

III

Programación Concurrente

Clases 7 a 9: Memoria distribuida

Ramiro Martínez D'Elía

2021

Índice general

1. Programa distribuido	3
1.1. Definición	3
1.2. Canales	3
2. Patrones de resolución	5
2.1. Interacting peers: intercambiando valores	5
2.1.1. Solución centralizada	5
2.1.2. Solución simétrica	6
2.1.3. Solución circular	6
2.1.4. Conclusiones	7
3. PMS - Pasaje de Mensajes Asíncrono	8
3.1. CSP y Comunicación guardada	8
4. RPC, Rendezvous y Notaciones primitivas	10
4.1. Introducción	10
4.2. Remote Procedure Calls (RPC)	10
4.2.1. Sincronización	11
4.2.2. Time Server	11
4.3. Rendezvous	12
4.3.1. Sentencias Input	12
4.3.2. Buffer limitado	13
4.3.3. ADA	13
4.3.4. Buffer limitado con ADA	14
4.4. Notaciones primitivas	14
5. Paradigmas de interacción entre procesos	16
5.1. Introducción	16
5.2. Manager and workers	16
5.3. Heartbeat	16
5.4. Pipeline	17
5.5. Probes and echoes	17
5.6. Broadcasts	17
5.7. Token passing	17
5.7.1. Exclusión mutua con token ring	17

5.8. Servidores replicados	18
--------------------------------------	----

Capítulo 1

Programa distribuido

1.1. Definición

Los mecanismos de sincronización vistos hasta ahora, están pensados para programas concurrentes que ejecutan en hardware con procesadores que comparten memoria (arquitecturas de memoria compartida).

Sin embargo, hay arquitecturas donde los CPU solo comparten una red de comunicación. Bajo este contexto, los procesos ya no comparten variables sino que comparten *canales*.

Los *canales* son abstracciones de una red de comunicación física y al ser el único recurso compartido entre procesos, toda variable es local a un solo proceso y solo accedida por ese proceso. Esto descarta la necesidad de exclusión mutua.

La ausencia de variables compartidas, permite que los procesos puedan ejecutarse en procesadores distribuidos. Por esta razón, los programas concurrentes que utilizan pasaje de mensajes son llamados *programas distribuidos*.

1.2. Canales

Los canales son *estructuras FIFO* (First In First Out) que contienen mensajes pendientes. Estas estructuras soportan 2 (dos) **operaciones atómicas *send* y *receive***. Para iniciar una comunicación, un proceso envía un mensaje por el canal; y otro lo adquiere recibiendo desde el mismo canal.

La operación *send* puede ser **asincrónica** (no demora al proceso que la invoca) y **sincrónica** (demora al proceso que la invoca, hasta que su mensaje fuese recibido). La operación *receive* **siempre demora** al proceso que la invoca.

La estructura FIFO y la atomicidad de las primitivas send y receive aseguran que; los mensajes eventualmente serán recibidos, los mensajes no serán corrompidos y serán entregados en el orden en que fueron encolados.

```
1  channel ch1(string),           8  End.
2      ch2(string);              9
3                                10 Program Dos
4  Program Uno                   11      string saludo;
5      string respuesta;         12      receive ch2(saludo);
6      send ch2("Hola");         13      send ch1("Aca esta tu respuesta ...");
7      receive ch1(respuesta);    14  End.
```

Según la forma en que se utilicen, los canales, pueden clasificarse en los siguientes tipos:

- **Mailbox:** Cualquier proceso puede enviar y recibir datos, por alguno de los canales declarados. Relación $n:n$.
- **Input Port:** El canal tiene un único receptor y (n) emisores. Relación $1:n$.

- **Link:** El canal tiene un único receptor y un único emisor. Relación *1:1*.

Capítulo 2

Patrones de resolución

Los tipos de procesos que podemos encontrar en los programas distribuidos, son los mismos que en un programa concurrente con memoria compartida (peers, filters, clientes y servidores). Lo mismo ocurre, con los patrones de resolución.

A continuación, abordaremos un patrón en particular: *interacting peers*, donde se describirá la forma de interconexión entre procesos. Todos los ejemplos, implementan *pasaje de mensajes asincrónicos (PMA)*.

💎 Soluciones para ordenar números: Merged Network, Odd/Even Exchange Sort, Pipeline. ¿Cantidad de procesos? ¿Cantidad de mensajes? (Ver apunte de gemas).

