Universidad Nacional de la Patagonia San Juan

Bosco Licenciatura en Sistemas O.P.C.G.P.I.

Sistemas Distribuidos

TP N.º 1

Docentes: Mg. Ing. Ricardo Antonio López
Lic. Cristian Javier Parise

Alumnos: Aguila Maximiliano
Krmpotic Lucas

Año 2018

Trabajo de Laboratorio 2

Java RMI. Concurrencia. Sincronización. Criptografía. PKI .Transacciones

1. Basándose en el proyecto java rmisum, implemente dos servidores con Java RMI, cada uno de ellos implementando funciones distintas (uno de ellos suma y resta y el otro multiplicación y división). El cliente debe tomar de la línea de comando tanto el indicador de operación como los operandos a utilizar. Valiéndose de un debugger y/o un analizador de protocolos, informar:

_Para la solución de este ejercicio nosotros planteamos dos servidores. En uno de ellos se atiende a los clientes y se les brinda servicio sobre dos operaciones: Suma y Resta (Servidor-suma-resta), mientras que en el otro (De la misma manera) se les da servicio sobre las operaciones de Multiplicación y División (Servidor-mult-div).

En la implementación nos basamos de los cascarones provistos por la cátedra, modificando las funciones y los Objetos Remotos que viajan para asi poder brindar los servicios nombrados anteriormente.

Luego para poder simular que un mismo cliente se comunica con dos servidores remotos distintos mediante RMI (Poniéndose en evidencia la abstracción sobre los puertos en los que RMIRegistry atiende a los clientes), solicitando servicios de cada uno de los Servidores remotos se realizaron los siguientes pasos:

_Uno de los servidores corriendo en la maquina local (RMIRegistry activo) y para simular otro servidor utilizamos una imagen de Docker.

Imagen utilizada para correr el contenedor de uno de los servidores:

https://hub.docker.com/_/openjdk/

Se corre con el comando:

docker run -it --name servidor-suma-resta -v $PWD:/usr/src/servidor-suma-resta openjdk bash

Recordar que en wireshark aparecerá el cliente con distintas direcciones IP. En un caso con la 127.0.0.1 y en el otro con la 172.17.0.1 que es el gateway por defecto de la red en la que se encuentra el contenedor del servidor-suma-resta. Esto lo maneja el domonio de docker mediante su interfaz de red que se llama docker0 y que funciona como un bridge entre todas las redes virtuales de contenedores y el sistema operativo anfitrion.

a) En cuáles puertos atiende cada uno de los objetos remotos de los servidores?.

_Utilizando Wireshark se puede observar que la comunicación del Cliente con RMI Registry es mediante su puerto por defecto (Port=1099), y luego la comunicación del Cliente con los objetos remotos es mediante un puerto diferente, el cual RMI Registry le asocia.

_ Suma/Resta = 34703

172.17.0.1	172.17.0.2	TCP	74 47240 → 34703 [SYN, ECN, CWR]
172.17.0.2	172.17.0.1	TCP	74 34703 → 47240 [SYN, ACK, ECN]
172.17.0.1	172.17.0.2	TCP	66 47240 → 34703 [ACK]

_ Multiplicacion/Divicion = 42529

```
127.0.0.1    127.0.0.1    TCP    74 46952 → 42529 [SYN, ECN, CWR]
127.0.0.1    127.0.0.1    TCP    74 42529 → 46952 [SYN, ACK, ECN]
127.0.0.1    127.0.0.1    TCP    66 46952 → 42529 [ACK]
```

b) Cuál es el puerto en el que atiende RMIRegistry?. _

RMI Registry por defecto atiende en el puerto 1099.

Tambien el puerto se le puede asignar al momento de ejecutar rmiregistry.

c) Identificar el mensaje que transfiere el cliente con RMIRegistry.

_En las siguientes capturas puede verse claramente el handshacke (o 3-way) entre el Cliente y RMI

Registry. Cliente - Servidor-mult-div

```
127.0.0.1    127.0.0.1    TCP    74 56050 → 1099 [SYN, ECN, CWR]
127.0.0.1    127.0.0.1    TCP    74 1099 → 56050 [SYN, ACK, ECN]
127.0.0.1    127.0.0.1    TCP    66 56050 → 1099 [ACK]
```

Cliente - Servidor-suma-resta

```
172.17.0.1   172.17.0.2   TCP    74 38886 → 1099 [SYN, ECN, CWR]
172.17.0.2   172.17.0.1   TCP    74 1099 → 38886 [SYN, ACK, ECN]
172.17.0.1   172.17.0.2   TCP    66 38886 → 1099 [ACK]
```

d) Los mensajes hacia RMI Registry se valen de TCP o UDP.?

_Los mensajes hacia RMI Registry se valen e TCP, podemos ver en capturas anteriores que las comunicaciones tienen en handshake (o 3-way) de inicio de conecion. Y tambien tienen el cierre de coneccion, por lo cual podemos deducir que la comunicación es orientado a la conexion.

2. Implemente con Java RMI el servidor de archivos remoto del punto 3.b de la práctica anterior. Debe tener las cuatro operaciones básicas: open, close, read y write.

a) Compare la especificación de interfaz entre cliente y servidor de la implementación con RMI con la basada en sockets de Java. Efectúe los comentarios que haya lugar.

La diferencia es notable tanto desde el punto de vista de la simpleza como de la transparencia de cuestriones que son propias de la comunicación entre la capa de aplicación y la de transporte, y del manejo de la concurrencia (definición de puertos, creación de hilos, mashalling/unmarshalling).
Las cuestiones relacionadas a la eficiencia, nivel de abstracción y concurrencia se tocan en otros apartados. En relación a la simpleza RMI no sólo nos ahorra la definición, diseño e implementación de los stubs, sino también

la creación de las clases necesarias para el envío y recepción de objetos. En nuestro caso concretamente pasamos de tener 13 clases a tener 3 interfaces y 3 clases.

b) Compare la complejidad de la implementación con Java sockets y con RMI tanto para el servidor como para el cliente. Efectúe los comentarios que haya lugar.

