PROGRAMACION CONCURRENTE

II.2 Invocación remota de procedimientos

J.M. Drake

Notas:

Mecanismos de sincronización

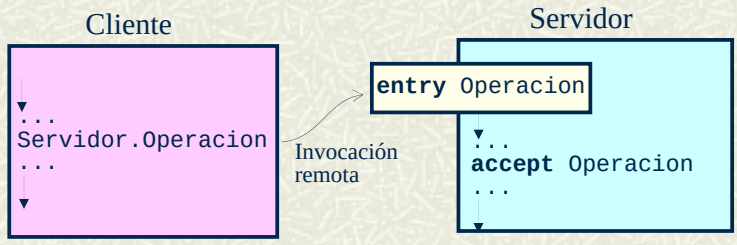
- Implementa el tercer mecanismo de sincronización de las primitivas Send-Wait:
  - **Envío Asíncrono**: El proceso emisor continua con independencia del estado del receptor.
  - **Envío Síncrono (Rendezvous simple):** El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido recibido. Las sentencias Send y Wait terminan síncronamente.
  - **Invocación Remota (Rendezvous completo):** El proceso emisor permanece suspendido hasta confirmar que el mensaje ha sido aceptado. Emisor y receptor ejecutan síncronamente un segmento de código. Las sentencias Send y Wait terminan síncronamente.
- La invocación remota de procedimiento es un mecanismo de comunicación **síncrona, con denominación directa** y **asimétrica**, que sigue un formalismo semejante a la declaración e invocación de procedimientos (RPC).

Notas:

La **invocación remota de procedimientos** ("remote procedure invocation") corresponde a un modelo de comunicación síncrona entre procesos basada en intercambio de mensajes, y que sigue un formalismo semejante a la llamada de procedimientos. Conceptualmente, la invocación remota se pueden definir como un mecanismo por el que un proceso puede **ejecutar un procedimiento que pertenece al entorno de otro proceso**. Esta es la estrategia de comunicación entre procesos que ha sido adoptada por el lenguaje Ada.

Invocación de procedimiento remoto.

- La invocación de procedimiento remoto formalmente consiste en que los procesos receptores del mensaje (servidores) ofrecen procedimientos públicos (**entry**) que otros procesos pueden invocar.
- Un procedimiento remoto (entry) es formalmente similar a un procedimiento público, pero se diferencian en:
  - Un procedimiento público se ejecuta incondicionalmente utilizando el thread del proceso invocante.
  - Un entry se ejecuta solo cuando se sincronizan el thread del proceso que invoca y el thread del proceso propietario. Este puede aceptar o no la invocación.



Notas:

Para comprender la motivación de este modelo, consideremos inicialmente la relación que se establece entre un procedimiento ordinario y el proceso que lo invoca:

* Un mismo procedimiento puede ser llamado desde cualquier punto de un programa que tenga visibilidad sobre su nombre.

* En la llamada a un procedimiento se produce intercambio de información en ambos sentidos, esto es, en la cabecera de un procedimiento existen argumentos de entrada y de salida.

* Un procedimiento posee un bloque de sentencias que en una invocación al mismo son ejecutadas dentro de la línea de flujo de control del proceso que realiza la llamada.

Los procedimientos son componentes básicos de todos los lenguajes de programación modernos, y dentro de un lenguaje concurrente pueden ser incluso invocados concurrentemente con otros procesos que incluso también han invocado el mismo procedimiento.

Componentes de la invocación remota.

- En el **proceso propietario (Servicio)** del procedimiento remoto existen sentencias para declarar el procedimiento remoto y para aceptar su invocación:

```
process propietario;
    entry procedimientoRemoto(dato:TipoDato);
begin
....
    accept procedimientoRemoto(dato:TipoDato) do
        -- Bloque de sentencias
....
end;
```

- En el **proceso invocante (Cliente)** existe sentencias de invocación del procedimiento:

```
    ....
    propietario.procedimientoRemoto(nuevoDato);
    .....
```


Notas:

El proceso Propietario ofrece el procedimiento remoto Procedimiento_Remoto, con un argumento de entrada Dato. En su cuerpo la ejecución de la sentencia accept por su thread representa la aceptación de una posible invocación del procedimiento.

En el proceso invocante se invoca el procedimiento remoto siguiendo un formalismo compuesto similar al que se utiliza para invocar un procedimiento de un paquete.

