Paso de mensajes

Lecturas:
Burns & Wellings, Cap. 6
Transparencias y apuntes de la asignatura

Manuel Carro

Universidad Politécnica de Madrid

Este texto se distribuye bajo los términos de la *Creative Commons License*

Paso de mensajes

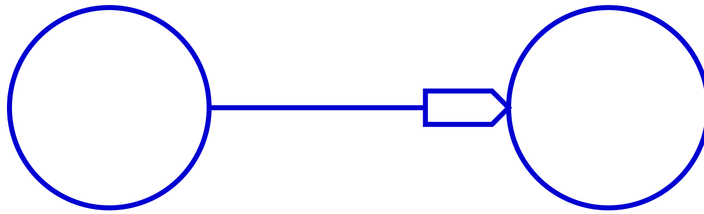
- Motivación: sistemas distribuidos:
  - Ausencia de memoria compartida
  - Datos repartidos entre procesos / tareas
  - Quizás en máquinas distintas
  - Acceso a datos mediante comunicación de procesos: paso de mensajes
  - También para sincronización
- Aparecen al conectar ordenadores a una red
- Exclusión mutua *gratis* (no hay datos físicamente compartidos)
- Problema: comunicación entre procesos

Filosofía cliente-servidor

- Procesos *servidor* ejecutándose en un ordenador
- Procesos *cliente* se ejecutan donde resulte más conveniente al usuario
- Filosofía cliente-servidor adapta al contexto distribuido los conceptos de recursos compartidos del tema anterior
- Procesos resultantes del análisis darán lugar a procesos cliente
- Interacción se convierte en proceso servidor que encapsula el recurso

Canales de Comunicación

Los procesos se ejecutan en espacios de memoria independientes y se comunican intercambiando mensajes, que contienen **copias** de variables del emisor



Envío: Send (Canal, Mensaje)
Recepción: Receive (Canal, Mensaje)

Lenguajes para paso de mensajes

Clasificación según las características de los canales de comunicación entre procesos:

- Por el **comportamiento dinámico** de los canales (síncrono o asíncrono)
- Por el **nombrado** de los canales (explícito o implícito)
- Por la **cardinalidad** de los esquemas de comunicación (1:1, n:1, n:m)

Sincronicidad de los Canales

- ¡La recepción **siempre** es bloqueante!

Asíncronos: Envío **no bloqueante**; el emisor sigue ejecutándose sin esperar a que la comunicación tenga efecto (telegramas, emails, . . .)
- Ventajas: mayor concurrencia — útil si el tiempo de respuesta es elevado
- Inconvenientes: Necesita de colas de mensajes (puede ser costoso)

Síncronos: Envío **bloqueante**; emisor y receptor se sincronizan para intercambiarse cada mensaje (hablar en persona, por teléfono. . .)

- Mayoría de lenguajes optan por canales síncronos
- Canales asíncronos: protocolos de comunicaciones (TCP/IP) o computación paralela

Esquemas de comunicación

(1:1) Cada canal tiene un emisor y un receptor.

(n:1) Varios procesos pueden enviar mensajes por el canal y sólo uno puede recibirlos, el habitual admite nombrado explícito (*ports*)

(n:m) Varios emisores y varios receptores (no ha tenido mucho éxito)

Nombrado de Canales

Nombrado explícito: Los canales son un tipo de datos del lenguaje y el programador debe declararlos, asignarlos, destruirlos, etc.

Ventajas:

- Pueden declararse vectores o matrices de canales
- Pasar canales como datos: útil en entornos cliente-servidor

Inconvenientes:

- Más trabajo para el programador

Nombrado implícito: Canales ocultos al programador: se identifican proporcionando un nombre del proceso receptor

Ventajas:

- Menos código, no es necesario declarar, ni destruir

Inconvenientes:

- Sólo permiten comunicación 1:1

Ejemplo: Productor-Consumidor

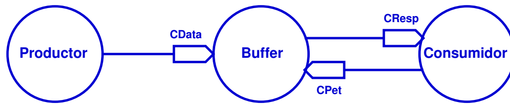
Primer intento: envío directo

- Asumimos una notación de paso de mensajes con canales explícitos síncronos
- Send (Canal,*mensaje*) $\Longrightarrow$ Receive (Canal,*mensaje*)
- La codificación resulta inmediata
- Conocimiento de Canal: aparece aquí como variable compartida
- En un verdadero entorno distribuido sería una dirección / nombre conocido por ambos procesos
- ¿Dónde está el *buffer*?

