

Lonja Online

[1 hora, 3 puntos]

En una subasta de pescado, el que subasta anuncia el lote de pescado que se va a subastar, junto con un precio de salida que será una cota superior del precio que realmente se va a pagar. Tras el anuncio inicial irá cantando precios cada vez menores hasta que alguno de los compradores puje, siéndole adjudicado el lote por el último precio cantado.¹

La globalización también ha llegado a las subastas de pescado y se está diseñando un sistema para poder participar en ellas a través de Internet. Tras un primer análisis, se han identificado dos tipos de procesos, *Vendedor* y *Comprador*, cuyo comportamiento viene dado por el siguiente pseudocódigo:

```
task body Vendedor is
  Adjudicado : Boolean := False;
  NuevoPrecio : Precio;
begin
  <iniciar Lote>
  Gestor.IniciarSubasta(Lote.pescado, Lote.precio);
  loop
    <calcular NuevoPrecio>
    <esperar un cierto tiempo>
    Gestor.ActualizarPrecio(NuevoPrecio, Adjudicado);
    exit when Adjudicado;
  end loop;
end Vendedor;
task body Comprador is
  Ps : Pescado;
  Pr : Precio;
  NuevoPrecio : Precio;
  Concedido : Boolean := False;
  Hay : Boolean;
begin
  Gestor.UnirseASubasta(Hay, Ps, Pr);
  if Hay and then <interesa Ps y Pr> then
    while not Concedido and Hay loop
      Gestor.OirValor(Pr,NuevoPrecio,Hay); Pr := NuevoPrecio;
      if Hay and then <queremos comprar> then
        Gestor.Pujar (Concedido, PrecioFinal)
      end if;
    end loop;
    if Concedido then
      <comprar lote>
    end if;
  end if;
end Comprador;
```

Sólo habrá una instancia de *Vendedor* en ejecución y varias de *Comprador*. Como se puede ver, una característica fundamental del sistema es el escaso acoplamiento entre vendedor y compradores: aquél ignora cuántos compradores hay atendiendo a la subasta en un momento determinado y éstos ignoran si la subasta está o no abierta. La interacción con el gestor se ha pensado para que las operaciones proporcionen en todo momento esa información — variables *Hay*, etc.

¹Se asume, por simplicidad, que siempre acaba pujando alguien.

Las operaciones que el gestor ha de proporcionar son, pues, las siguientes:

```

IniciarSubasta (Ps : in Pescado;
               Pr : in Precio);
-- Usada por el vendedor para notificar la apertura de una subasta

UnirseASubasta (Hay : out Boolean;
               Ps  : out Pescado;
               Pr  : out Precio);
-- Usada por un comprador para saber si la subasta está activa y,
-- en tal caso, las características de lo subastado

OirValor (PrecioAnterior : in Precio;
         NuevoPrecio     : out Precio;
         Hay              : out Boolean);
-- Usada por un comprador para conocer el nuevo precio
-- ‘Hay’ sirve para saber si la subasta sigue abierta

Pujar (Concedido : out Boolean;
      Pr : out Precio);
-- ‘Concedido’ indica si la puja tuvo éxito y, si es así, ‘Pr’ es el precio
-- a que se ha adjudicado el lote (puede haber variado desde que se pujó).

ActualizarPrecio (NuevoPrecio : in Precio;
                 Adjudicado   : out Boolean);
-- La llama el vendedor, y pasa en ‘NuevoPrecio’ el nuevo valor de la oferta.
-- Si algún comprador ha pujado desde la última actualización,
-- ‘Adjudicado’ se hará cierto.

```

Un requisito que debe cumplir el sistema es la ausencia de bucles de espera activa, tanto en el vendedor como en los compradores. En el vendedor esto se consigue mediante una pequeña espera entre actualizaciones consecutivas del precio, pero en los compradores esto no es posible, ya que introducir retardos entre llamadas consecutivas a *OirValor* podría resultar en la pérdida de precios intermedios. Es, por tanto, responsabilidad del gestor el sincronizar las llamadas a *OirValor* para que los compradores no llamen varias veces sin haberse producido variación en el precio del pescado. En otras palabras, la llamada a *OirValor* no debe retornar hasta que no se haya producido una variación de precio con respecto al anterior que el comprador conocía — de ahí la necesidad del parámetro de entrada *PrecioAnterior*.

