Un modello base di *rendez-vous*

SCD

Anno accademico 2018/19
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@math.unipd.it

Modello base – 1

- **Interazione di tipo cliente-servente**
  - **Il servente specifica i servizi che intende fornire ai clienti**
  - **La specifica del servente dichiara i canali d'accesso (*entry*) che corrispondono a ciascuno dei servizi esposti**
    - Ogni canale specifica il suo proprio protocollo di scambio parametri
  - **Il cliente effettua richiesta (*entry call*) nominando servente e canale**
  - **Il servente fornisce uno dei servizi richiesti esprimendone esplicitamente la propria accettazione**
  - **La comunicazione tra servente e cliente è sincrona e non richiede necessariamente il passaggio di dati**

Modello base – 2

```
task type Operator is
   entry Query (A_Person   : in Name;
                An_Address : in Address;
                A_Number   : out Number);
end Operator;
Ann : Operator;
```

Specifica di punto di accesso e protocollo

```
task type User;
task body User is
   My_Number : Number;
begin
   …
   Ann.Query(
      "…", "…",
      My_Number);
   …
end User;
```

```
task body Operator is
begin
   …
   loop
      accept Query(A_Person   : in Name;
                   An_Address : in Address;
                   A_Number   : out Number) do
      …;
   end loop;
   …
end Operator;
```

Invocazione

Realizzazione di accettazione

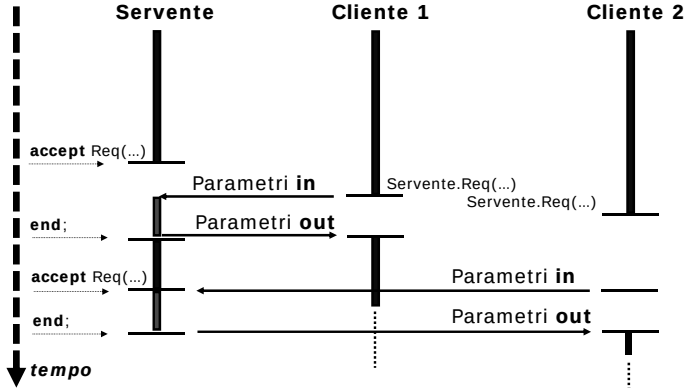
Modello base – 3

- **Storicamente chiamato *rendez-vous***
  - **Cliente e servente devono incontrarsi su uno specifico canale nello stesso istante temporale**
- **Al momento dell'incontro, i parametri di modo `in` passano dal cliente al servente**
- **Il servente esegue il servizio richiesto come una normale procedura e poi restituisce i parametri di modo `out` al cliente sul canale**
- **A quel punto la sincronizzazione si interrompe e i processi riprendono la loro esecuzione concorrente**

Modello base – 4

- Il servente si sospende in attesa di richieste
- Il cliente si sospende in attesa del servente
- La chiamata del cliente viene posta in una coda associata al canale (*entry queue*)
  - L'ordine di accodamento è normalmente FIFO
  - Può essere configurato diversamente, p.es. su base prioritaria, ma con rischio di *starvation*

Modello base – 5



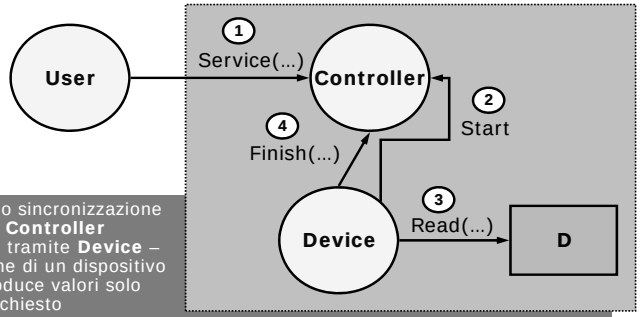
Sincronizzazione tripartita – 1

- Il modello *rendez-vous* è
  - Sincrono rispetto alla comunicazione
  - Asimmetrico rispetto all'interfaccia e alla denominazione
  - Bidirezionale rispetto al flusso dei dati
- Le azioni del servente nella sincronizzazione possono coinvolgere processi terzi
  - Per realizzare forme avanzate di sincronizzazione preservando separazione funzionale tra le parti

