


# Appunti di Middleware Technologies

Matteo Gianello

4 settembre 2014

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/4.0/>. .

# Indice

<b>1</b>	<b>Concorrenza</b>	<b>3</b>
1.1	Thread . . . . .	3
1.1.1	Introduzione ai threads . . . . .	3
1.1.2	Thread nei sistemi distribuiti . . . . .	5
1.1.3	Il modello preemptive . . . . .	5
1.2	I thread in C . . . . .	5
1.3	Concorrenza in Java . . . . .	12
<b>2</b>	<b>Core Communication Facilities</b>	<b>18</b>
2.1	Socket . . . . .	18
2.1.1	I socket in C . . . . .	18
2.1.2	I socket in Java . . . . .	21
2.1.3	IP Multicast . . . . .	23
2.2	Remote procedure call . . . . .	23
2.3	Remote method invocation . . . . .	24
2.3.1	Java RMI . . . . .	25
<b>3</b>	<b>CORBA</b>	<b>26</b>
3.1	CORBA e Java . . . . .	27
<b>4</b>	<b>Programmazione concorrente con OpenMP</b>	<b>29</b>

# 1 Concorrenza

In questo capitolo vedremo come i processi giochino un ruolo fondamentale nei sistemi distribuiti. Il concetto di processo proviene dall'ambito dei sistemi operativi ed è definito come un programma in esecuzione.

Per organizzare efficacemente un sistema client-server è spesso necessario utilizzare tecniche di *multithreading* in quanto questa tecnica permette ai client e ai server di essere costruiti in modo tale che la comunicazione e l'elaborazione locale siano sovrapposti ottenendo un alto livello di prestazione.

## 1.1 Thread

Anche se i processi costituiscono la base di tutti i sistemi, la loro granularità non è sufficiente a soddisfare i bisogni dei sistemi distribuiti. una gestione più fine, sotto forma di **thread**, rende più facile la costruzione di applicazioni distribuite e ottenere prestazioni migliori.

### 1.1.1 Introduzione ai threads

Prima di capire che cos'è un thread e che ruolo esso gioca nella costruzione di applicazioni distribuite è utile capire che cos'è in realtà un processo e che ruolo ha con i thread.

Per eseguire un programma, un sistema operativo crea un certo numero di processi virtuali. Per tener traccia di questi processi il sistema operativo mantiene aggiornata una **tabella dei processi** contenente elementi che vanno dalla memorizzazione dei registri della CPU, alla mappa della memoria, alla lista dei file aperti alle informazioni sugli *account* e così via.

Il sistema operativo fa sì che processi indipendenti non possano in alcun modo influire sulla correttezza degli altri processi, ovvero, è reso trasparente il fatto che più processi possano condividere concorrentemente la stessa CPU e le altre risorse hardware. Questa concorrenza però è ottenuta ad un prezzo abbastanza alto; ogni volta che viene creato un processo il sistema operativo deve creare uno spazio degli indirizzi completamente indipendente. Allocare memoria può voler dire inizializzare segmenti di memoria azzerando segmenti dati, copiare il programma in un segmento di testo e preparando uno *stack* per i dati temporanei. Altrettanto costoso è il passaggio tra un processo ed un altro a livello di CPU, in quanto oltre a salvare il contesto è necessario cambiare i registri ed invalidare la cache.

Come un processo un *thread* esegue il suo pezzo di codice indipendentemente dagli altri threads. A differenza dei processi nei threads non si cerca di ottenere un alto grado di trasparenza, in quanto il fatto di cercare di mantenere la trasparenza fa degradare le prestazioni; di conseguenza un sistema basato sui thread gestisce l'insieme minimo delle informazioni per gestire la CPU. Infatti, il **contesto di un thread** è spesso costituito solamente dal contesto della CPU e dalle informazioni per gestire il thread stesso come ad esempio lo stato dovuto al blocco di una variabile *mutex*. È quindi compito degli sviluppatori proteggere l'accesso ai dati tra i vari threads di un singolo processo.

**Utilizzo dei thread nei sistemi non distribuiti** Il vantaggio principale dell'utilizzo dei thread nei sistemi non distribuiti deriva dal fatto che in un processo a singolo thread quando viene effettuata una chiamata di sistema bloccante l'intero processo viene messo in pausa. Come nel caso di un foglio elettronico dove più celle sono collegate tra loro; in questo caso quando l'utente modifica il valore di una cella anche altre celle vengono rielaborate, ma tale rielaborazione è impensabile in un sistema a singolo thread in quanto il processo resterebbe bloccato in attesa di input e non calcolerebbe il valore delle altre celle.