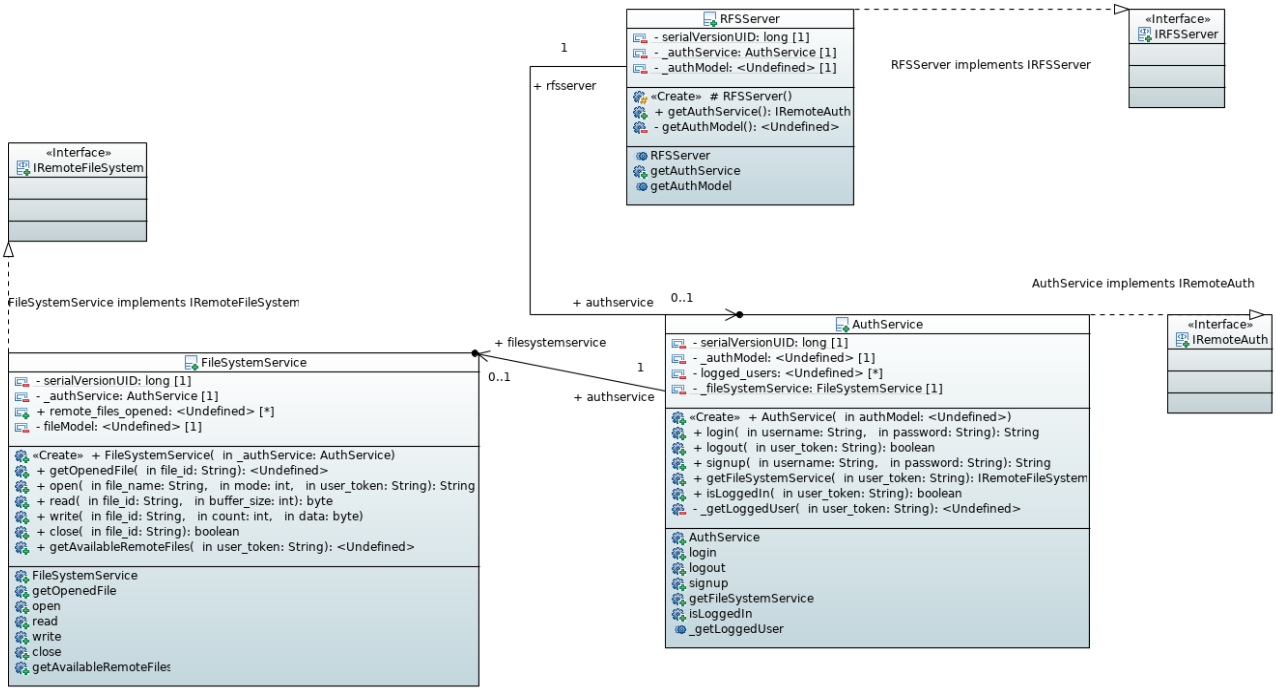
A nivel de aplicación no hay grandes cambios. La diferencia fundamental se encuentra en el manejo de la comunicación.

La implementación para el cliente prácticamente no varía dado que más allá de cambios menores, se podía adaptar perfectamente la solución anterior reemplazando la llamada al stub por la llamada al objetoRemoto.

```
String file_id = stub.rfs_open(file_name, user_token, mode);
String dir = Config.getProperties().getProperty("home_path");
FileProxy file = new FileProxy(dir+file_name);
file.setFileID(file_id);
```

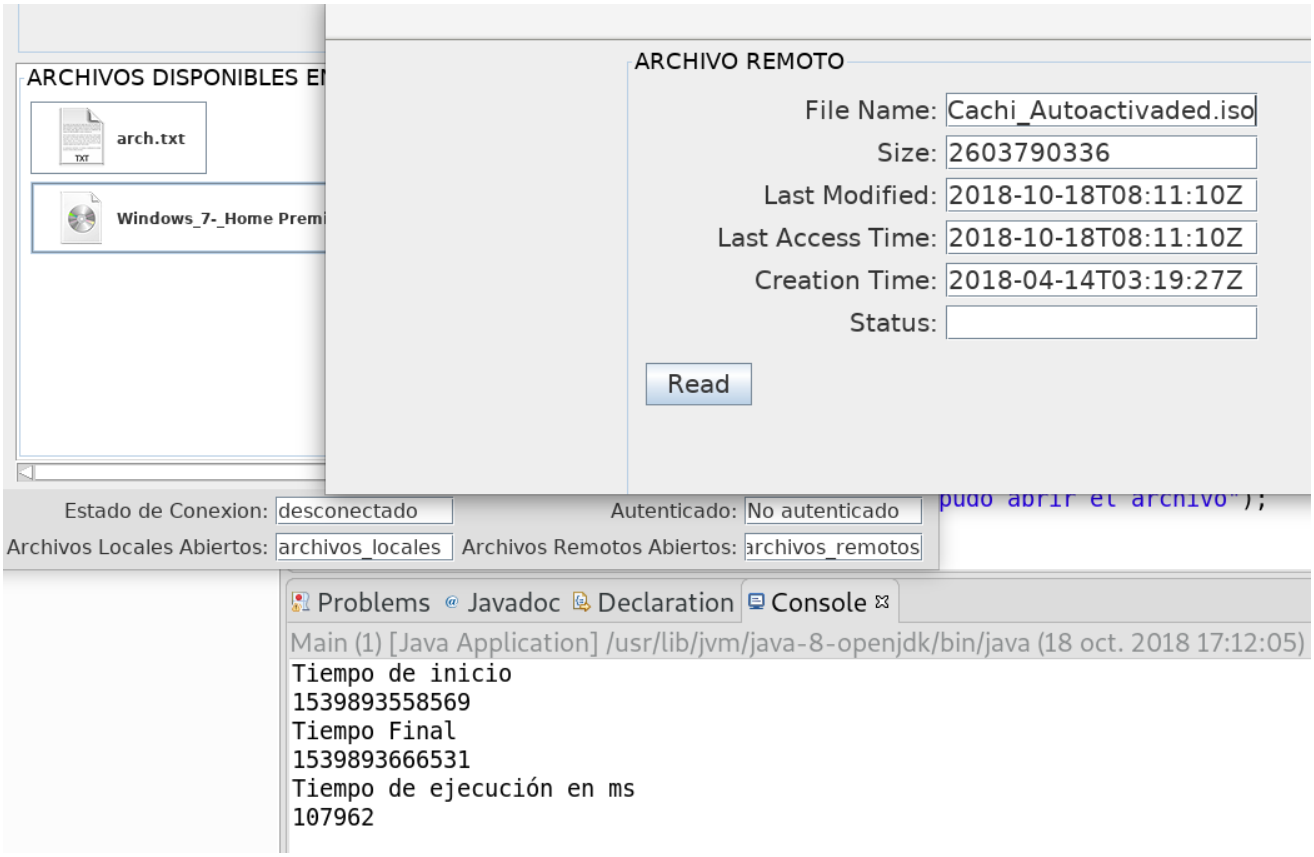
```
String file_id = this.remoteFileSystem.open(file_name, mode, user_token);
String dir = Config.getProperties().getProperty("home_path");
FileProxy file = new FileProxy(dir+file_name);
file.setFileID(file_id);
```

