

## “MONITOR ATTIVI”

Il progetto consiste di due parti:

- I) la prima è dedicata alla realizzazione, in java, di un meccanismo di interazione tra processi simile (ma non uguale) al meccanismo del rendez-vous (sincronizzazione estesa) come visto a lezione con riferimento agli statement `entry call/entry accept`;
- II) la seconda consiste nella utilizzazione del precedente meccanismo per scrivere il codice di un `thread server` che alloca dinamicamente a un insieme di thread clienti le risorse di un pool di risorse equivalenti.

Il meccanismo realizzato nella prima parte deve essere del tutto generale e non calibrato sulla realizzazione dell'esempio previsto nella seconda parte che, viceversa, deve essere concepita solo come un esempio di uso del meccanismo stesso.

### PRIMA PARTE

Come noto, il meccanismo di comunicazione “*tipo sincronizzazione estesa*” (rendez-vous) è stato introdotto con lo scopo di programmare le interazioni tra processi utilizzando un paradigma di programmazione analogo a quello proprio dei sistemi a memoria comune. In particolare, come nei sistemi a memoria comune, quando un processo cliente desidera eseguire un'operazione `op()` su una risorsa condivisa, invoca tale operazione come se fosse un'operazione di monitor. In realtà, a questa similitudine sintattica corrisponde una profonda differenza semantica. Infatti, il meccanismo `entry call/entry accept` presuppone un sottostante meccanismo a scambio di messaggi avendo le seguenti caratteristiche:

- a) l'operazione `op()` (operazione remota o `entry`) invece che essere eseguite dal chiamante, viene eseguita da un processo server che è l'unico ad avere accesso alla risorsa su cui il cliente desidera operare. Per questo alla chiamata della `entry` corrisponde l'invio al server da parte del cliente dei valori relativi ai parametri di input, che, una volta ricevuti dal processo server consentono a questo di eseguire l'operazione richiesta. Alla fine dell'esecuzione dell'operazione il server invia al cliente i valori restituiti dall'operazione stessa;
- b) la sincronizzazione che ne deriva implica che il cliente resti bloccato dal momento in cui esegue la chiamata fino a quando questa non viene accettata dal server e non termina l'esecuzione del servizio richiesto (corpo dello statement `accept`). Il server si blocca solo quando esegue un `accept` di entries su cui non sono state eseguite chiamate.
- c) Il server quindi corrisponde a un *monitor attivo* in quanto esegue un suo specifico flusso di controllo che gli permette di implementare un qualunque algoritmo in base al quale accettare le singole entries, tenendo conto dello stato interno della risorsa gestita.

Il progetto consiste nel realizzare in java un meccanismo analogo a quello sopra indicato, implementando in particolare i due precedenti punti b) e c). Viceversa, per quanto riguarda il punto a), tenendo conto che java definisce un ambiente a memoria comune, possiamo consentire al generico thread cliente di eseguire direttamente una qualunque operazione `op()` definita come `public` nell'ambito del thread server, evitando così ogni problema di invio e ricezione di messaggi.

Per chiarire meglio la specifica del progetto, supponiamo di fare riferimento a una risorsa condivisa che, in ambiente a memoria comune, potrebbe essere realizzata come istanza di un `monitor M`.

E supponiamo che una delle operazioni definite nel monitor, ad es. l'operazione `op()` preveda che per essere eseguita sia necessario verificare che la risorsa sia in un particolare stato interno. Per questo motivo, come noto, nel monitor viene riservata una variabile `condition c` associata alla condizione logica (`cond`) che deve essere verificata all'inizio dell'esecuzione della funzione con lo scopo di bloccare il processo chiamante sulla variabile `condition` se la condizione logica è falsa. Il

processo sarà poi svegliato mediante una `signal` quando la risorsa si troverà nello stato interno desiderato:

```
monitor m {
    .....
    condition c;

    public void op() {
        if (!cond) wait(c);
        <corpo della funzione>;
    }
    .....
}
```

dove il test può essere un semplice `if`, come sopra indicato o un `while` a seconda della semantica della `signal`.

Come noto, volendo simulare un tale schema in ambiente a scambio di messaggi ed utilizzando il meccanismo `entry call/entry accept`, il precedente monitor `M` potrebbe essere realizzato come un processo server che offre la `entry op()` che i processi clienti possono chiamare e che il server, quando la risorsa gestita si trova in uno stato interno che soddisfa la condizione logica `cond`, può decidere di accettare eseguendo lo statement

```
accept op() { <corpo della funzione>; }
```

Lo schema che si propone di realizzare è molto simile a questo con la sola differenza che l'operazione `op()` dichiarata come funzione `public` nel corpo del thread server viene, non solo chiamata, ma anche eseguita dal thread cliente. Compito del metodo `run()` del thread server è proprio quello di decidere quando accettare le chiamate delle funzioni offerte ai clienti e implementando una sincronizzazione esattamente uguale a quella del meccanismo `entry call/entry accept`.

Per questo motivo si richiede l'implementazione di una classe (chiamiamola `Entries`) il cui scopo è quello di definire un array di `entry`. La dimensione `n` dell'array viene passata come parametro al costruttore della classe:

```
Entries e = new Entries(n);
```

La classe deve offrire i seguenti tre metodi:

```
public void call(int x);
public void fineRendez_vous();
public void accept(int vet, int n);
```

necessari per implementare la richiesta sincronizzazione.