Un altro vantaggio del multithreading è la possibilità di sfruttare il parallelismo quando si esegue il programma su sistemi multiprocessore. Il multithreading è usato anche nelle grandi applicazioni, le quali solitamente sono sviluppate come un insieme di processi cooperanti; tale cooperazione è realizzata tramite meccanismi di comunicazione tra processi (*IPC*, *interprocess communication*), ma questi meccanismi solitamente richiedono molti cambi di contesto che ne rallentano notevolmente le prestazioni. Invece di usare i processi un'applicazione può essere costruita mediante l'utilizzo di threads e la comunicazione tra questi avviene mediante l'uso dei dati condivisi, ed il passaggio da un thread all'altro può essere eseguito a livello utente.

**Implementazione dei thread** I thread sono spesso forniti sotto forma di pacchetto contenente le operazioni di creazione e distruzione dei threads sia le operazioni per la loro sincronizzazione come *mutex* e *condition*. Gli approcci per implementare un pacchetto di thread sono due. Il primo è costruire una libreria che viene eseguita completamente a livello utente, il secondo è lasciare che il kernel sia conscio dei threads e si occupi del loro scheduling.

Usare una libreria utente ha notevoli vantaggi, prima di tutto la creazione e la distruzione dei threads a livello utente è molto meno costosa in quanto il costo è dovuto solo all'allocazione della memoria per creare uno *stack*. Inoltre il cambio di contesto a livello utente può essere fatto con poche istruzioni. L'inconveniente principale dei thread a livello utente però è che una chiamata bloccante di sistema bloccherà l'intero processo e quindi bloccherà tutti i thread del processo.

Questo problema può essere aggirato implementando i threads a livello del kernel ma questo comporta che ogni operazione eseguita su un thread (creazione, distruzione, sincronizzazione e così via) dovrà essere eseguita a livello del kernel richiedendo quindi una chiamata a sistema che risulta essere molto più lenta e costosa come quella di un processo.

La soluzione ai problemi sta nell'uso di una forma ibrida chiamata **processi lightweight**. Un processo leggero viene eseguito nel contesto di un singolo processo (pesante) e per ogni processo ci possono essere più processi leggeri. Oltre a questi il sistema fornisce un pacchetto a livello utente per i threads mettendo a disposizione le solite operazioni. Il pacchetto dei thread è condivisibile da tutti i processi leggeri; questo significa che ogni processo leggero può eseguire il suo thread. Le applicazioni multithread vengono costruite creando dei thread e successivamente assegnando questi thread a un processo leggero.

Il pacchetto dei thread ha una singola routine per pianificare il thread successivo. Quando si crea un processo leggero gli si assegna uno *stack* e lo si mette alla ricerca di un thread da eseguire. I thread in esecuzione sono salvati in una tabella alla quale i processi leggeri accedono in mutua esclusione tramite l'uso di *mutex* nello spazio utente. Questo significa che la sincronizzazione tra threads è interamente eseguita a livello utente senza la necessità di informare il kernel.

Nel caso in cui vi sia una chiamata di sistema bloccante il contesto di esecuzione passa dalla modalità utente a quella kernel ma continua comunque nel contesto del processo leggero attuale. Nel momento in cui il processo leggero non può più proseguire allora il sistema può decidere di proseguire con un altro processo leggero ritornando alla modalità utente.

I vantaggi di utilizzare un sistema ibrido sono molti. Innanzitutto la creazione, la distruzione e la sincronizzazione dei threads è relativamente poco costosa in quanto avviene a livello utente. Se un processo ha abbastanza processi leggeri allora una chiamata bloccante di sistema non bloccherà l'intero processo. A livello di architetture multiprocessore processi leggeri diversi possono essere eseguiti su CPU diverse. L'unico inconveniente che si presenta è che i processi leggeri devono essere creati e distrutti ma fortunatamente tali operazioni non sono comuni.

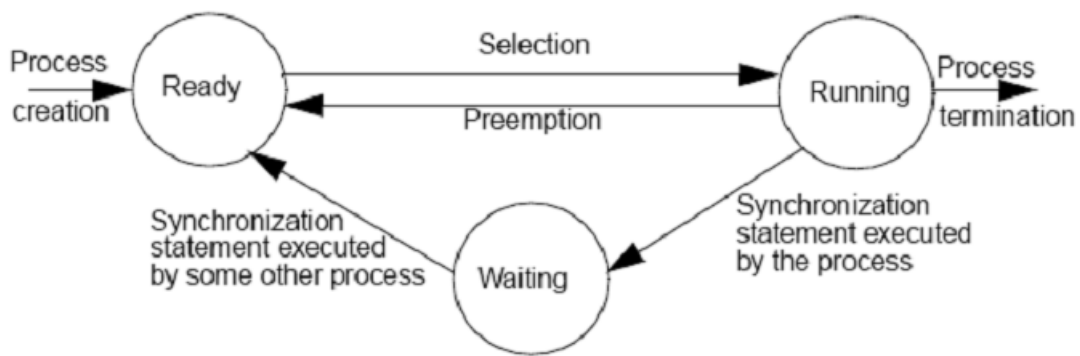


Figura 1: Modello preemptive

### 1.1.2 Thread nei sistemi distribuiti

Come abbiamo visto il vantaggio principale dell'uso dei threads è che una chiamata di sistema bloccante non blocca l'intero processo. Questa caratteristica è molto vantaggiosa nel caso di realizzazione di comunicazioni multiple come ad esempio la gestione di comunicazioni client-server.

