Practica 2 Ampliación de Sistemas Operativos

Gestión de supermercados

Autor: Gabriel Morales Dato
Titulación: Doble grado en Ingeniería de Computadores e Ingeniería Informática
Curso: 2023/2024

Fecha: 31 de diciembre de 2023

Índice

1. Introducción

En esta práctica se pide implementar un sistema basado en una cola rápida, que acelera el tiempo en el que los clientes son atendidos de manera óptima y eficiente.

Se pide realizar dos implementaciones: un sistema maestro-esclavo donde el maestro gestiona la cola y envía los clientes a las esclavos, que representan las cajas; y otro sistema peer-to-peer donde los clientes se reparten equitativamente entre las cajas y la carga de las cajas se distribuye durante la ejecución.

Las características del sistema son:

- Hay dos tipos de clientes: clientes prioritarios, que tardan entre 10 y 20 segundos en ser atendidos. Y clientes no prioritarios, que tardan entre 5 y 10 segundos en ser atendidos.
- Hay dos tipos de cajas: cajas prioritarias, que solo atienden a clientes prioritarios, y cajas no prioritarias, que atienden a cliente no prioritarios y a clientes prioritarios, en caso de que todas las cajas prioritarias estén ocupadas.
- Al comienzo de la ejecución se abren el 50 % de las cajas disponibles, y un 25 % de las cajas abiertas se abren como prioritarias.
- Durante la ejecución, si el numero de clientes esperando en la cola es el doble que el de cajas abiertas, se abre otra caja si hay alguna disponible. En caso de que haya mas cajas abiertas que clientes esperando en cola, se procede a cerrar cajas hasta ajustar el numero de cajas abiertas con el numero de clientes en cola.
- Los clientes que terminan de ser atendidos vuelven a ponerse en la cola, pero lo hacen volviendo a establecer su tiempo de espera y su condición de prioridad.
- Estas condiciones se aplican a los dos tipos de sistema. Además, en el sistema peer-to-peer, las cajas deben equilibrar la carga para repartirse el trabajo que tiene cada una. Para ello los clientes que esperan en una caja pueden ser transferidos a otra con el fin de que el tiempo de atención sea óptimo.

2. Sistema maestro-esclavo

2.1. cola.h

El fichero cola.h contiene estructuras y métodos relacionados con las colas de clientes y la cola de prioridad.

La estructura de cola de prioridad contiene dos colas, una para clientes prioritarios y otra para no prioritarios. Esto nos permite sacar un usuario u otro dependiendo de la caja a la que se vaya a enviar.

```
typedef struct cola {
    int n;
    cliente_t *primero;
    cliente_t *ultimo;
} cola_t;

typedef struct colaPrioridad {
    cola_t *prioritarios;
    cola_t *no_prioritarios;
} colaPrioridad_t;
```

Existen métodos para crear y destruir colas en memoria:

```
cola_t *crearCola();
colaPrioridad_t *crearColaPrioridad();
int liberarColaPrioridad(colaPrioridad_t *cola);
```

Y para sacar o meter usuarios en la cola de prioridad:

```
cliente_t *pop(colaPrioridad_t *cola);
cliente_t *popNP(colaPrioridad_t *cola);
int push(colaPrioridad_t *cola, cliente_t *cliente);
```

El método popNP sirve para sacar un cliente no prioritario de la cola de prioridad. Normalmente se saca un cliente prioritario, pero habrá ocasiones en las que tenemos que sacar un cliente no prioritario si sabemos que los clientes prioritarios que hay en cola tienen una caja reservada para ser atendidos (se explica más adelante). Esto agiliza el envío de clientes a cajas.

2.2. cliente.h

El fichero cliente.h contiene la estructura de un cliente y métodos relacionados con el manejo de clientes.

Un cliente tiene un identificador único que será utilizado para identificar los clientes en las cajas a la hora de depurar el programa. También tiene un puntero al siguiente cliente en la cola, que se asigna cuando un cliente se pone detrás en la cola.

```
typedef struct cliente {
    int id;
    int prioritario;
    int tiempoEspera;
    struct cliente *siguiente;
} cliente_t;
```

Hay un método para crear clientes y otro para simular la espera en la caja, durmiendo x segundos.

```
cliente_t *crearCliente();
int esperar(cliente_t *cliente);
```

2.3. main.c

Contiene el código MPI del programa. Al comienzo se inicializa MPI y se comprueba el rango para ejecutar el código del maestro (rank = 0) o del esclavo (cualquier otro rank).

