UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Práctica 3. Implementación de algoritmos distribuidos con MPI.

Carlos Ureña / Jose M. Mantas / Pedro Villar / Manuel Noguera

Curso 2025-26 (archivo generado el 8 de septiembre de 2025)

Grado en Ingeniería Informática,
Grado en Informática y Matemáticas,
Grado en Informática y Administración de Empresas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Práctica 3. Implementación de algoritmos distribuidos con MPI.

Índice.

Objetivos

Los objetivos de esta práctica son:

- Iniciar a los alumnos en la programación de algoritmos distribuidos.
- Conocer varios problemas sencillos de sincronización y su solución distribuida mediante el uso de la interfaz de paso de mensajes MPI:
  - Diseñar una solución distribuida al problema del **productor - consumidor** con buffer acotado, para varios productores y varios consumidores. El planteamiento del problema es similar al ya visto para múltiples hebras en memoria compartida.
  - Diseñar diversas soluciones al problema de la **cena de los filósofos**.

Sección 1.
Productor-Consumidor con buffer acotado.

Subsección 1.1.
Aproximación inicial.

Aproximación inicial en MPI

En la solución distribuida habrá (inicialmente) **tres procesos**:

- **Productor**: produce una secuencia de datos (números enteros, comenzando en 0), y los envía al proceso buffer.
- **Buffer**: Recibe (de forma alterna) enteros del proceso productor y peticiones del consumidor. Responde al consumidor enviándole los enteros recibidos, en el mismo orden.
- **Consumidor**: realiza peticiones al proceso buffer, como respuesta recibe los enteros y los consume.

El esquema de comunicación entre estos procesos se muestra a continuación:

Aproximación inicial. Estructura del programa.

En `prodcons.cpp` podemos ver una solución inicial al problema. La estructura del programa es como sigue:

```
#include ......   // includes varios
#include <mpi.h>  // includes de MPI
using ..... ;     // using varios

// contantes: asignación de identificadores a roles
const int id_productor           = 0, // identificador del proceso productor
          id_buffer              = 1, // identificador del proceso buffer
          id_consumidor          = 2, // identificador del proceso consumidor
          num_procesos_esperado = 3, // número total de procesos esperado
          num_iteraciones        = 20;// núm. de datos producidos/consum.
// funciones auxiliares
int  producir()           { ... } // produce un valor (usada por productor)
void consumir( int valor ){ ... } // consume un valor (usada por consumidor)
// funciones ejecutadas por los procesos en cada rol:
void funcion_productor()  { ... } // función ejecutada por proceso productor
void funcion_consumidor() { ... } // función ejecutada por proceso consumidor
void funcion_buffer()     { ... } // función ejecutada por proceso buffer
// función main (punto de entrada común a todos los procesos)
int main( int argc, char *argv[] ) { ... }
```

Aproximación inicial. Función `main`.

`main` se encarga de que cada proceso ejecute su función:

```
int main( int argc, char *argv[] )
{
  int id_propio, num_procesos_actual; // ident. propio, núm. de procesos
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );

  if ( num_procesos_esperado == num_procesos_actual )
  {
    if ( id_propio == id_productor )  // si mi ident. es el del productor
      funcion_productor();            //     ejecutar función del productor
    else if ( id_propio == id_buffer )// si mi ident. es el del buffer
      funcion_buffer();               //     ejecutar función buffer
    else                              // en otro caso, mi ident es consumidor
      funcion_consumidor();           //     ejecutar función consumidor
  }
  else if ( id_propio == 0 )  // si hay error, el proceso 0 informa
    cerr << "error: número de procesos distinto del esperado." << endl ;
  MPI_Finalize( );
  return 0;
}
```

Aproximación inicial. Productor y consumidor

Los procesos productor y consumidor usan envío **síncrono seguro**

```
void funcion_productor()
{
 for ( unsigned i = 0 ; i < num_items ; i++ )
 {
   int valor_prod = producir(); // producir (espera bloqueado tiempo aleat.)
   cout << "Productor va a enviar valor " << valor_prod << endl;
   MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, 0, MPI_COMM_WORLD );
 }
}
void funcion_consumidor()
{
 int peticion, valor_rec = 1 ; MPI_Status estado ;

 for( unsigned i = 0 ; i < num_items; i++ )
 {
   MPI_Ssend( &peticion,  1,MPI_INT, id_buffer, 0,MPI_COMM_WORLD);
   MPI_Recv ( &valor_rec, 1,MPI_INT, id_buffer, 0,MPI_COMM_WORLD, &estado);
   cout << "Consumidor ha recibido valor " << valor_rec << endl;
   consumir( valor_rec ); // consumir (espera bloqueado tiempo aleat.)
 }
}
```

