Instituto Tecnológico de Aeronáutica
Engenharia Aeroespacial e Mecânica

CE-288
Programação Distribuída

Professor: Dr. Celso Massaki Hirata

Resolução da Serie de Exercícios nº2

Lucas Kriesel Sperotto

30 de Setembro de 2011

Primeira Parte: Java

1) Dê exemplo de como implementar os métodos *suspend()* e *resume()* para Threads sem usar os métodos que foram "deprecated" (entraram em desuso):

Os métodos *suspend() e resume()* estão sujeitos a "deadlock", como no código abaixo:

```
private boolean threadSuspended;
…
if (threadSuspended)
        resume();
else
        suspend();  // DEADLOCK-PRONE!

threadSuspended = !threadSuspended;
```

Para evitar seu uso estes métodos foram "deprecated", não podendo o programador invoca-los. Uma alternativa é fazer uso dos métodos *wait()* e *notify()* da seguinte forma:

```
private boolean threadSuspended;
…
//Dentro de um método synchronized
threadSuspended = !threadSuspended;
if (!threadSuspended)
        notify();
…
//Dentro do método run()
synchronized(this) {
        while (threadSuspended)
                wait();
}
```

O método *wait()* deve estar dentro de uma clausula "**try...catch**" com tratamento para "**InterruptedException**", além disso a chamada de *notify()* deve estar em um método "**synchronized"**.

2) Por que o código abaixo deve estar dentro de um método sincronizado?

```
While (!condition)
        wait();
```

Isto é necessário pela linguagem, e garante que *wait()* e *notify()* são devidamente serializados. Em termos práticos, isso elimina as condições de corrida em que uma thread em estado de espera não receberia um *notify()* e continuam suspensas por tempo indeterminado.

Uma alternativa para se evitar o uso explicito de sincronização é utilizar um parâmetro do tipo **volatile** para garantir imediata comunicação do pedido para *suspend()*, como no código abaixo:

```
private boolean volatile threadSuspended;
…
//Dentro do método run()

if (threadSuspended) {
        synchronized(this) {
                while (threadSuspended)
                        wait();
        }
}
```

3) O que são applets? Quais as restrições de uma applet em execução? Qual a hierarquia da classe Applet? Quais os métodos de Applet vistos em aula?

Applets são programas JAVA que podem ser embutidos em documentos HTM e podem ser executados em um host remoto (cliente). Quando o navegador carrega uma pagina WEB que contém um *applet*, o *applet* é baixado para o navegador e começa a ser executado. Applets possuem algumas restrições de segurança, entre elas:

1) Um *applet* não pode carregar biblioteca ou definir métodos nativos do sistema operacional; 2) Não pode ler ou escrever arquivos no cliente onde está sendo executado (exceto *applets* assinadas); 3) Não pode fazer conexões de comunicação exceto para o host de origem (servidor); 4) Não pode iniciar programas; 5) Não pode ler algumas propriedades do sistema;

Quanto a sua hierarquia de classes, podemos destacar o diagrama de heranças seguinte:

java.lang.Object
  ∟ java.awt.Component
      ∟ java.awt.Container
          ∟ java.awt.Panel
              ∟ **java.applet.Applet**

A classe Applet possui implementação das seguintes interfaces: Accessible, ImageObserver, MenuContainer, Serializable. E como principal subclasse temos a classe JApplet. Para finalizar a hierarquia vamos descrever um pouco suas superclasses.

Component: classe básica para representação gráfica de componentes em tela com interação com o usuário. Contém variáveis que representam a localização, forma, aparência geral e estado do objeto, bem como métodos para desenhar, como paint() e repaint(), e manipular objetos. Suas principais classes derivadas são: Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

Container: como o próprio nome diz, é um contêiner que contém componentes AWT entre outros. Quando se adiciona um componente ao contêiner, ele cria uma lista de componentes filhos e métodos para agrupá-los e arranjá-los. Por exemplo, o método

add(Button) adiciona o componente Button no final da lista de componentes do objeto do tipo Container de onde o método é chamado.

Panel: é uma classe recipiente simples. Um painel fornece o espaço em que um aplicativo pode conectar outros componentes, incluindo outros painéis.