Productor-Consumidor (Cont.)

Segundo intento: *buffer* en proceso intermedio

- Evitar acoplamiento con encolado de datos: *buffering*



Productor	Buffer	Consumidor
...	...	...
Producir (Dato);	Receive (CDato,MiDato);	Send (CPet,CResp);
Send (CDato,Dato);	Insertar (MiBuffer,MiDato);	Receive (CResp,MiDato);
...	Receive (CPet,P');	Consumir (MiDato);
	Extraer (MiBuffer,MiDato);	...
	Send (P',MiDato);	

- Acoplamiento de velocidades

Veremos:

1. Versión Ada (nativa) de paso de mensajes (*Rendez-Vous*)
2. Versión Ada de selección alternativa

Rendez-Vous

La versión edulcorada de paso de mensajes en Ada

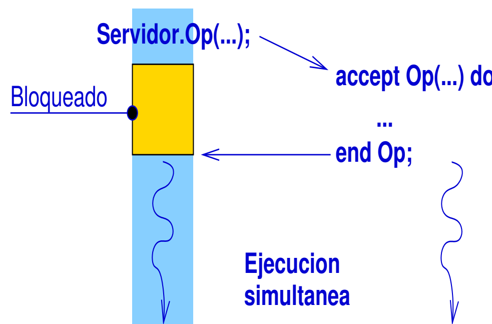
Espera por una *llamada* desde otra tarea

```
accept Op (A1, ..., An) do
   ...
end Op;
```

Tarea llamante

```
Servidor.Op (A1, ..., An);
```

- Op como *nombre estático* de canal
- Argumentos: pueden ser de entrada / salida
- Tarea llamante bloqueada hasta final *Rendez-Vous*



Recepción alternativa

```
select
    when Cond1 =>
        <accept1>
        I1;
or
    when Cond2 =>
        <accept2>
        I2;
or
    ...
end select;
```

- $accept_i$ una construcción como las ya vistas
- I_i una secuencia arbitraria de instrucciones
- Espera llamadas a $accept_i$ de entre aquellas en que $Cond_1$ es cierto
- $Cond_1$: únicamente variables visibles
  - **Nunca** parámetros de $accept_i$

Ejemplo: Buffer con *Rendez-Vous*

```
loop
  select
    when not Lleno (Buff) =>
    accept Poner (D: in T_Dato)
    do
      Insertar (Buff, D)
    end Poner;
  or
    when not Vacio (Buff) =>
    accept Tomar (D: out T_Dato)
    do
      Retirar (Buff, D);
    end Tomar;
  end select;
end loop;
```

```
loop -- Productor
  Producir (Dato);
  Tarea.Poner (Dato);
end loop;
```

```
loop -- Consumidor
  Tarea.Tomar (Dato);
  Consumir (Dato);
end loop;
```


Crítica *Rendez-Vous*

- Adecuado para problemas simples
- Pero oculta características propias de los sistemas basados en mensajes:
  - Devolución explícita de datos mediante canales
  - Send / Receive para sincronizar
  - Comunicación **fuera** del *espacio* del *Rendez-Vous*
- Necesita una implementación de canales *explícitos*
- Implementados como un CTAD: `channels.ads`

```
procedure Send (C : in out Channel;
                M : in     Message);
procedure Receive (C : in out Channel;
                   M :    out Message);
```

- Síncrono, nombrado explícito, $n : m$
- Lo usaremos sólo como $1 : 1$

Rendez-Vous + canales

- accept para recibir canales de comunicación
- Canales para pasar datos

```
select
  when not Lleno (B) =>
    accept Poner (Chan : in out Channel_P) do
      Local_Channel := Chan;
    end Poner;
  Receive (Local_Channel,D);
  Insertar (B, D);
or
  when not Vacio (B) =>
    accept Tomar (Chan: in out Channel_P) do
      Local_Channel := Chan;
    end Tomar;
  Retirar (B, D);
  Send (Local_Channel,D);
end select;
```

- ¿Por qué código fuera de accept?

