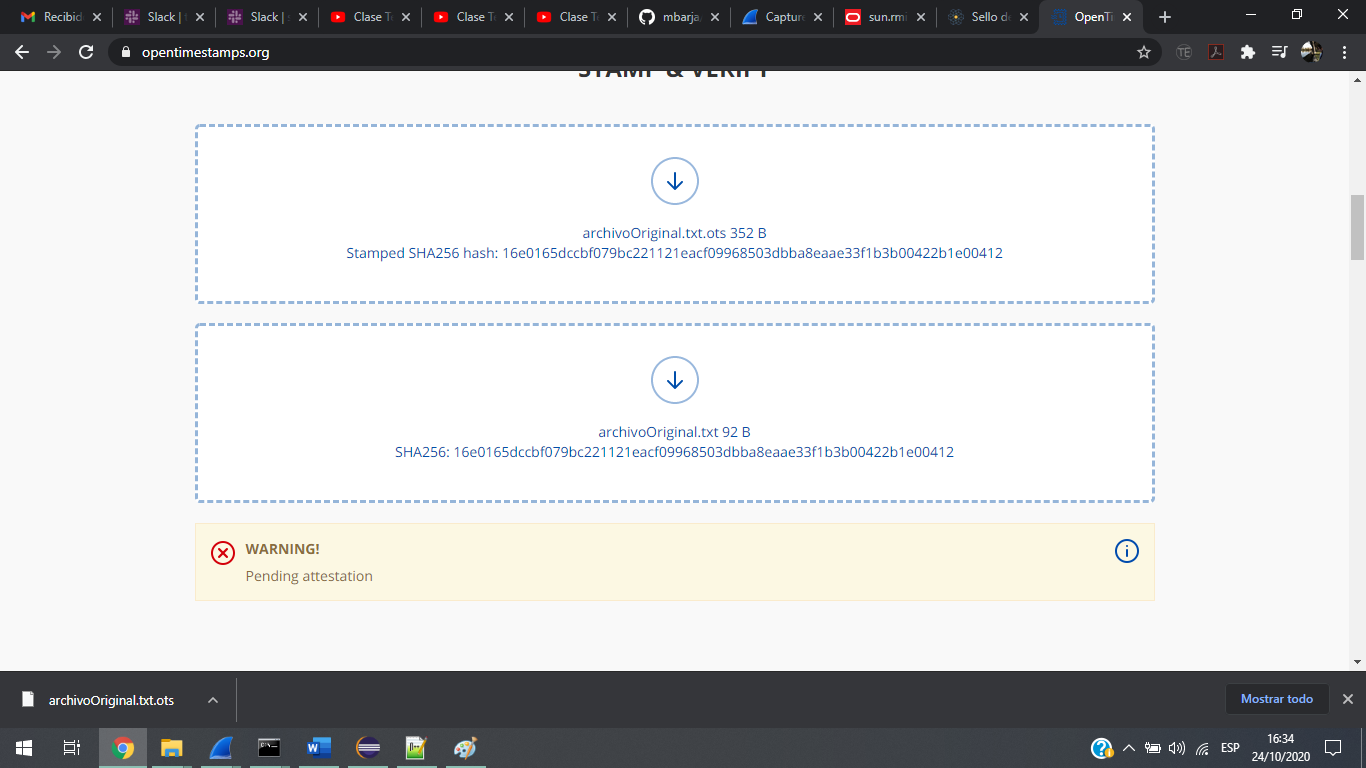
Luego se alteró el archivo original, y se intento verificar nuevamente. Se observa que no se valida el archivo una vez que fue modificado.



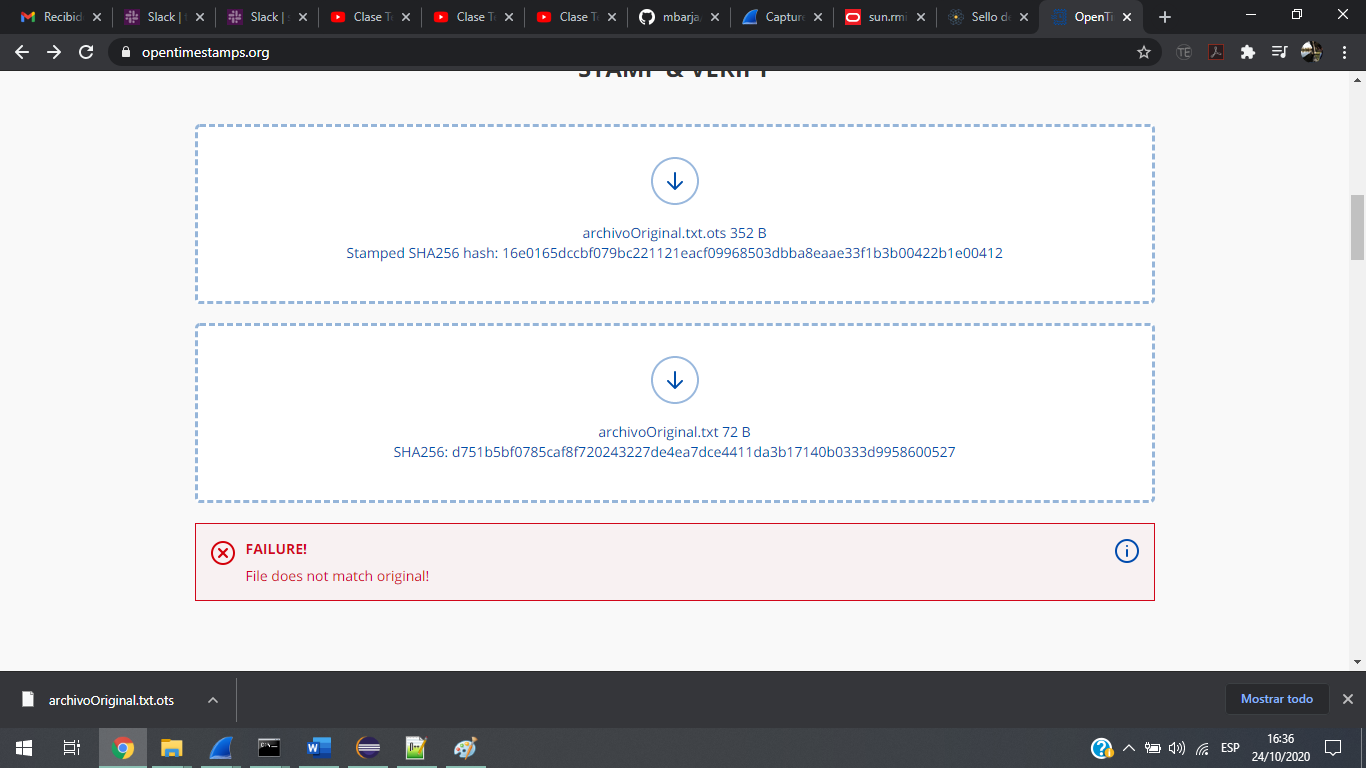
<https://opentimestamps.org/>

Se inserto el mismo archivoOriginal.txt y automáticamente se descarga un archivo con extensión .ots

Luego se importaron el archivo original y su extensión .ts, y se observa que lo verifica, pero muestra una advertencia por qué aun no fue incluido en ningún bloque.



Se modificó el archivo original y se intentó verificar nuevamente y se observó que no fue verificado.



Se realizó nuevamente la verificación para luego consultar a qué bloque fue agregado el archivo.

