

|  |
| --- |
| class RR { static int requestID\_G = 0;    int messageType;  int requestId;  RemoteRef remoteReference;  int operationId;  byte \* args;  // Constructor de Request RR (char \* args, RemoteRef & remoteReference, int opID){   this.messageType = 0; this.args = args; this.remoteReference = remoteReference;  if(requestID\_G == max(int))  requestID\_G = 0;  requestId = requestID\_G ;  requestID\_G++; this.operationId = opID; }    // Constructor de Reply   RR(RR request, RemoteRef & remoteReference){    this.messageType = 1;  this.remoteReference = remoteReference;    // Para saber qué es la respuesta a esa petición  this.requestId = request.requestId;   //Según el id de operación realiza la operación  operationID=request.operationID;   //arguments serán los resultados de la operación  // Se realiza la operación  }    byte \* marshalling(){  byte \* datos;  datos.append(obtenerBytes(messageType, int));  datos.append(obtenerBytes(requestID, int));  datos.append(obtenerBytes(remoteReference, RemoteReference));  datos.append(obtenerBytes(operationID, int));  datos.append(args);    return datos;  }   RR unmarshalling(byte [] msg){  RR m;  m.messageType = obtener(msg, sizeof(int),0);  m.requestID = obtener(msg, sizeof(int),sizeof(int));  m.remoteReference = obtener(msg, sizeof(RemoteReference), 2\*sizeof(int));  m.operationID = obtener(msg, sizeof(int), 2\*sizeof(int)+sizeof(RemoteReference));  m.args = obtener(msg,//Lo que queda hasta acabar el vector,3\*sizeof(int)+sizeof(RemoteReference));    return m;  } } |
|  |

**5.3 Dé un resumen de la implementación del servidor, mostrando cómo las operaciones getRequest() y sendReply() son utilizados por un servidor que crea un nuevo hilo para ejecutar cada solicitud del cliente.**

**Indique cómo el servidor copiará el ID de solicitud del mensaje de solicitud al mensaje de respuesta y cómo obtendrá la dirección IP y el puerto del cliente.**

1. El servidor inicia y se configura para escuchar en un puerto específico.
2. El servidor entra en un bucle infinito para procesar solicitudes de clientes:
   1. El servidor invoca getRequest para recibir un mensaje de solicitud de un cliente. getRequest bloquea hasta que llegue una solicitud.
   2. Una vez que se recibe la solicitud, el servidor extrae la información relevante del mensaje de solicitud, como requestId, la dirección IP del cliente, el puerto del cliente y la operación a realizar.
   3. El servidor crea un nuevo hilo para procesar la solicitud del cliente. Esto permite al servidor manejar múltiples solicitudes de clientes de manera concurrente.
   4. El hilo creado para la solicitud del cliente ejecuta la operación solicitada y obtiene el resultado.
   5. El hilo del servidor utiliza sendReply para enviar un mensaje de respuesta al cliente. En este mensaje de respuesta, se incluye el requestId original (copiado del mensaje de solicitud) para que el cliente pueda asociar la respuesta con su solicitud correspondiente. Además, el mensaje de respuesta incluye la dirección IP y el puerto del cliente para que el servidor sepa dónde enviar la respuesta.
   6. Una vez que se envía la respuesta, el hilo del servidor termina y se libera.
3. El servidor continúa procesando solicitudes en su bucle infinito.