**Client multithread** Per raggiungere un buon grado di trasparenza alla distribuzione i sistemi distribuiti che operano su reti globali hanno la necessità di nascondere lunghi tempi di propagazione dei messaggi. La tecnica più comune per nascondere la latenza dei messaggi è quella di avviare la comunicazione ed immediatamente iniziare a fare qualcos'altro. Un esempio molto diffuso sono i browser web che iniziano la comunicazione, ricevono una parte del codice HTML ed iniziano a visualizzare la pagina prima ancora di aver concluso la comunicazione.

**Server multithread** Anche se l'uso di client multithread offre notevoli vantaggi, il vero uso del multithreading è lato server. La pratica dimostra come l'uso del multithreading semplifica la codice e rende più facile lo sviluppo di applicazioni parallele per ottenere un alto livello di prestazioni.

Vediamo il caso di un *file server* dove un **dispatcher** riceve in ingresso su di una porta le richieste provenienti da diversi client. Dopo averla esaminata il dispatcher seleziona un **worker thread** inattivo a cui assegnare la richiesta. Il *worker* procede con la richiesta ed esegue una lettura bloccante sul file system locale; questo può far sì che il thread venga bloccato in attesa della lettura da disco, in tal caso viene selezionato un altro thread (*worker* o *dispatcher*) che procede con la sua esecuzione.

### 1.1.3 Il modello preemptive

Nei sistemi moderni oltre ai thread viene utilizzato il modello *preemptive*, ovvero è possibile forzare un processo ad abbandonare il suo stato di esecuzione. Solitamente questo meccanismo è utilizzato per implementare un meccanismo di *time slicing* come mostrato in Figura 1.

## 1.2 I thread in C

Tutti i sistemi UNIX sono multitasking con il sistema preemptive; tradizionalmente tutti i processi sono creati allo stesso modo tramite l'uso della primitiva `fork()`. La *fork* produce una copia del processo chiamante; questa copia è esattamente identica all'origina tranne per il valore

restituito dalla `fork` che per il processo figlio vale `0` mentre nel padre il valore restituito è il `pid` del figlio. Un piccolo esempio: La `fork` restituisce due copie completamente indipendenti

```

1  /*do parent stuff*/
2  ppid = fork ();
3  if (ppid < 0) {
4      fork_error_function ();
5  } else if (ppid == 0) {
6      child_function ();
7  } else {
8      parent_function ();
9  }

```

Listing 1: Esempio di uso della `fork`

dello stesso processo, questa indipendenza permette la protezione della memoria e la stabilità ma causa dei problemi quando si vuole che diversi processi lavorino sullo stesso problema; infatti sarebbe necessario usare *pipes* oppure *SysV IPC*. Inoltre il costo di switching tra processi multipli è molto alto, la sincronizzazione è lenta ed esistono dei limiti sul numero di processi che possono essere schedulati efficacemente.

Per questo sono stati introdotti i *threads* che invece possono essere schedulati all'interno del processo e risolvono molti problemi del lavoro multi processo. L'API più popolare per creare una applicazione multithread in ambiente UNIX è la *pthread* (*POSIX thread*).

Le operazioni che si possono eseguire con quest'API sono la creazione, la distruzione, la sincronizzazione (*join*), lo scheduling, il controllo dei dati e l'interazione con il processo principale. I *threads* dello stesso processo condividono le istruzioni di processo, gran parte dei dati, i descrittori dei file aperti, i segnali e lo user e il group id. Mentre per ogni thread abbiamo un distinto *ThreadID*, un certo numero di registri, uno stack pointer ed una certa priorità come possiamo vedere Figura 2. Vediamo ora quali sono le funzioni della API *pthread*