Sincronizzazione tripartita – 2

- Il coinvolgimento di processi terzi nella sincronizzazione di lato *server* ammette due forme distinte e duali
  - Annidamento di accettazioni
    - Modellando una **macchina a stati** in cui alcuni stati sono raggiungibili solo a partire da un dato stato iniziale
  - Invocazione di richieste nella realizzazione di una accettazione
    - Realizzando un **servizio composito** che racchiude il possibile contributo di più serventi che si siano ripartiti tra loro il lavoro

Sincronizzazione tripartita – 3



- Utilizzando sincronizzazione tripartita, **Controller** realizza – tramite **Device** – l'astrazione di un dispositivo **D** che produce valori solo quando richiesto
- L'entità **Device** è una macchina a stati che usa l'accettazione delle sue *entry call* come evento di transizione

Sincronizzazione tripartita – 4

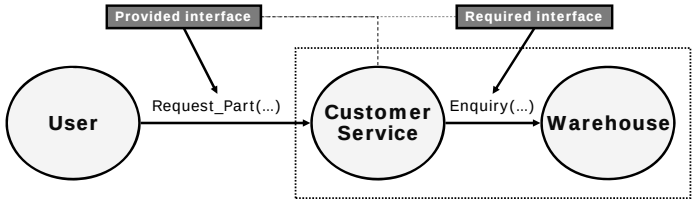
```
task User;
task Device;
task Controller is
 entry Service (I : out Integer);
 entry Start;
 entry Finish (K : out Integer);
end Controller;
```

```
task body User is
…
 Controller.Service (Val);
 …
end User;
```

```
task body Controller is
begin
 loop
  accept Service (I : out Integer) do
   accept Start;
   accept Finish (K : out Integer) do
    I := K; -- azione sincronizzata
   end Finish;
  end Service;
 end loop;
end Controller;
```

```
task body Device is
 Val : Integer;
 procedure Read
   (I : out Integer);
begin
 loop
  Controller.Start;
  Read(Val);
  Controller.Finish(Val);
 end loop;
end Device;
```


Sincronizzazione tripartita – 5



- Strutturazione gerarchica con *encapsulation*
- Il servizio Request_Part(…) nasconde al cliente la necessità di eventuale approvvigionamento presso componenti incapsulati all'interno del servente

Sincronizzazione tripartita – 6

```
task Warehouse is
 entry Enquiry
   (Item : Part_Number;
    In_Stock : out Boolean);
end Warehouse;

task Customer_Service is
 entry Request_Part
   (Order : Part_Number;
    Part  : Spare_Part;
    Order : Order_Number);
end Customer_Service;
```

Difetto: i servizi di Warehouse sono in questo modo visibili a tutti e non solo a Customer_Service

```
task body Customer_Service is
 In_Stock : Boolean;
 …
begin
 loop
  …
  accept Request_Part
    (Order : Part_Number;
     Part  : Spare_Part;
     Order : Order_Number) do
  …
  if In_Stock then
   Part  := The_Part; Order := None;
  else
   Warehouse.Enquiry(Order, In_Stock);
   if In_Stock then
    … -- go get part from Warehouse
    Part  := The_Part; Order := Next_Order_Nr;
   end if;
  end if;
 end Request_Part;
 end loop;
end Customer_Service;
```

Punti d'accesso privati – 1

- Un servente non deve necessariamente esporre al pubblico tutti i suoi canali
- Alcuni possono essere ristretti per motivi di incapsulazione e/o di astrazione
- La dichiarazione dei canali deve in tal caso distinguere tra <u>pubblici</u> e <u>privati</u>