**5.4 Definir una nueva versión del método doOperation() que establece un tiempo de espera para el mensaje de respuesta. Después de un tiempo de espera, retransmite el mensaje de solicitud n veces. Si aún no hay respuesta, informa a la persona que llama.**

|  |
| --- |
| public byte[] doOperationTimeout(RemoteRef s, int operationId, byte[] arguments, int timeout, int retries) {  int attempts = 0;  byte[] reply = null;  while (attempts < retries) {  // Enviar el mensaje de solicitud al servidor remoto  sendRequest(s, operationId, arguments);  try {  // Establecer el tiempo de espera para recibir la respuesta  reply = receiveReply(timeout);  // Si se recibe una respuesta, salir del bucle y devolverla  if (reply != null) {  break;  }  } catch (SocketTimeoutException e) {  // Si se produce un tiempo de espera, incrementar el contador de intentos y reintentar  attempts++;  }  }  // Si se agotan los intentos y no se recibe una respuesta, informar al llamador  if (attempts == retries && reply == null) {  throw new RuntimeException("No se ha recibido respuesta del servidor después de " + retries + " intentos");  }  return reply;  } |

**5.5 Describa un escenario en el que un cliente podría recibir una respuesta de una llamada anterior**

**Profesor:** *Utilización de RPC asíncrona, es decir, imaginemos que el cliente envía un mensaje que no va a querer requerir una respuesta del servidor inmediata, luego mas tarde cuando el cliente le envié una solicitud para otro mensaje que si lo requiera se le podrá devolver todo, un ejemplo:*

*Imaginemos una aplicación en la que un cliente realiza varias solicitudes a un servidor mediante RPC asíncrona para realizar operaciones de larga duración, como procesar grandes conjuntos de datos o realizar cálculos complejos. El cliente no espera a que se complete cada solicitud antes de enviar la siguiente, ya que las llamadas RPC asíncronas no bloquean al cliente.*

*Aquí hay un ejemplo de secuencia de eventos:*

1. *El cliente envía una solicitud A al servidor.*
2. *Antes de recibir la respuesta A, el cliente envía una solicitud B al servidor.*
3. *El servidor comienza a procesar la solicitud A, pero se demora debido a la carga de trabajo.*
4. *Mientras tanto, el servidor recibe la solicitud B y la procesa rápidamente.*
5. *El servidor envía la respuesta B al cliente.*
6. *Finalmente, el servidor completa el procesamiento de la solicitud A y envía la respuesta A al cliente.*

**5.6 Describir las formas en que el protocolo de solicitud-respuesta enmascara la heterogeneidad de los sistemas operativos y de las redes informáticas.**

Preguntar a tirado cuales eran.

Cuando se establece conexión entre distintos sistemas, no se tiene en cuenta el sistema operativo, ya que simplemente realizas la retransmisión de bytes, contando con el marshalling y unmarshalling, sin ser relevante la estructura del sistema operativo.

**5.7 Comenta si las siguientes operaciones son idempotente:**

i) presionar un botón de solicitud de ascensor (ascensor); **→ Idempotente**

ii) escribir datos en un archivo; **→ Idempotente**

iii) agregar datos a un archivo. **→ No es idempotente**

**¿Es condición necesaria para la idempotencia que la operación no esté asociada a ningún estado?**

Si es necesario, no podrías distinguir entre algunas operaciones.

**5.8 Explique las opciones de diseño que son relevantes para minimizar la cantidad de datos de respuesta almacenados en un servidor. Compare los requisitos de almacenamiento cuando se utilizan los protocolos RR y RRA.**

Para responder a esta pregunta, tenemos los fallos por omisión o cualquier otro fallo, que se puede corregir de tal manera que se utilice los Tiempos de Espera, Filtrado de duplicados, volver a enviar solicitudes remitentemente, historial de respuestas con tiempo de vida y las cookies?