Se pide:

- 1 Especificar formalmente un CTADSOL *GestorSubasta* que permita realizar la interacción entre vendedor y compradores. **[3 puntos]**

Solución propuesta

La solución propuesta es la especificación que se encuentra al principio de la siguiente parte del examen. Algunos comentarios sobre ella:

- Utilizar dos campos booleanos para llevar si la subasta está abierta y si se ha adjudicado no es incorrecto, pero se puede correr el riesgo de permitir que dos (o más) clientes diferentes pujen uno inmediatamente tras el otro, y el gestor asigne el lote a ambos. Es necesario prever este caso. Una sola variable salva esa situación.
- En *OirValor* es necesario tener en cuenta que la subasta puede haber acabado. Varios clientes pueden estar bloqueados esperando un cambio de precio, y un cliente rezagado puede pujar por un precio más alto que el actual. Entonces la subasta terminaría sin que se baje el precio y por tanto sin que se desbloqueen aquellos que esperaban un cambio del mismo.

Lonja Online — Implementación [2 h. 30 min., 7 puntos]

Asumamos la siguiente especificación formal del recurso *GestorSubasta*:

C-TADSOL GestorSubasta

USA Pescado, Precio

OPERACIONES

ACCIÓN IniciarSubasta: $Subasta[es] \times Pescado[e] \times Precio[e]$

ACCIÓN UnirseASubasta: $Subasta[e] \times Booleano[s] \times Pescado[s] \times Precio[s]$

ACCIÓN OirValor: $Subasta[e] \times Precio[e] \times Precio[s] \times Booleano[s]$

ACCIÓN Pujar: $Subasta[es] \times Booleano[s] \times Precio[s]$

ACCIÓN ActualizarPrecio: $Subasta[es] \times Precio[e] \times Booleano[s]$

SEMÁNTICA

DOMINIO:

TIPO: $Subasta = (abierta: Booleano \times precio: Precio \times pescado: Pescado)$

INICIAL(s): $\neg s.abierta$

PRE: $\neg subasta.abierta$

CPRE: Cierto

IniciarSubasta (subasta, pescado, precio)

POST: $subasta^{sal}.abierta \wedge subasta^{sal}.pescado = pescado \wedge$
 $subasta^{sal}.precio = precio$

CPRE: Cierto

UnirseASubasta (subasta, hay, pescado, precio)

POST: $hay = subasta.abierta \wedge$
 $hay \rightarrow (pescado = subasta.pescado \wedge precio = subasta.precio)$

CPRE: $subasta.precio < precioanterior \vee \neg subasta.abierta$

OirValor (subasta, precioanterior, nuevoprecio, hay)

POST: $hay = subasta.abierta \wedge$
 $hay \rightarrow nuevoprecio = subasta.precio$

CPRE: Cierto

Pujar (subasta, concedido, precio)

POST: $concedido = subasta^{ent}.abierta \wedge$
 $concedido \rightarrow precio = subasta.precio \wedge$
 $subasta^{sal} = subasta^{ent} \setminus \neg subasta^{sal}.abierta$

CPRE: Cierto

ActualizarPrecio (subasta, precio, adjudicado)

POST: $adjudicado = \neg subasta^{ent}.abierta \wedge$
 $subasta^{sal} = subasta^{ent} \setminus subasta^{sal}.precio = precio$

Se pide:

- 2 Implementación en Ada 95 del sistema (procesos *Vendedor*, *Comprador* y gestor) usando procesos distribuidos y *rendez-vous* y/o paso de mensajes. [4 puntos]
- 3 Implementación mediante objetos protegidos (sólo el código del tipo protegido). [3 puntos]

Observación: No será necesario tener en cuenta posibles problemas de vivacidad provocados exclusivamente por la implementación indeterminista en Ada 95 de las llamadas a *entries* o ramas de una *select*.