Aproximación inicial. Proceso buffer

El proceso buffer recibe un dato, luego recibe una petición, y finalmente responde a la petición enviando el dato recibido:

```
void funcion_buffer()
{
 int        valor, peticion ;
 MPI_Status estado ;

 for ( unsigned int i = 0 ; i < num_items ; i++ )
 {
   // recibir valor del productor
   MPI_Recv( &valor,    1,MPI_INT, id_productor, 0,MPI_COMM_WORLD,&estado);
   cout << "Buffer ha recibido valor " << valor << endl ;

   // recibir petición de consumidor, enviarle el dato
   MPI_Recv( &peticion, 1,MPI_INT, id_consumidor,0,MPI_COMM_WORLD,&estado);
   cout << "Buffer va a enviar " << valor << endl;
   MPI_Ssend( &valor,   1,MPI_INT, id_consumidor,0, MPI_COMM_WORLD);
 }
}
```

Valoración de la solución inicial

Sin embargo, esta solución **fuerza una excesiva sincronización entre productor y consumidor**. A largo plazo, el tiempo promedio empleado en `producir` será similar al empleado en `consumir`, sin embargo:

- En cada llamada hay diferencias arbitrarias entre ambos tiempos.
- Frecuentemente la hebra productora o la hebra consumidora quedará esperando un tiempo hasta que el buffer puede procesar su envío o solicitud.
- Si las hebras consumidora y productora se ejecutan en dos procesadores distintos (en exclusiva para ellas), esos procesadores quedarán sin usar (desocupados) una fracción del tiempo total y el programa puede tarda más en acabar.

Necesitamos algún mecanismo de reducción de las esperas.

Subsección 1.2.
Solución con selección no determinista.

Solución con selección no determinista

Para lograr nuestro objetivo, permitimos que el proceso buffer acomode diferencias temporales en la duración de producir y consumir:

- El proceso buffer puede guardar un vector de valores pendientes de consumir, en lugar de un único valor.
- De esta forma: el productor puede producir varios valores seguidos (sin esperar al consumidor), y el consumidor puede consumir varios seguidos (sin esperar al productor)
- Las esperas se reducen, las CPUs están menos tiempo desocupadas.
- El tiempo total hasta acabar el programa se reduce.

Para lograr esto, necesitamos que el proceso buffer pueda recibir una petición cuando hay valores pendientes de enviar, y a la vez pueda recibir un valor cuando hay celdas donde se pueda guardar.

Espera selectiva en MPI

El comportamiento del buffer que queremos implementar se llama **espera selectiva**

- En cada iteración, el buffer podrá aceptar un mensaje exclusivamente del productor (si el vector está vacío), exclusivamente del consumidor (si el vector está lleno), o de ambos (ni vacío ni lleno).
- MPI permite implementar este comportamiento usando para ello la posibilidad de especificar, en cada operación de recepción, un emisor concreto o cualquier emisor (igualmente con las etiquetas).
- Por tanto, el buffer aceptará, en función del estado del vector, un mensaje solo del productor, solo del consumidor, o de cualquier emisor (es decir, de ambos)

Estructura del proceso buffer

El proceso buffer tiene esta estructura (archivo **prodcons2.cpp**)

```
void funcion_buffer()
{
   int        buffer[tam_vector],       // buffer con celdas ocupadas y vacías
              valor,                    // valor recibido o enviado
              primera_libre       = 0,  // índice de primera celda libre
              primera_ocupada     = 0,  // índice de primera celda ocupada
              num_celdas_ocupadas = 0,  // número de celdas ocupadas
              id_emisor_aceptable ;     // identificador de emisor aceptable
   MPI_Status estado ;                  // metadatos del mensaje recibido

   for( unsigned int i=0 ; i < num_items*2 ; i++ )
   {
      // 1. determinar si puede enviar solo prod., solo cons, o todos
      ....
      // 2. recibir un mensaje del emisor o emisores aceptables
      .....
      // 3. procesar el mensaje recibido
      .....
   }
}
```