Agora vamos descrever os principais métodos vistos da classe Applet:

public void init ():

Chamado pelo browser ou visualizador do applet uma única vez quando o applet é carregado para a execução.

public void start():

Chamada pelo browser ou visualizador do applet depois do método *init()*, para informar que a applet deseja iniciar sua execução (chamado sempre que a applet torna-se vizivel).

public void stop():

Chamada pelo browser ou visualizador do applet para informar que o applet deseja parar sua execução (chamado sempre que a applet torna-se não-vizivel). Serve para aliviar a carga de processamento da CPU.

public void destroy():

Chamada pelo browser ou visualizador do applet quando o usuário sai da seção de navegação, esse método desaloca recursos do applet, encerra conexões de comunicação etc.

public void paint(Graphics g):

É chamado para desenhrar a applet na tela.

public void repaint():

Escalona o método *paint()* para execução. Redefine, redesenha o applet através do método *update()*.

4) Como se faz para tratar eventos de mouse dentro de objeto Applet?

A classe MouseEvent fornece métodos para o tratamento de eventos relacionados ao Mouse. Para utilizar os eventos do mouse o programador deve implementar uma (ou mais, dependendo do tipo de evento) dessas interfaces :

MouseListener: para quando os botões do mouse forem clicados em um componente. Devemos adicionar esse evento com a chamada do método **addMouseListener(this)**;

MouseMotionListener: para quando o mouse for movimentado sobre um componente. Devemos adicionar esse evento com a chamada do método **addMouseMotionListener(this)**;

MouseWheelListener: para quando o botão rotatório do mouse for usado sobre um componente. Devemos adicionar esse evento com a chamada do método **addMouseWheelListener(this)**;

Com a implementação dessas interfaces, devemos sobrescrever métodos que permitem utilizar os eventos do mouse. Por exemplo mouseReleased(), mouseClicked(). Os eventos são passados para o método, podendo o programador utilizar o evento em cada uma das situações esperadas.

5) O que se deve ter em código Java no servidor e no cliente para estabelecer uma comunicação TCP/IP? De um exemplo.

Do lado do servidor, precisamos criar um ServerSocket passando o numero da porta no construtor, e um Socket para aguardar a conexão do cliente. Através do socket podemos criar dous fluxos de dados, um para entrada e outro para saída (in, out). Exemplo pode ser visto no código a seguir, não mostrarei a manipulação dos dados:

```
private ServerSocket server;
private InputStream in;
private OutputStream out;
…
//Dentro de um método:

server = new ServerSocket(PortRef);

Socket aClient = server.accept();
in = aClient.getInputStream();
out = aClient.getOutputStream();
```

Do lado do cliente, devemos criar um socket com o hostname do servidor e o numero da porta. Através desse socket podemos criar dois fluxos de dados, um para entrada e outro para saída. Exemplo no código a seguir:

```
private Socket client;
private InputStream in;
private OutputStream out;
…
//Dentro de um método:

client = new Socket(HostName, PortRef);

in = client.getInputStream();
out = client.getOutputStream();
```

6) O que se deve ter em código Java no servidor e no cliente para estabelecer uma comunicação UDP? De um exemplo.

Do lado do servidor, precisamos criar um "DatagramaSocket" especificando a porta que será usada, e um "DatagramaPacket" especificando o dado (um Array de bytes) e seu tamanho. Exemplo no código que segue:

```
private DatagramSocket serverSocket;
private DatagramPacket receivePacket;
…
//Dentro de um método:

serverSocket = new DatagramSocket(PortRef);
receivePacket = new DatagramPacket(receiveData,
receiveData.length);
serverSocket.receive(receivePacket);
```

Já para o cliente necessitamos criar um "DatagramaSocket" com o construtor vazio, e um "DatagramaPacket" com para enviar as informações, o DatagramaPacket deve ter além das informações sobre os dados, o endereço e a porta para onde o pacote será enviado. Exemplo no código a seguir:

```
private DatagramSocket clientSocket;
private InetAddress IPAddress;
private DatagramPacket sendPacket;
…
//Dentro de um método:

clientSocket = new DatagramSocket();
IPAddress = InetAddress.getByName (serverHostname);
sendPacket = new DatagramPacket(sendData, sendData.length,
IPAddress, PortRef);
clientSocket.send(sendPacket);
```

7) Descreva as declarações que são necessárias para se ter uma implementação RMI. Justifique a descrição.