2.1. Interacting peers: intercambiando valores

Supongamos el problema donde existen n procesos, cada uno con un valor local v . El objetivo es, lograr que todos los procesos conozcan el máximo y mínimo valor de v de entre todos.

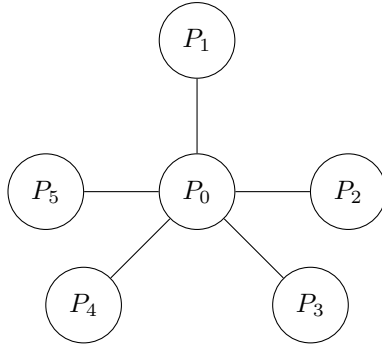
En este caso, los procesos son idénticos. Por lo cual, podemos diseñar una solución basada en el esquema de pares que interactúan.

A continuación se detallan soluciones, empleando 3 (tres) formas distintas de conexión entre procesos: *centralizada*, *simétrica* y *circular*.

2.1.1. Solución centralizada

En esta solución, utilizaremos un proceso central que; recibirá los valores de los demás. Computará los máximos y mínimos e informará al resto.

```
1  channel values(int),
2      results[n](int, int);
3
4  Process Peer[i = 1 to n-1]
5      int v, min, max;
6      send values(v);
7      receive results[i](min, max);
8  End.
9
10 Process CentralPeer[0]
11     int v, min, max, temp;
12
13     for(i=1 to n-1)
14         receive values(temp);
15         # Compara y actualiza ...
16     end;
17
18     for(i=1 to n-1)
19         send results[i](min, max);
20 End.
```



2.1.2. Solución simétrica

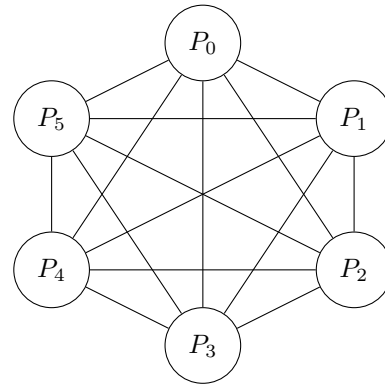
En esta solución, todos los procesos realizan el mismo algoritmo. Cada par de procesos, se encuentran conectados mediante un canal.

Cada proceso, envía los datos al resto, y así cada uno puede conocer todos los valores y efectuar el cálculo de máximos y mínimos en paralelo.

```

1  channel values[n](int, int);
2
3  Process Peer[i = 0 to n]
4      int v, min, max, temp;
5
6      for(j=0 to n st j != i)
7          send values[j](v);
8
9      for(j=0 to n st j != i)
10         receive receive[i](temp)
11         # Compara y actualiza.
12 End.

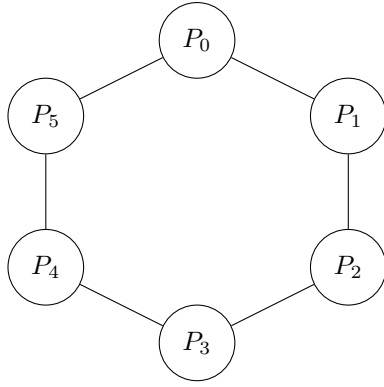
```



2.1.3. Solución circular

En esta solución, cada proceso recibe mensajes de su antecesor y envía mensajes a su sucesor. Con este enfoque, harán falta 2 (dos) iteraciones al anillo. La primera, para obtener el máximo y el mínimo. Y la segunda, para notificar a todos los procesos.

<pre> 1 channel values[n](int, int); 2 3 Process Initial[0] 4 int v, min = v, max = v; 5 6 # Inicia primera iteracion ... 7 send values[1](min, max); 8 receive values[0](min, max); 9 10 # Inicia segunda iteracion ... 11 send values[1](min, max); 12 End. 13 14 Process Peer[i = 1 to n-1] 15 int v, min, max; 16 17 # Primera iteracion ... 18 receive values[i](min, max); </pre>	<pre> 19 # Compara y actualiza valores ... 20 send values[(i+1) mod n](min, max); 21 22 # Segunda iteracion 23 receive values[i](min, max); 24 send values[(i+1) mod n](min, max); 25 End. </pre>
---	---



2.1.4. Conclusiones

Desde el punto de vista de la cantida de mensajes enviados y la eficiencia, podemos obtener las siguientes conclusiones para las soluciones descriptas.

