Active Monitors

Carmine Benedetto

10/02/2011

1 Introduzione

Questa relazione ha lo scopo di descrivere il progetto Active Monitors, realizzato in Java, illustrando le strutture dati utilizzate, i metodi implementati ed i test effettuati per verificare la correttezza dell'elaborato.
Il progetto consta di due parti:

- la prima verte sulla realizzazione di un meccanismo di interazione tra processi simile al meccanismo di sincronizzazione estesa del rendez-vous;
- la seconda consiste nella realizzazione di un thread server che utilizza il meccanismo precedentemente implementato per allocare dinamicamente a un insieme di thread clienti le risorse di un pool di risorse equivalenti.

2 Locazione dei sorgenti

I sorgenti dell'elaborato finale sono posizionati all'interno della cartella */src*, mentre i sorgenti e i relativi eseguibili dei test effettuati, sono distribuiti nelle varie cartelle */testx*.

3 Istruzioni di compilazione e di esecuzione

Per compilare l'applicazione da shell, posizionarsi nella directory interessata e digitare:

javac Server.java per compilare il server senza nessuna particolare strategia di allocazione;

javac ServerP.java per compilare il server che alloca le risorse in modo tale da privilegiare le richieste doppie rispetto alle richieste singole.

Per eseguire l'applicazione da shell, posizionarsi nella directory interessata e digitare:

java Server per eseguire il server senza nessuna particolare strategia di allocazione;

java ServerP per eseguire il server che alloca le risorse in modo tale da privilegiare le richieste doppie rispetto alle richieste singole.

IMPORTANTE: in ogni directory relativa ai test */testx* è presente uno script di shell *Testx.sh* che effettua la compilazione dei sorgenti e specifica le istruzioni di esecuzione relative al test. Per eseguire lo script, digitare da shell *./Testx.sh*

(controllare che lo script abbia i permessi di esecuzione e in caso contrario assegnarli allo stesso).

Procediamo con la descrizione delle classi utilizzate per realizzare il progetto.

4 Classe FIFO (FIFO.java)

La classe FIFO realizza una classica coda First In First Out.

4.1 Variabili e strutture dati:

private class Elem: elemento della coda. Contiene il puntatore all'elemento successivo *elem succ* e un oggetto di tipo Thread *Thread t*.

private elem testa: puntatore alla testa della coda.

4.2 Metodi pubblici:

public FIFO(): costruttore della classe FIFO. Inizializza testa a null.

public void insert(Thread id): effettua l'inserimento in coda di un oggetto di tipo Thread. La funzione crea un nuovo elemento, ne inizializza i valori e, effettuando gli opportuni controlli, lo inserisce in coda.

public Thread extract(): effettua l'estrazione dalla testa di un oggetto di tipo Thread. La funzione estrae l'oggetto Thread presente in testa dalla coda e lo restituisce, mentre restituisce null se la coda risulta vuota.

5 Classe Semaphore (Semaphore.java)

La classe Semaphore si occupa di realizzare un semaforo con i suoi relativi metodi atti a bloccare/sbloccare un oggetto su di esso.

5.1 Variabili e strutture dati:

private FIFO coda: oggetto di tipo FIFO utilizzato per realizzare una coda First In Firt Out di oggetti bloccati sul semaforo.

private int valore: valore associato al semaforo.

private int bloccati: numero di oggetti totali bloccati sul semaforo.

private Thread sbloccato: oggetto Thread che viene sbloccato sul semaforo e quindi estratto dalla coda delle risorse bloccate.

5.2 Metodi pubblici:

public Semaphore(int s): costruttore della classe Semaphore. Inizializza opportunamente le variabili della classe e in particolar modo segnala s come valore

del semaforo (*valore* = *s*).

public synchronized void p(): blocca l'oggetto sul semaforo. La funzione controlla il valore del semaforo e se questo risulta maggiore di 0 (semaforo verde) lo decrementa. Se il valore del semaforo non risulta maggiore di 0 (semaforo rosso) inserisce l'oggetto Thread che ha chiamato la funzione nella coda FIFO di risorse bloccate sul semaforo e incrementa il valore di bloccati.

public synchronized void v(): sblocca l'oggetto sul semaforo. La funzione controlla il valore di bloccati e se questo risulta pari a 0 (nessun elemento bloccato sul semaforo) incrementa la variabile valore. Se bloccati risulta diverso da 0 (ci sono elementi bloccati sul semaforo) estrae un oggetto Thread dalla testa della coda delle risorse bloccate sul semaforo, lo memorizza in sbloccato e decrementa bloccati.