O RMI (Remote Method Invocation) facilita o desenvolvimento de aplicações distribuídas. Essa tecnologia permite que o programador invoque métodos de objetos remotos, ou seja, permite que objetos Java em hosts diferentes comuniquem-se entre si, sendo que um dado cliente ao invocar um método contido em um servidor, a execução desse método ocorre no servidor.

Para isso ser possível, temos que declarar uma interface remota. E implementa-la em outra classe. Nesta interface declaramos os métodos que serão chamados remotamente. Sendo que a comunicação é transparente ao programador (o RMI cuida dessa parte), para satisfazer a comunicação o RMI é dividido em camadas como mostra a figura 1.

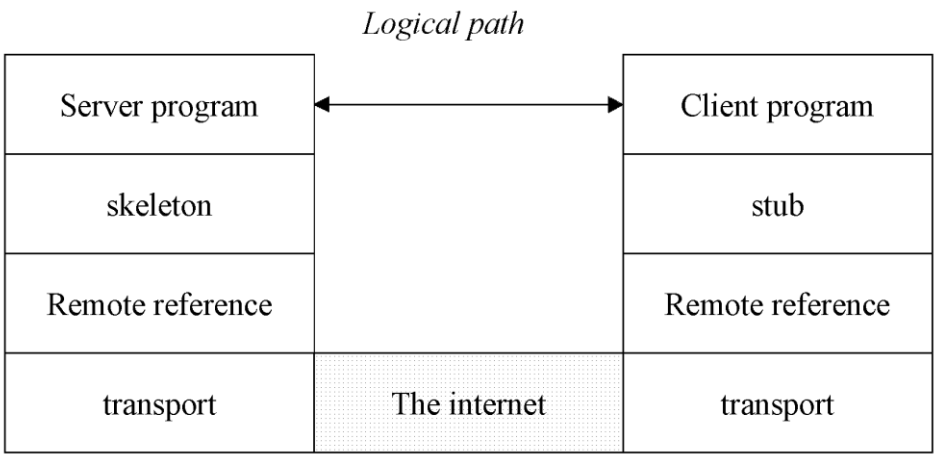


Figura 1 - Camadas RMI

Como vemos na figura 1, necessitamos de duas aplicações, uma cliente (que usará o método) e uma servidora (disponibiliza o método), ambas as aplicações se comunicam-se diretamente com as camadas Skeleton e Stub. Os Stub's são classes usadas do lado da aplicação do cliente e funcionam como Proxies entre a aplicação cliente e o objeto remoto. Os Stubs recebem os parâmetros dos métodos exportados pelo objeto remoto (definidas pela interface da classe remota) e reencaminham-nos para o lado do servidor onde serão interpretados por uma instância de uma classe Skeleton.

O Skeleton recebe os parâmetros enviados pelo Stub e executa as respectivas chamadas no objeto remoto. Em sentido inverso, os Skeletons são também responsáveis por receber o valor de retorno do método remoto (local na sua perspectiva) e direcioná-los para os Stubs dos clientes correspondentes.

As camadas de referencia remota mantém as referencias entre os clientes e os objetos remotos e estabelece a semântica da ligação RMI. Esta camada funciona como um "router" entre o cliente e os vários objetos remotos. A camada de transporte cria a comunicação (TCP/IP) que conecta as maquinas virtual.

Podemos ver nos códigos seguintes, uma aplicação que calcula a área e o perímetro de um retângulo. O primeiro passo é a definição da interface do objeto remoto, esta interface deve herdar da classe java.rmi.remote e cada método declarado deve indicar o envio de exceções do tipo RemoteException (relacionadas a falhas de comunicação de rede).