Código del maestro

En el código del maestro declaro las variables que voy a utilizar para la gestión de cajas y de clientes. Para la gestión de cajas utilizo los siguientes vectores:

```
int *estaAbierta;
int *estaOcupada;
int *estaTrabajando;
int *seHaPreguntado;
int *esPrioritaria;
```

Estos vectores los utilizo para realizar comprobaciones con las cajas:

- estaAbierta: indica si una caja esta abierta y admite clientes. Antes de entrar al bucle principal se abren la mitad de las cajas.
- estaOcupada: indica si la caja "ha llamado" a un cliente y va a estar ocupada. Esta variable principalmente la uso para que no se ejecute el MPI_Irecv dos veces para la misma caja antes de terminar de atender un cliente, porque provocaría un bloqueo.
- estaTrabajando: indica que la caja ya ha recibido un cliente y lo esta atendiendo. Desde que el cliente entra en caja hasta que sale la caja esta trabajando.
- seHaPreguntado: esta variable la uso en la parte del código donde compruebo si una caja ha terminado de trabajar o no, para así poder enviarle otro cliente. La variable se pone a 1 cuando se ha enviado un MPI_Irecv a una caja ocupada, para no enviarle mas MPI_Irecv.
- esPrioritaria: indica si la caja es prioritaria o no. Antes de entrar en el bucle principal, un 25 % de las cajas abiertas se establecen como prioritarias.

También creo e inicializo la cola de prioridad, que contiene una cola de clientes prioritarios y otra de clientes no prioritarios. Pongo en la cola tantos clientes como se haya indicado por parámetro.

A continuación creo dos vectores de MPI_Request.

```
MPI_Request *requestsCajaLibre =
        (MPI_Request *) malloc(nCajas * sizeof(MPI_Request));
for (i = 0; i < nCajas; i++) requestsCajaLibre[i] = MPI_REQUEST_NULL;
MPI_Request *requestsCAtendido =
        (MPI_Request *) malloc(nCajas * sizeof(MPI_Request));
for (i = 0; i < nCajas; i++) requestsCAtendido[i] = MPI_REQUEST_NULL;
```

Estos vectores los uso para almacenar las Request que envío a las cajas y así poder comprobar más tarde si se han completado. Los inicializo a NULL para evitar posibles errores al hacer Test sobre una Request no inicializada.

- requestsCajaLibre: guarda la Request que se genera al realizar el MPI_Irecv en el paso 1 (se explica mas adelante), cuando se pregunta a los esclavos si están libres.
- requestsCAtendido: guarda la Request que se genera al realizar el MPI_Irecv en el paso 3 (se explica mas adelante), cuando el esclavo avisa que ha terminado de atender a un cliente.

Después de inicializar estas variables comienza el bucle principal del maestro, que se ejecuta mientras haya clientes en cola. Este bucle lo he dividido en 3 pasos que se corresponden con tres pasos en los esclavos, para que sea más fácil identificar que parte del código se corresponde con cada acción en el supermercado.

- Paso 0: Abrir o cerrar cajas. Se comprueba el numero de clientes en cola y el numero de cajas abiertas para abrir o cerrar cajas según proceda. Se marcan o desmarcan las variables estaAbierta y esPrioritaria.

  Código del paso 0:

```
/* 0. Se ajusta el numero de cajas abiertas */
/* Se abre 1 caja si hay el doble de cola que de cajas */
if (clientesEnCola >= cajasAbiertas * 2 && cajasAbiertas < nCajas) {
    for (i = 0; i < nCajas; i++) {
        if (!estaAbierta[i] && !estaOcupada[i] && !estaTrabajando[i]) {
            estaAbierta[i] = 1;
            cajasAbiertas++;
            /* Comprobamos si se necesitan cajas prioritarias */
            if (cPrioritarias < cajasAbiertas / 4) {
                esPrioritaria[i] = 1;
                cPrioritarias++;
            } else {
                esPrioritaria[i] = 0;
            }
            break;
        }
    }
/* Si hay mas cajas abiertas que clientes en cola se cierran cajas */
} else if (clientesEnCola < cajasAbiertas && cajasAbiertas > 1)  {
    i = 0;
    while (clientesEnCola < cajasAbiertas) {
        if (i >= nCajas - 1) break;
        if (estaAbierta[i] && !estaOcupada[i] && !estaTrabajando[i]) {
            if ((esPrioritaria[i] && (cPrioritarias > cajasAbiertas / 4)) ||
                (!esPrioritaria[i] && (cPrioritarias <= cajasAbiertas / 4))) {
                estaAbierta[i] = 0;
                cajasAbiertas--;
                if (esPrioritaria[i]) cPrioritarias--;
            }
        }
        i++;
    }
}
```