Recepción de un mensaje

En el cuerpo del bucle, en primer lugar se calcula (en **`id_emisor_aceptable`**) de que proceso o procesos podemos aceptar un mensaje. Después, lo recibimos:

```
// 1. determinar si puede enviar solo prod., solo cons, o de ambos

if ( num_celdas_ocupadas == 0 )                 // si buffer vacío
   id_emisor_aceptable = id_productor ;         //    solo prod.
else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
   id_emisor_aceptable = id_consumidor ;        //    solo cons.
else                                            // si no vacío ni lleno
   id_emisor_aceptable = MPI_ANY_SOURCE ;       //    cualquiera

// 2. recibir un mensaje del emisor o emisores aceptables:

MPI_Recv( &valor, 1, MPI_INT, id_emisor_aceptable, 0,
          MPI_COMM_WORLD, &estado );
```

Procesamiento del mensaje

Una vez recibido el mensaje, el tercer paso es actualizar el buffer en función de que proceso haya sido el que lo ha enviado:

```
// 3. procesar el mensaje recibido
switch( estado.MPI_SOURCE )   // leer emisor del mensaje en metadatos
{
  case id_productor:  // si ha sido el productor: insertar en buffer
    buffer[primera_libre] = valor ;
    primera_libre = (primera_libre+1) % tam_vector ;
    num_celdas_ocupadas++ ;
    cout << "Buffer ha recibido valor " << valor << endl;
    break;

  case id_consumidor:  // si ha sido el consumidor: extraer y enviarle
    valor = buffer[primera_ocupada] ;
    primera_ocupada = (primera_ocupada+1) % tam_vector ;
    num_celdas_ocupadas-- ;
    cout << "Buffer va a enviar valor " << valor << endl ;
    MPI_Ssend( &valor, 1, MPI_INT, id_consumidor, 0, MPI_COMM_WORLD);

    break;
}
```

Ejercicio propuesto: múltiples productores y consumidores

Extenderemos el programa anterior (para 1 productor y 1 consumidor) a múltiples productores y consumidores:

- Habrá $n_p = 4$ procesos productores y $n_c = 5$ procesos consumidores.
- Sigue habiendo un único proceso buffer.
- El número m total de items a producir o consumir (constante `num_items`) debe ser múltiplo de n_p y múltiplo de n_c.
- Los procesos con identificador entre 0 y $n_p - 1$ son productores.
- El proceso con identificador n_p es el buffer.
- Los procesos con identificdor entre $n_p + 1$ y $n_p + n_c$ son consumidores.
- Debes declarar dos constantes enteras con el núm. de prods. (n_p) y el núm. de consum. (n_c), asegurate que el programa es correcto aunque se usen otros valores distintos de 4 y 5 para n_p y n_c.

Números de orden de los procesos y producción de valores

Puesto que ahora tenemos múltiples productores y consumidores:

- La función de los productores y la de los consumidores reciben como parámetro el número de orden del productor o del consumidor, respectivamente (esos números son los números de orden en cada rol, comenzando en 0, no son los identificadores de proceso).
- Los números de orden deben calcularse en **`main`**.
- La función de producir dato recibe como parámetro el número de orden del productor que la invoca. Esto permite que los productores usen cada uno su contador (variable **`contador`** de **`producir_dato`**) para producir un rango distinto de valores: el productor con número de orden i producirá los valores entre ik y $ik + k - 1$ (ambos incluidos), donde k es el número de valores producidos por cada productor (es decir $k = m/n_p$).

Diseño de la solución con etiquetas

Para solucionar el problema con múltiples prods./cons.:

- Según el estado del buffer, debemos de aceptar un mensaje de cualquier productor, de cualquier consumidor, o de cualquier proceso.
- No es posible usar exactamente la misma estrategia que antes: con MPI no es posible restringir el emisor aceptable a cualquiera de un subconjunto de procesos dentro de un comunicador (o aceptamos de un proceso concreto o aceptamos de todos los del comunicador)
- El problema se puede solucionar usando múltiples comunicadores, pero no hemos visto como definirlos.
- También se puede solucionar **usando dos etiquetas distintas para diferenciar los mensajes de los productores y los consumidores**

Actividad

Partiendo de `prodcons2.cpp`, crea un nuevo archivo (llamado `prodcons2-mu.cpp`) con tu solución al problema descrito, ahora para múltiples productores y consumidores.