\*\* Pregunta al tío\*\*

**5.9 Suponga que el protocolo RRA está en uso. ¿Cuánto tiempo deben conservar los servidores los datos de respuesta no reconocidos? ¿Deberían los servidores enviar repetidamente la respuesta en un intento de recibir un acuse de recibo?**

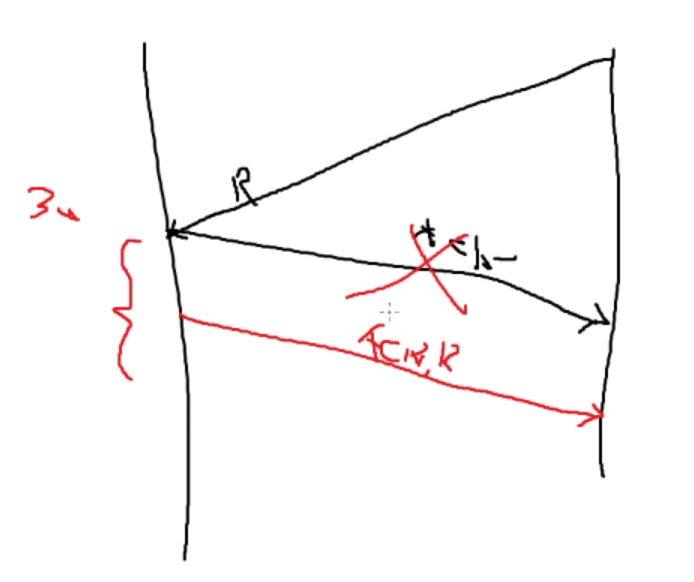
Tenía que mantener la información hasta que el cliente realice un request distinto o hasta que le llegase el ack, pero podría haber un tiempo de espera (time out) hasta poder eliminarlo.

No debería estar indefinidamente, ya que si el cliente ha caído siempre va a estar enviando la respuesta, así que debería tener también un timeout.

**5.10¿Por qué la cantidad de mensajes intercambiados en un protocolo podría ser más importante para el rendimiento que la cantidad total de datos enviados?**

**Diseñe una variante del protocolo RRA en la que el acuse de recibo se superponen (es decir, se transmita en el mismo mensaje que) a la siguiente solicitud cuando corresponda y, de lo contrario, se envíe como un mensaje separado. (Sugerencia: use un temporizador adicional en el cliente).**

Es más importante enviar la menor cantidad de mensajes, ya que se pierde mucho tiempo a la hora de enviar los mensajes, por lo que será más eficiente enviar menos mensajes con mayor cantidad de datos que al revés.



El cliente esperará por cada conexión que deba hacer con el servidor un periodo de tiempo determinado, por si en ese periodo se da que tiene que mandarle más mensajes al servidor mandarlo todo junto, en vez de mandarlos por separados, siendo así más eficiente.

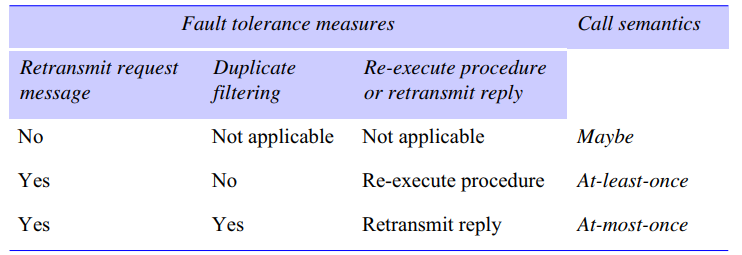
**5.11La interfaz “Elección” proporciona dos métodos remotos:**

**votar: Este método tiene dos parámetros a través de los cuales el cliente proporciona el nombre de un candidato (una cadena) y el 'número de votante' (un número entero que se utiliza para garantizar que cada usuario vote una sola vez). Los números de los votantes se asignan escasamente del rango de números enteros para que sean difíciles de adivinar.**

**Resultado: Este método tiene dos parámetros a través de los cuales el servidor proporciona al cliente el nombre de un candidato y el número de votos para ese candidato.**

**¿Cuáles de los parámetros de estos dos procedimientos son “input” y cuales son “output” parámetros?**

Los 3 primeros parámetros “input” y el número de votos “output”.