En el servidor sí se vé con claridad la diferencia ya que no es necesario ocuparnos de la creación de hilos ni del manejo de puertos. Además, en nuestro caso se aprovechó la forma en la que RMI maneja la devolución de objetos remotos para lograr un esquema controlado de acceso a los servicios. Así, el cliente accede al servidor, de él obtiene el servicio de autenticación y sólo si se autentica accede al objeto file system:



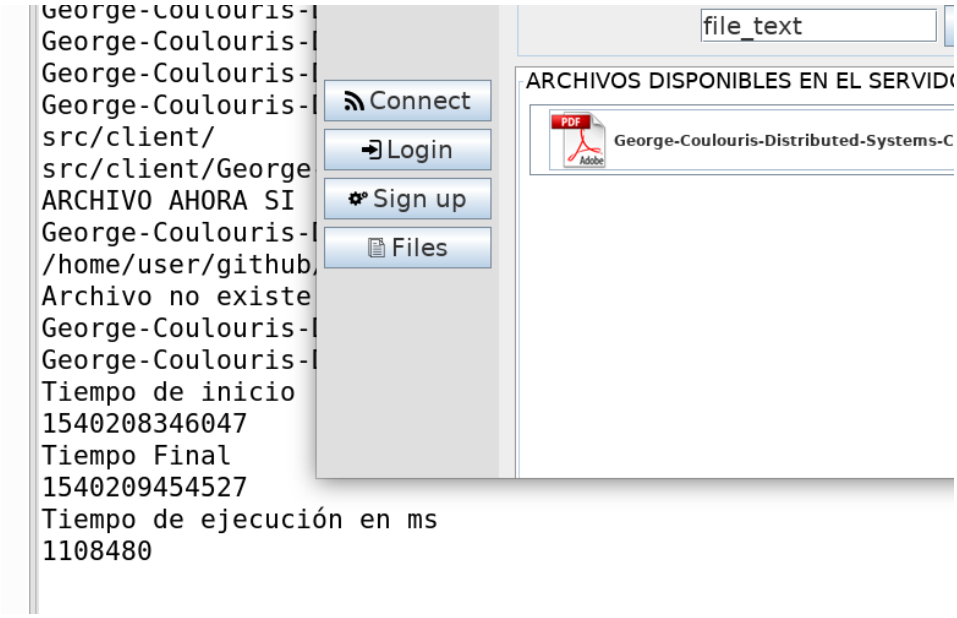
c) Transferir el mismo archivo de gran tamaño (GBs) que se utilizó para el experimento con Java Sockets y con RPC, pero ahora en la versión Java RMI (utilizando un tamaño de buffer igual que en los anteriores) y cronometrar (que el mismo cliente muestre cuanto demoró) sacando las conclusiones comparando la performance con las anteriores implementaciones.

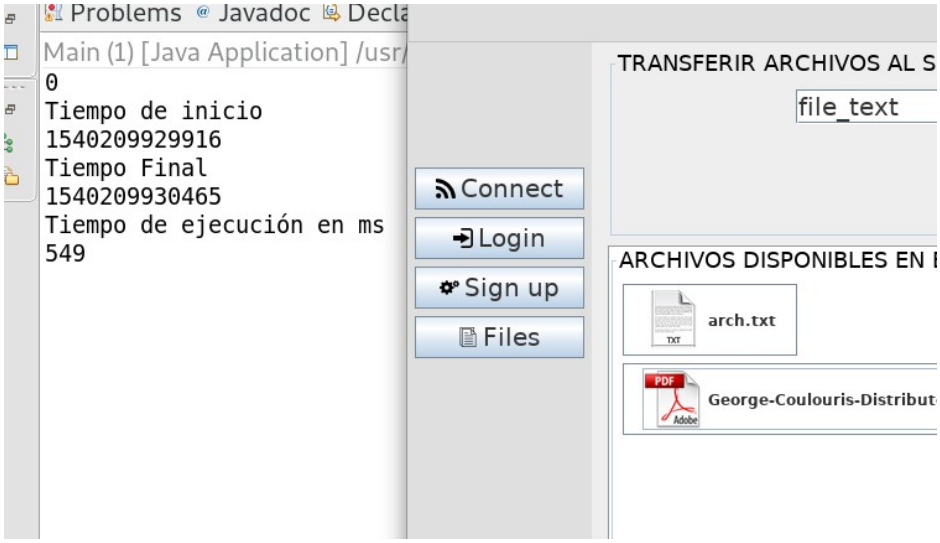
La primera prueba que se hizo fue con un archivo de 2.6GB (una iso de windows 7). En el caso de RMI el resultado fue bastante satisfactorio. Como puede verse en la imagen el tiempo de transferencia (local) fue de un poco más de 10 seg.