Punti d'accesso privati – 2

```
task User;
task Controller is
 entry Service (I : out Integer);
private
 entry Start;
 entry Finish (K : out Integer);
end Controller;
```

```
task body User is
…
 Controller.Service(Val);
 …
end User;
```

In questo modo la visibilità ai canali privati è ristretta al solo ambito (*scope*) del processo **Controller**

```
task body Controller is
 task Device;
 task body Device is
  Val : Integer;
  procedure Read (I : out Integer) is … ;
 begin
  loop
   Controller.Start;
   Read(Val);
   Controller.Finish(Val);
  end loop;
 end Device;
-- continues in sidebox
```

```
begin -- Controller
 loop
  accept Service (I : out Integer) do
   accept Start;
   accept Finish (K : out Integer) do
    I := K;
   end Completed;
  end Service;
 end loop;
end Controller;
```


Casi d'errore

- Una eccezione sollevata durante la sincronizzazione ne causa l'abbandono e si propaga a entrambi i partecipanti
- Emettere una richiesta d'accesso verso un processo terminato è un errore a tempo di esecuzione e solleva una eccezione nel chiamante

Limiti del modello base

- Il servente può accettare richieste su <u>un solo canale</u> alla volta
  - Il cliente può inviare <u>una sola richiesta</u> alla volta
- Una volta postosi in attesa su canale, il servente vi resta fino all'arrivo di una richiesta
  - Inviata la richiesta, il cliente resta sospeso in attesa della sua accettazione e del completamento delle relative azioni

Requisiti di estensione – 1

- **Requisiti servente (RS) – più critici**
  1. **Poter attendere su più canali simultaneamente**
  2. **Limitare l'attesa a un tempo limite (*time-out*)**
  3. **Poter abbandonare immediatamente l'attesa su un canale che non abbia richieste in coda**
  4. **Poter terminare quando nessun cliente fosse più in grado di emettere richieste**
     - Il comportamento più naturale per un vero *server*

Osservazione

- **I requisiti RS 1 e RS 3 sono assimilabili a quanto previsto dal modello "*Guarded Commands*" di Dijkstra**
- **I requisiti RS 2 e RS 4 hanno motivazione più pragmatica e meno algebrica**
  - **Ma con conseguenze da considerare con cautela**

Requisiti di estensione – 2

- **Requisiti cliente (RC) – meno critici**
  - **A un singolo cliente non serve poter inviare più richieste alla volta**
    - Un processo cliente coeso ha una logica interna sequenziale
    - Per inviare più richieste simultaneamente servono più clienti paralleli
  2. **Poter fissare un tempo limite all'attesa dell'accettazione di una richiesta (cf. RS 2)**
  3. **Poter abbandonare l'attesa di accettazione ove essa non fosse immediatamente disponibile (cf. RS 3)**

Estensioni di lato servente – 1

- **RS 1. Attesa su più punti d'accesso**
  - **Il servente può erogare più servizi, ciascuno attraverso messaggi su canale tipato dedicato (*entry*)**

```
task Server is
 entry S1 (…);
 entry S2 (…);
end Server;
```

```
task body Server is
…
begin
 loop
  select
   accept S1(…) do … end S1;
  or
   accept S2(…) do … end S2;
  end select;
 end loop;
end Server;
```

Estensioni di lato servente – 2

- **RS 1. (continua)**
  - **Se nessuna richiesta fosse disponibile su alcun canale al momento della valutazione il servente si pone in attesa**
    - Sulla `select`
  - **La valutazione avviene sempre simultaneamente su tutti i canali considerati**
  - **Qualora canali diversi avessero richieste in attesa, la scelta tra essi è non deterministica**
    - Come nel modello di Dijkstra
  - **La politica base per l'attesa su canale è FIFO**
    - Altre politiche (p.es. per urgenza) possono essere contemplate