**5.12. Discuta la semántica de invocación que se puede lograr cuando el protocolo de solicitud-respuesta se implementa sobre una conexión TCP/IP, lo que garantiza que los datos se entregan en el orden en que se envían, sin pérdida ni duplicación. Tenga en cuenta todas las condiciones que provocan la interrupción de una conexión.**

Contando con el filtro de duplicación del TCP/IP solo podríamos reproducir la semántica de invocación de como mucho uno, ya que al tener ACK tanto del cliente al servidor como del servidor al cliente vas controlando que lleguen los mensajes, filtrando así duplicados a su vez.

**5.14. El servicio “Elección” debe asegurarse de que se registre un voto cada vez que un usuario crea que ha emitido un voto.**

**Discutir el efecto de “tal vez” semántica de llamadas en el servicio “Elección”.**

**“Al menos una vez” la semántica de llamadas sea aceptable para el servicio “Elección” o recomendaría como “máximo una vez” llamar semántica?**

Se recomienda el “máximo una vez” ya que al tratarse de un sistema que recoge los votos si implementamos otra semántica de invocación podríamos estar contando más veces el número de votos para un candidato, dando así un “pucherazo”.

**5.15 Se implementa un protocolo de solicitud-respuesta sobre un servicio de comunicación con fallos de omisión para proporcionar una semántica de invocación al menos una vez. En el primer caso, el implementador asume un sistema distribuido asíncrono. En el segundo caso, el implementador asume que el tiempo máximo para la comunicación y la ejecución de un método remoto es T. ¿De qué manera esta última suposición simplifica la implementación?**

Ajustándonos al tiempo real de ejecución del método remoto, estamos ajustando a su vez el tiempo del timeout.

Simplificación del control de flujo: Al tener un tiempo máximo T para la comunicación y la ejecución de un método remoto, el control de flujo entre el cliente y el servidor puede simplificarse.

En lugar de lidiar con la incertidumbre de los tiempos de comunicación y respuesta, el sistema puede adaptarse a un ritmo de envío y recepción de mensajes más predecible, lo que facilita la implementación de mecanismos de control de flujo.

**5.16 Esboza una implementación para el servicio de Elecciones que garantice que sus registros permanezcan consistentes cuando es accedido simultáneamente por múltiples clientes. página 199**

Sincronización: Utiliza mecanismos de sincronización, como bloqueos o semáforos, para controlar el acceso concurrente a los datos compartidos (en este caso, los registros de votos). Esto garantizará que solo un cliente a la vez pueda modificar los registros, evitando condiciones de carrera e inconsistencias en los datos.

**5.23 Diseñe una tabla de objetos remotos que pueda admitir la recolección de basura distribuida, así como la traducción entre referencias de objetos locales y remotos. Dé un ejemplo que involucre varios objetos remotos y proxies en varios sitios para ilustrar el uso de la tabla. Muestre los cambios en la tabla cuando una invocación provoca la creación de un nuevo proxy. Luego, muestre los cambios en la tabla cuando uno de los proxies se vuelve inaccesible. página 215**

**5.24 Una versión más simple del algoritmo de recolección de basura distribuida descrito en la Sección 5.4.3 simplemente invoca addRef en el sitio donde vive un objeto remoto cada vez que se crea un proxy y removeRef cada vez que se elimina un proxy. Describa todos los posibles efectos de las fallas de comunicación y procesos en el algoritmo. Sugiera cómo superar cada uno de estos efectos, pero sin utilizar arrendamientos (leases).**

**Problemas de chaty de paso de mensajes:**

**Un problema de cálculo distribuido:**

**Se tienen n procesos que quieren calcular la suma de m números.**

**Cada proceso tiene un conjunto de números, representado como un array de tamaño k.**

**Los procesos deben enviar mensajes para intercambiar sus números, de manera que al final cada proceso tenga la suma total de todos los números.**

**Los procesos pueden enviar y recibir mensajes a través de un único canal de comunicación.**

n procesos

m numeros

// Variables de procesos

double \* numeros;

double suma;

double suma\_total;

double suma\_procesos;