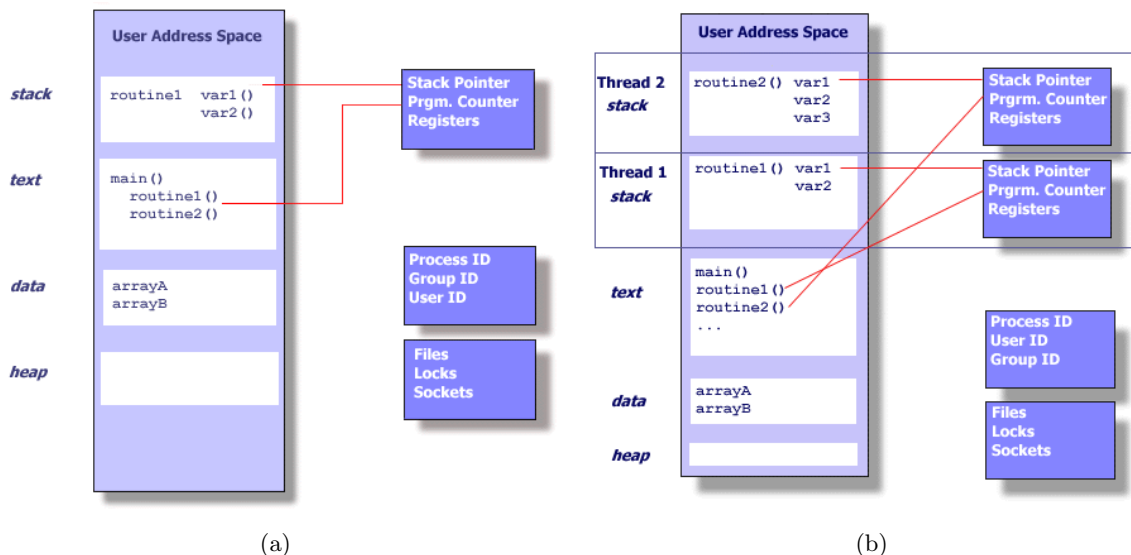


Figura 2: Memoria nel caso di processo (a) e di thread (b)

**Thread creation** La funzione per la creazione dei thread è: dove i valori sono:

```

1 int pthread_create (pthread_t *id, const pthread_attr_t *attr, void *(*routine)(void
    *), void *arg)

```

Listing 2: Funzione di creazione dei thread

**id:** un valore che identifica il thread che viene restituito dalla funzione.

**attr :** un attributo che può essere utilizzato per impostare alcuni valori del thread. Se viene impostato a *NULL* vengono impostati i valori di default.

**routine:** indica la funzione C che il thread eseguirà una volta creato.

**arg:** un singolo argomento che può essere passato a routine, deve essere passato come riferimento ad un puntatore di tipo *void*; in caso non vi siano valori si imposta a *NULL*.

**Thread termination** Esistono diversi modi in cui un pthread può terminare:

- Il thread termina la sua routine.
- Nel thread viene richiamata la `pthread_exit`.
- Il thread è cancellato da un altro thread tramite la chiamata della funzione `pthread_cancel`.
- L'intero processo termina quando viene chiamata una delle funzioni `exec` o `exit`.

Tramite la `pthread_exit` è possibile specificare uno stato di terminazione che può essere restituito alla sincronizzazione del thread. Inoltre è molto importante ricordare che la `pthread_exit` non chiude i file ed ogni file aperto all'interno del thread rimane aperto anche alla sua terminazione. Se la funzione *main* termina con una `pthread_exit` prima che i threads siano conclusi i threads proseguono la loro esecuzione altrimenti terminano alla conclusione del *main*. Vediamo un esempio di creazione e terminazione dei thread in C nel Listato 3

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 #define NUM_THREADS 5
6
7 /* Esempio di programma che utilizza la libreria pthread */
8 void *PrintHello(void * threadid) {
9     int *temp;
10    temp = (int *) threadid;
11    printf("\n%d: Hello World!\n", *temp);
12    pthread_exit(NULL);
13 }
14
15 int main(int argc, char *argv[]) {
16    pthread_t threads[NUM_THREADS];
17    int rc,t;
18
19    for (t = 0; t < NUM_THREADS; t++) {
20        printf("Creazione del thread %d\n", t);
21        rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
22        if(rc) {

```

```

23         printf("ERROR;_return_code_from_thread_create()_%d\n",rc);
24         exit(-1);
25     }
26 }
27
28 for (t = 0; t < NUM_THREADS; t++) {
29     pthread_join(threads[t],NULL);
30 }
31 pthread_exit(NULL);
32 }

```

Listing 3: Esempio di uso della API pthread

**Passaggio di argomenti** Come abbiamo visto nella *pthread\_create* è possibile impostare l'ultimo attributo con un attributo da passare alla routine che il thread eseguirà. Tale attributo deve essere convertito in un puntatore di tipo void. Tale passaggio presenta però alcuni tranelli, vediamo come nel Listato 4 come il passaggio per indirizzo crei un errore nell'esecuzione. Infatti, provando ad eseguire tale programma si rischia che più di un thread acceda contemporaneamente alla variabile *t* e si rischiano quindi di ottenere dei valori sbagliati.

```

1  int rc, t;
2  for(t=0; t<NUM_THREADS; t++) {
3      printf("Creating_thread_%d\n", t);
4      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
5      ...
6  }

```

Listing 4: Errore nel passaggio di argomenti ad un thread

Un possibile risultato di questo codice è quello seguente dove si può vedere che il thread numero 3 stampa il valore 4 anche se il thread numero 4 non è ancora stato creato (In realtà tutti i thread accedono alla variabile *t* con un ritardo in quanto manca la stampa del thread numero 0).

```

Creazione del thread 0
Creazione del thread 1
1: Hello World!
Creazione del thread 2
2: Hello World!
Creazione del thread 3
3: Hello World!
4: Hello World!
Creazione del thread 4

```

Per passare un argomento ad una routine è necessario controllare l'accesso ai dati da parte dei threads in modo che non vi siano possibili conflitti come nel caso del Listato 5. Nel quale viene passato ad ogni routine un puntatore ad un dato diverso.