Solución propuesta a la pregunta 2

La declaración del interfaz del tipo tarea que implementa el gestor aparece debajo. En ella se ha elegido utilizar *rendez-vous* en todas las operaciones, excepto en *Oir_Valor*. En esta última se ha elegido el uso de canales explícitos para implementar la dependencia entre la precondition de concurrencia y los parámetros de entrada. Esta operación, por tanto, cambia los tipos de sus parámetros con respecto a la especificación, y el proceso que simula un cliente cambia la llamada correspondiente de forma acorde.

```

-- Gestor de la subasta implementado como una tarea. Algunas
-- operaciones han cambiado ligeramente su cabecera para incluir
-- canales usados para realizar un desbloqueo explícito.
task type Gestor_Subasta is
  -- Indica qué pescado está siendo subastado y el precio inicial
  -- que se pide por él.
  entry Iniciar_Subasta (Pescado_Subastado : in T_Pescado;
                        Precio_Inicial    : in T_Precio);
  -- Cambia el precio del pescado a la baja y devuelve una
  -- indicación de si algún comprador ha pujado por él.
  entry Actualizar_Precio (Nuevo_Precio : in T_Precio;
                          Adjudicado    : out Boolean);
  -- Un comprador quiere unirse a una subasta. Quizá no haya
  -- subasta abierta en este momento, o quizá la mercancía no nos
  -- interese. En otro caso, conocemos el valor inicial del lote.
  entry Unirse_A_Subasta (Hay                : out Boolean;
                          Pescado_Subastado : out T_Pescado;
                          Precio_Inicial    : out T_Precio);
  -- Esperamos a que se cambie el valor del pescado con respecto
  -- al que ya conocíamos, y adquirimos conocimiento del nuevo
  -- valor del pescado; otro cliente puede haber pujado, en cuyo
  -- caso se nos comunica que la subasta no sigue.
  entry Oir_Valor (Precio_Anterior : in T_Precio;
                  Canal_Comprador : in out Channel_P);
  -- Cuando se llega a un valor que nos conviene, pujamos por él.
  -- Si hemos sido los primeros en pujar, nos quedamos con el
  -- lote; esto se nos indica con una variable booleana.
  entry Pujar (Concedido : out Boolean;
              Precio_Final : out T_Precio);
end Gestor_Subasta;

```

El cuerpo de la tarea traduce casi directamente la especificación dada. La entrada correspondiente a Oir_Valor almacena todas las peticiones que llegan en una cola. Dichas peticiones se atienden en un bucle al final de la construcción **select**.

```

task body Gestor_Subasta is
  -- Estado del recurso; refleja exactamente lo que había en la
  -- especificación
  Precio : T_Precio;
  Pescado : T_Pescado;
  Abierta : Boolean := False;

  -- Cola de peticiones de Oir_Valor pendientes. Las peticiones
  -- se almacenan según llegan, y se van despertando conforme su
  -- precondition se va haciendo cierta. Utilizamos canales
  -- explícitos únicamente en Oir_Valor, porque es la única en la
  -- que los necesitamos.
  Oir_Pendientes : Cola := Crear_Vacia;

  -- Variables auxiliares
  Peticion : Tipo_Pet_Oir;
  Oir_Pendientes_Aux : Cola;

begin
  loop
    select
      when True =>
        accept Iniciar_Subasta (Pescado_Subastado : in T_Pescado;
                                Precio_Inicial    : in T_Precio)
        do
          Pescado := Pescado_Subastado;
          Precio := Precio_Inicial;
          Abierta := True;
        end Iniciar_Subasta;
    or
      when True =>