- Diseña una solución basada en el uso de etiquetas, que por lo demás es similar a la ya vista
- Define constantes enteras para las etiquetas: el programa será mucho más legible. Estas constantes deben tener nombres que comienzen con `etiq_`

Sección 2.
Cena de los Filósofos.

2.1. Aproximación inicial.

2.2. Uso del proceso camarero con espera selectiva

Subsección 2.1.
Aproximación inicial..

Cena de los filósofos en MPI.

Consideramos un programa MPI para el problema de **la cena de los filósofos**. En este problema intervienen 5 **filósofos** y 5 **tenedores**:

- **Los filósofos son 5 procesos** (numerados del 0 al 4) que ejecutan un bucle infinito, en cada iteración comen primero y piensan después (ambas son actividades de duración arbitraria).
- Los filósofos, para comer, se disponen en una mesa circular donde hay un tenedor entre cada dos filósofos. Cuando un filósofo está comiendo, **usa en exclusión mutua sus dos tenedores adyacentes**.

En programación distribuida cada recurso compartido (de uso exclusivo por un único proceso en un instante) debe de implementarse con un proceso gestor adicional (específico para ese recurso), proceso que alterna entre dos estados (libre o en uso).

Procesos tenedor. Sincronización

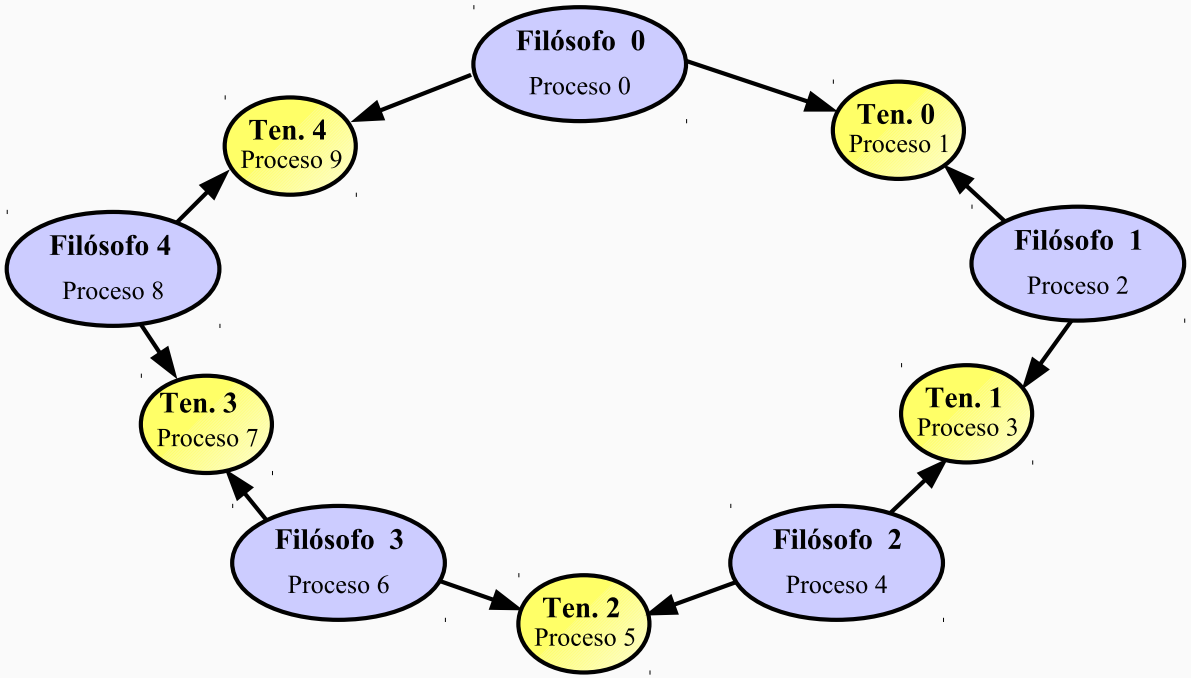
Por tanto, para implementar el problema, **debemos de ejecutar** 5 **procesos de tipo tenedor**, numerados del 0 al 4.

- Cuando un proceso filósofo va a usar un tenedor, debe de enviar (al proceso tenedor correspondiente) un mensaje síncrono antes de usarlo y otro mensaje después de haberlo usado.
- Cada proceso tenedor ejecuta un bucle infinito, al inicio de cada iteración está libre, y da estos dos pasos:
  1. Espera hasta recibir un mensaje de cualquier filósofo, al recibirlo el tenedor pasa a estar ocupado por ese filósofo.
  2. Espera hasta recibir un mensaje del filósofo que lo adquirió en el paso anterior. Al recibirlo, pasa a estar libre.
- Puesto que el envío (por el filósofo) del mensaje previo al uso es síncrono, supone para dicho filósofo una espera bloqueada hasta adquirir el tenedor en exclusión mutua.