Solución	# Mensajes	Orden	Conclusión
Centralizada	$2(n - 1)$	Lineal $O(n)$	No hay paralelismo, toda la carga del cómputo recae en CentralPeer. Menor overhead en el pasaje de mensajes. Menor eficiencia a mayor n .
Simétrica	$n(n - 1)$	Cuadrático $O(n^2)$	Cómputo paralelo pero, mayor overhead en el pasaje de mensjaes.
Circular	$2(n - 1)$	Lineal $O(n)$	Cómputo paralelo. Mayor eficiencia para n muy grandes.

Nota: Con la primitiva **broadcast**, la cantidad de mensajes se reduce a n para todos los casos. Lo que hace, dicha primitiva es envíar un mensaje (de forma concurrente) a todos los procesos que escuchen el canal utilizado.

Capítulo 3

PMS - Pasaje de Mensajes Asincrónico

3.1. CSP y Comunicación guardada

Con CSP (*Communicating Sequential Processes*) se introduce el concepto de *PMS y comunicación guardada*. Los procesos, se enviarán mensajes, entre ellos, mediante links directos. La forma general de comunicación, entre procesos, en CSP es de la siguiente forma:

```
1 DestinationProc!portName(...);
2 SourceProc?portName(...);
```

Siendo los operadores *!* y *?* para el envío y recepción de mensajes respectivamente. Dos procesos, entablabran comunicación cuando ejecuten sentencias de comunicación que hagan match entre sí.

El siguiente ejemplo, muestra un proceso (Copy) que copia caracteres entre 2 (dos) procesos East y West.

```
1 Process Copy
2
3 Char c;
4
5 while (true)
6     west?(c)
7     east!(c);
8 end.
9
10 End.
```

La principal limitación de *!* y *?* es que, son primitivas bloqueantes. Usualmente, se desea, que un proceso pueda comunicarse con los demás; sin importar el orden en que los otros quieran comunicarse con el.

Por ejemplo, si modificamos el ejemplo de Copy de forma tal que; ahora se deben copiar de a 2 caracteres. El proceso Copy, entonces, debe esperar a recibir los 2 caracteres desde West, antes de retransmitirlos. Esto provoca que, East quede bloqueado.

```
1 Process Copy
2
3 Char c1, c2;
4
5 while (true)
6     west?(c1); west?(c2);
7     east!(c1); east!(c1);
8 end.
9
10 End.
```

La comunicación guardada, ofrece un tipo de *comunicación no determinística* que resuelve la limitación planteada. *Combinando sentencias condicionales y de comunicación* de la siguiente forma $B; C \rightarrow S$, donde:

- B es una sentencia condicional, que de no existir se asume *verdadera*.
- C es una sentencia de comunicación.
- D es un conjunto de sentencias a ejecutar.
- Juntas B y C forman una *guarda* que "protegen" la ejecución de S

Con respecto al funcionamiento de las *guardas*, se debe tener en cuenta lo siguiente:

- La guarda tiene *éxito* si, B es verdadera y C no causa demora (tiene mensajes \rightarrow ejecución inmediata).
- La guarda *falla* si, B es falsa.
- La guarda se *bloquea* si, B es verdadera y C causa demora.

Podemos aplicar comunicación guardada sobre 2 (dos) tipos de estructuras del control: el *if* y el *do*. Cada una, con la siguiente semántica de ejecución:

- Si *todas las guardas fallan* (en simultáneo); la estructura de control finaliza sin efectos.
- Si al menos una guarda tiene éxito; se elige una de manera no determinística, se ejecuta su B y luego su C . En el caso del *if*, la estructura finaliza.
- Si todas las *guardas están bloqueadas*; se espera hasta que alguna tenga éxito.

La versión del proceso Copy, para 2 caracteres, se puede mejorar utilizando comunicación guardada del siguiente modo:

```

1  Process Copy
2
3  char c1, c2;
4  west?(c1);
5
6  do true; west?(c2) -> east!(c1); c1=c2;
7  [] true; east!(c1) -> west?(c1);
8  end;
9
10 End.
```

Aplicando comunicación guardada, el proceso *Copy*, una vez recibido el primer caracter podrá (de forma no determinística) esperar al segundo caracter o enviar el primero a *East*. De esta forma, logramos reducir el bloqueo en la comunicación entre *Copy* y *East*.