Nelle funzioni vengono utilizzati metodi low-level synchronized assieme a metodi wait e notifyAll in modo tale che tutti i Thread chiamanti si sospendano escluso quello presente in sbloccato.

6 Classe Clients (Entries.java)

La classe Clients è una classe privata inserita all'interno della classe Entries che implementa una coda First In First Out di richieste effettuate dai client su ogni entry.

6.1 Variabili e strutture dati:

private int bloccati: numero di richieste effettuate dai client bloccate in coda.

private FIFO coda: oggetto di tipo FIFO utilizzato per realizzare una coda First In First Out di richieste effettuate dai client.

private Thread sbloccato: oggetto Thread estratto dalla coda di richieste effettuate dai client.

6.2 Metodi pubblici:

public Clients(): costruttore della classe Clients. Inizializza opportunamente le variabili della classe.

public synchronized void blocca(): inserisce la richiesta effettuata dal client in coda. La funzione inserisce l'oggetto Thread relativo alla richiesta effettuata dal client in coda, incrementa il valore di bloccati, rilascia la mutua esclusione che era stata attivata dalla chiamata di una call (vedi sezione successiva).

public synchronized boolean da_sbloccare(): estrae la richiesta effettuata dal client dalla coda di richieste. La funzione rilascia la mutua esclusione che era stata attivata dalla chiamata di una call (vedi sezione successiva), controlla il

valore della variabile bloccati e se questo risulta maggiore di 0 (ci sono richieste in coda) estrae un oggetto Thread dalla testa della coda e lo memorizza in sbloccato, decrementa bloccati e restituisce true segnalando di aver estratto la richiesta dalla coda. Se bloccati non risulta maggiore di 0 (non ci sono richieste in coda) restituisce false. Da notare che l'oggetto Thread sbloccato è quello presente in testa alla coda che corrisponde, quindi, alla prima delle richieste giunte in coda.

public synchronized boolean vuota(): controlla se la coda delle richieste effettuate dai client è vuota. La funzione controlla il valore della variabile bloccati e se questo risulta pari a 0 (non ci sono richieste in coda) restituisce true segnalando che la coda è vuota, altrimenti restituisce false.

Nelle funzioni vengono utilizzati metodi low-level synchronized assieme a metodi wait e notifyAll in maniera del tutto analoga a come erano stati utilizzati nelle funzioni della classe Semaphore. Nel caso ci siano Thread in attesa di essere sbloccati viene segnalato quale deve essere sbloccato tra tutti quelli in attesa tramite l'utilizzo della notifyAll a seguito dell'estrazione del Thread dalla coda delle richieste bloccate.

7 Classe Entries (Entries.java)

La classe definisce le entries e si occupa di realizzare i metodi necessari per implementare la richiesta di sincronizzazione con riferimento agli statement entry call/entry accept.

7.1 Variabili e strutture dati:

private Semaphore mutex: semaforo di mutua esclusione.

private Semaphore richiesta: semaforo utilizzato nella gestione delle richieste arrivate sulle entries e servite dal server.

private int num_en: numero di entries.

boolean accettata[]: arrey booleano che tiene traccia dell'accettazione di una richiesta su una entry.

private Clients c[]: array di tipo Clients che indicizza le richieste effettuate dai client sulle entries.

7.2 Metodi pubblici:

public Entries(int n): costruttore della classe Entries che ha come parametro il numero di entries. Inizializza in maniera opportuna le variabili e le strutture dati. In particolar modo inizializza tutti gli elementi dell'array accettata[] a false segnalando che inizialmente nessuna entry è abilitata ad accettare richieste dei client e crea una coda di richieste per ogni entry.

public void call(int x): metodo che richiede la mutua esclusione. La funzione, che ha come parametro l'indice della entry su cui effettuiamo la richiesta, attiva il semaforo di mutua esclusione e controlla che la richiesta sulla specifica entry sia stata accettata. Se la richiesta risulta essere stata accettata e quindi il server risulta abilitato a servire la richiesta, rilascia la mutua esclusione segnalando allo stesso tempo che le richieste su tutte le entries non possono essere più accettate perchè il server è già impegnato a servire la richiesta corrente. Se la richiesta non risulta essere stata accettata e quindi il server non risulta abilitato a servire la richiesta, inserisce in coda la richiesta effettuata dal client sulla entry.