Identificadores de los procesos

Para facilitar la comunicación entre filósofos y tenedores:

- Los procesos filósofos tienen identificadores MPI pares, es decir, el filósofo número i tendrá identificador $2i$.
- Los procesos tenedor tienen identificadores MPI impares, es decir, el tenedor número i tendrá identificador $2i + 1$.



Cena de los filósofos. Programa principal

La función **main** (en `filosofos-plantilla.cpp`) es esta:

```
const int num_filosofos = 5 ,                // número de filósofos
          num_filo_ten  = 2*num_filosofos,   // núm. de filo. + ten.
          num_procesos  = num_filo_ten;      // núm. total de procs.
.....
int main( int argc, char** argv )
{
  int id_propio, num_procesos_actual ;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
  if ( num_procesos == num_procesos_actual )
  { if ( id_propio % 2 == 0 )          // si es par
      funcion_filosofos( id_propio ); //    es un filósofo
    else                              // si es impar
      funcion_tenedores( id_propio ); //    es un tenedor
  }
  else if ( id_propio == 0 )
    cerr << "Error: se esperaban 10 procesos. Programa abortado." <<endl;
  MPI_Finalize( );
  return 0;
}
```

Procesos filósofos

En cada iteración del bucle un filósofo realiza repetidamente estas acciones:

1. Tomar los tenedores (primero el tenedor izquierdo y después el derecho).
2. Comer (bloqueo de duración aleatoria).
3. Soltar tenedores (el orden da igual).
4. Pensar (bloqueo de duración aleatoria).

- Las acciones pensar y comer pueden implementarse mediante un mensaje por pantalla seguido de un retardo durante un tiempo aleatorio.
- Las acciones de tomar tenedores y soltar tenedores deben implementarse enviando mensajes **síncronos seguros** de petición y de liberación a los procesos tenedor situados a ambos lados de cada filósofo.

Esquema de los procesos filósofos

```
void funcion_filosofos( int id )
{
  int id_ten_izq = (id+1)                % num_filo_ten, // id. ten. izq.
      id_ten_der = (id+num_filo_ten-1) % num_filo_ten; // id. ten. der.

  while ( true )
  {
    cout <<"Filósofo " <<id << " solicita ten. izq." <<id_ten_izq <<endl;
    // ... solicitar tenedor izquierdo (completar)
    cout <<"Filósofo " <<id <<" solicita ten. der." <<id_ten_der <<endl;
    // ... solicitar tenedor derecho (completar)

    cout <<"Filósofo " <<id <<" comienza a comer" <<endl ;
    sleep_for( milliseconds( aleatorio<10,100>() ) );

    cout <<"Filósofo " <<id <<" suelta ten. izq. " <<id_ten_izq <<endl;
    // ... soltar el tenedor izquierdo (completar)
    cout<< "Filósofo " <<id <<" suelta ten. der. " <<id_ten_der <<endl;
    // ... soltar el tenedor derecho (completar)

    cout << "Filosofo " << id << " comienza a pensar" << endl;
    sleep_for( milliseconds( aleatorio<10,100>() ) );
 }
}
```

Procesos tenedor

Cada proceso tenedor ejecutará en un bucle estas dos acciones:

1. Esperar hasta recibir un mensaje de cualquier filósofo (lo llamamos *mensaje de petición*)
2. Esperar hasta recibir un mensaje del mismo filósofo emisor del anterior (lo llamamos *mensaje de liberación*)

```
void funcion_tenedores( int id ) // id es el identificador del proceso tenedor
{
  int valor, id_filosofo ;  // valor recibido, identificador del filósofo
  MPI_Status estado ;       // metadatos de las dos recepciones
  while ( true )
  {
    // ..... recibir petición de cualquier filósofo (completar)
    // ..... guardar en id_filosofo el id. del emisor (completar)
    cout <<"Ten. " <<id <<" cogido por filo. " <<id_filosofo <<endl;

    // ..... recibir liberación de filósofo id_filosofo (completar)
    cout <<"Ten. "<< id<< " liberado por filo. " <<id_filosofo <<endl ;
  }
}
```