// funciones de comunicación

funciones = []

for(int i=0;i<n;i++)

{

send numeros[i](numeros);

receive numeros[i](suma\_procesos);

suma\_total += suma\_procesos;

}

int n\_pasados, n\_recibidos= 0;

//canales comienzo\_com[1..n](), recibo\_num[1..n](numero), envio\_num[1..n](numero)

**Diseñar un sistema de mensajería instantánea entre usuarios utilizando primitivas de paso de mensajes utilizando distintos canales de comunicación.**

**Diseñar un sistema de mensajería instantánea entre usuarios en un sistema distribuido. Los usuarios pueden conectarse, desconectarse y enviar mensajes a otros usuarios conectados. El servidor debe mantener un registro del estado de conexión de cada cliente y se encargará de reenviar los mensajes a los destinatarios adecuados. Utilice primitivas de paso de mensajes y distintos canales de comunicación para llevar a cabo este diseño.**

type clase\_op = enum (CONECTAR, DESCONECTAR, ENVIAR\_MENSAJE);

port peticion (int, clase\_op, int, string)

chan respuesta 1..n;

Cliente[i:1..n]::

var destino:int; mensaje:string;

// Conectar al sistema

send peticion (i, CONECTAR, -1, "");

receive respuestai;

// Enviar un mensaje a otro usuario

send peticion (i, ENVIAR\_MENSAJE, destino, mensaje);

// Desconectar del sistema

send peticion (i, DESCONECTAR, -1, "");

// Código del Servidor

type cliente\_info = record

id: int;

estado: bool;

var clientes: array [1..n] of cliente\_info;

id\_cliente: int;

clase: clase\_op;

destino: int;

mensaje: string;

Servidor:

begin

for i in 1..n do clientes[i].estado := false;

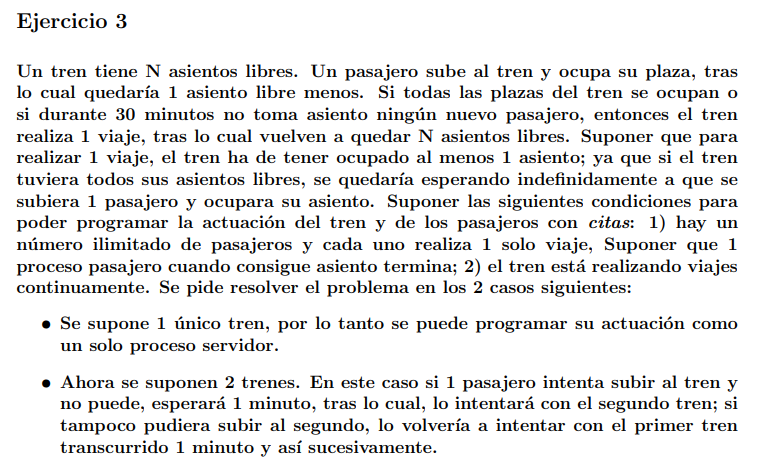
// Inicializar los estados de los clientes como desconectados

Solucion:

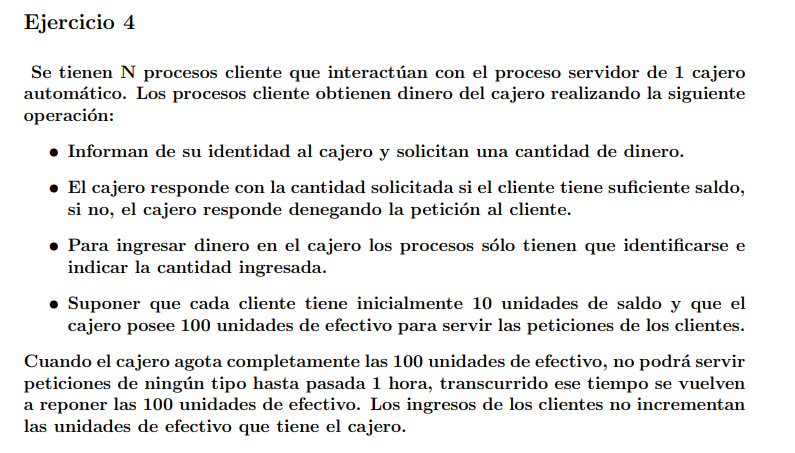
|  |
| --- |
| while(true){ receive peticion(id\_cliente, tipo,destino, contenido); if(tipo==CONECTAR){ clientes[id\_cliente]=true; } else if(tipo==DESCONECTAR){ clientes[id\_cliente=false; } else if(tipo == ENVIAR\_MENSAJE){ if(clientes[destino]==true){ send respuesta[destino](mensaje); } } } |
|  |