Sin embargo con la aplicación desarrollada con sockets se decidió no concluir la prueba luego de que pasados 20 minutos no terminara la transferencia. Por lo tanto se decidió reducir drásticamente el tamaño del archivo a transferir con el objetivo de poder efectuar la comparación.

Se utilizó el archivo pdf del libro ”Distributed Systems Concepts and Design” de George Coulouris, que pesa 11,1MB. Como puede verse en las imágenes a continuación, la aplicación con sockets tardó un poco más de 11seg en transferir el archivo localmente, mientras que la aplicación con rmi demoró solamente 549 ms





3. Responda las siguientes cuestiones:

a) Cuáles son los problemas que tiene un servidor concurrente?

_Principalmente, los problemas que se pueden dar en un servidor concurrente tienen que ver con la sincronización. Un ejemplo es el de la atomicidad de las operaciones que se realizan. Supongamos que un proceso realiza un chequeo sobre un valor (Digamos, verificar que no es nulo) y lo encuentra como que no es nulo, entonces su siguiente acción es mostrar ese valor. Si toda esa acción no se considera como atómica, puede suceder que se la interrumpa en el medio, otro proceso ponga nulo ese valor y, al retomar la ejecución, muestra un nulo en lugar del

valor que debía contener. Existen otros ejemplos también, como podría ser el de suponer que existe cierto valor inicializado, e incluso algunos problemas llamados Deadlocks.

Este tipo de problemas de sincronización es importante tenerlos en cuenta en sistemas distribuidos, ya que un servidor puede tener réplicas que conviven entre sí bajo algún mecanismo de sincronización que debe estar bien programado para no dar lugar a inconsistencias (O caer en la necesidad de interrumpir el servicio para llevar a cabo la sincronización).

b) Qué mecanismos se tienen disponibles con Java RMI?

_ La invocacion de metodos remotos es la version Orientada a Objetos de RPC.

_Uno de los mecanismos más importantes es la sincronización de métodos, que nos soluciona la implementación de la concurrencia en los servidores que se conecten con RMI.

c) Podría considerar que RPC y RMI tienen el mismo nivel de abstracción? Justifique.

_ En nuestra opinión no, ya que (si bien en RPC uno tiene los archivos de los stubs a la vista en RMI podrias tenerlos con el comando rmic -d, pero ya es obsoleto) ambos realizan una configuración automática del mecanismo con el que el servidor y los clientes se comunican, mientras que el programador sólo define la interfaz de los servicios que se van a ofrecer.

Obviamente esta tambien en que RPC corre en C y RMI corre en java, por lo cual esta el nivel de abstraccion del lenguaje mismo. RPC realiza invocacion de funciones remotamente, mientras que RMI invoca Metodos de forma remota.

La invocacion de metodos remotos es la version Orientada a Objetos de RPC.

d) Identifique similitudes y diferencias entre RPC y Java RMI.

_Como similitud podemos encontrar el hecho de que se define la interfaz en una clase y RPC/RMI se encargan de realizar la implementación de los stubs.

Obviamente esta tambien en que RPC corre en C y RMI corre en java, por lo cual esta el nivel de abstraccion del lenguaje mismo. RPC realiza invocacion de funciones remotamente, mientras que RMI invoca Metodos de forma remota.

Otra diferencia es que con RPC tenes los Stubs a la vista. Rmi automaticamente no lo hace, pero se podria tenerlos con el comando “*rmic -d”*, aunque ya es obsoleto.

e) Determine si el servidor concurrente con Java RMI tiene un thread por requerimiento, por conexión o por recurso (en términos del cap. 6 de Coulouris).

_ Para determinar de qué manera atiende el servidor, se realizo un escenario diferente para que el servidor se identifique (por PID de hilo). Entonces, en cada llamada, el servidor dirá qué pid la está procesando. (Se utiliza el mismo escenario que el Punto 4)

Luego, evaluamos el comportamiento de la siguiente manera:

- Si cada cliente llama al mismo método y el servidor atiende en pids diferentes, de tiene un hilo por conexión.
- Si cada cliente llama al mismo método y el servidor devuelve el mismo pid, y al llamar a métodos diferentes envía diferentes pids, se tiene un hilo por recurso .
- Y, finalmente, si un mismo cliente llama a un método (Cualquiera sea) y, a la siguiente llamada al mismo método el pid cambia, se tiene un hilo por requerimiento.

En la primera prueba, cada cliente llama al mismo método (Cuenta), siendo atendidos concurrentemente.

```
0 - PID--> 14
1 - PID--> 14
0 - PID--> 16
2 - PID--> 14
1 - PID--> 16
3 - PID--> 14
2 - PID--> 16
4 - PID--> 14
3 - PID--> 16
5 - PID--> 14
4 - PID--> 16
6 - PID--> 14
5 - PID--> 16
7 - PID--> 14
6 - PID--> 16
8 - PID--> 14
7 - PID--> 16
9 - PID--> 14
8 - PID--> 16
10 - PID--> 14
PID--> 14     FIN
9 - PID--> 16
10 - PID--> 16
PID--> 16     FIN
```

Con esto no se puede determinar mucho, salvo que la atención no es por recurso. Por lo pronto todo indica que el servidor atiende por conexión. Vamos a determinar ahora si es por requerimiento.