- Paso 1: Comprobar si hay cajas libres. Se itera sobre todas las cajas y se compruebas las variables:
  - estaAbierta: para ver si la caja esta abierta. Puede darse el caso de que la caja 4 este abierta pero la 3 cerrada, por eso itero sobre todas las cajas.
  - estaOcupada: para ver si ya se ha recibido la solicitud de atención de una caja y no volver a recibirla.
  - estaTrabajando: para ver si la caja no esta atendiendo ya algún cliente. Si una caja esta trabajando, necesariamente esta ocupada, por lo que esta ultima comprobación es redundante.

Si una caja cumple las tres condiciones, esta preparada para recibir a un cliente. En ese caso pasamos a comprobar si hay clientes en cola que puedan ser atendidos por esta caja. Para ello se tiene que dar una de las siguientes condiciones:

- La caja es prioritaria y hay clientes prioritarios en la cola.
- La caja no es prioritaria y hay cliente no prioritarios en la cola.
- La caja no es prioritaria y hay clientes prioritarios en cola, pero todas las cajas prioritarias están ocupadas.

Si se cumple alguna de las tres condiciones, la caja puede aceptar algún cliente de la cola, por lo que se pasa a marcar la caja como ocupada y se realiza un MPI_Irecv para recibir la petición de trabajo de la caja. La request de este MPI_Irecv se guarda en requestsCajaLibre. La comunicación MPI de este primer paso se realiza con la etiqueta TAG_CAJA_LIBRE 1.

Código del paso 1:

```
/* 1. Se comprueba si hay alguna caja libre */
for (i = 1; i < nCajas + 1; i++) {
    if (estaAbierta[i - 1] && !estaOcupada[i - 1] && !estaTrabajando[i - 1]) {
        /* Solo se envia el request si sabemos que va a ser aceptado */
        int seVaAEnviar = 0;
        if (esPrioritaria[i - 1] && cola->prioritarios->n > 0) {
            seVaAEnviar = 1;
            /* fprintf */
        } else if (!esPrioritaria[i - 1] && (cola->no_prioritarios->n > 0 ||
                   (cola->prioritarios->n > 0 &&
                    cPrioritarias - cPrioritariasOcupadas <= 0))) {
            seVaAEnviar = 1;
            /* fprintf */
        }
        /* Solo se envia el request si sabemos que va a ser aceptado */
        if (seVaAEnviar) {
            estaOcupada[i - 1] = 1;
            MPI_Irecv(&esclavoPreparado,
                      1,
                      MPI_INT,
                      i,
                      TAG_CAJA_LIBRE,
                      comm,
                      &requestsCajaLibre[i - 1]);
        }
    }
}
```

- Paso 2: Enviar un cliente a la caja. Sabiendo que la caja esta libre y puede atender a un cliente de la cola, se saca un cliente de la cola para enviárselo a la caja. El primer paso a realizar es un MPI_Testany para comprobar si alguna de las requestCajaLibre que solicitamos en el paso 1 ha terminado. En el caso de que alguna request haya terminado, la variable flag se pondrá a 1, y la variable cajaLibre contendrá el índice de la caja que ha enviado la petición de trabajo.

  A continuación comprobamos que la caja libre esta dentro del rango de cajas, y que la caja esta abierta y ocupada. La caja va a estar ocupada si le hemos enviado la solicitud en el paso 1.

  Después, comprobamos que se da algunas de las siguientes condiciones:

  - La caja es prioritaria y hay clientes prioritarios en cola: se realiza un pop de la cola y se envía el cliente con un MPI_Isend.

- La caja no es prioritaria y hay clientes prioritarios en cola, pero todas las cajas prioritarias están ocupadas: se realiza un pop para sacar al cliente prioritario de la cola y se envía a la caja no prioritaria con un MPI_Isend.
- La caja no es prioritaria y hay clientes no prioritarios en cola: en este caso puede que haya clientes prioritarios en cola porque acaban de llegar a la cola justo en este paso. Por eso se realiza un popNP para sacar al cliente no prioritario y se envía a la caja con un MPI_Isend.