Cuando un procedimiento remoto es invocado, el proceso que invoca se suspende hasta que el proceso que contiene el procedimiento llamado, lo acepta. A partir de ese instante, ambos procesos unifican su línea de flujo de control, y ejecutan conjuntamente el cuerpo del procedimiento remoto. Cuando este termina, las dos líneas de control de flujo quedan independizadas y ambos procesos evolucionan de acuerdo con su estructura.

La invocación de un procedimiento remoto es un mecanismo de comunicación síncrono, de nominación directa, y multipunto. Cualquier proceso que conozca el nombre del proceso y el nombre del procedimiento, puede invocarlo, e incluso concurrentemente. Dado que el procedimiento remoto es único, el proceso al que pertenece es el responsable que las invocaciones concurrentes al procedimiento sean secuencializadas de forma compatible con su estructura.

Invocación remota y programación orientada a objetos.

- El formalismo de sincronización por procedimientos remotos está concebida para implementar objetos que se comunican según el paradigma cliente-servidor.
  - **Objetos activos**: Se implementa como procesos que no ofrecen procedimientos remotos. En su cuerpo invocan procedimientos remotos de otros objetos.
  - **Objetos pasivos**: Se implementan como procesos que ofrecen como procedimientos remotos los servicios que ofrecen.
  - **Objetos neutros**: Son estructuras de datos a los que se acceden mediante una interfaz compuesta de procedimientos públicos ordinarios.

Notas:

El concepto de invocación de procedimientos remotos, esta en gran medida concebido para ser la base de una estrategia de programación orientada a objetos, y siguiendo el paradigma de configuraciones cliente-servidor. Cuando se utiliza los mecanismos de invocación remota de procedimientos, se deben emplear las siguientes utilizaciones de los componentes de programación:

* Los **objetos activos** se realizan como procesos que no tienen entradas, y que en su código si hacen invocaciones (uso) a las entradas de los objetos pasivos.

* Los **objetos neutros** son estructuras de datos recubiertas de una interfaz de procedimientos ordinarios de acceso.

* Los **objetos pasivos** se realizan también mediante procesos, que gestionan activamente los recursos que contienen, y a los que se acceden mediante una interfase de procedimientos remotos (entry).

Es importante contrastar la diferencia entre la invocación remota de procedimiento y la **llamada remota a procedimientos o RPC** ("remote procedure call"). Mientras que la primera es un mecanismo del lenguaje por el que un proceso puede ejecutar un procedimiento de otro proceso, dentro de un programa concurrente, las RPC son mecanismos de comunicación de alto nivel entre computadores vía una red telemática, por las que un programa puede ejecutar un procedimiento ubicado en otro computador. Aunque el nivel de abstracción de ambos conceptos sea muy diferente, ambos juegan un papel similar, uno dentro de un programa concurrente, y el otro dentro de un sistema software distribuidos. Ambos tienen la función de constituir la interfaz de los objetos pasivos servidores, dentro del paradigma de diseño orientado al objeto.

Estructura de los procedimientos remotos.

- Los procedimientos remotos se declaran en la cabecera del procedimiento mediante sentencias entry y en las que se declaran los argumentos de forma idéntica a un procedimiento ordinario.

```
process propietario;
entry intercambia( entrada: Integer; var salida: Integer);
var d1, d2 : integer;
begin
   ....
   d2:= AlgunValor;
   ....
   accept  intercambia( entrada: Integer; var salida: Integer) do
      begin
          d1:= entrada;
          salida := d2;
      end;
   ....
end;
```


Notas:

Los procedimientos remotos están definidos dentro de la estructura de los procesos. Existen dos componentes específicas para definir los procedimientos remotos, su declaración en la cabecera del proceso mediante la sentencia **entry**, y la declaración del punto del proceso en la que se acepta su ejecución **accept**.

Todos los procedimiento remotos de un proceso deben estar declarado justamente tras la cabecera del proceso mediante sentencias **entry.** En la declaración de un procedimiento remoto se declaran también los argumentos de entrada y salida del procedimiento. La declaración de los procedimientos siguen una nomenclatura semejante a la declaración de los argumentos en los procedimientos ordinarios:

Ejemplos de declaración de procedimientos remotos son:

entry Coloca (C: char);

entry Toma (**var** C : char);

entry Intercambia(Entrada: integer; **var** Salida: integer);

entry Opera (A, B : integer; **var** C, D : integer ; E : boolean);

Una invocación a un procedimiento remoto solo se atiende si el flujo de control del proceso al que pertenece se encuentra ejecutando una sentencia **accept** relativa a ese procedimiento.