Podemos ver no código a seguir a interface remota para realizar duas operações matemáticas:

```
import java.rmi.*;

public interface InterfaceServidorMat extends Remote{

public double soma(double a, double b) throws RemoteException;
public double multiplica(double a, double b) throws
RemoteException;
}
```

Tendo definido a interface, o próximo passo é declarar os métodos em uma outra classe. Esta classe além de implementar a interface do objeto remoto, deve herdar da classe UnicastRemoteObject que realiza a ligação com o sistema RMI. Como na interface, seus métodos devem informar exceção do tipo RemoteException. Podemos ver o código dessa classe a seguir:

```
import java.rmi.*;
import java.rmi.server.*;
public class ServidorSoma extends UnicastRemoteObject
implements InterfaceServidorMat{

        public double soma(double a, double b) throws
RemoteException{
                return a+b;
        }
        public double multiplica(double a, double b)
throws RemoteException{
                return a*b;
        }
}
```

Com o objeto remoto definido, vamos criar uma aplicação servidora. Esta aplicação tem como objetivo criar o objeto remoto e disponibilizá-lo no registo RMI. O registro é feito através do método Naming.rebind() que recebe como parâmetro o nome pelo qual objeto remoto deverá ficar conhecido e a referencia do próprio objeto remoto. Podemos ver detalhes no código seguinte:

```
import java.rmi.*;

public class StartServidor{
        public static void main(String argv[]){
                try{
                        Naming.rebind("ServidorMat_1", new
ServidorMat());
                }catch (Exception e){
                        System.err.println(e.toString());
                }
        }
}
```

Por ultimo criamos a aplicação cliente, ela usará os métodos do objeto remoto para realizas o cálculo da área e do perímetro de um retângulo, para invocar os métodos remotos, o cliente deve consultar o registro RMI, isso é feito através do método Namming.lookup(), que tem como parâmetro a identificação em URL RMI do objeto remoto, este identificador tem a seguinte forma:

rmi://<Servidor onde Corre o Serviço de Registos>[:<porta de rede (opcional)>]/<nome do serviço remoto>

Podemos ver no código seguinte a aplicação cliente e a chamada do objeto remoto e seus métodos:

```
import java.rmi.*;

public class Cliente {

    private InterfaceServidorMat msi;
    public Cliente() {
        try {
            msi = (InterfaceServidorMat)
Naming.lookup("rmi://127.0.0.1/ServidorMat_1");
        } catch (Exception e) {
            System.err.println(e);
            System.exit(0);
        }
    }

    public double area(double a, double b) throws
RemoteException {
        return msi.multiplica(a, b);
    }

    public double perimetro(double a, double b) throws
RemoteException {
        double metade = msi.soma(a, b);
        return msi.multiplica(2.0, metade);
    }

    public static void main(String[] argv) {
        Cliente c = new Cliente();
        try {
            System.out.println("Area: " + c.area(20.0,
40.0));
            System.out.println("Perimetro: " +
c.perimetro(20.0, 40.0));
        } catch (Exception e) {
            System.err.println(e);
        }
    }

}
```

A execução é feita compilando os fontes com o comando javac, e depois para criar o os Stub's e Skeleton usamos o comando rmic <classe que implementa a interface>, chamamos o registro com o comando start rmiregistry e executamos com o comando java o cliente e o servidor.

Segunda parte: Algoritmos

1) Descreva uma implementação do comando SELECT RECIVE do exemplo BOUNDEDBUFFER na apostila Java.