public void fineRendex_vous(): metodo che permette al server di riprendere la sua esecuzione all'interno del metodo run(). La funzione segnala che la richiesta del client è stata soddisfatta sbloccando la condizione sul semaforo richiesta (*richiesta.v()*).

public void accept(int[] vet, int n): pone il server in attesa di ricevere una richiesta. La funzione, che ha come parametri un vettore di interi rappresentante l'indice delle varie entries e un intero rappresentante il numero di entries, richiede la mutua esclusione, controlla per ogni entry se la coda di richieste effettuate dai client sull'entry specifica è vuota (non c'è nessuna richiesta sulla specifica entry) e tiene conto di quante code vuote sono presenti. Se il numero di code vuote è pari al numero di entries, segnala di poter accettare una richiesta per ogni entry (*accettata[indice] = true*). Se il numero di code vuote è diverso dal numero di entries ci sarà almeno una richiesta accodata su una qualsiasi entry, segnalerà quindi di dover servire la richiesta *if(c[indice].da_sbloccare()) break;*. Prima di concludere, la funzione rilascia la mutua esclusione e si blocca sul semaforo richiesta.

8 Classe Server (Server.java)

La classe Server implementa un server che alloca un insieme di risorse equivalenti ad un insieme di Thread client. Il server in questione non predilige nessuna particolare strategia di allocazione delle risorse.

8.1 Variabili e strutture dati:

static int disponibili: numero di risorse disponibili.

static boolena libera[]: array che tiene conto delle risorse libere e di conseguenza di quelle utilizzate.

final static int op1: indice dell'entry relativa all'operazione op1;

final static int op2: indice dell'entry relativa all'operazione op2;

final static int op3: indice dell'entry relativa all'operazione op3;

final static int op4: indice dell'entry relativa all'operazione op4;

static Entries e: oggetto di tipo Entries.

8.2 Metodi pubblici:

public Server: costruttore della classe Server. Inizializza in maniera opportuna le variabili e le strutture dati, in particolar modo inizializza tutti gli elementi dell'array libera a true segnalando che tutte le risorse sono inizialmente libere, quindi non utilizzate.

public static int op1(): richiede una singola risorsa. La funzione effettua una chiamata al metodo call, segnala di aver utilizzato una risorsa settando un elemento dell'array libera a false e decrementando disponibili. Prima di concludere effettua una chiamata al metodo fineRendez_vous e restituisce l'indice della risorsa richiesta.

public static void op2(int r): rilascia una singola risorsa. La funzione, che ha come parametro l'indice della risorsa da rilasciare, effettua una chiamata al metodo call, segnala di aver rilasciato la risorsa ponendo l'elemento dell'array libera di indice r a true e incrementando disponibili. Prima di concludere effettua una chiamata al metodo fineRendez_vous.

public static int[] op3(): richiede contemporaneamente due risorse. La funzione effettua una chiamata al metodo call, segnala di aver utilizzato due risorse settando due elementi dell'array libera a false e diminuendo di 2 il valore di disponibili. Prima di concludere effettua una chiamata al metodo fineRendez_vous e restituisce il vettore di interi contenente i due indici delle risorse richieste.

public static void op4(int[] r): rilascia contemporaneamente due risorse. La funzione, che ha come parametro un vettore di interi contenete gli indici delle risorse da utilizzare, effettua una chiamata al metodo call, segnala di aver rilasciato le risorse ponendo gli elementi dell'array libera di indice r[0] e r[1] a true e aumenta di 2 il valore di disponibili. Prima di concluedere effettua una chiamata al metodo fineRendez_vous.

public void run(): metodo run. Il metodo effettua una chiamata al metodo accept all'interno di un ciclo infinito permettendo quindi al server di rimanere sempre attivo. All'interno del cliclo while viene effettuata una selezione della corretta chiamata al metodo accept da attuare in base al numero di risorse disponibili. La casistica si riduce a 3 situazioni particolari: nessuna risorsa disponibile (*disponibili == 0*); una risorsa disponibile (*disponibili == 1*); più di 2 risorse disponibili (*disponibili >= 2*).

Estratto di codice: ciclo while all'interno del metodo run()

```
while(true) {
    if(disponibili == 0) {
    vet[0] = op2;
    vet[1] = op4;
    e.accept(vet, 2);
```

```
    continue;
    }
    if(disponibili == 1) {
    vet[0] = op1;
    vet[1] = op2;
    vet[2] = op4;
    e.accept(vet, 3);
    continue;
    }
    if(disponibili >= 2) {
    vet[0] = op1;
    vet[1] = op2;
    vet[2] = op3;
    vet[3] = op4;
    e.accept(vet, 4);
    continue;
    }
}
```

public static void main(String[] args): metodo main. All'interno del metodo viene effettuato il controllo sulla correttezza del comando di esecuzione e vengono effettuate le chiamate ai metodi run del server e dei client attraverso la chiamata del metodo start.