```

1  int *taskids[NUM_THREADS];
2  for(t=0; t<NUM_THREADS; t++) {
3      taskids[t] = (int *) malloc(sizeof(int));
4      *taskids[t] = t;
5      printf("Creating_thread_%d\n", t);

```



```

6   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
7   ...
8 }

```

Listing 5: Metodo corretto nel passaggio di argomenti ad un thread

**Joining threads** L'operazione di *join* è uno dei modi nel quale si può implementare la sincronizzazione tra thread.

```

1 int pthread_join(pthread_t thid, void **thread_return)

```

dove i diversi campi sono:

**thid** è l'identificativo del thread su cui fare la join

**thread\_return** è il possibile valore di ritorno che si ottiene dall'invocazione della `pthread_exit`

Il funzionamento della funzione di *join* è specificato in Figura3

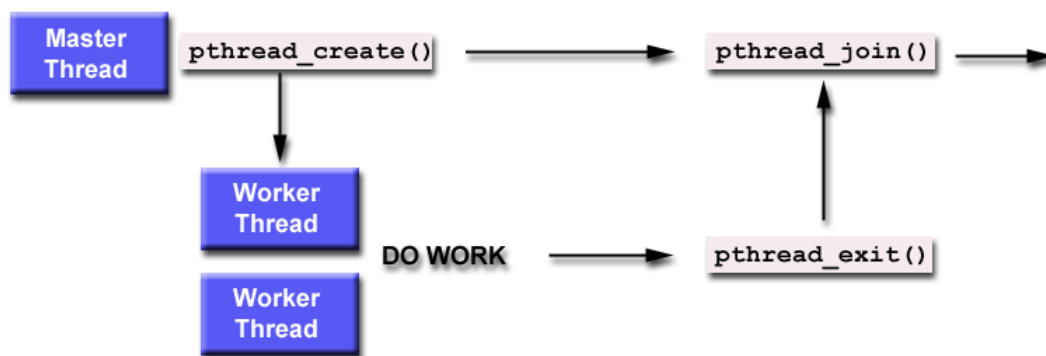


Figura 3: Funzionamento dell'operazione di join

**I mutex** Un *mutex* funziona come un *lock* proteggendo l'accesso a dei dati condivisi. Il concetto principale è che un solo thread alla volta può bloccare una variabile di tipo mutex, se più thread tenta di bloccare un mutex soltanto uno di questi effettuerà l'operazione con successo, inoltre i thread non possono bloccare un determinato mutex finché il thread che lo detiene non lo libera.

È compito del programmatore assicurarsi che ogni thread che utilizza dei dati condivisi usi i mutex.

Una variabile di tipo mutex può essere dichiarata sfruttando la parola chiave `pthread_mutex_t`, per inizializzare tale variabile, invece, è possibile sfruttare due metodi:

- Il primo è l'inizializzazione statica come ad esempio

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Il secondo metodo è l'inizializzazione dinamica richiamando la routine

```
pthread_mutex_init
```

In questo caso è possibile impostare alcuni parametri dell'oggetto mutex tramite le primitive `pthread_mutexattr_init` e `pthread_mutexattr_destroy` che rispettivamente creano e distruggono gli attributi di un mutex

In entrambi i casi di inizializzazione l'oggetto mutex è inizializzato *unlocked*. Infine, la routine `pthread_mutex_destroy` permette di rilasciare un mutex di cui non si ha più bisogno.

Esistono tre primitive per la gestione dei mutex, queste sono:

- `pthread_mutex_lock`: che si usa per acquisire un lock su di una variabile, nel caso in cui tale lock sia detenuto da un altro thread il thread che ha richiesto il lock si blocca finché il blocco non viene rilasciato.
- `pthread_mutex_trylock` molto simile alla routine precedente solo che nel caso in cui il blocco sia detenuto da un altro thread allora la routine restituisce un codice di errore che indica *busy*; è molto utile nel caso si vogliano prevenire condizioni di deadlock.
- `pthread_mutex_unlock`: questa routine permette di rilasciare il lock in possesso del thread ma restituisce un errore nel caso in cui si voglia rilasciare un lock su di una variabile già sbloccata o bloccata da un altro thread.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  typedef struct{
6      double *a;
7      double *b;
8      double sum;
9      int veclen;
10 } DOTDATA;
11
12 /*Define global variables and mutex*/
13 #define NUMTHREADS 4
14 #define VECLEN 100
15 DOTDATA dotstr;
16 pthread_t callThd[NUMTHREADS];
17 pthread_mutex_t mutexsum;
18
19 void *dotprod(void *arg) {
20     int i, start, end, offset;
21     double mysum;
22
23     offset = (int) arg;
24     start = offset * dotstr.veclen;
25     end = start + dotstr.veclen;
26     mysum = 0;
27     for (i = start; i < end; i++) {
28         mysum += (dotstr.a[i] * dotstr.b[i]);
29     }
30
31     pthread_mutex_lock (&mutexsum);
32     dotstr.sum += mysum;
33     pthread_mutex_unlock (&mutexsum);
34     pthread_exit((void *)0);
```

```

35 }
36
37 int main (int argc, char *argv[]) {
38     int i;
39     int status;
40
41     /* Assign storage and initialize values */
42     dotstr.a = (double*) malloc (NUMTHREADS*VECLEN*sizeof(double));
43     dotstr.b = (double*) malloc (NUMTHREADS*VECLEN*sizeof(double));
44     for (i=0; i<VECLEN*NUMTHREADS; i++) {
45         dotstr.a[i]=1;
46         dotstr.b[i]=1;
47     }
48     dotstr.veclen = VECLLEN;
49     dotstr.sum=0;
50     /* initialize the mutex */
51     pthread_mutex_init(&mutexsum, NULL);
52     /* Create threads to perform the dotproduct */
53     for(i=0;i<NUMTHREADS;i++) {
54         pthread_create(&callThd[i], NULL, dotprod, (void *)i);
55     }
56     /* Wait on the other threads */
57     for(i=0;i<NUMTHREADS;i++) {
58         pthread_join( callThd[i], (void **)&status);
59     }
60     /* After joining, print out the results */
61     printf ("Sum=%f\n", dotstr.sum);
62     free (dotstr.a);
63     free (dotstr.b);
64     pthread_mutex_destroy(&mutexsum);
65     pthread_exit(NULL);
66 }