Para determinarlo, sólo nos hace falta iniciar un sólo cliente y realizar dos llamadas consecutivas al mismo método. Si es un hilo por requerimiento, a cada llamada la debería atender un hilo diferente.

```
0 - PID--> 16
1 - PID--> 16
2 - PID--> 16
3 - PID--> 16
4 - PID--> 16
5 - PID--> 16
6 - PID--> 16
7 - PID--> 16
8 - PID--> 16
9 - PID--> 16
10 - PID--> 16
PID--> 16     FIN
0 - PID--> 16
1 - PID--> 16
2 - PID--> 16
3 - PID--> 16
4 - PID--> 16
5 - PID--> 16
6 - PID--> 16
7 - PID--> 16
8 - PID--> 16
9 - PID--> 16
10 - PID--> 16
PID--> 16     FIN
```

En esta última captura se puede ver que, para dos requerimientos subsecuentes de un mismo cliente, el pid es el mismo. Por lo que, finalmente, concluímos que el servidor atiende con un hilo por conexión.

4. Tomando el proyecto del ejercicio 1 elabore un escenario donde pueda verificarse:

a) Si RMI otorga un servicio concurrente en forma automática para el acceso a objetos remotos

_ RMI otorga por default servicio concurrente. En la siguiente captura podemos ver como dos clientes se comunican con un mismo servidor (Servidor-suma-resta) de manera concurrente. (Se ejecutaron en la maquina local).

127.0.0.1	127.0.0.1	TCP	111 56238 → 44835 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	111 56246 → 44835 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	92 44835 → 56246 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	92 44835 → 56238 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	111 56246 → 44835 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	111 56238 → 44835 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	92 44835 → 56246 [PSH, ACK]
127.0.0.1	127.0.0.1	TCP	92 44835 → 56238 [PSH, ACK]

_ Otra prueba ademas de esta (la captura de wireshark no demuestra totalmente si es concurrente, podria ser serializado), con el escenario de prueba posterior (4.b) podemos ver como dos diferentes hilos (distinguidos por su PID) estan trabajando sobre un mismo objeto remoto al mismo tiempo, lo cual nos muestra claramente la concurrencia.

b) Que las variables de instancia de un determinado objeto remoto, no se “contaminan” con sus equivalentes de otras instancias del mismo objeto a las que se está accediendo en forma concurrente (acceso sincronizado).

_ Se realizo un escenario claro en el cual se realiza una cuenta de 10 segundos y las muestra en el servidor indicando tambien el PID del hilo que lo esta ejecutando. La clase “Segundero” tiene un atributo en el cual se van seteando y leyendo cada vez que pasa un segundo. De esta manera podemos ver que si ejecutamos dos clientes de forma concurrente ejecutando el mismo metodo (Cuenta) del objeto remoto, si se ven o no afectados por la modificacion de ese atributo.

```
0 - PID--> 14
1 - PID--> 14
0 - PID--> 16
2 - PID--> 14
1 - PID--> 16
3 - PID--> 14
2 - PID--> 16
4 - PID--> 14
3 - PID--> 16
5 - PID--> 14
4 - PID--> 16
6 - PID--> 14
5 - PID--> 16
7 - PID--> 14
6 - PID--> 16
8 - PID--> 14
7 - PID--> 16
9 - PID--> 14
8 - PID--> 16
10 - PID--> 14
PID--> 14     FIN
9 - PID--> 16
10 - PID--> 16
PID--> 16     FIN
```

Como resultado obtenemos que cada cliente realiza la cuenta hasta 10, por lo cual no se contaminan ambas ejecuciones.

c) Que cuando se pasan objetos remotos como parámetros el pasaje es por referencia, contrariamente de cuando se pasan objetos locales (por valor) . Pag. 218 Colouris

_ Cualquier objeto que sea serializable, es decir, que implementa una interfaz serializable se puede pasar como un argumento o como resultado en Java RMI. Todos los tipos primitivos y los objetos remotos son serializables.

Pasaje por Referencia: cuando el tipo de un parámetro se define como una interfaz remota, el argumento siempre se pasa como referencia al objeto remoto.
Cuando una referencia de objeto remoto se recibe, se puede utilizar para realizar llamadas RMI en el objeto remoto al que hace referencia.

Pasaje por valor: Cuando un objeto se pasa por valor, un nuevo objeto se crea en el proceso del receptor. Los métodos de este nuevo objeto pueden ser invocado localmente, posiblemente causando que su estado difiera del estado del objeto original en el proceso del remitente.

5. ¿Hay una manera sencilla de establecer un timeout para el objeto/clase que hace un RMI?. Implemente una solución.

Para la solución de este ejercicio, se cambió la implementación del Cliente de la siguiente forma:

- Se agregó una clase MiSocketFactory, que sobreescribe la clase RMISocketFactory y define un socket al que se le puede asignar un tiempo de timeout al momento de crearlo.
- Se agregó en el cliente, previo a la creación de la conexión con rmiregistry, la creación de MiSocketFactory, especificando un timeout de 10 segundos.

```
class MiSocketFactory extends RMISocketFactory {
    private int timeout;

    public MiSocketFactory(int time) {
        this.timeout = time;
    }

    @Override
    public ServerSocket createServerSocket(int port) throws IOException {
        return getDefaultSocketFactory().createServerSocket(port);
    }

    @Override
    public Socket createSocket(String host, int port) throws IOException {
        Socket timeout_socket = getDefaultSocketFactory().createSocket(host, port);
        timeout_socket.setSoTimeout(timeout * 1000);
        return timeout_socket;
    }

}
```

Y luego desde el cliente se instancia un SocketFactory y se le manda como parametro el timeout.

```
// Se crea el socket factory, dandole un tiempo de 10 segundos
try {
    RMISocketFactory.setSocketFactory(new MiSocketFactory(10));
} catch (IOException e) {
    e.printStackTrace();
}
```

Luego, a cada llamada del lado del cliente se le agregó un bloque que captura la excepción en caso de recibir una excepción por timeout del socket (Y de esa manera no cortar con la ejecución del cliente).

```
try {
    System.out.println(objetoRemoto.suma(2, 10));
} catch (UnmarshalException e) {
    System.out.println("Timeout...");
}
try {
    System.out.println(objetoRemoto.resta(10, 5));
} catch (UnmarshalException e) {
    System.out.println("Timeout...");
}
```

Finalmente, para probar el caso de timeout, se le agregó al servidor un sleep de 20 segundos en la operación de suma. De esta forma, al llamar esa función, tardaría 20 segundos en responder, por lo que el socket daría timeout.