Si se ha enviado un cliente a una caja, esta se marca como trabajando. Para este paso se utiliza la etiqueta TAG_CLIENTE 2. En caso de que no se haya enviado ningún cliente, se espera un segundo antes de continuar la ejecución, para dar tiempo a que se liberen las cajas. Este ultimo paso es opcional.

Código del paso 2:

```
/* Se espera a que una caja quede libre */
int cajaLibre;
int flag;
MPI_Testany(nCajas,
            requestsCajaLibre,
            &cajaLibre,
            &flag,
            MPI_STATUS_IGNORE);
if (flag && cajaLibre >= 0 && estaAbierta[cajaLibre] &&
    estaOcupada[cajaLibre]) {
    /* 2. Se envia un cliente a la caja */
    MPI_Request request;
    if (clientesEnCola > 0) {
        /* Comprobamos si la caja libre es prioritaria */
        if (esPrioritaria[cajaLibre] && cola->prioritarios->n > 0) {
            cliente_t *clienteActual = pop(cola);
            /* En vez de enviar un cliente_t se envia solo su id y su
            * tiempo de espera. Y si es vip o no */
            int clienteEnviar[3] = {clienteActual->id,
                                    clienteActual->prioritario,
                                    clienteActual->tiempoEspera};
            clientesEnCola--;
            cPrioritariasOcupadas++;
            MPI_Isend(clienteEnviar,
                      3,
                      MPI_INT,
                      cajaLibre + 1,
                      TAG_CLIENTE,
                      comm,
                      &request);
            estaTrabajando[cajaLibre] = 1;
        } else if (!esPrioritaria[cajaLibre]) {
            /* Si el cliente es prioritario se comprueba si hay
                cajas prioritarias libres */
            if (cola->prioritarios->n > 0 &&
                (cPrioritarias - cPrioritariasOcupadas <= 0)) {
                cliente_t *clienteActual = pop(cola);
                int clienteEnviar[3] = {clienteActual->id,
                                        clienteActual->prioritario,
                                        clienteActual->tiempoEspera};
                clientesEnCola--;
                MPI_Isend(clienteEnviar,
                          3,
                          MPI_INT,
                          cajaLibre + 1,
                          TAG_CLIENTE,
                          comm,
                          &request);
                estaTrabajando[cajaLibre] = 1;
            } else if (cola->no_prioritarios->n > 0) {
                cliente_t *clienteActual = popNP(cola);
                int clienteEnviar[3] = {clienteActual->id,
                                        clienteActual->prioritario,
                                        clienteActual->tiempoEspera};
                clientesEnCola--;
                MPI_Isend(clienteEnviar,
                          3,
                          MPI_INT,
                          cajaLibre + 1,
                          TAG_CLIENTE,
                          comm,
                          &request);
                estaTrabajando[cajaLibre] = 1;
            }
        }
    }
} else sleep(1);
```

- Paso 3: Comprobar si alguna caja ha terminado de atender un cliente. En este paso se envía un MPI_Irecv a todas las cajas a las que se haya mandado un cliente en el paso anterior. Las cajas, cuando terminan de atender un cliente, envían un MPI_Send para avisar que han terminado. Es este MPI_Send el que se espera en el paso 3 del maestro, y para ello se usa la etiqueta TAG_CATENDIDO 3. Se crea un MPI_Request de este MPI_Irecv y se guarda en el vector de requests requestsCAtendido.

  Para evitar enviar MPI_Irecv innecesarios, hago uso de la variable seHa-Preguntado, marcándola cuando envío un MPI_Irecv para la caja a la que

lo haya enviado. En posteriores iteraciones del bucle en este paso, se comprueba si se ha preguntado a una caja y en caso que sea así, se realiza un MPI_Test de requestsCAtendido para comprobar si la caja ha terminado de atender. Cuando la caja termina de atender, se reinician todas las variables utilizadas en los pasos 0, 1, 2 y 3 para esa caja, y el cliente que estaba siendo atendido en esa caja vuelve a la cola, "renaciendo" con otra condición de prioridad y tiempo de espera, pero manteniendo su id.