Semántica de la sentencia accept.

- Una invocación a un procedimiento remoto solo se ejecuta si el procedimiento propietario ejecuta una sentencia **accept** relativa a ella.
- Cada **entry** tiene asociada una cola de procesos invocantes que tratan de ejecutar el procedimiento. De estas invocaciones es atendida una por cada ejecución de la sentencia accept relativa a ella.
- En el cuerpo del proceso propietario debe existir al menos una sentencia accept por cada entry declarada en la cabecera.
- La sentencia accept suspende el thread del proceso propietario, si no existe pendiente ninguna invocación de la correspondiente entry.
- El código del bloque incluido en un accept puede bien estar vacío o ser arbitrariamente grande e incluso incluir otras sentencias accept.

Notas:

Una invocación a un procedimiento remoto solo se atiende si el flujo de control del proceso al que pertenece se encuentra ejecutando una sentencia **accept** relativa a ese procedimiento.

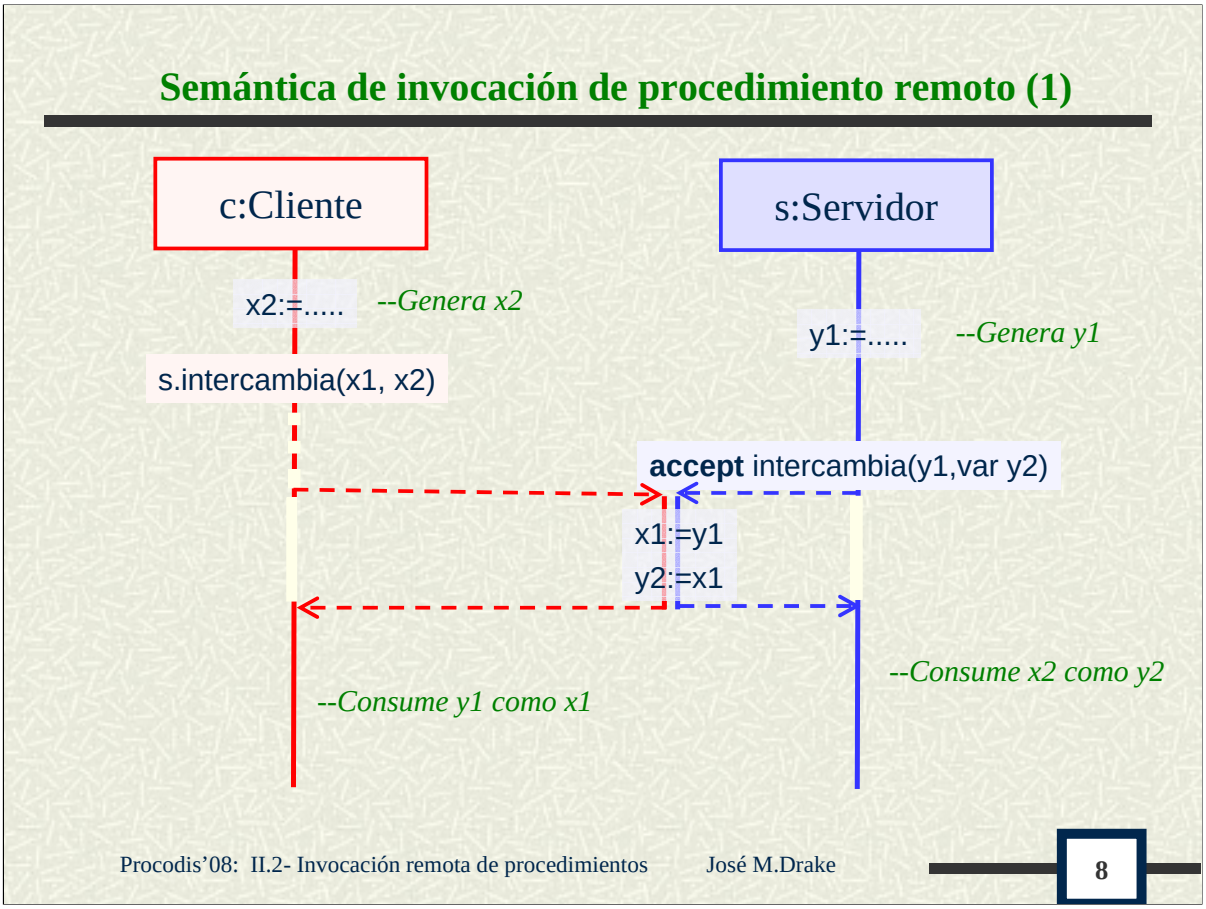
A cada procedimiento remoto o "entry" se le asigna una cola de procesos en espera de ser atendidos. Los procesos de la cola son atendidos uno a uno por cada una de las sentencias "accept" relativas a ese procedimiento que ejecute el proceso propietario. La selección de la llamada (entre las que se encuentra a la espera en la cola) se lleva a cabo siguiendo diferentes criterios según el lenguaje (Ada sigue una política FIFO, Pascal FC sigue una política de cola de prioridad, etc.).