9 Classe ServerP (ServerP.java)

Come richiesto da specifiche è stato realizzato il codice di un secondo server. La classe ServerP implementa un server del tutto analogo al precedente con la sola differenza di allocare le risorse in modo da privilegiare le richieste doppie rispetto alle richieste singole. Le variabili, le strutture dati e i metodi sono gli stessi della classe Server descritta in precedenza. L'elemento distintivo della classe ServerP risiede all'interno del ciclo while contenuto nel metodo run dove viene effettuata una selezione della corretta chiamata al metodo accept da attuare in base al numero di risorse disponibili prediligendo, dove possibile, un'operazione di richiesta doppia rispetto ad una di richiesta singola.

Estratto di codice: ciclo while all'interno del metodo run()

```
while(true) {
    if(disponibili == 0) {
    vet[0] = op4;
    vet[1] = op2;
    e.accept(vet, 2);
    continue;
    }
    if(disponibili == 1) {
    vet[0] = op4;
    vet[1] = op1;
    vet[2] = op2;
```

```
    e.accept(vet, 3);
    continue;
    }
    if(disponibili >= 2) {
    vet[0] = op3;
    vet[1] = op4;
    vet[2] = op1;
    vet[3] = op2;
    e.accept(vet, 4);
    continue;
    }
}
```

Continuiamo la relazione descrivendo i test intermedi realizzati per controllare la corretta esecuzione dell'elaborato.

10 Test

I test effettuati partono dai più semplici, che controllano il corretto funzionamento dei metodi implementati, fino a quelli più elaborati dove vengono simulate particolari scenari che potrebbero portare a situazioni di criticità.

10.1 Test 1: corretto funzionamento dei metodi che richiedono e rilasciano le risorse

10.1.1 Server standard

Vengono eseguite da parte di un solo client due richieste singole (op1), due rilasci singoli (op2), due richieste doppie (op3) e due rilasci doppi (op4) su un pool di 5 risorse equivalenti.

Estratto di codice: metodo main del Server

```
    public static void main(String args[]) {
    if(args.length <1) {
    System.out.println(COMANDO SERVER ERRATO);
    System.out.println(DIGITARE: java Server <numero_di_risorse >);
    return;
    }
    int risorse;
    risorse = Integer.parseInt(args[0]);
    Server s = new Server(risorse);
    s.start();
    Client cl = new Client();
    cl.start();
    }
```

Estratto di codice: metodo run del Client

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
int j;
int[] x = new int[2];
int[] y = new int[2];
i = Server.op1();
j = Server.op1();
Server.op2(i);
Server.op2(j);
x = Server.op3();
Server.op4(x);
y = Server.op3();
Server.op4(y);
}
```

Output di esecuzione

```
$ java Server 5

**********
SERVER ATTIVATO
**********
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 0
RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 3
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
```

Esecuzione metodo op2 :: Rilascia una singola risorsa
RISORSA RILASCIATA: 0
RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op2 :: Rilascia una singola risorsa
RISORSA RILASCIATA: 1
RISORSE DISPONIBILI: 5
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 0
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 3
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op4 :: Rilascia contemporaneamente due risorse
RISORSA RILASCIATA: 0
RISORSA RILASCIATA: 1
RISORSE DISPONIBILI: 5
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 0
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 3
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call

SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op4 :: Rilascia contemporaneamente due risorse
RISORSA RILASCIATA: 0
RISORSA RILASCIATA: 1
RISORSE DISPONIBILI: 5
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Come si può vedere le risorse vengono richieste e rilasciate correttamente. Il numero di richieste è pari al numero di rilasci quindi il valore finale delle risorse disponibili è pari a quello di partenza.

10.2 Test 2: esaurimento delle risorse a disposizione

10.2.1 Server standard

Vengono eseguite da parte di un solo client quttro richieste singole (op1) su un pool di 3 risorse equivalenti.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client cl = new Client();
cl.start();
}
```

Estratto di codice: metodo run del Client

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
int j;
int x;
int y;
i = Server.op1();
j = Server.op1();
```