   Podemos ver no código que segue uma possível implementação:

```
    public class BoundedBuffer {

    private UDPConnection putChar;// ENTRYPORT putchar: char
REPLY signaltype ;
    private UDPConnection getChar;//ENTRYPORT getchar: signaltype
REPLY char ;
    private static final int poolsize = 100;
    private static char[] pool = new char[poolsize];
    private static int inp, outp;//: 1..poolsize;
    private static int count;// : 0..poolsize;
    private static int signal = 1;
    private static Thread Send;
    private static Thread Reciv;

    public BoundedBuffer() {
        Send = new Thread(new OutBuffer());
        Reciv = new Thread(new InBuffer());
        putChar = new UDPConnection(0);
        getChar = new UDPConnection(1);
        inp = 1;
        outp = 1;
        count = 0;
        Send.start();
        Reciv.start();
    }

    public synchronized void setCount(int count) {
        this.count = count;
    }

    public synchronized int getCount() {
        return this.count;
    }

    public synchronized void setPool(char pool) {
        this.pool[inp] = pool;
    }

    public synchronized int getPool() {
        return this.pool[outp];
    }
```

```
    private class InBuffer extends Thread {
        public void run() {
            inOfBuffer();
        }

        public void inOfBuffer() {

            while (true) {
                if (getCount() < poolsize) {
                    //RECEIVE pool[inp] FROM putchar REPLY signal
                    setPool(putChar.recieve(signal));
                    inp = (inp % poolsize) + 1;
                    setCount(getCount() + 1);
                }
            }
        }
    }//final da classe InBuffer
    private class OutBuffer extends Thread {
        public void run() {
            outOfBuffer();
        }
        public void outOfBuffer() {

            while (true) {

                if (getCount() > 0) {
                    //RECEIVE signal FROM getchar REPLY pool[outp]
                    signal = getChar.recieve(getPool());
                    outp = (outp % poolsize) + 1;
                    setCount(getCount() - 1);
                }
            }
        }
    }//final da classe OutBuffer
}//final da classe BoundedBuffer
```

As classes “UDPConnection” são classes que criam uma conexão UDP dada uma porta de referencia. O código passado na apostila, possui a clausula “LOOP SELECT”, dessa forma, para implementar em Java usei duas Threads, que são as classes internas privadas (InBuffer e OutBuffer). Com essas duas threads, pode-se receber e enviar mensagens de forma concorrente.

Acredito que a implementação esta correta, pois esse algoritmo deve receber dados do tipo “char” (volume de dados definido em “poolsize”), armazena-los e envia-los em outra porta. Basicamente como o nome diz é um “buffer”. A variável “signal” é usada para confirmação da comunicação.

Os métodos “recieve(char)” e “recieve(int)”, enviam o dado passado no corpo da função e retornam o dado necessário, no caso o primeiro retorna o signal e o segunto retorna um char.

Tive que tomar algum cuidado para “setar” alguns parâmetros comuns as duas threads, para isso criei métodos synchronized o que garante acesso serial as variáveis e evita um erro no acesso e escrita de valores.

2) Complete o algoritmo de Misra (ping-pong) visto em aula. Explique com um exemplo porque não é permitida a ultrapassagem neste algoritmo.

Algoritmo de Misra (1983) completado de acordo com meu entendimento:

```
private int nbping = 1;
private int nbpong = -1;
private int token = 0;
private boolean pinghere = false, ponghere = false;
private Connection PortRef;

//Dentro de um método:
t = PortRef.recivPingPong();
if (t > 0) {
        pinghere = true;
        nbping = t;
        if (ponghere) {
                nbping = (nbping % 4) + 1;
                nbpong = (nbpong % 4) - 1;
        } else {
                if (token != nbping) {
                        token = nbping;
                } else {
                        nbping = (nbping % 4) + 1;
                        nbpong = -(nbping);
                        ponghere = true;//regenerou ele esta aki
                }
        }
} else {
        ponghere = true;
        nbpong = t;
        if (pinghere) {
                nbping = (nbping % 4) + 1;
                nbpong = (nbpong % 4) - 1;
        } else {
                if (token != nbpong) {
                        token = nbpong;
                } else {
                        nbpong = (nbpong % 4) + 1;
                        nbping = -(nbpong);
                        pinghere = true;//regenerou ele esta aki
                }
        }
}
if (pinghere){
        token = nbping;
        PortRef.sendPingPong(nbping);
        pinghere = false;
}
if (ponghere) {
        token = nbpong;
        PortRef.sendPingPong(nbpong);
        ponghere = false;
}
```

O algoritmo de Misra (1983) propõe uma modificação do algoritmo de “Token-Ring”, visando através de dois tokens (ping e pong) reestabelecer a perda de algum deles na comunicaçã. Inicialmente temos as variáveis “nbping” e “nbpong” iniciadas

com 1 e -1 respectivamente, a variável “token” com 0, “ponghere” e “pinghere” com “false”. A variável “token” guarda o valor do ultimo token (ping ou pong) que passou pelo cliente, de forma reconhecer se algum deles se perdeu na comunicação.