En el código del proceso debe haber al menos una sentencia "accept" por cada procedimiento remoto que esté declarado en él. Esta es una sentencia estructurada que incorpora el bloque de sentencias que constituyen el cuerpo del procedimiento remoto.

La interacción entre el proceso que realiza la invocación y el propietario del procedimiento llamado, es un encuentro (rendezvous) que implica sincronización de los procesos. El primero de los dos que llega al punto de comunicación (llamada o aceptación) debe suspenderse, y permanecer en ese estado hasta que el otro alcanza el punto complementario de comunicación. A partir de ese momento, el proceso que acepta la llamada continua ejecutando el cuerpo del "accept", y solo cuando este alcanza su "end", finaliza el rendezvous, y ambos procesos continúan de forma independiente.

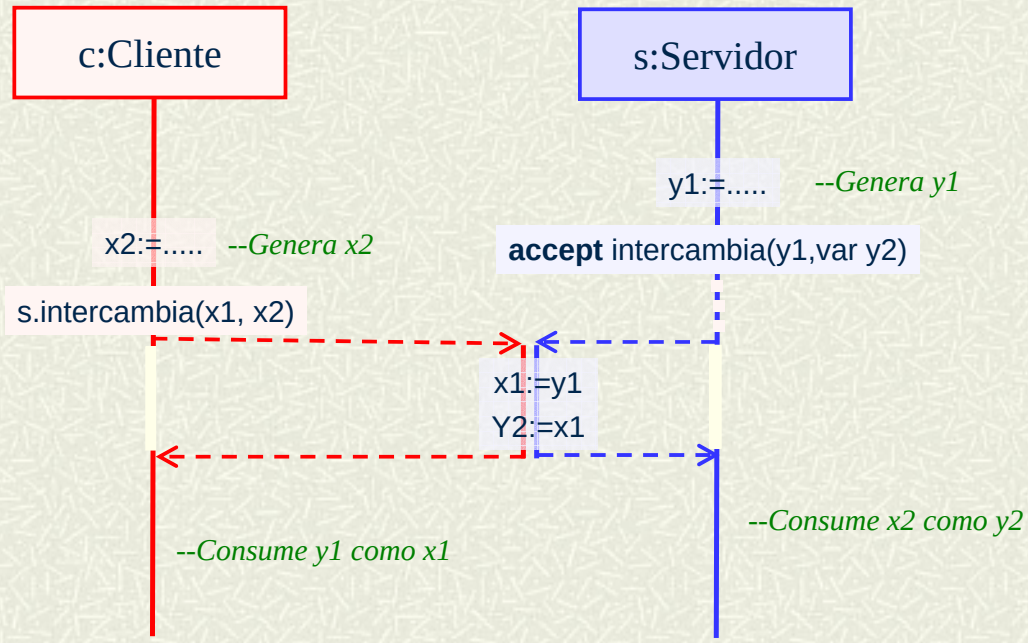
Es importante resaltar que el código del "accept" puede ser arbitrariamente complejo, y contener otros "accept", llamadas a otros procesos, o cualquier otra estructura del lenguaje.

En el otro extremo, el cuerpo del "accept" puede estar vacío, en cuyo caso la llamada al procedimiento solo tiene la finalidad de constituir un punto de sincronización.