Rendez-Vous + canales (Cont.)

Código del productor:

```
loop
   Producir (Dato);
   Servidor.Poner (ChanPoner);
   Send (ChanPoner, Dato);
end loop;
```

- Clientes esperando en Send / Receive: posibilidad de despertar explícito
- Además: más concurrencia (entre servidor y clientes)

Rendez-Vous y dependencia parámetros

- *Rendez-Vous*: guardas sólo usan estado
- Recurso activo:
  - No hay familias de *entries*
  - *requeue* no tiene mismo significado
- Pero tenemos canales:
  - Permiten comunicación explícita
  - Identifican procesos (cada canal almacenado corresponde a un proceso esperando)
  - Deben ser manejados / planificados explícitamente
- En general: canales como *entidades externas* que permiten realizar sincronización y paso de datos
- Paralelismo canal / identificador proceso

Buffer par/impar con *Rendez-Vous*

- Dependencia estado → proyección condiciones a nivel superior:

```
entry Tomar_Par (Item: out Integer);
entry Tomar_Impar (Item: out Integer);
```

- Respetar interfaz: englobar llamadas

```
procedure Tomar (S:    in out Server;
                 Req:  in     Req_T;
                 Dato:    out Integer) is
begin -- Descomponer según requerido
   if Req = Par then
     Server.Tomar_Par (Dato);
   else
     Server.Tomar_Impar (Dato);
   end if;
end Tomar;
```

- Idea idéntica a objetos protegidos

Buffer par / impar con *Rendez-Vous*

Tras la descomposición:

```
when Hay_Dato and then Dato mod 2 = 0 =>
   accept Tomar_Par (Item : out Integer) do
      Item := Dato;        -- Item: parte del estado
      Hay_Dato := False;   -- Hay_Dato: parte del estado
   end Tomar_Par;
```

- Como en objetos protegidos, puede haber problemas prácticos: muchos casos
- Solución más general: pasar canales, guardarlos, aceptar parámetros, decidir según parámetros

Dependencia de parámetros y paso de mensajes

- *entries* con dependencia de parámetros de entrada siempre aceptan canal (privado a cliente)
- Se requieren datos a clientes según sea necesario
  - Pero siempre necesitaremos los parámetros necesarios para evaluar condición de sincronización
- El aceptar / devolver datos permite que los clientes (re)arranquen
- Dos partes claramente diferenciadas en servidor:
  - Recepción de peticiones y canales, almacenamiento
  - Servicio de peticiones almacenadas
- Canales: vínculo entre clientes y recurso

Esquema de un servidor

```
when True =>
   accept .... do
      Guardar_Canal_Y_Datos (...);
   end accept;
```

- Aceptar todas las peticiones
- Con un canal asociado

```
while CPRE1 or CPRE2 or ... loop
   if CPRE1 then
      Extraer_Canal_Y_Datos (..., C);
      <Computar respuesta>
      Send (C, ...);
   elsif CPRE2 then
      ...
   end if;
end loop;
```

- ¿Qué peticiones <u>pueden</u> atenderse?
- Decidir entre ellas cual <u>debe</u> atenderse
- **Necesario** decidir:
  - Desbloqueo explícito
  - Análisis de vivacidad

Consumidor

- Se pasa tipo de petición y se espera dato

```
Data_Channel: Channel_Int.Channel_P := new ...

   loop
      Servidor.Tomar (Which_Type, Data_Channel);
      Receive (Data_Channel, Dato);
   end loop;
```

- Tipo de petición se acepta siempre
- Send en el recurso cuando se dan las condiciones
- **Nota:** productor no cambia

Recepción de peticiones: Tomar

Pend_Tomar: array (Req_Type) of Channel_Queue;

```
   when True =>
   accept Tomar (Pet: Req_Type;
                 C_D: in out Ch_Int.Ch_P) do
       Insertar (Pend_Tomar (Pet), C_D);
   end Tomar;
```