Ogni processo server che offre ai propri clienti un certo numero `n` di operazioni: `opa()`, `opb()`, ....., dichiarerà un'istanza `e` della classe `Entries` passando la dimensione `n` al relativo costruttore. Ognuna delle `n` `entry` è biunivocamente associata ad una delle `n` operazioni definite nel server. Inoltre, prescindendo dagli eventuali parametri e dal valore restituito, ogni operazione `opx()` viene così definita:

```
public void opx() {
    e.call(i);
```

```

    <corpo dell'operazione >;
    e.finerendez_vous();
}

```

se *i* è l'indice della entry associata all'operazione *opx()*.

L'esecuzione della *e.call(i)* deve bloccare il thread chiamante se il server, all'interno del proprio metodo *run()* non è in attesa di una chiamata della entry numero *i*. Il server, come indicato di seguito, si pone in attesa di una chiamata mediante l'esecuzione di un *accept*. Quando il server accetta tale chiamata, il thread chiamante esegue il corpo dell'operazione mentre il server si sospende in attesa che tale esecuzione sia terminata. Quando poi termina l'esecuzione del corpo dell'operazione, mediante l'esecuzione della *finerendez\_vous()* il processo *server* riprende la sua esecuzione all'interno del proprio metodo *run()*.

Il server accetta la chiamata di una entry o quella di una qualunque fra un sottoinsieme delle entries, invocando *accept* e passando come primo parametro un vettore di interi e come secondo parametro la dimensione del vettore. Il vettore passato come primo parametro contiene gli indici delle entries su cui si rimane in attesa se nessuna di esse è stata ancora chiamata, oppure ne viene accettata una qualunque fra quelle già chiamate, secondo il criterio degli statement non deterministici visti a lezione.

Per chiarire ulteriormente la specifica di ciò che viene richiesto, facciamo un semplice esempio. Supponiamo di fare riferimento al più semplice oggetto condiviso che, in ambiente a memoria comune potrebbe essere ad esempio un monitor che implementa un semplice semaforo:

```

monitor semaphore {
    int valore;
    condition verde;

    public void P() {
        if(valore>0) valore--;
        else wait(verde);
    }

    public void V() {
        if(empty(verde)) valore++;
        else signal(verde);
    }
}

```

Per realizzare un monitor attivo che implementi le stesse funzionalità del precedente monitor si dovrebbe realizzare un thread *server* che offra le due entries *P* e *V*.

Quindi nel server dovrà essere dichiarata una istanza della classe *Entries* con 2 elementi. Ad esempio riservando il primo di essi (quello di indice 0) alla *P* e l'altro (quello di indice 1) alla *V*. Ad esempio, a questo proposito, possiamo definire le due costanti *entry\_P* ed *entry\_V* come indicato di seguito

```

class Server extends Thread {
    Entries e=new Entries(2);
    int[] vet=new int[2];
    int valore;
    int n;
    final int entry_P=0, entry_V=1;
}

```

```

public Server(int v){
    valore=v;
}

public void P() {
    e.call(entry_P);
    valore--;
    e.finerendez_vous();
}

public void V() {
    e.call(entry_V);
    valore++;
    e.finerendez_vous();
}

public void run(){
    while(true)
        if (valore>0 {
            vet[0]=entry_P;
            vet[1]=entry_V;
            accept(vet, 2); }
        else { vet[0]=entry_V;
            accept(vet, 1); }
}
}

```

Come indicato, nel metodo `run` del `server`, quando il valore del semaforo è positivo, può essere accettata sia una `P` (entry 0) che una `V` (entry 1), passando all'`accept` un vettore di due elementi contenenti gli indici delle due entries; mentre se il semaforo è rosso viene accettata solo una entry, quella di indice 1 e cioè la `V`.

Concludendo, si tratta di implementare la classe `Entries` con i tre metodi sopra indicati, che fornisce il meccanismo per la realizzazione della sincronizzazione estesa e dell'accettazione di una qualunque entry fra quelle di un particolare sottoinsieme.

Può essere utilizzato qualunque meccanismo offerto da java. In particolare, per quanto riguarda la sincronizzazione è possibile utilizzare sia gli strumenti *low-level* (metodi `synchronized` insieme ai metodi `wait`, `notify` e `notifyAll`) che gli strumenti *high-level* (`Lock`, variabili `Condition` ecc.).

Qualunque strumento venga utilizzato, ricordarsi che una entry rappresenta una coda (FIFO) di chiamate, quindi l'ordine FIFO deve essere garantito.

## SECONDA PARTE

Utilizzando il meccanismo precedentemente realizzato, scrivere il codice di un thread server che alloca un insieme di risorse equivalenti (ed esempio 5 risorse equivalenti) ad un insieme di `n` thread clienti (ad esempio `n=4`). Ogni cliente può richiedere (e successivamente rilasciare) una singola risorsa, oppure può richiedere contemporaneamente due risorse, rilasciandole successivamente insieme.

- Per prima cosa scrivere il codice del server senza specificare nessuna particolare strategia di allocazione.
- Successivamente, scrivere il codice di un secondo server che, viceversa, alloca le stesse risorse in modo tale da privilegiare le richieste doppie rispetto alle richieste singole.