Actividades : soluciones con interbloqueo y sin interbloqueo

Se propone realizar las siguientes actividades

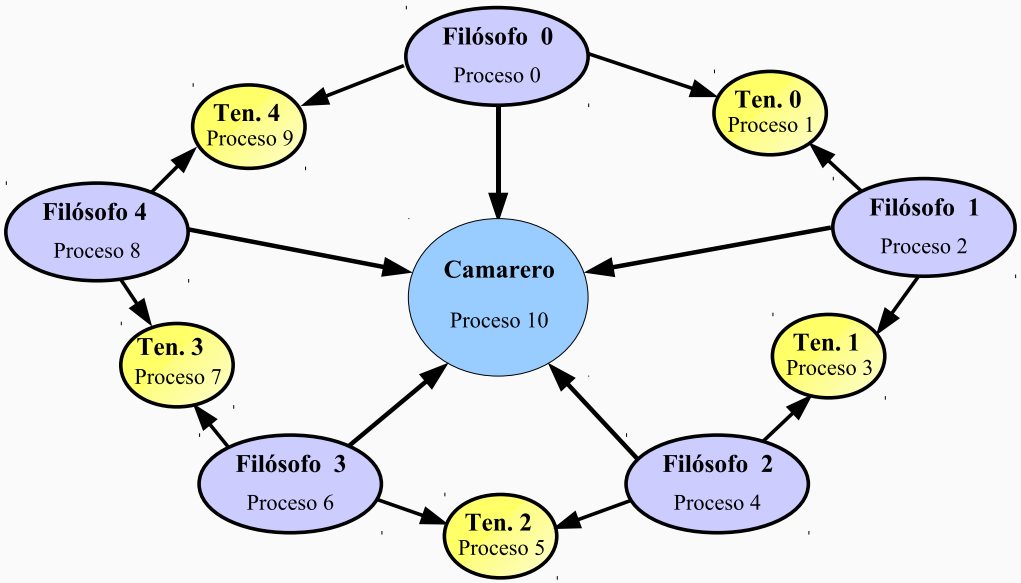
1. Implementar una solución distribuida al problema de los filósofos de acuerdo con el esquema descrito en las plantillas. Usar la operación síncrona de envío **`MPI_Ssend`**. Copia el archivo de la plantilla (`filosofos-plantilla.cpp`) en el archivo `filosofos-interb.cpp` y completa este último archivo.
2. El esquema propuesto (cada filósofo coge primero el tenedor de su izquierda y después el de la derecha) puede conducir a interbloqueo:
   - Identifica la secuencia de peticiones de filósofos que conduce a interbloqueo.
   - Diseña una modificación que solucione dicho problema.
   - Copia `filosofos-interb.cpp` en `filosofos.cpp` e implementa tu solución en este último archivo.

Subsección 2.2.
Uso del proceso camarero con espera selectiva.

Cena de los filósofos con camarero

Existe otra opción para solucionar el problema del interbloqueo:

- Se introducen dos pasos nuevos en los filósofos:
  - *sentarse en la mesa* (antes de coger los tenedores)
  - *levantarse de la mesa* (después de soltar los tenedores)
- Un proceso adicional llamado **camarero** (identificador 10) impedirá que haya 5 filósofos sentados a la vez.



Procesos filósofos y camarero

Ahora, cada filósofo ejecutará repetidamente esta secuencia:

1. Sentarse
2. Tomar tenedores
3. Comer
4. Soltar tenedores
5. Levantarse
6. Pensar

Cada filósofo pedirá permiso para sentarse o levantarse haciendo un envío síncrono al camarero. Debemos de implementar de nuevo una **espera selectiva** en el camarero, el cual

- llevará una cuenta (s) del número de filósofos sentados.
- solo cuando $s < 4$ aceptará las peticiones de sentarse.
- siempre aceptará las peticiones para levantarse.

De nuevo, debes de usar etiquetas para esta implementación. Recuerda definir constantes enteras para etiquetas, cuyos nombres deben comenzar por `etiq_`.

Actividades: solución con camarero

Realiza las siguientes actividades:

1. Copia tu solución con interbloqueo (`filosofos-interb.cpp`) sobre un nuevo archivo llamado `filosofos-cam.cpp`
2. Implementa, en este último archivo, el método descrito, basado en un proceso camarero con espera selectiva.

Fin de la presentación.