Capítulo 4

RPC, Rendezvous y Notaciones primitivas

4.1. Introducción

La técnica de pasaje de mensajes, resulta óptima para resolver problemas del tipo productores/consumidores y pares que interactúan. Ya que, estos plantean un tipo de comunicación unidireccional entre procesos.

No obstante, el uso de esta técnica para problemas del tipo cliente/servidor; no resulta del todo óptima. Ese tipo de problemas supone una comunicación bidireccional. Es decir, un cliente realiza una petición y, posteriormente, un servidor le otorga una respuesta. Esto nos obliga a definir un gran número de canales:

- Al menos 1 (un) canal por el cual, el servidor reciba peticiones.
- Un canal de respuesta para cada cliente.

Los mecanismos de *RPC* y *Rendezvous*, resuelven de forma más adecuada los problemas del tipo cliente/servidor. Combinando PMS con aspectos de monitores.

Como en *monitores*, un módulo (o proceso) expone operaciones públicas. Dichas operaciones, son invocadas por otro módulo (o proceso) utilizando la sentencia *call*.

Como en *PMS*, la ejecución de una sentencia *call* demora al llamador hasta obtener una respuesta.

Las *operaciones*, resultan *canales de comunicación bidireccional*. Desde el llamador hacia el proceso que sirve la llamada, y luego, de vuelta al llamador.

4.2. Remote Procedure Calls (RPC)

Los programas se descomponen en módulos (con procesos y procedimientos). Los procesos, de un módulo, son llamados *background* para diferenciarlos de los procesos exportados (*servers*). El siguiente ejemplo, muestra de forma trivial la especificación de un módulo.

```
1  Module module_name
2      op opname(params)[returns type];          # (1) Interfaz publica
3  Body
4      int x;                                    # (2) Definicion de variables locales
5      x = 0;                                    # e inicializacion
6
7      proc opname(params)[returns type]         # (3) Definicion de procedimientos exportados
8      begin
9          # Cuerpo del procedimiento.
10     end;
11
12     proc local_proc(params)[returns type]     # (4) Definicion de procedimientos locales
```

```

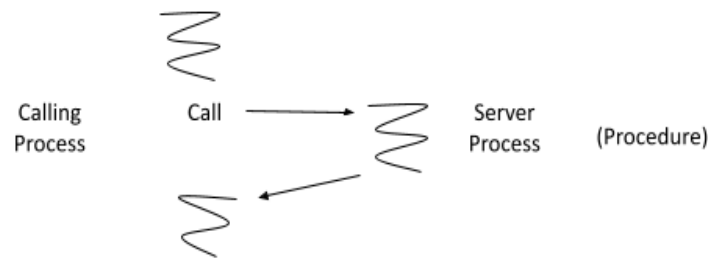
13     begin
14         # Cuerpo del procedimiento.
15     end;
16
17     Process Main                                # (5) Definicion de procesos locales
18         # Cuerpo del proceso background
19     end;
20 End module_name;

```

Los procesos background tienen acceso a las variables y pueden invocar procedimientos del mismo módulo. A su vez, como sucede en monitores, pueden invocar procedimientos definidos en otros módulos. Siempre y cuando, estos últimos, sean exportados por dicho módulo.

La ejecución de una llama intermódulo, difiere con respecto a una local.

- Un nuevo proceso atiende la llamada y los argumentos son pasados como mensajes entre el llamador y el proceso servidor.
- El proceso llamador que demorado mientras se ejecuta el cuerpo del procedimiento invocado.
- Cuando el proceso servidor finaliza su ejecución del procedimiento invocado, retorna los resultados y finaliza.
- El proceso llamador, continua su ejecución.



4.2.1. Sincronización

Con RPC, la *sincronización entre proceso llamador y servidor está implícita*. No obstante, necesitamos mecanismos para que procesos server y background sincronicen dentro de un módulo. Existen 2 (dos) enfoques para proveer sincronización dentro de un módulo.

A lo sumo, un proceso activo

Apropiado para entornos monoprocesador ya que, reduce el context switch.

- Mutex → Implícito.
- Condición → Explícito (semáforos o monitores).

Ejecución concurrente

Apropiado, de forma natural, para entornos multiprocesador.

- Mutex → Explícito (semáforos o monitores).
- Condición → Explícito (semáforos o monitores).