Semántica de invocación de procedimiento remoto (1)
c:Cliente
s:Servidor
x2:=.....
--Genera x2
y1:=.....
--Genera y1
s.intercambia(x1, x2)
accept intercambia(y1,var y2)
x1:=y1
y2:=x1
--Consume x2 como y2
--Consume y1 como x1
Procodis'08: II.2- Invocación remota de procedimientos
José M.Drake
8

Notas:

Semántica de invocación de procedimiento remoto (2)



Notas:

Select asimétrico e invocación remota.

- La **sentencia select** permite que un proceso quede suspendido a la espera de múltiples eventos.
- La **condición de guarda** en las alternativas de un select permiten adaptar las alternativas al estado instantáneo del proceso.
- Aunque los tipos de eventos que se pueden establecer como alternativa son diversos, en el **select asimétrico** limita los que pueden agruparse juntos:
  - **Select servidor**:
    - accept.
    - timeout.
    - terminate.
    - else.
  - **Select cliente**:
    - invocación de procedimientos remoto.
    - timeout

Notas:

Cuando un proceso realiza una llamada a un procedimiento remoto, o cuando un proceso ejecuta una sentencia accept, queda suspendido hasta que el evento esperado se produce. El hecho de que un proceso solo pueda encontrarse suspendido a la espera de un único tipo de evento es una limitación no aceptable. Todos los lenguajes introducen la sentencia **select** con la que un proceso puede permanecer suspendido a la espera de diferentes tipos de eventos.

La sentencia select es similar y con prestaciones semejantes a la descrita en la comunicación a través de canales. Al igual que allí, la estructura select admite sentencias de aceptación de llamadas (accept), llamadas a procedimientos remotos, y sentencias de temporización (timeout), de finalización (terminate) y "else". Así mismo, acepta **condiciones de guarda** para limitar los eventos que se admiten en una ejecución.

La diferencia más notable que introducen algunos lenguajes (Ada entre ellos) es establecer un **select asimétrico** respecto a las invocaciones de los procedimientos y a la aceptación de las mismas. Esta asimetría se introduce a fin de mantener la abstracción diferenciada entre los objetos activos y de los objetos pasivos. La principal naturaleza de la asimetría es que no se admite la mezcla en una misma estructura select de invocaciones y aceptaciones de llamadas a procedimientos remotos. Con ello se busca que los objetos pasivos solo dispongan estructuras select de aceptación de llamadas, mientras que los objetos activos solo posean estructura select de invocaciones. (En Ada, en las estructuras select que incluyen un invocación a un procedimiento remoto, solo se admite una alternativa de temporización).

Ejemplo: Control de acceso a variable compartida.

```
Program AccesoConcurrente;
  var dato:VariableCompartida;

  process escritor;
    i:Integer;
  begin
   ...
    dato.write(i);
   ...
  end;

  process lector;
     n: integer;
  begin
   ...
   dato.read(n);
   ...
  end;

begin
   dato.write(0);
   cobegin
      escritor;
      lector;
   coend;
end;
```

```
process type VariableCompartida;
   entry read(var v: integer);
   entry write(v: integer);

    var variable ; Integer;

begin
accept write(v : Integer) do variable:=  v;
   repeat
      select
         accept write(v : Integer) do variable:=  v;
      or
         accept read(var v: Integer) do v:= variable;
      or
         terminate;
      end;
    forever;
end;
```


Notas:

En el ejemplo se muestra el código de un proceso que proporciona el control de una variable compartida por cualquier número de otros procesos que la pueden leer o escribir de forma segura.

Ejemplo: Cena de los filósofos chinos (1).

```
program CenaFilosofosChino;
const N=5;
var i: Integer;

process type TipoPalillo:
  entry toma;
  entry deja; begin .. end;

process type TipoFilosofo(nombre:Integer); begin .. end;

process gestorSillas;
 entry seSienta;
 entry seLevanta; begin ... end;

var filosofo:array [1..N] of TipoFilosofo;
      palillo: array [1..N] of TipoPalillo;

begin
    cobegin
        for i:=1 to N do begin palillo[i]; filosofo[i] end;
        gestorSillas;
   coend;
end.
```


Notas:

En la solución que se propone, a fin de que no se produzca un bloqueo cuando los N comensales pretenden comer simultáneamente, se ha arbitrado que esta situación nunca se produzca. Para ello hemos introducido el concepto de silla, obligamos que los filósofos piensen de pie (como es habitual), tengan que sentarse antes de empezar a comer y para evitar el conflicto solo hemos introducido N-1 sillas.