Código del paso 3:

```
/* 3. Se comprueba si alguna caja ha terminado de atender */
for (i = 1; i < nCajas + 1; i++) {
    if (estaOcupada[i - 1] && estaTrabajando[i - 1]) {
        /* Se pregunta si ha terminado */
        if (!seHaPreguntado[i - 1]) {
            MPI_Irecv(&clienteTerminado,
                      1,
                      MPI_INT,
                      i,
                      TAG_CATENDIDO,
                      comm,
                      &requestsCAtendido[i - 1]);
            seHaPreguntado[i - 1] = 1;
        } else {
            int haTerminado;
            MPI_Test(&requestsCAtendido[i - 1],
                     &haTerminado,
                     MPI_STATUS_IGNORE);
            if (haTerminado) {
                estaOcupada[i - 1]     = 0;
                estaTrabajando[i - 1] = 0;
                seHaPreguntado[i - 1] = 0;
                if (esPrioritaria[i - 1]) cPrioritariasOcupadas--;
                cliente_t *cRenacido = crearCliente();
                cRenacido->id = clienteTerminado;
                push(cola, cRenacido);
                clientesEnCola++;
            }
        }
    }
}
```

Después de realizar los 4 pasos del bucle principal, se comprueba si quedan clientes en cola, en caso de que no queden más clientes, el trabajo habrá concluido y se pasa a liberar la memoria reservada para las variables del programa.

En este punto, el maestro habrá salido del bucle, pero los esclavos estarán en el paso 1, esperando trabajo por parte del maestro. Se envía un MPI_Recv con la etiqueta TAG_CAJA_LIBRE 1 a todos los esclavos para que pasen al paso 2. El Recv lo hago síncrono para asegurar que todos los esclavos lleguen al paso 2.

A continuación, el maestro envía un MPI_Send con la etiqueta TAG_TERMINAR 0 a todos los esclavos. Los esclavos recibirán este mensaje y comprobaran que lleva la etiqueta de finalización, así que procederán a salir del bucle sin continuar con el paso 3.

Finalmente se realiza un MPI_Barrier(comm) para asegurar que todas las comunicaciones han finalizado, y se termina la ejecución del programa con un MPI_Finalize;

Código posterior al bucle principal del maestro:

```
/* Se libera la memoria */
free(requestsCajaLibre);
free(requestsCAtendido);
free(estaOcupada);
free(estaTrabajando);
free(seHaPreguntado);
free(esPrioritaria);
liberarColaPrioridad(cola);

/* Se confirma que todos los nodos estan esperando */
int procListo;
for (int i = 1; i < size; i++) {
    MPI_Recv(&procListo,
                1,
                MPI_INT,
                i,
                TAG_CAJA_LIBRE,
                comm,
                MPI_STATUS_IGNORE);
}
/* Se envia una senal de terminacion a los nodos */
int terminar = 1;
for (int i = 1; i < size; i++) {
    MPI_Send(&terminar,
             1,
             MPI_INT,
             i,
             TAG_TERMINAR,
             comm);
}
MPI_Barrier(comm); /* Se espera a que la comunicacion haya terminado */
```

Código del esclavo

El esclavo realiza tres pasos que se corresponden con los pasos del maestro. Los tres pasos del esclavo son síncronos porque el esclavo solo realiza una tarea simultáneamente.

- Paso 1: Avisar al maestro que la caja esta libre. En este paso se envía un MPI_Send con la etiqueta TAG_CAJA_LIBRE 1 al maestro.

  Código del paso 1:

  ```
  /* 1. Avisar de que estamos libres */
  MPI_Send(&rank, /* Se podria enviar cualquier int */
              1,
              MPI_INT,
              0,
              TAG_CAJA_LIBRE,
              comm);
  ```

- Paso 2: Recibir un cliente del maestro. En este paso se espera recibir un cliente por parte del maestro. Se ejecuta un MPI_Recv sin especificar ninguna etiqueta porque, como se ha explicado en el código del maestro, en este paso también se puede recibir una señal de terminación del programa en caso de que no queden clientes en cola. Para realizar esta comprobación se utiliza una variable del tipo MPI_Status que guarda el estado de la comunicación realizada en este paso. Si la etiqueta utilizada en esta comunicación es un TAG_CLIENTE, se recibe ese cliente y se espera el tiempo que corresponda. Si la etiqueta es TAG_TERMINAR, se sale del bucle para terminar la ejecución del programa.