Servidor:

```
Sumando 2 + 10
Restando 10 - 5
```

Cliente:

```
➜ src git:(master) ✗ java Cliente localhost
Timeout...
5
```

6. Implemente un servidor de hora con Java RMI. Desarrollar un cliente que emule un reloj (utilizar hilos para la actualización del mismo) y que se sincronice dicho reloj con el servidor de hora implementado, utilizando el algoritmo de Cristian. Al utilizar la hora patrón, tome un grupo de valores (por ejemplo 5), eligiendo el que haya demandado menor tiempo de tránsito de los mensajes.

Para la solución de este punto, se implementaron 3 clases:

- Cliente: En donde está implementado el tanto el algoritmo de cristian como el método main (En el que se crean tanto el reloj como el mismo cliente, llamando al método RUN que inicia el hilo de ejecución).
- Clock: Se trata de la implementación de la interfaz de RMI.
- Server: Contiene su propio método main (Como es de esperarse) e instancia el Reloj. Sólo contiene un método (getHora), que devuelve la hora actual en milisegundos para que sea consumida por el cliente.

Se trata de un reloj con un desfase de 5 minutos (300 segundos) con respecto a un servidor de hora (que toma la hora directo del sistema). Nuestro reloj aumenta normalmente su tiempo en 1 segundo tras cada espera de 1 segundo + un tiempo delta (que inicialmente es 0).
Cuando se presiona el botón de actualizar hora, nuestro reloj pide la hora cinco veces al servidor de hora y, mediante el Algoritmo de Cristian, calcula la hora real y la diferencia en milisegundos entre ella y la hora propia de nuestro reloj.
Una vez obtenida la diferencia, el reloj cambia el valor de delta de acuerdo a si el desfase fue positivo o negativo y setea un valor de delta dependiendo el defasaje, antes de volver a cero. Con esto, el tiempo que tarda cada segundo

de nuestro reloj en avanzar habrá cambiado hasta que este y el reloj del servidor de hora sean cercanos con una diferencia menor al valor de delta.


```
La diferencia en milisegundos es: 299653
La diferencia en segundos es: 299
La diferencia en minutos es: 4
Trabajando con Delta: 1500
```

Ademas, como el defasaje en principio es de 5 minutos (300 segundos) y comienza con un Delta = 0, al precionar el boton de Actualizar, se verifica si el reloj esta adelantado/atrasado y dependiendo de eso tambien verifica cuanto defasaje tiene.

En el Cliente para que la sincronizacion sea mas rapida, se implementaron “Rangos” por los cuales el Delta va a ir variando para ajustar la Hora.

```
if(diferencia!=0){ //si el reloj no esta sicronizado a nivel milisegundos
    if (diferencia > 0){ //reloj adelantado
        if((diferencia/1000) > 200){
            this.setDelta(1500); //aumenta el tiempo entre cada segundo del reloj
        }else {
            if((diferencia/1000) > 50){
                this.setDelta(1000);
            }else{
                this.setDelta(granularidad);
            }

        }

    }else{ //reloj atrasado
        if(((Math.abs(diferencia))/1000) > 3){
            this.setDelta(-1500);//reduce el tiempo entre cada segundo del reloj
        }else{
            if(((Math.abs(diferencia))/1000) > 50){
                this.setDelta(-1000);
        }else {
            this.setDelta(-granularidad);
        }

    }
```

7. Desarrolle una aplicación que se comunique con un servidor NTP de internet y que extraiga 8 muestras del retardo con que correspondería ajustar la hora obtenida. Presentar en pantalla una tabla con cada muestra del tiempo obtenido del servidor en formato YYYY/MM/DD HH:MM:SS.mmm en relación con cada retardo.

Para este ejercicio se utilizó un cascarón provisto por la cátedra que nos abstrajo de la generación del mensaje NTP en sí, por lo que nosotros sólo implementamos un loop de 8 iteraciones, en las que se le pide la hora al servidor NTP elegido, se obtiene la timestamp inicial, la timestamp que devuelve el paquete de datos del servidor de hora y se calculan el RTT y el offset del reloj local.

En las pruebas se realiza con servidores NTP como :

- time.google.com
- pool.ntp.org (generico)

---------- Mensaje Recibido *----------*

---------- Mensaje Recibido *----------*

Muestra: 1

Originate timestamp: 10-oct-2018 00:28:14,496000

Receive timestamp: 10-oct-2018 03:28:13,853713

Transmit timestamp: 10-oct-2018 03:28:13,853741

Dest. timestamp: 10-oct-2018 00:28:14,677000

Round-trip delay: 180,97 ms

Local clock offset: 10799267,23 ms

---------- Mensaje Recibido *----------*

---------- Mensaje Recibido *----------*

Muestra: 2

NTP server: pool.ntp.org

Leap indicator: 0

Version: 3

Mode: 4

Stratum: 2

Poll: 3

Precision: -24 (6E-8 seconds)

Root delay: 0,23 ms

Root dispersion: 2,26 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:14,707000

Receive timestamp: 10-oct-2018 03:28:14,065829

Transmit timestamp: 10-oct-2018 03:28:14,065855

Dest. timestamp: 10-oct-2018 00:28:14,888000

Round-trip delay: 180,97 ms

Local clock offset: 10799268,34 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 3