```

Listing 6: Esempio di uso delle variabili mutex

**Condition variables** Mentre i mutex implementano la sincronizzazione tramite il controllo degli accessi sui dati le *condition variables* permettono ai thread di sincronizzarsi in base ad un determinato valore di un dato. Senza quest'aspetto dei thread bisognerebbe implementare un polling per verificare quando una particolare condizione viene riscontrata. Le *condition variables* sono un modo per ottenere lo stesso risultato senza il polling, e possono essere utilizzate anche insieme ai mutex. L'utilizzo principale sono tutti quei problemi della categoria **produttore-consumatore**.

Come per i mutex le condition variabile sono dichiarate utilizzando la parola chiave `pthread_cond_t`; per l'inizializzazione esistono due metodi:

- Statico

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER
```

- Dinamico tramite la funzione `pthread_cond_init` che permette di settare anche gli attributi della variabile tramite le due primitive

```
pthread_condattr_init
pthread_condattr_destroy
```

Che permettono rispettivamente di inizializzare e distruggere gli attributi della variabile

Infine tramite la primitiva `pthread_cond_destroy` è possibile liberare una variabile condizionale che non è più necessaria.

Per la gestione di questo tipo di variabile esistono diverse primitive:

- `pthread_cond_wait` è una routine che il thread finché una determinata condizione non si verifica, se chiamata quando vi è un lock attivo la routine sblocca il mutex e lo blocca nuovamente quando il thread si sveglia.
- `pthread_cond_signal` questa routine risveglia gli altri thread in attesa di una *condition variables*
- `pthread_cond_broadcast` può essere utilizzata al posto della routine precedente se ci sono più thread bloccati in uno stato di *wait*.

### 1.3 Concorrenza in Java

Come per il C anche il Java fornisce il supporto alla concorrenza a livello di linguaggio, esso mette a disposizione delle classi per istanziare ed eseguire nuovi thread più i metodi di sincronizzazione e le variabili di condizione. Il modo più semplice per creare un thread è quello di utilizzare la classe `java.lang.Thread` in questo caso è sempre necessario implementare un metodo `run()`.

```
1 public class MyThread extends Thread {
2     private String message;
3     public MyThread(String m) {message = m;}
4     public void run() {
5         for(int r=0; r<20; r++)
6             System.out.println(message);
7     }
8 }
9
10 public class ProvaThread {
11     public static void main(String[] args) {
12         MyThread t1,t2;
13         t1=new MyThread("primo_thread");
14         t2=new MyThread("secondo_thread");
15         t1.start();
16         t2.start();
17     }
18 }
```

Listing 7: Uso della classe Thread in Java

Come vediamo la nostra classe che implementa un thread estende l'oggetto *Thread*, in questa classe viene fatto l'override del metodo `run()` il quale non è altro che la routine che viene eseguita dal thread. Per far partire il thread è necessario richiamare il metodo `start()` dopo aver creato un nuovo oggetto.

Un'altra possibile soluzione è l'utilizzo dell'interfaccia `Runnable` la quale specifica soltanto che deve esistere un metodo `run()` che deve essere implementato. La classe `Thread` implementa anch'essa l'interfaccia `Runnable`. Come vediamo nel Listato 8 a differenza del caso precedente oltre all'oggetto *MyThread* deve anche essere creato un oggetto *Thread* corrispondente, al quale viene poi passato l'oggetto *MyThread*, ed infine il metodo `start()` viene invocato sull'oggetto *Thread*.