|  |
| --- |
| while (true) do begin  receive peticion(id\_cliente, clase, destino, mensaje);  case clase of  CONECTAR:   begin  clientes[id\_cliente].estado := true;  clientes[id\_cliente].id := id\_cliente;  send respuesta[id\_cliente]("Conectado");  end;  DESCONECTAR:  begin  clientes[id\_cliente].estado := false;  send respuesta[id\_cliente]("Desconectado");  end;  ENVIAR\_MENSAJE:  begin  if (clientes[destino].estado) then // Si el destinatario está conectado  send respuesta[destino]("Mensaje de " + toString(id\_cliente) + ": " + mensaje);  else  send respuesta[id\_cliente]("El destinatario no está conectado.");  end;  endcase; end; |

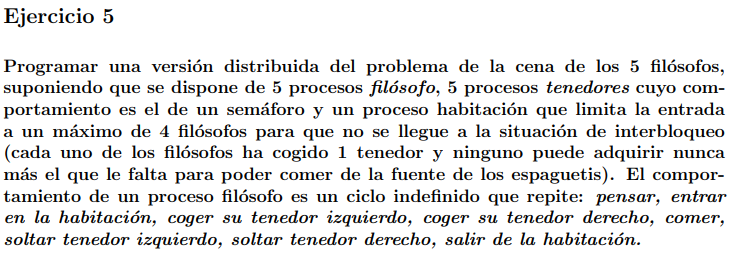
En este ejercicio, se diseñó un sistema de mensajería instantánea entre usuarios utilizando primitivas de paso de mensajes con distintos canales de comunicación. Los clientes pueden conectarse, desconectarse y enviar mensajes a otros usuarios que estén conectados. El servidor mantiene un registro del estado de conexión de cada cliente y se encarga de reenviar los mensajes a los destinatarios adecuados.



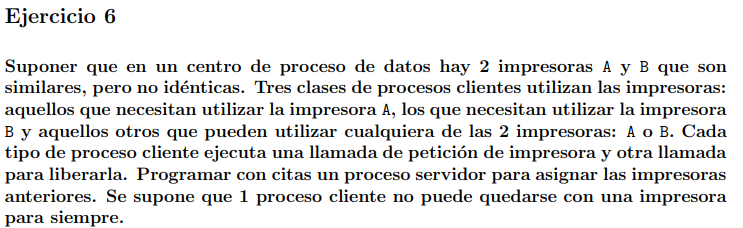
|  |
| --- |
| Tren:: int contador=0; const int N\_PLAZAS;  // Servidor (Tren)  while (true){  if(cronometro >= 30min && contador>0 || contador == N\_PLAZAS){  //Se marcha, vacia todo  }  elif (contador==0){  cronometro=0;  }  elif (!recibir\_pasajero.empty()){  receive recibir\_pasajero();  contador++;  cronometro=0;  } }  // Cliente (pasajero)  send enviar\_pasajero(); |