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:14,893000

Receive timestamp: 10-oct-2018 03:28:14,251861

Transmit timestamp: 10-oct-2018 03:28:14,251896

Dest. timestamp: 10-oct-2018 00:28:15,074000

Round-trip delay: 180,96 ms

Local clock offset: 10799268,38 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 4

NTP server: pool.ntp.org

Leap indicator: 0

Version: 3

Mode: 4

Stratum: 2

Poll: 3

Precision: -24 (6E-8 seconds)

Root delay: 0,23 ms

Root dispersion: 2,26 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:15,079000

Receive timestamp: 10-oct-2018 03:28:14,437853

Transmit timestamp: 10-oct-2018 03:28:14,437876

Dest. timestamp: 10-oct-2018 00:28:15,260000

Round-trip delay: 180,98 ms

Local clock offset: 10799268,36 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 5

Root dispersion: 2,26 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:15,265000

Receive timestamp: 10-oct-2018 03:28:14,623623

Transmit timestamp: 10-oct-2018 03:28:14,623651

Dest. timestamp: 10-oct-2018 00:28:15,446000

Round-trip delay: 180,97 ms

Local clock offset: 10799268,14 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 6

NTP server: pool.ntp.org

Leap indicator: 0

Version: 3

Mode: 4

Stratum: 2

Poll: 3

Precision: -24 (6E-8 seconds)

Root delay: 0,23 ms

Root dispersion: 2,26 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:15,451000

Receive timestamp: 10-oct-2018 03:28:14,809629

Transmit timestamp: 10-oct-2018 03:28:14,809652

Dest. timestamp: 10-oct-2018 00:28:15,632000

Round-trip delay: 180,98 ms

Local clock offset: 10799268,14 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 7

Root delay: 0,23 ms

Root dispersion: 2,27 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:15,637000

Receive timestamp: 10-oct-2018 03:28:14,995631

Transmit timestamp: 10-oct-2018 03:28:14,995690

Dest. timestamp: 10-oct-2018 00:28:15,818000

Round-trip delay: 180,94 ms

Local clock offset: 10799268,16 ms

**----------*

----------* Mensaje Recibido **----------

Muestra: 8

NTP server: pool.ntp.org

Leap indicator: 0

Version: 3

Mode: 4

Stratum: 2

Poll: 3

Precision: -24 (6E-8 seconds)

Root delay: 0,23 ms

Root dispersion: 2,27 ms

Reference identifier: 200.160.7.186

Reference timestamp: 10-oct-2018 03:27:40,864933

Originate timestamp: 10-oct-2018 00:28:15,823000

Receive timestamp: 10-oct-2018 03:28:15,181661

Transmit timestamp: 10-oct-2018 03:28:15,181688

Dest. timestamp: 10-oct-2018 00:28:16,004000

Round-trip delay: 180,97 ms

Local clock offset: 10799268,17 ms

8. Para cada afirmación determine si es verdadera o falsa:

d) En los criptosistemas simétricos no puede garantizarse el no repudio porque ambas partes de latransacción conocen la clave utilizada.

Verdadero. No puede garantizarse el no repudio porque aún en caso de canal seguro el destino puede haber creado el mensaje en lugar del origen.

e) Si únicamente me importara la eficiencia del método que uso para encriptar, debería optar por un algoritmo de cifrado asimétrico.

Falso. El cifrado asimétrico requiere mayor potencia de cómputo. Además los mensajes cifrados con esta técnica resultan más largos.

f) Con ambos tipos de criptosistemas necesito contar con un mecanismo seguro para transmitir la clave.

Falso. En cifrado asimétrico no hay intercambio de claves secretas. Dados dos hosts A y B, A encripta y desencripta con su clave privada y B con la clave pública provista en el certificado de A.

9. Responder

a) ¿Que información es necesaria para que quien recibe un mail firmado, pueda verificar la firma del mismo?

La clave pública del emisor y el hash que viene en el certificado.

b) ¿Que información necesito para poder enviar un mail encriptado?

La clave publica del receptor, para que solo el pueda desencriptarlo.

c) ¿Que información es necesaria para que quien recibe un mail encriptado, pueda abrirlo? La clave privada del receptor, dado para que solo el receptor pueda leerlo.

El emisor encripto con la clave publica del receptor.

10. Evalúe las siguientes situaciones en el marco de una infraestructura de

PKI: Su clave privada fue comprometida, entonces:

a) ¿Qué consecuencias sufro respecto de la información firmada con la clave privada asociada

a mi certificado?

_ Ya que para un atacante es posible suplantar mi identidad, un MITM permitiría interceptar mis mensajes, afectar su integridad y reenviarlo al destino sin que éste pudiera notarlo. Por lo tanto se ven afectados los principios básicos autenticación, integridad y no repudio.

b) ¿Qué consecuencias sufro respecto de la información encriptada para que solo usted pueda ver?

_ Se vería toda comprometida dado que la información encriptada con mi clave pública solo se puede desencriptar con mi clave privada y ésta última ha sido comprometida.

c) ¿Qué acciones se pueden llevar a cabo? ¿cómo?

_ Solicitar a la autoridad de certificación la revocación del certificado y la extensión de uno nuevo con la respectiva generación del nuevo par de claves.

11. Transacciones:

a. Explicar cómo el protocolo de compromiso de dos fases para transacciones anidadas, se asegura que si la transacción de nivel superior se compromete, todas las descendientes se comprometen o se abortan.

_El protocolo de consolidación en dos fases, confirmación en dos fases o commit en dos fases, es un protocolo de consenso distribuido que permite a todos los nodos de un sistema distribuido ponerse de acuerdo para consolidar a una transacción. Típicamente es usado en bases de datos distribuidas. El objetivo del protocolo es que todos los nodos realicen un commit de la transacción o la aborten.

El algoritmo está basado en la existencia de un coordinador (al resto de nodos de la red se les llama participantes) y se divide en dos fases:

Fase de prepare: En la cual el coordinador intenta preparar a todos para el commit contactando con todas las réplicas que participan, las cuales contestan si aceptan o abortan la transacción.