```

1 public class MyThread implements Runnable{
2     private String message;
3
4     public MyThread (String m) { message = m; }
5     public void run() {
6         for (int r = 0; r < 20; r++)
7             System.out.println(message);
8     }
9 }
10
11 class ProvaThread {
12     public static void main (String[] args) {
13         Thread t1, t2;
14         MyThread r1, r2;
15         r1 = new MyThread("PrimoThread");
16         r2 = new MyThread("SecondoThread");
17         t1 = new Thread(r1);
18         t2 = new Thread(r2);
19         t1.start();
20         t2.start();
21     }
22 }

```

Listing 8: Utilizzo dell'interfaccia Runnable

L'esecuzione dei thread non segue un ordine predefinito ma lo stesso codice può produrre risultati diversi su diversi computer o addirittura sullo stesso. Questa caratteristica è chiamata *non-determinismo* ed è un punto focale nella concorrenza.

Java di per sé implementa il modello *preemptive* e nel caso sia disponibile un meccanismo di *time-slicing* allora Java esegue i thread con la stessa priorità tramite un meccanismo di *round-robin*. Per definire quando un sistema multithread è corretto si devono rispettare due proprietà:

- *Sicurezza*: Un sistema si dice sicuro quando gli eventi malevoli non accadono.
- *Longevità*: Un sistema è longevo quando le cose buone possono accadere.

I possibili guasti che rientrano nella categoria Sicurezza sono quei guasti che avvengono a livello di esecuzione come i conflitti *read/write* e *write/write*. I meccanismi che invece riguardano la Longevità sono quei meccanismi che bloccano l'esecuzione del programma come:

- Lock
- Waiting
- CPU contention

Solitamente, purtroppo, le cose più semplici che si possono fare per aumentare la longevità ne riducono però la sicurezza e vice versa.

Vediamo ora quali sono i meccanismi che Java mette a disposizione per supportare la concorrenza.

**Exclusion** In un sistema sicuro ogni oggetto protegge se stesso da possibili violazioni della sua integrità. le tecniche di esclusione preservano l'invariante di un oggetto. Tre sono le tecniche principali per permettere l'*esclusione*:

- Immutabilità
- Esclusione dinamica (Locking)
- Esclusione strutturale

Per quanto riguarda l'**immutabilità** si ottiene creando le classi in modo che gli oggetti proteggano se stessi come nel Listato 9

```

1 class ImmutableAdder {
2     private final int offset;
3     public Immutableadder (int a) {
4         offset = a;
5     }
6     public int addOffset (int b) {
7         return offset + b;
8     }
9 }

```

Listing 9: Esempio di oggetto immutabile

I vantaggi di questa tecnica sono il fatto che non richiede sincronizzazione ed è molto utile per condividere degli oggetti tra i threads, ma sfortunatamente ha dei limiti di applicabilità. Per parlare di sincronizzazione introduciamo prima l'esempio del Listato 10

```

1 public class RGBColor {
2     private int r;
3     private int g;
4     private int b;
5
6     public void setColor (int r, int g, int b)
7         checkRGBVals(r, g, b);
8         this.r = r;
9         this.g = g;
10        this.b = b;
11    }
12 }

```

Listing 10: Esempio sincronizzazione

Ora immaginiamo che due thread chiamati *red* e *blue* vogliano impostare contemporaneamente il loro colore sullo stesso oggetto di tipo `RGBColor` a questo punto potrebbero verificarsi dei problemi in quanto i due thread tentano di scrivere lo stesso dato violando così la sua integrità. Per risolvere questo problema java tramite il locking serializza l'esecuzione del codice dichiarato *synchronized*. Ogni istanza di un oggetto possiede tali meccanismi di lock in quanto derivati dalla classe `Object`, l'unica eccezione si ha con l'utilizzo di array, infatti, bloccare un array non blocca gli elementi di tale array.

Esistono due modi per bloccare una parte di codice, si può dichiarare *synchronized* un intero metodo o un singolo blocco di codice, in caso di singolo blocco la funzione `synchronized` richiede l'oggetto sul quale effettuare il lock ( Listato 11 e Listato 12).

```

1 synchronized (object) {
2     //Lock is held
3     ...
4 }
5 //Lock is released

```

Listing 11: Sincronizzazione di una parte di codice

```

1 synchronized void f() {
2     //Lock is held
3     /* Body */
4 }
5 //Lock is released

```

Listing 12: Sincronizzazione di un metodo

I lock vengono automaticamente acquisiti all'ingresso del blocco o del metodo dichiarato `synchronized` e rilasciato all'uscita da esso.

Alcune regole chiave per l'uso della sincronizzazione sono:

- Sempre quando si effettua un aggiornamento a dei campi di un oggetto.

```

1 synchronized (point) {
2     point.x = 5; point.y = 7;
3 }

```

- Tutte le volte che si accede a dei dati che potrebbero essere aggiornati.