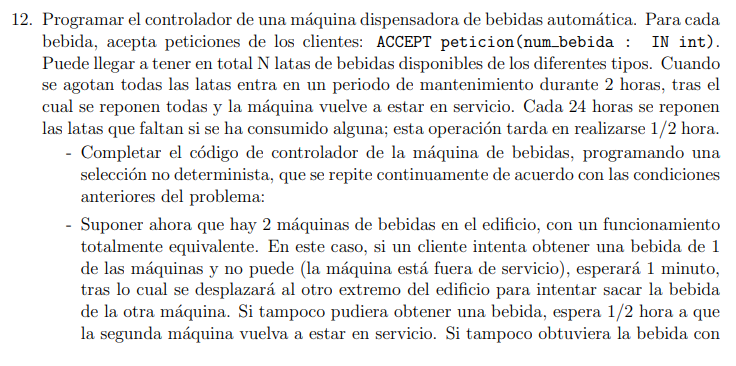
|  |
| --- |
| Cajero:: int saldoclientes[1..n]; int saldoCajero=100;  while true{  if saldoCajero>0;receive sacardinero(id\_cliente,cantidad){  if(saldoclientes[id\_cliente]<cantidad || saldoCajero<cantidad){  send respuesta[id\_cliente](//respuesta de error);  }  else{  saldoclientes[id\_cliente]-=cantidad;  saldoCajero-=cantidad;  send respuesta[id\_cliente](//respuesta de exito);  if saldoCajero==0;{  wait 1hora; saldoCajero=100;  }  }  elif ;receive meterdinero(id\_cliente,cantidad){  saldoCajero+= cantidad;  saldoclientes[id\_cliente]+=cantidad;  } } |



|  |
| --- |
| ProcesoHabitacion:: int cantidad=4; while true{ if cantidad>0;receive entrar\_habitacion(){  cantidad-; } elif ; receive salir\_habitacion(){  cantidad++; } }  ProcesoTenedor:: receive coger\_tenedor(); receive soltar\_tenedor();  Filoso:: id\_filosofo;  while(true){  //Piensa send entrar\_habitacion(); send coger\_tenedor[id\_filosofo](); send coger\_tenedor[id\_filosofo+1%n](); //comer send soltar\_tenedor[id\_filosofo+1%n](); send soltar\_tenedor[id\_filosofo](); send salir\_habitacion(); } |



|  |
| --- |
| ProcesoServidor:: bool A\_cogida, B\_cogida = false;  while(true){  if A\_cogida==false; receive cogerA(){  A\_cogida=true; } elif B\_cogida==false;receive cogerB(){  B\_cogida=true; } elif A\_cogida==false || B\_cogida==false; receive cogerAoB(){  if (!B\_cogida){  B\_cogida=true;   }  else if (!A\_cogida){  A\_cogida=true;  } } elif true; receive soltarA(){  A\_cogida=false; } elif true; receive soltarB(){  B\_cogida=false; } } |
|  |



|  |
| --- |
| ProcesoServidor:: int cantidadBebidas [1..i];//todas inicialmente a N int CANTIDAD\_TOTAL; while(true){  if CANTIDAD\_TOTAL;receive peticion(id\_bebida){  if(cantidadBebidas[id\_bebida]==0){  send respuesta(//error, no pudo obtenerla);  }  else {  cantidadBebidas[id\_bebida]--;  send respuesta(//exito);  CANTIDAD\_TOTAL-;  if(CANTIDAD\_TOTAL==0){  //mantenimiento, espera 2h  }  }  elif tiempo\_pasado==24h;{//reposición de todas las latas  } } |

— Asignador de recursos con citas.

//Para ello ya no tenemos peticion(id, tipo, n\_recurso) y tenemos adquirir(id\_cliente) y liberar(id\_cliente, recurso)

Servidor::

while true {

if disponibles>0; receive adquirir(id\_cliente){

disponibles–;

send respuesta[id\_cliente](unidades.poop());

}

elif ; receive liberar(id\_cliente,recurso){

disponibles++;

unidades.push(recurso);

}

}