Quando uma mensagem chega, o primeiro “if” compara para saber se é o “ping” ou o “pong”, no caso de ser o “ping”, ele seta “true” na variável “pinghere” (para indicar que o ping esta presente). No segundo instante ele verifica se o “pong” também esta presente (sinal de que eles se encontraram) e incrementam os valores de “nbping” e “nbpong”, esse tratamento no incremento é citado por Misra (1983) para que os valores das variáveis não venham a crescer demasiadamente e explodam a capacidade de um inteiro (já os tokens não irão se encontrar mais de n vezes).

Quando o “pong” não se encontra ele verifica se o “ping” que chegou é diferente do ultimo “token” que passou pelo processo (comparando com a variável token), no caso de ser igual é sinal que o “pong” se perdeu no caminho e necessita ser recuperado. Logo depois de recuperado a variável seta que o pong se encontra ali e o algoritmo faz o envio do ping e depois do pong. Notamos que o algoritmo regenera a perda de um dos tokens após uma volta completa. Para o caso do recebimento do pong é análogo.

Dado esse funcionamento, percebemos que se houver ultrapassagem de mensagens durante a comunicação, o algoritmo irá registrar a perda de um dos tokens, o que causaria o aparecimento de mais um token circulando na rede.

3) Complete a resolução do exercício algoritmo de Exclusão Mútua de Lamport. Explique com um exemplo porque não é permitida a ultrapassagem no algoritmo de Lamport.

1. Desempenho: O algoritmo melhorado (Ricart e Agrawala 1981) possui um melhor desempenho, dado o menor tráfego de mensagens na rede. Lamport 3(n-1) melhorado 2(n-1) e também as mensagens possuem um tamanho menor;

2. Ultrapassagem de mensagens é permitida?

Não, é necessária uma modificação no algoritmo. Para que possa ter ultrapassagem de mensagens deve-se adicionar uma mensagem de confirmação para pedidos de request e nela adicionar uma lista de destinatários.

Exemplo: Quando o nó 1 está realizando o processamento de seções críticas relacionadas com o seu pedido com número de seqüência 1, o nó 2 decide emitir uma mensagem de request com número de seqüência 2. Antes que a mensagem request chegar ao nó 1, nó 1 completa o processamento da seção crítica e transmite a mensagem reply aos outros nós. Sem uma lista de destinatários, o nó 2 pode pensar que a mensagem reply se aplica a sua mensagem de request e continuar. Na verdade, o nó 1 poderá fazer um novo request com número de seqüência 2 e ter direito a entrar em sua seção crítica primeiro, devido à regra de desempate.

3. Dê uma prova informal de que o algoritmo funciona: Assumindo que Pi e Pj estão acessando recurso. Considere as mensagens trocadas antes de isto ocorrer: cada processo transmitiu "request" e recebeu "reply" do outro.

Pi enviou "reply" ao "request" de Pj antes de selecionar o valor de c para sua mensagem. Portanto o valor de sua mensagem será maior que "request" de Pi. Então quando Pj recebe request de Pi, requestedi == true ⇒ Pj teria adiado reply.

Os dois enviaram reply após transmitir request. Então requested == true quando recebe request de outro processo. Cada processo vai comparar timestamps. Como existe ordem total, apenas um passará pelo comando if. Ou seja, um processo adia, o outro ganha acesso.

4. Tolerância à falha de processo:

Na prática alguns nós podem falhar e não vão responder as mensagens dirigidas a eles. Para evitar essa situação de parar o algoritmo de exclusão mútua, um mecanismo de "timeout" com recuperação podem ser adicionados. A detecção de tempo limite de um nó não responder se baseia no conhecimento de um limite superior sobre o tempo que pode decorrer antes de um nó de trabalho responde a uma mensagem e uma estimativa do tempo de processamento máximo dentro de uma seção crítica.