```
x = Server.op1();
y = Server.op1();
}
```

Output di esecuzione

$ java Server 3

```
**********
SERVER ATTIVATO
**********
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 0
RISORSE DISPONIBILI: 2
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 1
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 2
RISORSE DISPONIBILI: 0
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
```

Come si può vedere vengono richieste e servite correttamente tre risorse. La quarta non viene servita perchè non vi è nessun metodo che rilascia risorse e le

risorse a disposizione sono esaurite. Il server rimane in attesa di ricevere richieste di eventuali altre operazioni, ma non soddisfa l'ultima richiesta di risorsa singola non essendoci la risorsa a disposizione.

10.3 Test 3: esaurimento delle risorse a disposizione con due Client

10.3.1 Server standard

Vengono eseguite due richieste singole (op1) da parte di un client e due richieste doppie (op3) da parte di un secondo client su un pool di 5 risorse equivalenti. Il secondo client viene ritardato di 3 secondi (3000 msec) in modo tale che vengano prima consumate le due risorse dovute all'esecuzione di due richieste singole da parte del primo client.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client1 cl1 = new Client1();
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
}
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
int j;
i = Server.op1();
j = Server.op1();
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
try {
Thread.sleep(3000);
}
```

```
catch(InterruptedException e){}
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
int[] y = new int[2];
x = Server.op3();
y = Server.op3();
}
```

Output di esecuzione

```
$ java Server 5

********************
SERVER ATTIVATO
********************
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 0
RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 3
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 9 ATTIVATO
CLIENT CHIAMANTE: 9
Esecuzione metodo call
CLIENT CHIAMANTE: 9
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 2
RISORSA RICHIESTA: 3
RISORSE DISPONIBILI: 1
CLIENT CHIAMANTE: 9
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 9
```

Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Come si vede vengono consumate due risorse richieste dal primo client (CLIENT 8), dopo di che vengono consumate altre due risorse conseguentemente ad una chiamata di richiesta doppia del secondo client (CLIENT 9), mentre la seconda richiesta doppia del secondo client non viene soddisfatta poichè vi è solo una risorsa a disposizione. La richiesta doppia prevede che vengano consumate due risorse contemporaneamente, di conseguenza con una sola risorsa a disposizione la richiesta non può essere soddisfatta.

10.4 Test 4: esaurimento delle risorse a disposizione e conseguente liberazione della risorsa con due Client

10.4.1 Server standard

Viene effettuata un'operazione di richiesta doppia (op3) seguita da un'operazione di richesta singola (op1) da parte del primo client che poi attende 4 secondi (4000msec) prima di effettuare un'operazione di rilascio di una singola risorsa (op2) su un pool di 3 risorse equivalenti. Il secondo client effettua un'operazione di richiesta singola (op1) che inizialmente non potrà effettuare data la saturazione delle risorse e che eseguirà solo dopo che il primo client avrà liberato una risorsa.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client1 cl1 = new Client1();
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
}
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
int i;
```

```
int j;
x = Server.op3();
i = Server.op1();
try {
Thread.sleep(4000);
}
catch(InterruptedException e){}
Server.op2(i);
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
System.out.println();
System.out.println(CLIENT + Thread.currentThread().getId() + ATTI-
VATO);
int i;
i = Server.op1();
}
```

Output di esecuzione

$ java Server 3

```
**********
SERVER ATTIVATO
**********
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
SERVER
CLIENT CHIAMANTE: 8
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 0
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 1
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT 9 ATTIVATO
CLIENT CHIAMANTE: 9
Esecuzione metodo call
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 2
RISORSE DISPONIBILI: 0
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
```

SERVER
Esecuzione metodo accept :: Server in attesa di richieste
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op2 :: Rilascia una singola risorsa
RISORSA RILASCIATA: 2
RISORSE DISPONIBILI: 1
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 9
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 2
RISORSE DISPONIBILI: 0
CLIENT CHIAMANTE: 9
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Come si vede il secondo client (CLIENT 9) viene attivato, ma non essendoci risorse disponibili la sua richiesta di risorsa singola non viene soddisfatta. Il primo client (CLIENT 8) si sospende per 4 secondi e alla sua ripartenza libera una risorsa (RISORSA RILASCIATA: 2). A questo punto il secondo client è in grado di eseguire l'operazione di richiesta singola che non aveva potuto espletare in precedenza.

10.5 Test 5: richiesta di una singola risorsa in assenza di risorse disponibili

10.5.1 Server standard

Viene effettuata da parte di un solo client un'operazione di richiesta singola (op1) su un pool di risorse disponibi vuoto (risorse pari a 0).

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client cl = new Client();
cl.start();
```

```
}
```