Fase commit: Cuando todas las réplicas han respondido el coordinador busca los conflictos si los hay la transacción tiene que ser abotada, el acuerdo será abortado y el coordinador enviará un mensaje a todas las réplicas para que aborten la transacción.

Si no se detectan conflictos entonces el coordinador indica a todos realizar el commit. Después de que todas las réplicas realizan la consolidación, envían un mensaje de confirmación al coordinador.

Las dos fases del algoritmo son la «fase de petición de commit», en el cual el coordinador intenta preparar a todos los demás, y la «fase de commit» o consolidación, en la cual el coordinador completa las transacciones a todos los demás participantes.

b. Utilizando la API de Java para transacciones (JTA), hacer una aplicación que establezca conexión a dos diferentes bases de datos de postgres (banco1, banco2) para efectuar una transacción bancaria depositando un monto en el banco1 y extrayendo el dinero del bancos2 de una cuenta que debe ser especificada por el usuario.

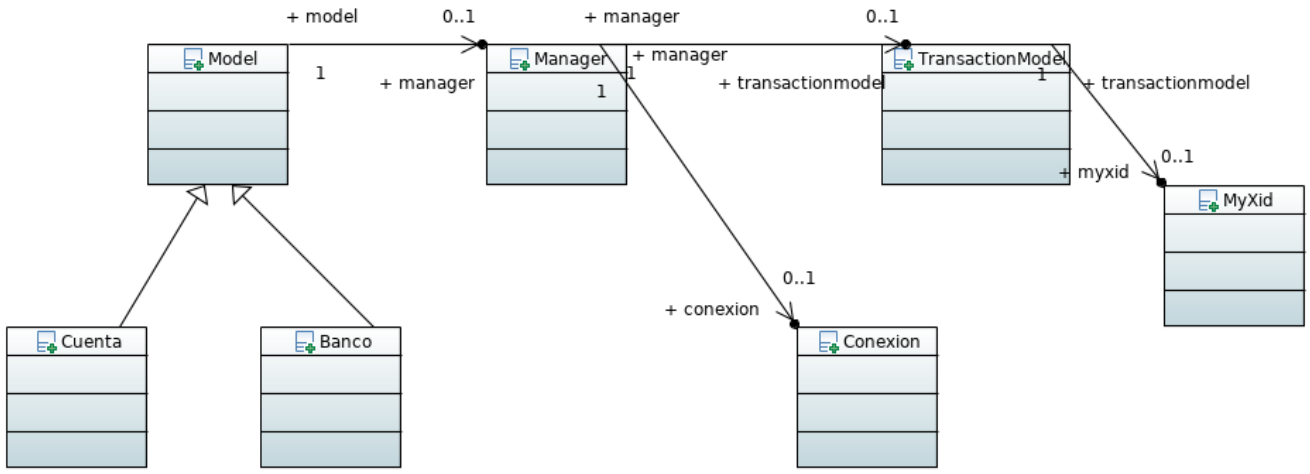
Los bases de datos a crear contendrán todas el mismo esquema consistente en una sola tabla:

Cuentas (id, titular, fecha_creacion, bloqueada (boolean), saldo (real))

(en el aula virtual se deja el script de creación dbCreate.sql de las 2 bases de datos con las mismas cuentas iniciales y diferentes saldos).

Utilizar como punto de partida los fuentes de la carpeta jta del aula virtual, que permite establecer una conexión jdbc a postgres y generar una transacción JTA con un solo branch a la base banco1)

El diseño de la solución desarrollada tiene esta forma:



Para poder realizar la transacción distribuída con el mismo XID es necesario que las bases de datos estén en 2 servidores diferentes (ya que los xid son por servidor y no por db). Se adjunta un archivo docker-compose.yml para poder ejecutar el ejemplo. En caso de querer ejecutarlo en un solo servidor descomentar la siguiente linea en el archivo main y comentar el if de abajo (además de corregir las direcciones ip de cada sitio):

```
//              if (cuentaOrigen.update() && cuentaDestino.update()) {
                if (cuentaOrigen.update() && cuentaDestino.update(cuentaOrigen.getTransactionId())) {
                    System.out.println("Valores resultantes de la operacion");
```

La idea es que todos los modelos heredan de Model, que utiliza reflection para asociarse con una tabla en la base de datos. La convención es que cada atributo del modelo tiene el mismo nombre que una columna en la base de datos y el nombre de la tabla es el nombre de la clase en plural.

De este modo al crearse un modelo, éste cuenta con un manager que será el encargado de gestionar la conexión y las transacciones. Una transacción puede instanciarse pasándole o no el XID, si no se lo pasa lo genera aleatoriamente.

Luego el modelo tiene los siguientes métodos:

```java
public boolean update() throws SQLException, XAException {
    return this.manager.update(this);
}

public void commit() throws UnknownTransactionException, XAException, SQLException {
    this.manager.commit();
}

public void rollback() throws UnknownTransactionException, XAException, SQLException {
    this.manager.rollback();
}
```

Que en el manager tienen la siguiente forma:

```java
public boolean update(Model model) throws SQLException, XAException {

    try {

        this.transaction = new TransactionModel(this.conexion, model.getUpdateQuery());
        return this.transaction.prepare();

    } catch (IllegalArgumentException | IllegalAccessException | NoSuchFieldException | SecurityException e) {

        e.printStackTrace();
        return false;
    }

}

public void commit() throws UnknownTransactionException, XAException, SQLException {
    if (this.transaction == null)
        throw new UnknownTransactionException("No hay ninguna transacción");
    this.transaction.commit();
    this.transaction = null;
}

public void rollback() throws UnknownTransactionException, XAException, SQLException {
    if (this.transaction == null)
        throw new UnknownTransactionException("No hay ninguna transacción");
    this.transaction.rollback();
    this.transaction = null;
}
```