4.2.2. Time Server

El siguiente módulo es una adaptación del ejemplo del libro.

```

1 module TimeServer
2
3 op get_time(): int;
4 op delay(int interval);
5
6 body
7
8 int time = 0;                                # Tiempo inicial del TimeServer (en ms).

```

```

 9 Queue delayed(waketime, id);      # Cola de procesos dormidos, por la operacion delay.
10 Sem m = 1;                        # Semaforo para garantizar mutex, en el acceso a delayed.
11 Sem d = ([n] 0);                  # Arreglo de semaforos para demorar procesos.
12
13 proc get_time(): int                # Retorna el tiempo actual del servidor en ms.
14     return time;
15 end;
16
17 proc delay(int interval)            # Demora un proceso tantos ms, como interval lo indique.
18     P(m);
19     waketime = time + interval;
20     delayed.push(waketime, id);
21     V(m);
22     P(d[id]);
23 end;
24
25 process Clock                       # Proceso background. Mantiene actualizada la variable time
26     while (true)                    # y despierta procesos demorados por la operacion delay.
27         time++;
28         P(m);
29         while (time >= menor waketime en delayed)
30             delayed.pop(waketime, id);
31             V(d[id]);
32         end;
33         V(m);
34     end;
35 end;
36 End.

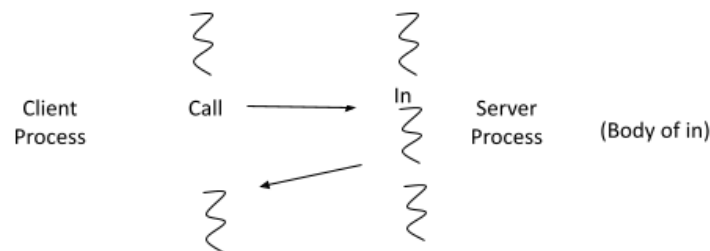
```

4.3. Rendezvous

Al igual que RPC, el programa se descompone en módulos, se brindan los mecanismos para la *comunicación intermódulo* y se demoran a los procesos llamadores.

La cabecera de un módulo, debe especificar las operaciones exportadas. Mientras que el cuerpo, del módulo, contendrá *un único proceso*; que atenderá las llamadas a las operaciones en lugar de crear un nuevo proceso servidor.

Dicho proceso, esperara por una call y actuar. Las llamadas, a operaciones, serán atendidas de a una por vez aunque los procesos se ejecuten concurrentemente. Esto *provee sincronización dentro del módulo*.



4.3.1. Sentencias Input

Suponiendo que, un módulo exporta la siguiente operación: `op opname(formals)`. El proceso servidor realiza rendezvous con un llamador de `opname`, ejecutando una sentencia *input*. Dicha sentencia, respeta la siguiente forma:

```

1 in opname(formals) -> S; ni

```

Las partes entre `in` y `ni` son conocidas como *operaciones guardadas*. La guarda nombra las operaciones y provee sus parámetros. En cuanto a `S`, es un conjunto de sentencias a ejecutar en la invocación de la

operación. Los parámetros, de la operación, tienen alcance de a toda la guarda; por consiguiente *S* podrá leer y escribir sus valores.

Una sentencia *input*, demora al proceso servidor hasta que haya al menos una llamada pendiente a *opname*. Entonces, elige la llamada pendiente más antigua, copia los argumentos en los parámetros formales, ejecuta *S* y finalmente, retorna los resultados al proceso llamador.

Es posible ofrecer *comunicación guardada*, en una sentencia *input* de la siguiente forma:

```

1  In opname1 (formals) and B1 by E1 -> S1
2  [] opname2 (formals) and B2 by E2 -> S2
3  Ni

```

Donde *E* supone una *condición booleana de sincronización*. Mientras que, *E* una *expresión de scheduling*. Ambas expresiones son opcionales.

4.3.2. Buffer limitado

```

1  Module Buffer
2
3  op depositar(int);
4  op retirar(int);
5
6  Body
7  Process Buffer
8      Queue buffer;
9      int cant = 0;
10     int cantMax = n;
11
12     while(true)
13         in depositar(x) and (cant < cantMax) ->
14             buffer.push(x);
15             cant++;
16         [] retirar(x) and (cant > 0) ->
17             buffer.pop(x);
18             cant--;
19     ni
20 end;
21 end;
22 End.