Código del paso 2:

```
/* 2. Recibir un cliente del maestro */
MPI_Recv(clienteRecibido,
                3,
                MPI_INT,
                0,
                MPI_ANY_TAG,
                comm,
                &status);
/* Se comprueba si se ha recibido un cliente */
if (status.MPI_TAG == TAG_CLIENTE) {
    sleep(clienteRecibido[2]);
} else if (status.MPI_TAG == TAG_TERMINAR) {
    MPI_Barrier(comm); /* Se espera a que la comunicacion haya terminado */
    break;
}
```

- Paso 3: Avisar al maestro que el cliente ha terminado. En este paso se envía otro MPI_Send al maestro con el id del cliente recibido, para avisar que el cliente ha terminado, y que puede renacer para volver nuevamente a la cola.

  Código del paso 3:

```
/* 3. Avisar de que el cliente ha terminado */
MPI_Send(&clienteRecibido[0],
                1,
                MPI_INT,
                0,
                TAG_CATENDIDO,
                comm);
```

3. Sistema peer-to-peer

3.1. cola.h y cliente.h

Los métodos y estructuras de cola.h y cliente.h siguen siendo los mismos que en el sistema maestro-esclavo.

3.2. main.c

Antes de entrar en el bucle principal del programa, se declaran e inicializan las variables de ejecución comunes. También se distribuyen los clientes entre las cajas de la siguiente forma:

```
/* Se distribuyen los clientes entre las cajas */
int nClientesTotal    = atoi(argv[1]);
int nClientesPorCaja  = nClientesTotal / nCajas;
int clientesRestantes = nClientesTotal % nCajas;
int nClientes = nClientesPorCaja + (rank < clientesRestantes ? 1 : 0);
int cInicial = rank * nClientesPorCaja +
               (rank < clientesRestantes ? rank : clientesRestantes);
clientesEnCola = nClientes;
colaPrioridad\_t *cola = crearColaPrioridad();
/* Se asignan clientes a cajas */
for (i = 0; i < nClientes; i++) {
    cliente\_t *clienteCreado = crearCliente();
    clienteCreado->id = cInicial + i;
    push(cola, clienteCreado);
}
```

Antes de entrar al bucle principal se ejecuta un MPI_Barrier para asegurar que todos los nodos empiecen a la vez.

El bucle principal esta dividido en 4 pasos:

- Paso 1: Atender a los clientes propios. En este paso se comprueba si quedan clientes en la cola propia, y se procede a atenderlos en caso de que los haya. Si se atiende a un cliente, se marca la variable estaOcupada[rank], y se resta una unidad a las variables clientesEnCola y nClientesTotal.

  estaOcupada[] es un vector que contiene información sobre que cajas están ocupadas. Esto sirve para pedir ayuda u ofrecerla en los siguientes pasos.

  clientesEnCola es un variable que indica el numero de clientes en la cola propia, y nClientesTotal es una variable que indica cuantos clientes quedan en todas las cajas. Esta variable se inicializa con el numero de clientes totales al comienzo del programa y se va reduciendo según se atiendan clientes.

  Después de atender al cliente, se hace un MPI_Isend al resto de cajas con el tag TAG_INFORME 4. En este send se envía el numero de clientes que nos queda en cola, pero realmente lo utilizo para notificar que hemos atendido un cliente, y así el resto de cajas al recibir este aviso, restaran una unidad a su variable nClientesTotal, para que así todas las cajas conozcan cuantos clientes quedan en el resto de cajas y saber cuando hay que terminar el programa. La recepción de este mensaje se realiza en el paso 4.

  Código del paso 1:

```
/* 1. Atender a los clientes propios */
if (clientesEnCola > 0) {
    estaOcupada[rank] = 1;
    cliente_t *cliente = pop(cola);
    clientesEnCola--;
    nClientesTotal--;
    sleep(cliente->tiempoEspera);
    /* Avisar a los demas que hay un cliente menos */
    for (i = 0; i < nCajas; i++) {
        if (i != rank) {
            MPI_Isend(&clientesEnCola,
                      1,
                      MPI_INT,
                      i,
                      TAG_INFORME,
                      comm,
                      &requestInforme);
        }
    }
} else {
    estaOcupada[rank] = 0;
}
```