```

```

    accept Unirse_A_Subasta (Hay                : out Boolean;
                           Pescado_Subastado : out T_Pescado;
                           Precio_Inicial    : out T_Precio)
    do
        Hay := Abierta;
        if Hay then
            Pescado_Subastado := Pescado;
            Precio_Inicial := Precio;
        end if;
    end Unirse_A_Subasta;
or
    -- La precondition de Oir_Valor depende de los parámetros
    -- de entrada. Aceptamos siempre las llamadas a Oir_Valor
    -- y las encolamos en "Oir_Pendientes". El cliente espera
    -- en una recepción de un canal para saber cuando se ha
    -- servido su petición. En la cola se almacenan tanto el
    -- precio que conoce el (posible) comprador como el canal
    -- por el que espera recibir una respuesta.
    when True =>
        accept Oir_Valor (Precio_Anterior : in    T_Precio;
                        Canal_Comprador : in out Channel_P)
        do
            Insertar(Oir_Pendientes, (Precio_Anterior, Canal_Comprador));
        end Oir_Valor;
or
    when True =>
        accept Pujar (Concedido      : out Boolean;
                     Precio_Final : out T_Precio)
        do
            Concedido := Abierta;
            if Concedido then
                Precio_Final := Precio;
                Abierta := False;
            end if;
        end Pujar;
or
    when True =>
        accept Actualizar_Precio (Nuevo_Precio : in    T_Precio;
                                Adjudicado      : out Boolean)
        do
            Adjudicado := not Abierta;
            if not Adjudicado then
                Precio := Nuevo_Precio;
            end if;
        end Actualizar_Precio;
end select;

-- Implementación de desbloques explícitos. Un comprador se
-- desbloquea cuando el valor de la subasta es menor que el
-- que ha declarado oir, o cuando la subasta ha acabado.
-- Utilizamos una cola auxiliar para almacenar las peticiones
-- que no han sido atendidas, y después las pasamos de nuevo
-- a la cola principal.

Oir_Pendientes_Aux := Crear_Vacia;
-- Necesitamos recorrer toda la cola (no está ordenada por precios).
while not Es_Vacia(Oir_Pendientes) loop
    Primero(Oir_Pendientes, Peticion);
    Borrar(Oir_Pendientes);
    -- Comprobamos la precondition de Oir_Valor
    if Peticion.Precio >= Precio or not Abierta then
        -- Enviamos comprador un mensaje con el precio y la
        -- indicación de si la subasta sigue o no abierta.
        Peticion.Canal_Comprador.Send((Precio, Abierta));
    else

```

```

        Insertar(Oir_Pendientes_Aux, Peticion );
    end if;
end loop;
Destruir(Oir_Pendientes);
Oir_Pendientes := Oir_Pendientes_Aux;
end loop;
end Gestor_Subasta;

```

El código de la tarea comprador cambia ligeramente para aguardar la respuesta a la llamada *Oir_Valor*, que señala cuándo el precio requerido por el lote ha cambiado con respecto al anterior.

```

task body Tipo_Comprador is
    Mi_Precio_Minimo: T_Precio;
    Pescado          : T_Pescado;
    Precio,
    Precio_Final     : T_Precio;
    Concedido        : Boolean := False;
    Hay_Subasta      : Boolean;
    Canal_Comprador  : Channel_P :=
        new Item_Channel.Channel;
    Nuevo_Estado     : Tipo_Dato_Canal;
begin
    Gestor.Unirse_A_Subasta(Hay_Subasta, Pescado, Precio);
    if Hay_Subasta and <<quiero el Pescado>> then
        while Hay_Subasta and not Concedido loop
            Gestor.Oir_Valor(Precio, Canal_Comprador); -- Envía canal
            Canal_Comprador.Receive(Nuevo_Estado);    -- Espera por respuesta
            Precio := Nuevo_Estado.Precio;             -- Examina respuesta
            Hay_Subasta := Nuevo_Estado.Hay;
            if Hay_Subasta and then <<me conviene Precio>> then
                Gestor.Pujar(Concedido, Precio_Final);
            end if;
        end loop;
    end if;
end Tipo_Comprador;

```

Algunos comentarios sobre **vuestros ejercicios**:

- Hay un número sorprendentemente alto de ejercicios que incluyen el parámetro *PrecioAnterior* en la guarda de *Oir_Valor*.
- En algunos casos se detecta el problema con dicho parámetro pero se ignora cómo resolverlo. Ejemplos de esto son: poner la guarda a *True* y no hacer nada más (lo cual provoca la espera activa que se trataba de evitar), ponerla a *not Abierta* (lo cual provoca un interbloqueo inmediato), etc.
- La inmensa mayoría de las soluciones que aplican un esquema más o menos similar al de la solución que proponemos sufren, sin embargo, de algún problema en el código de desbloques. Algunos de los más habituales son:
 - Desencolar sin comprobar la CPRE. Esto provoca espera activa, lo cual se trataba de evitar.
 - Desencolar todos y contestar sólo a los que cumplen la CPRE. Esto provoca el bloqueo de parte de los procesos *Comprador*.
 - Desencolar hasta que se encuentra un elemento de la cola que no cumple la CPRE. Provoca que algunas peticiones aplazadas que se podrían atender tengan que esperar innecesariamente a la siguiente actualización. No es un fallo crítico en esta aplicación, pero podría serlo en otro contexto.
- Finalmente, hay una serie de fallos que, sin afectar al esquema fundamental de la solución, hemos de comentar por haber aparecido más veces de lo deseado, como:
 - Envíos o recepciones dentro del rendez-vous (provoca interbloqueo).
 - Mantener los parámetros de salida de *Oir_Valor* tras haberla reformado para usar paso de mensajes explícito.

Solución propuesta a la pregunta 3

Notas relativas a la solución:

- Es completamente incorrecto, al igual que en la pregunta de paso mensajes, utilizar directamente variables que aparecen en la cabecera de una *entry* o de un *accept* en la evaluación de una guarda. Hacer un *requeue* tampoco soluciona esto de por sí.
- Puede implementarse una solución similar a la utilizada para paso de mensajes asignando un identificador diferente a cada proceso, y almacenándolo en una estructura de datos tras cada llamada a *Oir_Valor* junto con el precio oído. La llamada a ésta operación se reencolaría internamente en una familia de entradas. Los pares identificador/precio se revisarían tras cada actualización del precio y tras cada puja y liberarían los procesos que estuviesen esperando por ese cambio. Esta técnica (que es la más general, al permitir controlar el rearranque al nivel de tareas individuales), necesita una implementación relativamente laboriosa, más propensa a errores y necesita cambiar el código de los procesos.
- Un rearranque *uno a uno*, en que se permite la existencia de a lo sumo un proceso bloqueado en *Oir_Val* dentro del objeto protegido y espacio para guardar los parámetros de esa llamada, es válido. La justificación de la validez es que *Oir_Valor* no realiza ninguna puja; existe un tiempo, no determinado *a priori* entre el momento en que se escucha un precio interesante y el momento en que se puja por él. Por ello, aún en el caso de clientes que aceptan el mismo precio máximo, existe la posibilidad de una inversión del orden de llegada a la puja con respecto al orden de llegada a *Oir_Valor*. Lo mismo puede aplicarse a clientes con *ideas* diferentes sobre el valor máximo deseable. Muchos han intentado aplicar esta solución de forma incorrecta, sin bloquear la *entry* más externa. Esto hace que llamadas posteriores pueden sobrescribir el estado que guarda argumentos de llamada de llamadas ya almacenados.
- Realizar la operación *Oir_Valor* con una familia de entries tiene varios problemas:
 - No sabemos realmente cómo está implementado *Tipo_Precio*. Podría ser perfectamente un registro con la parte entera y decimal de un precio.
 - Incluso si el tipo es un tipo simple, podría no ser escalar (por ejemplo, un *Float*).
 - Aún en el caso de un escalar, su rango podría ser tan amplio que no llevase a una eficiencia excesivamente baja (o incluso a errores en tiempo de ejecución).