```

4.3.3. ADA

Un programa, escrito en ADA, está compuesto por subprogramas, paquetes y tareas. Donde *rendezvous* es el principal mecanismo de comunicación y sincronización.

Las tareas (*tasks*), son procesos independientes que cuentan con una especificación y un cuerpo (*body*). En la especificación, se definen las operaciones visibles por otros procesos (*entry*). El cuerpo, contiene variables locales (junto con su inicialización) y sentencias de atención (*entry calls*).

```

1  Procedure nombreProceso
2
3  Task nombreTarea Is
4      # Declaracion de entries
5  end;
6
7  Task Body nombreTarea Is
8      # Declaracion de locales e inicializacion
9  Begin
10     # Sentencias ...
11 end;
12

```

```
13 End nombreProceso;
```

Un programa en ADA, atiende solicitudes a procedimientos mediante la sentencia `accept`. La cual equivale a `input` pero, su ejecución es determinística y no posee guardas. Dicha sentencia, posee la siguiente forma:

```
1 Accept E (formals) do
2     # Sentencias
3 end;
```

De ser necesario, comunicación guardada con un comportamiento no determinístico. ADA ofrece la sentencia `select`. No obstante, `select` resulta menos poderosa que la notación formal establecida en rendezvous; por 2 (dos) motivos.

1. No soporta expresiones de scheduling.
2. La expresión de sincronización, no puede referenciar a los parámetros formales de la operación.

4.3.4. Buffer limitado con ADA

```
1 Procedure nombreProceso
2
3 Task Mailbox Is
4     Entry Depositar (msg: IN string)
5     Entry Retirar (msg: OUT string)
6 End;
7
8 Task Body Mailbox Is
9     Array datos[n];
10    Int count, first, last = 0;
11 Begin
12     Loop
13         Select
14             When (count < n) -> Accept Depositar (msg: IN string)
15                 # Sentencias de la operacion Depositar
16             End Depositar;
17             Or
18             When (count > 0) -> Accept Retirar (msg: OUT string)
19                 # Sentencias de la operacion Retirar
20             End Retirar;
21         End;
22     End;
23 End;
24 End nombreProceso;
```

4.4. Notaciones primitivas

Mecanismo que *combina RPC, Rendezvous y PMA en un paquete coherente*. Con este mecanismo, será posible implementar una solución óptima al problema de filtros y pares que interactúan mediante el uso de las técnicas de RPC y Rendezvous.

Los programas, son colecciones de módulos que exportan operaciones visibles para otros módulos o procesos.

Una operación puede ser invocada por `call` (sincrónico) o por `send` (asincrónico). Como también una operación puede ser servida por RPC o Rendezvous (según la elección del programador del módulo).

Invocación	Servicio	Efecto
call	proc	Llamado a procedimiento.
call	in	Rendezvous.
send	proc	Creación dinámica de un proceso.
send	in	PMA.

Capítulo 5

Paradigmas de interacción entre procesos

5.1. Introducción

Existen 3 (tres) esquemas básicos de interacción entre procesos: productores/consumidores, cliente/servidor y pares que interactúan. Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a nuevos modelos o paradigmas de interacción entre procesos.

5.2. Manager and workers

Es la implementación distribuida del modelo *bag of tasks*. Diversos procesos workers comparten una bolsa de tareas. Cada worker repetidamente toma una tarea de la bolsa, la ejecuta y posiblemente genere una (o más) tareas nuevas para depositar en la bolsa.

La principal ventaja de este paradigma, en cuanto a cómputo paralelo, es la facilidad para escalar y balancear la carga de trabajo de cada worker.

Por el otro lado un proceso manager representa la bolsa de tareas. Es decir, mantiene la lista de tareas, recolecta los resultados y detecta la finalización de todas las tareas.

En esencia, un manager funciona como un server. Mientras que, un worker como un cliente.

Caso de uso: multiplicación de matrices.

5.3. Heartbeat

Los procesos, periódicamente, deben intercambiar información con mecanismos de tipo send/receive.

En particular, puede ser utilizado cuando los datos están divididos entre los procesos workers. Donde, cada worker es responsable de actualizar una parte de los datos. Dicha actualización, dependerá de los valores que contenga cada worker o sus vecinos inmediatos.

La lógica de este paradigma, tiene relación con su nombre. Los procesos se expanden enviando información a sus vecinos y se contraen recibiendo información actualizada desde sus vecinos.