- Paso 2: Ayudar a otras cajas si estamos libres. Este paso se ejecuta únicamente si estamos libres, es decir, la variable estaOcupada[rank] no esta marcada. Este paso esta dividido en dos partes:

  Paso 2a: Solicitar clientes a otras cajas. Se envía un MPI_Isend al resto de cajas con el tag TAG_AYUDA y se guarda la request requestsAyudaOfr[i]. Esta solicitud solo se envía a las cajas que estén ocupadas y no se les haya enviado una solicitud antes. Para manejar esto, utilizo una variable seHaOfrecidoAyuda[i] que marco cada vez que se envía el MPI_Isend a i, y se desmarca cuando termina la comunicación de ayuda.

  Paso 2b: Recibir clientes de otras cajas. Se vuelve a iterar sobre el resto de cajas y se comprueba con MPI_Test si la requestsAyudaOfr[i] ha sido recibida. En caso de que sea así, se ejecutan dos MPI_Iprobe para comprobar si el cliente que se puede recibir tiene un tag TAG_CLIENTE 2 o un TAG_CANCELAR 3. En esta parte del código he decidido hacer dos MPI_Iprobe cada uno comprobando un tag diferente, en vez de realizar un solo MPI_Iprobe con MPI_ANY_TAG, porque con este último podría recibir otros mensajes que no corresponden a este paso, como los que llevan el TAG_INFORME o el TAG_AYUDA. Una vez que se ha identificado que es lo que podemos recibir, se pasa a recibirlo. Si es un cliente se añade a la cola, se aumenta en uno la variable clientesEnCola, es marca la variable estaOcupada[rank] y se desmarca seHaOfrecidoAyuda[i]. En caso de que se reciba una cancelación de envió, se desmarcan las variables estaOcupada[i] y seHaOfrecidoAyuda[i]. Una caja que cancela un envío de cliente es porque ya no esta ocupada y ha dejado de necesitar ayuda. Es por eso que desmarco la variable estaOcupada[i].

Código del paso 2:

```
/* 2. Ayudar a otras cajas si estamos libres */
if (!estaOcupada[rank]) {
    /* 2a. Solicitar clientes */
    for (i = 0; i < nCajas; i++) {
        if (i != rank && !seHaOfrecidoAyuda[i] && estaOcupada[i]) {
            MPI_Isend(&rank,
                      1,
                      MPI_INT,
                      i,
                      TAG_AYUDA,
                      comm,
                      &requestsAyudaOfr[i]);
            seHaOfrecidoAyuda[i] = 1;
        }
    }
    /* 2b. Recibir clientes de otras cajas */
    for (i = 0; i < nCajas; i++) {
        if (i != rank && seHaOfrecidoAyuda[i]) {
            MPI_Test(&requestsAyudaOfr[i],
                     &flagAyudaRecibida,
                     MPI_STATUS_IGNORE);
            if (flagAyudaRecibida) {
                MPI_Iprobe(i,
                           TAG_CLIENTE,
                           comm,
                           &flagEnviaCliente,
                           MPI_STATUS_IGNORE);
                MPI_Iprobe(i,
                           TAG_CANCELAR,
                           comm,
                           &flagCancelaEnvio,
                           MPI_STATUS_IGNORE);
                if (flagEnviaCliente) {
                    MPI_Recv(cRecibido,
                             3,
                             MPI_INT,
                             i,
                             TAG_CLIENTE,
                             comm,
                             MPI_STATUS_IGNORE);
                    estaOcupada[i] = 1;
                    if (cRecibido[0] != -1) {
                        cliente_t *cRecibidoED =
                            (cliente_t *) malloc(sizeof(cliente_t));
                        cRecibidoED->id           = cRecibido[0];
                        cRecibidoED->prioritario  = cRecibido[1];
                        cRecibidoED->tiempoEspera = cRecibido[2];
                        push(cola, cRecibidoED);
                        clientesEnCola++;
                        estaOcupada[rank] = 1;
                        seHaOfrecidoAyuda[i] = 0;
                    } else {
                        /* Se ha recibido un NULL */
                        seHaOfrecidoAyuda[i] = 0;
                    }
                } else if (flagCancelaEnvio) {
                    seHaOfrecidoAyuda[i] = 0;
                    estaOcupada[i] = 0;
                }
            }
        }
    }
}
```

- Paso 3: Solicitar ayuda si estamos ocupados. Este paso solo se ejecuta si el numero de clientes en cola supera el de la constante MUCHOS_CLIENTES, definida al comienzo del programa. Para que la atención sea óptima esta variable debería ser 0 para que no haya ningún cliente esperando habiendo otras cajas libres.