Servicio petición en pares e impares

- Sólo hemos almacenado peticiones de Tomar
- Sólo debemos atender peticiones de Tomar

```
while (Hay_Dato and ( (not Es_Vacia(Pend_Tomar(Par)))
                      and Dato mod 2 = 0) or ... ) loop
    if not Es_Vacia(Pend_Tomar(Par) and then
          (Dato mod 2 = 0) then
       Primero (Pend_Tomar(Par), Canal_Tomar);
       Borrar (Pend_Tomar(Par));
       Send (Canal_Tomar, Dato);
       Hay_Dato:= False;
    elsif not Es_Vacia(Pend_Tomar(Impar) and then
          (Dato mod 2 = 1) then
       ...
    end if;
end loop;
```

- Puede acortarse con un bucle for I in Req_Type loop ...
- **Atención:** preferencia por un tipo de petición ⟶ inanición

Un caso particular

- A veces atender en orden de llegada es suficiente
- Sólo necesitamos guardar un registro de activación
- Similar a doble bloqueo
- Idea de esquema:
  - Sólo una instancia suspendida de cada operación
  - Con parámetros/canales respuesta guardados
  - Al recibir operación:
    - Aceptar sólo si no instancia suspendida ya
    - Si hay instancia suspendida, guardar en registro de activación, atender después

Esquema de código

```
select
  when not Op_1_Delayed =>
  accept Op_1 (Par: in ...;
               Ans: in out Channel_P) do
    Par_Op_1:= Par;
    Ans_Chan_1:= Ans;
    Op_1_Delayed := True;
  end Op1;
  ⋮
end select

while ... loop
   if Op_1_Delayed and then
      Pre_Op_1 (State, Par_Op_1) then
        State:= ...;
        Send (Ans_Chan_1,...);
        Op_1_Delayed := False;
   end if;
«comprobaciones de otras operaciones»
end loop;
```


Multibuffer con doble bloqueo

- Multibuffer *estilizado*: sólo número elementos
- **Poner**, **Tomar** dependen de argumentos
- Dos pasos:
  1. Aceptar número elementos a poner/tomar
  2. Aceptar elementos en sí
- Segundo paso vacío en este ejemplo: lo usamos para despertar cliente

Cliente de multibuffer

```
loop
   << Produce Num datos >>
    Pool.Get (Num, Get_Channel);
    Receive (Get_Channel, Data);
end loop;
```

- Data sería el vector de elementos
- Send en servidor despierta clientes cuando se acepta su petición

Servidor multibuffer: aceptar peticiones

- Veremos esquema para el Get (Put sería similar)

```
when not Delayed_Get =>
accept Get (Num_Items : in  Buffer_Quantity;
            The_Get_Channel: in out Channel_P) do
  Num_Items_To_Get := Num_Items;
  Get_Channel := The_Get_Channel;
  Delayed_Get := True;
end Get;
```

- Máx. una petición suspendida en cada momento
- Aceptamos únicamente los datos necesarios para decidir si atender o no

Multibuffer: atender petición almacenada

- Código inmediatamente **detrás** de la selección alternativa
- Se ejecuta siempre tras aceptar una operación

```
if Delayed_Get and then
  Item_Counter >= Num_Items_To_Get then
    Item_Counter:= Item_Counter - Num_Items_To_Get;
    Send (Get_Channel, Data);
    Delayed_Get := False;
end if;
```

- Se ejecuta cuando hay **Get** suspendido y la llamada suspendida puede atenderse
- Data contendría secuencia de elementos a devolver

Políticas explícitas

Cómo despertar la operación / proceso que queramos

- Fácil:
  - Admitir canal en servidor (ya sea para devolver o para recibir datos)
  - Guardar canal (quizás con datos asociados a la llamada)
  - Escoger qué proceso (canal) servir
- La elección puede (debe):
  - Evaluar estado recurso
  - Evaluar parámetros asociados a la llamada
- Sustituye PID proceso por canal (único por cliente)

Políticas explícitas (Cont.)

Cuenta compartida algún tipo de prioridad explícita

```
select
    ...
    accept Retirar (Cant: in Integer; Acep: in out Chan)
    when True do
        <<Guardar canal asociado a Cant>>
    end Retirar;
or
...
end select;

while <podemos atender petición> loop
    <seleccionar mejor petición>
    <responder por canal>
end loop;
```

- El cliente de Retirar espera en Acep
- Al depositar: examinar canales, satisfacer mejor petición
- Estructura de datos dependiente de noción de *mejor petición*