La interacción entre vecinos, dada por la disposición de las primitivas send/receive, produce una especie de barrera de sincronización entre ellos. Impidiendo que, cada worker inicie su fase de actualización antes de que su vecino no finalice la suya.

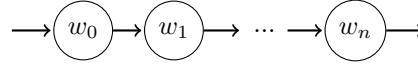
Caso de uso: paralelizar soluciones iterativas → topología de una red, procesamiento de imágenes.

5.4. Pipeline

Los procesos, funcionan como filtros y se encuentran conectados entre sí; formando una colección de filtros. De esta manera, la información fluye por los procesos utilizando alguna forma de receive/send. Existen 3 (tres) formas de conectar los procesos:

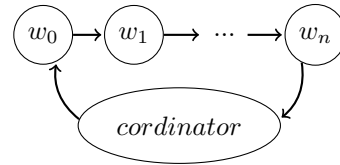
Pipeline Abierto

La fuente de entrada y salida, del pipeline, no está especificada.



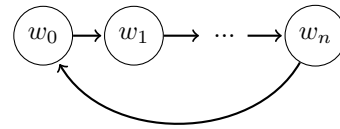
Pipeline Cerrado

Pipeline abierto conectado por un coordinador. El coordinador, produce los datos de entrada para w_0 y consume la salida de w_n .



Pipeline Circular

La salida de w_n está conectada a la entrada de w_0 . Útil cuando la solución, no puede obtenerse en una sola pasada.



5.5. Probes and echoes

La interacción entre procesos permite recorrer estructuras dinámicas, disseminando y recolectando información.

La estructura de muchos programas distribuidos, puede verse como árboles o gráficos. En el cual, los procesos son nodos y los canales aristas.

Una primitiva **probe** es un mensaje enviado por un nodo, a todos sus sucesores. Mientras que, una primitiva **echo** es su respectiva respuesta. Este paradigma, es el análogo al algoritmo Deep Search First (DFS).

Caso de uso: Cuando se desea recorrer una red, de la cual no se conoce la cantidad de nodos → redes móviles.

5.6. Broadcasts

Los procesos suelen enviar mensajes a todos los demás, de forma paralela, mediante una primitiva **broadcast**.

Dicha primitiva, no se asume atómica, por lo cual los procesos pueden recibir los mensajes en distinto orden al cual fueron enviados. La primitiva **receive** funciona de manera tradicional, sobre el canal de cada proceso.

Caso de uso: Mensaje broadcast en redes LAN.

5.7. Token passing

Los procesos se pasan, entre sí, un token que garantiza el acceso exclusivo a un recurso compartido.

Caso de uso: Exclusión mutua en ambientes distribuidos → Exclusión mutua con token ring.

5.7.1. Exclusión mutua con token ring

Existe un proceso *helper* por cada proceso que quiera solicitar acceso a una SC. Los procesos *helper*, esperan por una solicitud de acceso (*entryRequest*).

En caso de tener solicitudes pendientes (y poseer el token), concederán el acceso al proceso correspondiente (*grant*). Luego, esperaran a recibir una solicitud de salida (*exitRequest*).

Una vez finalizada la atención, pasarán el token al proceso *helper* siguiente.

```
1  # Exclusion mutua con token ring (PMA).
2  Chan token[n];
3  Chan entryRequest[n];
4  Chan exitRequest[n];
5  Chan grant[n];
6
7  Process Worker[i=1..n]
8  while(true)
9      send entryRequest[i]();
10     receive grant[i]();
11     # Seccion Critica ...
12     send exitRequest[i];
13 end;
14 end;
15
16 Process Helper[i=1..n]
17 while(true)
18     receive token[i]();
19     if (!empty(entryRequest[i]))
20         receive entryRequest[i]();
21         send grant[i]();
22         receive exitRequest[i]();
23     end;
24     send token[(i mod n) + 1]();
25 end;
26 end;
```

5.8. Servidores replicados

Los procesos servidores gestionan el acceso a un recurso. Al existir n instancias del recurso, se replicará un servidor por cada una. También, puede utilizarse para abstraer a los clientes la cantidad real de recursos.

Caso de uso: En el problema de los filósofos, estos le solicitan sus cubiertos (recursos) a un mozo (servidor). Los filósofos, desconocen la existencia de cubiertos ajenos.