Estratto di codice: metodo run del Client

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = Server.op1();
}
```

Output di esecuzione

```
$ java Server 0

********************
SERVER ATTIVATO
********************
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
```

Si può osservare che l'operazione di richiesta singola non verrà effettuata poichè non è disponibile la risorsa necessaria.

10.6 Test 6: richiesta doppia di risorse in presenza di una risorsa disponibile

10.6.1 Server standard

Viene effettuata da parte di un solo client un'operazione di richiesta doppia (op3) su un pool di risorse disponibi pari a 1.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client cl = new Client();
cl.start();
```

```
}
```

Estratto di codice: metodo run del Client

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int x = new int[2];
x = Server.op3();
}
```

Output di esecuzione

```
$ java Server 1

********************
SERVER ATTIVATO
********************
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
```

Notiamo che ci troviamo in una situazione del tutto analoga a quella verificatasi nel test precedente. L'operazione di richiesta doppia non verrà effettuata poichè vi è solo una risorsa a disposizione.

10.7 Test 7: Server temporaneamente disabilitato con due Client

10.7.1 Server standard

Viene disabilitato temporaneamente il server, mentre vengono avviati due client che effettuano rispettivamente un'operazione di richiesta singola (op1) e un'operazione di richiesta doppia (op3) su un pool di 5 risorse equivalenti.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client1 cl1 = new Client1();
```

```
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
}
```

Estatto di codice: metodo run del Server

```
public void run() {
try {
Thread.sleep(4000);
}
catch(InterruptedException e){}
....
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = Server.op1();
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = Server.op3();
}
```

Output di esecuzione

$ java Server 5

```
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT 9 ATTIVATO
CLIENT CHIAMANTE: 9
Esecuzione metodo call
**********
SERVER ATTIVATO
**********
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 0
```

RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 9
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 1
RISORSA RICHIESTA: 2
RISORSE DISPONIBILI: 2
CLIENT CHIAMANTE: 9
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Si può notare che i due client vengono attivati anche se il server non è ancora attivo. Le loro richieste verranno soddisfatte dopo 4 secondi (4000 msec) quando il server diventerà attivo.

10.8 Test 8: Server temporaneamente disabilitato con quattro Client

10.8.1 Server standard

Viene disabilitato temporaneamente il server, mentre vengono avviati quattro client che effettuano rispettivamente un'operazione di richiesta singola (op1), un'operazione di richiesta doppia (op3), un'operazione di richiesta doppia (op3) e un'operazione di richiesta singola (op1) su un pool di 6 risorse equivalenti.

Estratto di codice: metodo main del Server

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
Server s = new Server(risorse);
s.start();
Client1 cl1 = new Client1();
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
Client3 cl3 = new Client3();
cl3.start();
Client4 cl4 = new Client4();
cl4.start();
```

```
}
```

Estatto di codice: metodo run del Server

```
public void run() {
try {
Thread.sleep(6000);
}
catch(InterruptedException e){}
....
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = Server.op1();
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = Server.op3();
}
```

Estratto di codice: metodo run del Client3

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = Server.op3();
}
```

Estratto di codice: metodo run del Client4

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = Server.op1();
}
```

Output di esecuzione