Estensioni di lato servente – 3

- **RS 1. (continua)**
  - **Opportuno aderire più pienamente al modello di Dijkstra**
  - **Porre "guardie" sui canali per specificare le condizioni logico-funzionali sotto le quali richieste su quel canale possano essere accettate**

```
select
   Guard_1 => accept …;
or
   Guard_2 => accept …;
or
   …
or
   Guard_N => accept …;
end select;
```

- La guardia è una espressione Booleana, di tipo "**when** <condizione>" il cui verificarsi abilita la considerazione del canale
- Le guardie entro un comando **select** sono valutate simultaneamente e una sola volta all'inizio del comando

Estensioni di lato servente – 4

- **RS 2 e RS 3 hanno entrambi l'obiettivo di limitare il tempo massimo di attesa ma con semantiche diverse**
  - **RS 2. Fissare un tempo limite non nullo entro il quale il servente è disposto ad attendere l'arrivo di richieste su uno dei canali considerati**
    - Tempo di attesa relativo o assoluto secondo bisogno
  - **RS 3. Abbandonare immediatamente l'attesa in assenza di richieste all'instante di valutazione**
    - Equivalente ad ammettere solo tempo di attesa nullo

Estensioni di lato servente – 5

- **RS 2.**
  - **Un limite temporale di attesa non nullo consente al servente di adempiere al suo compito base**
    - Senza però diventare incapace di fare altro a causa di attese infinite
  - **Al perdurare di assenza di richieste sui suoi canali il servente può assumere che i clienti si trovino in una condizione di errore**
    - Ma così l'eventuale anomalia non si propaga al servente

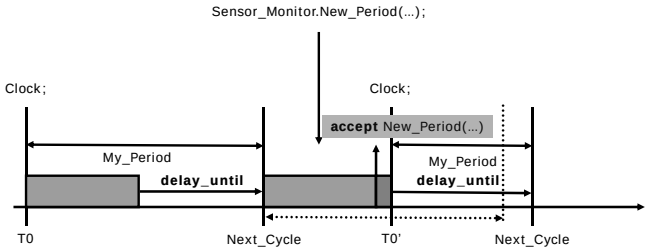
Esempio – 1

```
with Ada.Real_Time; use Ada.Real_Time;
task Sensor_Monitor is
 entry New_Period (Period : Time_Span);
end Sensor_Monitor;
…
task body Sensor_Monitor is
 My_Period : Time_Span := Milliseconds(10_000);
 Next_Cycle : Time := Clock + My_Period;
begin
 loop
  select
   accept New_Period (Period : Time_Span) do
    My_Period := Period;
   end New_Period;
   Next_Cycle := Clock + My_Period;
   delay until Next_Cycle;                    (1)
  or
   delay until Next_Cycle;
   -- do periodic work (e.g., read sensor)
   Next_Cycle := Next_Cycle + My_Period;
  end select;
 end loop;
end Sensor_Monitor;
```

- Comportamento di base periodico, con lettura di sensore ogni **10** secondi
- Capace di variare dinamicamente l'ampiezza del periodo in caso di richiesta del cliente

L'effetto del cambiamento è ottenuto dalla presenza del blocco di comandi in (1)

Esempio – 1: l'effetto



- **L'arrivo di una richiesta "`New_Period`" crea un nuovo riferimento T0' per i successivi periodi**
  - **Interrompendo la periodicità precedente**
  - **La soluzione data in (1) comporta discontinuità temporale**

Esempio – 2

```
task type Watchdog (Minimum_Distance : Duration) is
 entry All_is_Well;
end Watchdog;

task body Watchdog is
begin
 loop
  select
   accept All_is_Well;
   … -- client is alive and well
  or
   delay Allowable_Distance;
   … -- client may have failed, raise alarm
  end select;
 end loop;
end Watchdog;
```