  Para este paso se itera sobre el resto de cajas, y primero se comprueba con un MPI_Iprobe si hay algún mensaje con el tag TAG_AYUDA pendiente de ser recibido. Si este es el caso, hay otras cajas que están dispuestas a ayudarnos. Cuando una caja nos va a ayudar procedemos a recibir su mensaje de ayuda que debió enviar en el paso 2. Esta recepción es síncrona porque nos hemos asegurado con MPI_Iprobe que hay un mensaje pendiente de

ser recibido.

Después de saber que la caja i nos va a ayudar, comprobamos si tenemos clientes en cola, ya que puede que en este punto hayamos enviado clientes a otras cajas y no queden clientes en nuestra cola. Si quedan clientes, se saca uno de la cola y se envía con un MPI_Send a la caja i con el tag TAG_CLIENTE. Si no quedan clientes, se envía un cliente nulo con el tag TAG_CANCELAR para que la caja que ayuda sepa que ya no es necesaria la ayuda.

Código del paso 3:

```
/* 3. Enviar clientes si estamos ocupados */
/* 3a. Preguntar si alguna caja esta libre */
if (clientesEnCola > MUCHOS_CLIENTES) {
    for (i = 0; i < nCajas; i++) {
        if (i != rank) {
            MPI_Iprobe(i,
                       TAG_AYUDA,
                       comm,
                       &flagAyudaOfrecida,
                       MPI_STATUS_IGNORE);
            /* 3b. Esperar que una caja nos ayude y enviarle un cliente */
            if (flagAyudaOfrecida) {
                int procLibre;
                MPI_Recv(&procLibre,
                         1,
                         MPI_INT,
                         i,
                         TAG_AYUDA,
                         comm,
                         MPI_STATUS_IGNORE);
                if (clientesEnCola > 0) {
                    cliente_t *cEnviadoED = pop(cola);
                    clientesEnCola--;
                    int cEnviado[3] = {
                        cEnviadoED->id,
                        cEnviadoED->prioritario,
                        cEnviadoED->tiempoEspera
                    };
                    MPI_Send(cEnviado,
                             3,
                             MPI_INT,
                             procLibre,
                             TAG_CLIENTE,
                             comm);
                } else {
                    int cEnviado[3] = {-1, -1, -1};
                    MPI_Send(cEnviado,
                             3,
                             MPI_INT,
                             procLibre,
                             TAG_CANCELAR,
                             comm);
                }
            }
        }
    }
}
```

- Paso 4: Comprobar si se ha terminado. Primero se comprueba si otras cajas han atendido clientes. Se itera sobre el resto de cajas y se comprueba con un MPI_Iprobe si se puede recibir un mensaje con el tag TAG_INFORME. En caso afirmativo se procede a recibir dicho mensaje, que indica que la caja i ha terminado de atender a un cliente, y se resta una unidad de nClientesTotal.

  Después se comprueba si quedan clientes por atender, es decir, si nClientesTotal no vale 0, para continuar con la ejecución del programa o salir del bucle principal.

Código del paso 4:

```
/* 4. Comprobar si se ha terminado */
/* 4a. Comprobar si otras cajas han atendido clientes */
for (i = 0; i < nCajas; i++) {
    if (i != rank) {
        MPI_Iprobe(i,
                   TAG_INFORME,
                   comm,
                   &flagInformeRecibido,
                   MPI_STATUS_IGNORE);
        if (flagInformeRecibido) {
            int procInforme;
            MPI_Recv(&procInforme,
                     1,
                     MPI_INT,
                     i,
                     TAG_INFORME,
                     comm,
                     MPI_STATUS_IGNORE);
            nClientesTotal--;
        }
    }
}
/* 4b. Comprobar si no quedan clientes por atender */
if (nClientesTotal == 0) {
    break;
}
```

Finalmente se libera la memoria del programa y termina la ejecución MPI.

Para este sistema he preparado dos casos de prueba que se activan definiendo CASO_DE_PRUEBA y CASO_DE_PRUEBA_2 con el preprocesador. Estos casos los he utilizado para comprobar situaciones donde se producen interbloqueos.