```
$ java Server 6

    CLIENT 8 ATTIVATO
    CLIENT CHIAMANTE: 8
    Esecuzione metodo call
    CLIENT 9 ATTIVATO
    CLIENT CHIAMANTE: 9
    Esecuzione metodo call
    CLIENT 10 ATTIVATO
    CLIENT CHIAMANTE: 10
    Esecuzione metodo call
    CLIENT 11 ATTIVATO
    CLIENT CHIAMANTE: 11
    Esecuzione metodo call
    ********************
    SERVER ATTIVATO
    ********************
    SERVER
    CLIENT CHIAMANTE: 8
    Esecuzione metodo op1 :: Richiede una singola risorsa
    Esecuzione metodo accept :: Server in attesa di richieste
    RISORSA RICHIESTA: 0
    RISORSE DISPONIBILI: 5
    CLIENT CHIAMANTE: 8
    Esecuzione metodo fineRendez_vous
    SERVER
    Esecuzione metodo accept :: Server in attesa di richieste
    CLIENT CHIAMANTE: 11
    Esecuzione metodo op1 :: Richiede una singola risorsa
    RISORSA RICHIESTA: 1
    RISORSE DISPONIBILI: 4
    CLIENT CHIAMANTE: 11
    Esecuzione metodo fineRendez_vous
    SERVER
    Esecuzione metodo accept :: Server in attesa di richieste
    CLIENT CHIAMANTE: 9
    Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
    RISORSA RICHIESTA: 2
    RISORSA RICHIESTA: 3
    RISORSE DISPONIBILI: 2
    CLIENT CHIAMANTE: 9
    Esecuzione metodo fineRendez_vous
    SERVER
    Esecuzione metodo accept :: Server in attesa di richieste
    CLIENT CHIAMANTE: 10
    Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
    RISORSA RICHIESTA: 4
    RISORSA RICHIESTA: 5
    RISORSE DISPONIBILI: 0
    CLIENT CHIAMANTE: 10
```

Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Notiamo che i quattro client vengono attivati, mentre il server risulta ancora inattivo. Le loro richieste verranno soddisfatte dopo 6 secondi (6000 msec), quando il server verrà attivato.

Nota 1: il server attivo è quello standard che allocherà le risorse in modo tale da prediligere le richieste singole alle richieste doppie. Verranno quindi eseguite per prime le operazioni di richiesta singole (op1) del primo client (CLIENT 8) e del quarto client (CLIENT 11) e successivamente le operazioni di richiesta doppia (op3) del secondo client (CLIENT 9) e del terzo client (CLIENT 10).

Nota 2: in caso di richieste dello stesso tipo, vengono servite quelle giunte prima in coda. Nel nostro caso le operazioni di richiesta singola (op1) verranno eseguite in ordine di chiamata: primo client (CLIENT 8), quarto client (CLIENT 11). Lo stesso discorso è valido per le operazioni di richiesta doppia (op3): secondo client (CLIENT 9), terzo client (CLIENT 10). Questo comportamento ci garantisce che ogni entry rappresenta una coda FIFO di chiamate.

10.9 Test 9: Server temporaneamente disabilitato con due Client

10.9.1 Server con priorità

Viene disabilitato temporaneamente il server, mentre vengono avviati due client. Il primo client effettua un'operazione di richiesta singola (op1) seguita da un'operazione di rilascio singolo (op2). Il secondo client effettua un'operazione di richiesta doppia (op3) seguita da un'operazione di rilascio doppio (op4). Le operazioni vengono effettuate su un pool di 5 risorse equivalenti.

Estratto di codice: metodo main del Server con priorità

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
ServerP sp = new ServerP(risorse);
sp.start();
Client1 cl1 = new Client1();
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
}
```

Estatto di codice: metodo run del Server con priorità

```
public void run() {
try {
Thread.sleep(4000);
}
catch(InterruptedException e){}
....
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = ServerP.op1();
ServerP.op2(i);
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = ServerP.op3();
ServerP.op4(x);
}
```

Output di esecuzione

$ java ServerP 5

```
CLIENT 9 ATTIVATO
CLIENT CHIAMANTE: 9
Esecuzione metodo call
CLIENT 8 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
**********
SERVER ATTIVATO
**********
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 9
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 0
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 3
CLIENT CHIAMANTE: 9
```

Esecuzione metodo fineRendez_vous
CLIENT CHIAMANTE: 9
Esecuzione metodo call
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 9
Esecuzione metodo op4 :: Rilascia contemporaneamente due risorse
RISORSA RILASCIATA: 0
RISORSA RILASCIATA: 1
RISORSE DISPONIBILI: 5
CLIENT CHIAMANTE: 9
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 0
RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT CHIAMANTE: 8
Esecuzione metodo op2 :: Rilascia una singola risorsa
RISORSA RILASCIATA: 0
RISORSE DISPONIBILI: 5
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Notiamo che i due client vengono attivati, mentre il server risulta ancora intattivo. Dopo 4 secondi (4000 msec) il server verrà svegliato è sarà in grado di servire le richieste inoltrate dai client.

Nota: il server in esecuzione è quello con priorità che allocherà, quindi, le risorse in maniera tale da privilegiare le richieste doppie (e di conseguenza i rilasci doppi) alle richieste singole (e di conseguenza i rilasci singoli). Nel caso specifico verranno soddisfatte prima le operazioni di richiesta doppia (op3) e di rilascio doppio (op4) del secondo client (CLIENT 9) e successivamente le operazioni di richiesta singola (op1) e rilascio singolo (op2) del primo client (CLIENT 8).