Il modello delle guardie di Dijkstra applica anche all'alternativa di attesa temporale che può pertanto ammettere una guardia

Estensioni di lato servente – 6

- **RS 3. Attesa nulla**
  - **Il servente può voler considerare solo canali che abbiano richieste in attesa <u>al momento del controllo</u> altrimenti effettuare azioni alternative**
    - Questo rende possibile l'attesa attiva anche se indesiderabile!

```
select
 accept A;
or
 accept B;
else
 C;
end select;
```

L'effetto richiesto può essere ottenuto in 2 modi alternativi

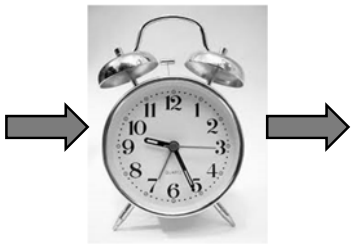
Forma esplicita (preferibile)

Forma implicita per T=0.0

```
select
 accept A;
or
 accept B;
or
 delay T;
 C;
end select;
```

RS 2 vs. RS 3

- **Il *runtime* fa cose diverse nei due casi**
- **Per RS 2 deve «armare la sveglia»**
  - **A meno di valore costante fissato a 0 a tempo di compilazione**
- **Per RS 3 non ne ha bisogno**



Estensioni di lato servente – 7

- **RS 4. Terminazione in mancanza di clienti**
  - **L'indipendenza tra clienti e servente può far sì che il servente sopravviva al completamento dei suoi clienti**
    - In questo caso è desiderabile che anche il servente possa terminare
  - **La terminazione del servente può essere gestita programmaticamente**
    - Per esempio con valori sentinella nella versione bis del crivello di Eratostene
    - Ma in quel programma non agirebbero serventi "puri" ma degeneri!
  - **Trattandosi però di un requisito generale di modello di *rendez-vous* è bene disporre di una soluzione generale**
    - Basta aggiungere un'alternativa `terminate` nel comando `select`

Estensioni di lato servente – 8

- **RS 4. (continua)**
  - **Un servente sospeso su comando `select` con alternativa `terminate` aperta viene considerato "completo" allorché**
    - Il master da cui esso dipende ha completato la propria esecuzione
    - Ogni altro processo dipendente da quello stesso master è
    - Già terminato, oppure a sua volta sospeso su un comando `select` con alternativa `terminate` aperta
  - **La condizione 1 assicura che non vi possano essere nuove richieste di servizio in arrivo**
  - **La condizione 2 applica transitivamente**

Ultime volontà ☺

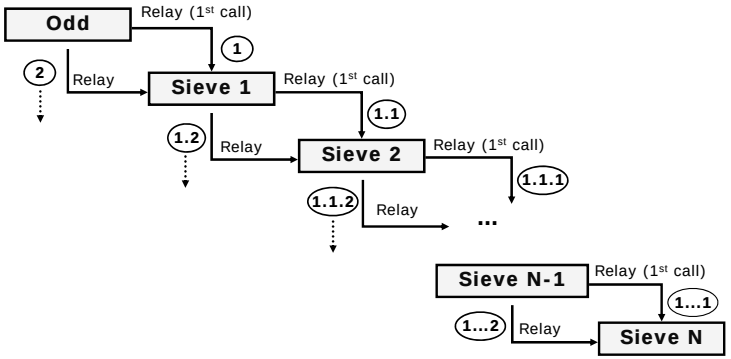
- **La semantica di terminazione RS 4 va arricchita in modo da permettere al processo terminante di effettuare azioni esplicite di finalizzazione**
  - **Le ultime volontà …**

- **Alcuni tipi esportano un metodo di finalizzazione che viene invocato dalla macchina astratta quando un oggetto di quel tipo esce di *scope***
  - **La terminazione di un processo il cui *scope* contenga istanze di tali tipi comporta l'invocazione automatica dei loro metodi di finalizzazione**