Los procesos Cliente y Vendedor no cambian, y las cabeceras de los procedimientos son las mismas que en la especificación. Se incluyen a continuación la declaración de variables privadas del tipo protegido y el código del mismo.

```
private
-- Variables de estado básicas; deducidas directamente de la
-- especificación del recurso.
Abierta: Boolean := False;
Precio: T_Precio;
Que_Pescado: T_Pescado;
-- Variables que almacenan el estado de la última llamada a
-- Oir_Valor recibida.
Escucha_Bloqueada: Boolean := False;
Precio_Bloqueado: T_Precio;
-- Procedimiento interno de Oir_Valor en casos bloqueados.
entry Oir_Valor_Blq ( Precio_Anterior : in      T_Precio;
                    Nuevo_Precio    :      out T_Precio;
                    Sigue_Subasta   :      out Boolean);

end Subasta;

protected body Subasta is
-- Trivial; da valor inicial al pescado, precio del lote, etc.
-- De cara al resto de las operaciones y los procesos, abre la
-- subasta.
entry Iniciar_Subasta (Lote           : in T_Pescado;
                      Precio_Entrada : in T_Precio)

when True is
begin
```

```

    Que_Pescado:= Lote;
    Abierta:= True;
    Precio:= Precio_Entrada;
end Iniciar_Subasta;

-- Trivial; determina si alguien ha pujado (y entonces se
-- entiende que se le ha adjudicado la subasta), y actualiza el
-- precio para utilizar uno nuevo.
entry Actualizar_Precio (Otro_Precio : in      T_Precio;
                        Adjudicado  : out Boolean)

when True is
begin
    Adjudicado:= not Abierta;
    if Abierta then
        Precio:= Otro_Precio;
    end if;
end Actualizar_Precio;

-- Trivial. Si hay subasta, devuelve el tipo del lote y el
-- precio inicial.
entry Unirse_A_Subasta (Hay_Subasta : out Boolean;
                        Pescado      : out T_Pescado;
                        Precio_Ahora : out T_Precio)

when True is
begin
    Hay_Subasta:= Abierta;
    if Hay_Subasta then
        Pescado:= Que_Pescado;
        Precio_Ahora:= Precio;
    end if;
end Unirse_A_Subasta;

-- No trivial. Implementa una escucha por orden de llegada. El
-- primero en escuchar es el primero en saber el nuevo precio.
-- Un comprador podría perder un lote interesante si se retrasa
-- en escuchar. Pero también puede perderlo si, escuchando el
-- primero, tarda en decidirse a pujar. La vida es así.
entry Oir_Valor (Precio_Anterior : in      T_Precio;
                 Nuevo_Precio    : out T_Precio;
                 Sigue_Subasta   : out Boolean)

when not Escucha_Bloqueada is
begin
    Escucha_Bloqueada := True;
    Precio_Bloqueado := Precio_Anterior;
    requeue Oir_Valor_Blq;
end Oir_Valor;

-- Implementa la verdadera espera por el cambio de precio. *NO
-- DETERMINA, EN MODO ALGUNO, UNA PUJA*. Esta viene después.
entry Oir_Valor_Blq (Precio_Anterior : in      T_Precio;
                    Nuevo_Precio     : out T_Precio;
                    Sigue_Subasta     : out Boolean)

when Precio < Precio_Bloqueado or not Abierta is
begin
    Escucha_Bloqueada := False;
    Sigue_Subasta:= Abierta;
    if Sigue_Subasta then
        Nuevo_Precio:= Precio;
    end if;
end Oir_Valor_Blq;

-- Notifica al recurso que desea realizar una puja. Si somos
-- los primeros se nos concede. Si no, hemos perdido el lote.
entry Pujar (Concedido : out Boolean;
             Precio_Final : out T_Precio)

```

```

when True is
begin
  Concedido:= Abierta;
  Abierta:= False;
  if Concedido then
    Precio_Final:= Precio;
  end if;
end Pujar;
end Subasta;

```

Por completitud incluimos a continuación la parte relevante de una implementación con familias de *entries*, suponiendo que es posible utilizarla con el tipo de los precios del pescado:

```

-- Implementa la dependencia de la precondition con respecto de
-- los datos de entrada con una familia de "entries". Esto
-- asume que el precio es un escalar discreto y, por motivos de
-- eficiencia, que el posible rango de precios no es demasiado
-- elevado.

```

```

entry Oir_Valor ( Precio_Anterior : in      T_Precio;
                  Nuevo_Precio   :      out T_Precio;
                  Sigue_Subasta  :      out Boolean)

```

```

when True is
begin
  requeue Oir_Valor_Int(Precio_Anterior);
end Oir_Valor;

```

```

-- Familia de entries, propiamente dicha.

```

```

entry Oir_Valor_Int(for P in T_Precio)
  ( Precio_Anterior : in      T_Precio;
    Nuevo_Precio   :      out T_Precio;
    Sigue_Subasta  :      out Boolean)

```

```

when Precio < P or not Abierta is
begin
  Sigue_Subasta:= Abierta;
  if Sigue_Subasta then
    Nuevo_Precio:= Precio;
  end if;
end Oir_Valor_Int;

```