La solución se plantea como un programa concurrente en el que los filósofos (objetos activos)son procesos que requieren los recursos que necesitan (silla, palillo derecho y palillo izquierdo). Los palillos son recursos, y se implementan como procesos que contienen el valor (ocupado o libre) y a los que se accede de forma segura mediante los procedimientos toma y deja. Por último, las sillas es de nuevo un objeto pasivo que contiene el valor (número de sillas libres que quedan), y al que se accede de forma segura mediante los procedimientos Se_sienta y Se_levanta.

Ejemplo: Cena de los filósofos chinos (2)

```
process type TipoPalillo;
  entry toma;
  entry deja;
begin
    repeat
       select
           accept toma do null;
           accept deja do null;
       or
           terminate
       end;
    forever;
end;
```


Notas:

Ejemplo: Cena de los filósofos chinos (3)

```
process gestorSillas;
  entry seSienta;
  entry seLevanta;
var sillasLibres : Integer;
begin
   sillasLibres:= N-1;
   repeat
     select
       when sillasLibres >0 =>
            accept seSienta do null;
            sillasLibres:= sillasLibres -1;
      or
         accept seLevanta do null
            sillasLibres:= sillasLibres +1;
       or
         terminate;
     end;
   forever;
end;
```


Notas:

Ejemplo: Cena de los filósofos chinos (4)

```
process type TipoFilosofo (nombre : Integer);
var i : Integer;
      izquierdo, derecho : Integer;
begin
    izquierdo := nombre;
    derecho := (nombre mod N)+1;
    for i:= 1 to 10 do begin
            sleep(random(5));          -- Está pensando
            gestorSillas.seSienta;
            palillo[Izquierdo].toma;
            palillo[Derecho].toma;
            sleep (Random(5));         -- Está comiendo
            palillo[Derecho].deja;
            palillo[Izquierdo].deja;
            gestorSillas.seLevanta;
     end;
end;
```


Notas:

Procesos auxiliares.

- Se describen componentes auxiliares reutilizables en programación concurrente:
  - **Buffer**: Es un componente que permite el acceso seguro a una variable compartida. Permite que la variable sea escrita y leída en cualquier secuencia de acciones, de forma que el valor que se lee sea siempre el último que se ha escrito. Se exige que la primera acción sea siempre una escritura del buffer.
  - **Cola**: Es una lista con capacidad de recibir y entregar mensajes de un tipo dado (TipoDato). Tienen capacidad de almacenar hasta un número determinado de datos. La política de entrega de los datos almacenados es la FIFO, esto es, en cada solicitud de dato, la cola entrega el que lleva mas tiempo almacenado en la misma.
  - **Stack**: Es una lista similar a la cola salvo que la política de entrega es de tipo LIFO, esto es, en cada solicitud de entrega de dato, el stack entrega el dato que más recientemente ha sido introducido.
  - **Buzón (Mailbox):** Es un proceso que actúa como buffer temporal entre otros dos procesos. Admite dos operaciones put y get, que son atendidas de forma alternativa.
  - **Agente**: Es un proceso que actúa directamente como intermediario de la interacción entre dos procesos.

Notas:

Buffer

```
type TipoDato = .....
process type Buffer;
  entry write ( dato: TipoDato);
  entry read (var dato : TipoDato);
  var valor :  TipoDato;
begin
    accept write(dato : TipoDato) do valor:=  dato;
    repeat
        select
          accept write(dato : TipoDato) do valor :=  dato;
        or
          accept read(var dato: TipoDato) do dato:=  valor;
        or
          terminate;
        end;
    forever;
end;
```


Notas:

Buffer: Básicamente es un componente que permite el acceso seguro a una variable compartida. Permite que la variable sea escrita y leída en cualquier secuencia de acciones, de forma que el valor que se lee sea siempre el último que se ha escrito. Se exige que la primera acción sea siempre una escritura del buffer.

Cola (1)

```
type TipoDato= .....

process type Cola;
  entry put(dato : TipoDato);
  entry get(var dato : TipoDato);

const LONG_COLA = .....;
var datos : array [0..LONG_COLA - 1] of TipoDato;
      tope, base : Integer;
       nDatos : Integer;
begin
      ....
 end;
```


Notas:

Cola: Es una lista con capacidad de recibir y entregar mensajes de un tipo dado (Tipo_dato). Tienen capacidad de almacenar hasta un número determinado de datos. La política de entrega de los datos almacenados es la FIFO, esto es, en cada solicitud de dato, la cola entrega el que lleva mas tiempo almacenado en la misma.