10.10 Test 10: Server temporaneamente disabilitato con quattro Client

10.10.1 Server con priorità

Viene disabilitato temporaneamente il server con priorità, mentre vengono avviati quattro client che effettuano rispettivamente un'operazione di richiesta singola (op1), un'operazione di richiesta doppia (op3), un'operazione di richiesta doppia (op2) e un'operazione di richiesta singola (op1) su un pool di 6 risorse equivalenti.

Estratto di codice: metodo main del Server con priorità

```
public static void main(String args[]) {
if(args.length <1) {
System.out.println(COMANDO SERVER ERRATO);
System.out.println(DIGITARE: java Server <numero_di_risorse >);
return;
}
int risorse;
risorse = Integer.parseInt(args[0]);
ServerP sp = new ServerP(risorse);
sp.start();
Client1 cl1 = new Client1();
cl1.start();
Client2 cl2 = new Client2();
cl2.start();
Client3 cl3 = new Client3();
cl3.start();
Client4 cl4 = new Client4();
cl4.start();
}
```

Estatto di codice: metodo run del Server con priorità

```
public void run() {
try {
Thread.sleep(6000);
}
catch(InterruptedException e){}
....
```

Estratto di codice: metodo run del Client1

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = ServerP.op1();
```

```
}
```

Estratto di codice: metodo run del Client2

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = ServerP.op3();
}
```

Estratto di codice: metodo run del Client3

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int[] x = new int[2];
x = ServerP.op3();
}
```

Estratto di codice: metodo run del Client4

```
public void run() {
System.out.println();
System.out.println(CLIENT+Thread.currentThread().getId()+ATTIVATO);
int i;
i = ServerP.op1();
}
```

Output di esecuzione

$ java ServerP 6

```
CLIENT 8 ATTIVATO
CLIENT 9 ATTIVATO
CLIENT CHIAMANTE: 8
Esecuzione metodo call
CLIENT 10 ATTIVATO
CLIENT CHIAMANTE: 9
Esecuzione metodo call
CLIENT CHIAMANTE: 10
Esecuzione metodo call
CLIENT 11 ATTIVATO
CLIENT CHIAMANTE: 11
Esecuzione metodo call
**********
SERVER ATTIVATO
**********
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
```

CLIENT CHIAMANTE: 9
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 0
RISORSA RICHIESTA: 1
RISORSE DISPONIBILI: 4
CLIENT CHIAMANTE: 9
Esecuzione metodo fineRendez_vous
SERVER
CLIENT CHIAMANTE: 10
Esecuzione metodo op3 :: Richiede contemporaneamente due risorse
RISORSA RICHIESTA: 2
RISORSA RICHIESTA: 3
RISORSE DISPONIBILI: 2
CLIENT CHIAMANTE: 10
Esecuzione metodo fineRendez_vous
Esecuzione metodo accept :: Server in attesa di richieste
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 8
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 4
RISORSE DISPONIBILI: 1
CLIENT CHIAMANTE: 8
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste
CLIENT CHIAMANTE: 11
Esecuzione metodo op1 :: Richiede una singola risorsa
RISORSA RICHIESTA: 5
RISORSE DISPONIBILI: 0
CLIENT CHIAMANTE: 11
Esecuzione metodo fineRendez_vous
SERVER
Esecuzione metodo accept :: Server in attesa di richieste

Notiamo che i quattro client vengono attivati, mentre il server con priorità risulta ancora inattivo. Le loro richieste verranno soddisfatte dopo 6 secondi (6000 msec), quando il server con priorità verrà attivato.

Nota 1: il server in esecuzione è quello con priorità che allocherà, quindi, le risorse in maniera tale da privilegiare le richieste doppie alle richieste singole. Nel caso specifico il server servirà le richieste con il seguente ordine: richiesta doppia (op3) del secondo client (CLIENT 9), richiesta doppia (op3) del terzo client (CLIENT 10), richiesta singola (op1) del primo client (CLIENT 8), richiesta singola (op1) del quarto client (CLIENT 11).

Nota 2: in caso di richieste dello stesso tipo, vengono servite quelle giunte prima in coda. Nel nostro caso le operazioni di richiesta doppia (op3) verranno eseguite in ordine di chiamata: secondo client (CLIENT 9), terzo client (CLIENT

10). Lo stesso discorso è valido per le operazioni di richiesta singola (op1): primo client (CLIENT 8), quarto client (CLIENT 11). Questo comportamento ci garantisce che ogni entry rappresenta una coda FIFO di chiamate.