Esempio – 3

- **Il crivello di Eratostene sincrono**
  - **Ogni coppia di processi nella sequenza comunica tramite *rendez-vous***
  - **L'effetto di sincronizzazione rende superflua la mutua esclusione sui dati del servizio**
    - L'accodamento FIFO sulla coda d'accesso preserva l'ordine dei valori da esaminare (proprietà di serializzazione)
  - **Aggiungiamo controllo di terminazione alle istanze dei processi «crivello»**
  - **In caso di terminazione, vogliamo che il processo terminante ce ne fornisca notifica**

Esempio – 4



Estensioni di lato cliente

- **Per il lato cliente avevamo solo 2 esigenze**
  - **RC 2. Limite temporale non nullo all'attesa di servizio**
    - Equivalente al requisito **RS 2** di lato servente
    - Il limite riguarda solo la durata massima di attesa fino a inizio sincronizzazione
    - Nessuna relazione con la durata effettiva della sincronizzazione!
  - **RC 3. Abbandonare l'attesa a servente non immediatamente disponibile**
    - Equivalente al requisito **RS 3** del lato servente
    - Sta al *runtime* trattare di più clienti che desiderino simultaneamente conoscere la disponibilità istantanea di uno stesso servente

Usi del modello cliente-servente

- **Un servente è un'entità reattiva capace di garantire mutua esclusione**
  - **Eseguendo una sola alternativa `accept` alla volta**
- **L'esecuzione della sincronizzazione rappresenta la sezione critica**
- **La risorsa condivisa deve però essere visibile soltanto al processo servente**

```
task body Buffer (…) is
 … -- the shared resource
begin
 …
 loop
  select
   when …
    accept Put (…) do … end Put;
   … -- local housekeeping
  or
   when …
    accept Get (…) do … end Get;
   … -- local housekeeping
  or
   terminate;
  end select;
 end loop;
end Buffer;
```

```
task type Buffer (…) is
 entry Put (…);
 entry Get (…);
end Buffer;
```

Abusi del modello cliente-servente

- **Programmazione incauta può causare situazioni di stallo**
  - **Anche il modello *rendez-vous* esteso <u>non</u> è capace di impedirne strutturalmente il rischio**

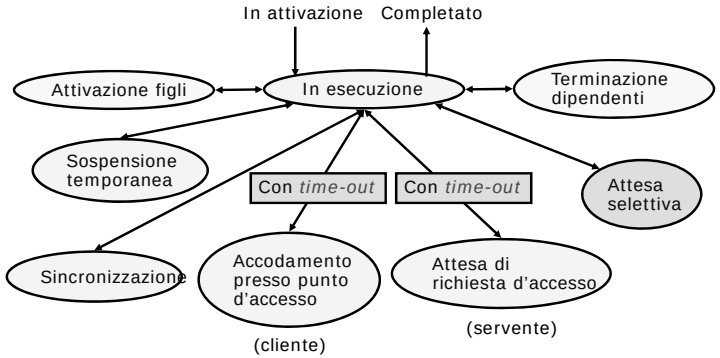
```
task T1 is
 entry A;
end T1;
…
task body T1 is
begin
 T2.B;
 accept A;
end T1;
```

```
task T2 is
 entry B;
end T2;
…
task body T2 is
begin
 T1.A;
 accept B;
end T1;
```


Una buona prassi

- **I processi dovrebbero essere usati soltanto per realizzare entità attive <u>oppure</u> *server***
  - **Secondo una ben precisa architettura di sistema**
  - **Trattando ogni dipendenza esterna tramite incapsulazione**
- **Le entità con ruolo attivo non dovrebbero possedere canali ma solo inviarvi messaggi**
- **I *server* "puri" accettano richieste di accesso ma non ne fanno alcuna**

Stati d'esecuzione di processo



Stati d'esecuzione di processo