A única mensagem no algoritmo original, que exige uma resposta é a mensagem de request. Um nó requerente deverá iniciar um temporizador quando as mensagens request são enviadas. O temporizador deve ser reiniciado quando uma resposta é recebida e cancelado quando o processamento de seção crítica começa.

5. Dê uma descrição mais detalhada do algoritmo.

Podemos ver o algoritmo abaixo:

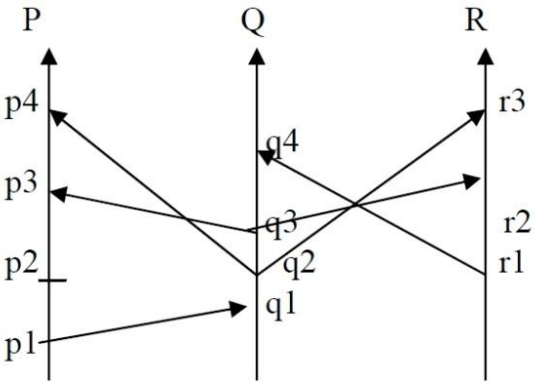
```
int osn=0, hsn=0;
int numrepexpected;
bool requested = false, repdeferred[N-1] = {false, ...};
//para pedido de exclusão
requested = true;
osn = hsn + 1;
numrepexpected = N − 1;
forall j ≠ i
        send (req, osn, i) to j;
wait ( numrepexpected == 0 );
< acesso exclusivo >
requested = false;
forall j ≠ i
        if ( repdeferred[j] ) {
                repdeferred[j] = false;
                send (rep) to j;
        }
// se receber request
when receive (req, k, j):
        hsn = max (hsn, k) + 1;
        if ( requested && ( osn < k | | (osn == k && i < j) )
                repdeferred[j] = true;
        else
                send (rep) to j;
// se receber reply
when receive rep:
        numrepexpected − −;
```

O processo Pi necessita exclusão, então envia um broadcast com o pedido “request” (req). Pj recebendo request pode enviar mensagem de “reply” (rep) imediatamente ou adiar envio. Quando Pi receber reply de todos pode acessar recurso. Ao terminar o processamento na exclusão, Pi envia “reply” adiados. Resumindo:

• Pi não quer acesso ⇒ envia reply

• Pi quer acesso ⇒ compara timestamp de seu próprio reply com o recebido; se o seu for mais antigo, decide adiar reply.

4) Rotule o diagrama abaixo com:



(a) valores dos relógios lógicos (inicialmente 0);

p4[6];	q4[5];	
p3[5];	q3[4];	r3[6];
p2[2];	q2[3];	r2[5];
p1[1];	q1[2];	r1[1];
[0]	[0]	[0]

(b) Tempo de Vetor (inicialmente [0,0,0]).

p4[4, 3, 0];	q4[1, 4, 1];	
p3[3, 3, 0];	q3[1, 3, 0];	r3[1, 3, 3];
p2[2, 0, 0];	q2[1, 2, 0];	r2[1, 3, 2];
p1[1, 0, 0];	q1[1, 1, 0];	r1[0, 0, 1];
[0, 0, 0];	[0, 0, 0];	[0, 0, 0];

5) Suponha a seguinte transação de banco de dados distribuídos: transferência de R$100,00 de uma conta A do Banco Alfa para uma conta B do Banco Beta. Suponha que os bancos têm seus sistemas instalados em cidades diferentes e a transação está sendo feita de uma terceira cidade e a conta A está com um saldo de R$ 300.

a) Descreva as operações de leitura e gravação necessárias nos dois locais:

A: Sa = Sa–Vt; **B:** Sb = Sb+Vt;
Sa = Saldo da conta A; Vt = Valor a transferir; Sb = Saldo da conta B;

A: R(A, Sa) < R(A, Vt) < W(A, Sa);
B: R(B, Sb) < R(B, Vt) < W(B, Sb);

b) Descreva a troca de mensagens usando Protocolo de Cometimento de Duas Fases para uma transação com sucesso:

Coordenador:
send prepare to Alfa;
send debit $100 from A to Alfa;
send prepare to Beta;
send credit $100 to B to Beta;
receive ready from Alfa;
receive ready from Beta;
send commit to Alfa;
send commit to Beta;
receive ack from Alfa;
receive ack from Beta;

Agente Alfa: (init A= 300)
receive prepare from coordinator;
receive debit $100 from coordinator
write < modify, A, 300, 200> to log
send ready to coordinator;
receive commit from coordinator;
update A;
send ack to coordinator;

Agente Beta: (init B= 000)
receive prepare from coordinator;
receive credit $100 from coordinator
write < modify, B, 000, 100> to log
send ready to coordinator;
receive commit from coordinator;
update B;
send ack to coordinator;

c) Mostre o conteúdo dos arquivos log para uma transação com sucesso:

Coordenador:

<global-begin>
<prepare, Alfa, Beta>
<global-commit>
<complete>

Agente Alfa:

<local-begin>
<modify, A, 300, 200>
<ready>
<commit>

Agente Beta:

<local-begin>
<modify, B, 000, 100>
<ready>
<commit>

d) Mostre o conteúdo dos arquivos log para uma transação com uma falha do host do Banco Beta depois de gravar o ready:

Coordenador:

<global-begin>
<prepare>
<global-abort>
<prepare>
<global-commit>
<complete>

Agente Alfa:

<local-begin>
<modify, A, 300, 200>
<ready>
<abort>
<modify, A, 300, 200>
<ready>
<commit>

Agente Beta:

<local-begin>
<modify, B, 000, 100>
<ready>
<abort>
<modify, B, 000, 100>
<ready>
<commit>

6) Verifique se as execuções distribuídas abaixo são serializáveis e caso elas sejam dê a sequência das transações.

a) Execução 1: S1: R(1,a) < R(3,d) < W(2,a);
S2: W(2,b) < R(1,e) < R(3,b);
S3: W(3,c) < R(2, f) < R(2,g) < R(1,c);

S1: T(1) < T(2);
S2: T(2) < T(3);
S3: T(3) < T(1);

Temos um deadlok nesta execução, pois T(1) depende de T(3) que depende de T(2) que depende de T(1). Uma possível ordenação parcial seria:

S1: R(3,d) < R(1,a) < W(2,a);
S2: R(1,e) < W(2,b) < R(3,b);
S3: R(2, f) < R(2,g) < W(3,c) < R(1,c);

b) Execução 2: S1: R(2,b)< R(2,c)< R(1,a) < W(1,b) < R(1,c) < R(3,c) < W(3,a);
S2: R(2,d) < R(2,e) < R(3,e) < W(1,e) < W(3,d);
S3: R(2,g) < R(2,h) < W(2,h) < W(1,g) < W(3,g);

S1: T(2) < T(1) < T(3);
S2: T(2) < T(3) < T(1);
S3: T(2) < T(1) < T(3);

Neste caso, teremos a conclusão da transação T(2) porém T(1) e T(3) ficarão bloqueadas.

S1: R(2,b) < R(2,c)< R(1,a) < R(1,c) < W(1,b) < R(3,c) < W(3,a);
S2: R(2,d) < R(2,e) < R(3,e) < W(3,d) < W(1,e);
S3: R(2,g) < R(2,h) < W(2,h) < W(1,g) < W(3,g);

Referências:

DEITEL, H. M. ;DEITEL, P. J. Java: Como Programar. 6ª Ed.Trad. Edson Furmankiewicz. São Paulo: Pearson Pretince Hall, 2005.

http://download.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html

http://download.oracle.com/javase/1.4.2/docs/api/java/applet/Applet.html

http://www.lac.inpe.br/~rafael.santos/Docs/JaVale/applets.pdf

http://download.oracle.com/javase/1.4.2/docs/api/java/awt/event/MouseMotionListener.html

http://download.oracle.com/javase/1.4.2/docs/api/java/awt/event/MouseEvent.html

http://www.ime.uerj.br/~alexszt/cursos/topesp_inter/trabs/992/g6/

http://paginas.fe.up.pt/~eol/AIAD/aulas/JINIdocs/rmi1.html

http://en.wikipedia.org/wiki/Vector_clock