Cola (2)

```
.....
begin
tope:= 0;  base:=0; nDatos:= 0;
repeat
   select
      when (nDatos<LONG_COLA) => accept put(dato: TipoDato) do begin
            datos[tope]:= dato; tope := (tope +1) mod LONG_COLA;
            nDatos:= nDatos +1;  end;
     or
        when (nDatos>0) => accept get(var dato: TipoDato) do begin
            dato:= datos[base]; base:= (base +1) mod LONG_COLA;
            nDatos:=nDatos-1;  end;
     or
        terminate;
    end;
forever
end;
```


Notas:

Stack

```
type TipoDato= .....
process type Stack;
  entry pull(dato : TipoDato);
  entry push(var dato : TipoDato);
  const LONG_STACK = .....;
  var datos : array [0..LONG_STACK - 1] of TipoDato; tope : Integer:=0;
begin
 repeat
   select
     when (tope < LONG_COLA) => accept pull(dato : TipoDato) do
              begin Datos[tope]:= dato; tope := (tope +1);  end;
   or  when (tope > 0) => accept push(var dato : TipoDato) do
              begin  tope:= tope-1; dato:= datos[Tope];  end;
   or terminate;
   end;
 forever
end;
```


Notas:

Stack: Es una lista similar a la cola salvo que la política de entrega es de tipo LIFO, esto es, en cada solicitud de entrega de dato, el stack entrega el dato que más recientemente ha sido introducido.

Buzón (Mailbox)

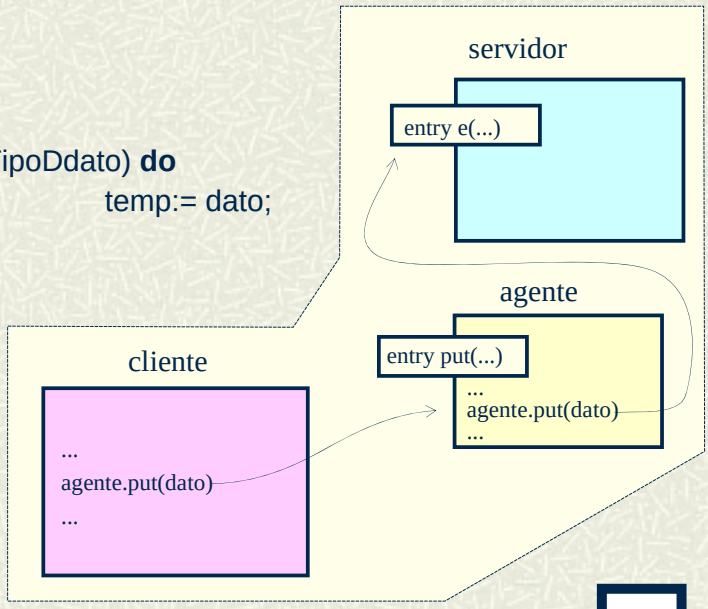
```
type TipoDato = .....
process Buzon;
  entry put (dato : TipoDato);
  entry get (var dato : TipoDato);
var valor : TipoDato;
begin
    repeat
       select
           accept put(dato : TipoDato) do valor :=  dato;
           accept get(var dato : TipoDato) do dato:=  valor;
        or
           terminate;
       end;
    forever
end;
```


Notas:

Buzón (Mailbox): Es un proceso que actúa como buffer temporal entre otros dos procesos. Admite dos operaciones Put y Get, que se exige que sean atendidas de forma alternativas.

Agente

```
Process type Agente;
   entry put (dato :  TipoDato);
var  temp: TipoDato;
begin
    repeat
        select
            accept put(dato : TipoDdato) do
                                          temp:= dato;
            servidor.e(temp);
        or
            terminate
        end;
    forever;
end;
```



Notas:

Agente: es un proceso que actúa directamente como intermediario de la interacción entre dos procesos. Supóngase que un proceso P invoca un procedimiento remoto E de cierto proceso Q. Si P llama directamente y Q no está dispuesto a atenderla, P queda suspendida hasta que sea atendida. Si se desea que la interacción sea asíncrona, y P quede libre con independencia de que la interacción sea o no atendida, P debe utilizar un proceso agente que haga el encuentro con Q por él.