```

1 synchronized (point) {
2     if (point.x > 0) {...}
3 }

```

- Si può fare a meno di sincronizzare parti di metodo stateless

```

1 public void f() {
2     synchronized (this) {
3         state = ...;
4     }
5     operations();
6 }

```

- **Mai** sincronizzare parti di codice che contengono invocazioni ad altri oggetti

```

1 public void f() {
2     synchronized (this) {
3         ...
4     }
5     h.foo();
6 }

```

La strategia più sicura (ma non la più efficace) per realizzare un'applicazione OO concorrente è quella di utilizzare oggetti completamente sincronizzati, anche detti oggetti *atomici*, nei quali tutti i metodi sono sincronizzati, non esistono campi pubblici o altri tipi di violazione nell'incapsulamento, tutti i metodi sono finiti e hanno modo di rilasciare il lock, tutti i campi sono

inizializzati ad un valore consistente nel costruttore, ed infine lo stato dell'oggetto è consistente sia all'inizio che alla fine di ogni metodo anche in presenza di eccezioni.

Uno dei problemi principali della programmazione concorrente è il *deadlock*, tale problema si verifica quando due o più oggetti sono acceduti da due o più threads e tali thread detengono un lock mentre tentano di acquisire un lock detenuto da un altro thread.

L'assegnamento di un valore ad una variabile è un'operazione atomica (a parte per i *long* e i *double*), questo significa che generalmente non è necessario sincronizzare l'accesso ad una variabile. Tuttavia i thread solitamente memorizzano il valori delle variabile in memoria locale, questo significa che se un thread cambia il valore di una variabile un altro thread non vede il cambiamento. Per evitare questo meccanismo bisogna sincronizzare la variabile oppure dichiararla di tipo *volatile* che significa che ogni volta che una variabile è usata deve prima essere letta dalla memoria principale.

Il confinamento implementa l'incapsulamento garantendo che al massimo un'attività alla volta acceda agli oggetti. Questo meccanismo permette l'accesso ad un solo thread alla volta senza utilizzare i locking dinamici. Il punto principale è quello avere un punto di uscita dal thread. Esistono quattro categorie per verificare che un riferimento *r* ad un oggetto *x* può uscire da un metodo *m*:

- *m* passa *r* come argomento di un'invocazione ad un metodo o ad un costruttore di un oggetto
- *m* passa *r* come valore di ritorno di un metodo.
- *m* registra *r* in un campo accessibile da altre attività
- *m* rilascia un riferimento che però permette l'accesso ad *r*

Per quanto riguarda le collezioni, il framework `java.util.Collection` basata su uno schema *Adapter* permette la sincronizzazioni delle classi collection, infatti, ad eccezione di `Vector` e `Hashtable` le classi base per le collezioni (come `java.util.ArrayList`) sono non sincronizzate. Sono state così costruite una serie di classi sincronizzate attorno alle classi base come la `Collection.synchronizedList`.

Come abbiamo detto prima però la sincronizzazione non è molto efficiente infatti richiamare un metodo sincronizzato richiede un tempo quattro volte maggiore rispetto a metodi non sincronizzati; inoltre, esso riduce la concorrenza e diminuisce le performance, infine non vi è alcun modo di controllare il meccanismo dei lock.

Con java versione 5 sono stati introdotti nuovi meccanismi di sincronizzazione come la *sincronizzazione condizionata*. Prendiamo come esempio un parcheggio con una certa capacità e dei metodi che permettono l'arrivo e la partenza di automobili come esemplificato nel Listato 13.

```
1 public class CarParkControl {
2     protected int space;
3     protected int capacity;
4
5     public CarParckControl (int n) {
6         capacity = space = n;
7     }
8
9     synchronized public void arrive() {
10         ...; --space; ...;
11     }
12     synchronized public void depart() {
```



```

13         ...; ++space; ...;
14     }
15 }

```

Listing 13: Esempio di controllore di un parcheggio

Come per il C esistono però dei metodi che permettono una gestione più efficiente del controllore rispetto all'uso della `synchronized`; questi metodi sono:

- `public final void notify()`: che risveglia un singolo thread in attesa.
- `public final void notifyAll()`: risveglia tutti i thread in attesa.
- `public final void wait() throws InterruptedException`: pone il thread in attesa di un *notify*. Quando un thread viene posto in uno stato di wait esso rilascia il lock acquisito e lo riacquista al suo risveglio.

```

1 public class CarParkControl {
2     private int space;
3     private int capacity;
4
5     public CarParkControl (int n) {
6         capacity = space = n;
7     }
8     synchronized public void arrive() throws InterruptedException {
9         while (space == 0) wait();
10        --space;
11        notifyAll();
12    }
13    synchronized public void depart() throws InterruptedException {
14        while (space == capacity) wait();
15        ++space;
16        notifyAll();
17    }
18 }

```

Listing 14: Esempio di variabili condizionali in Java

Si può ridurre l'overhead dovuto al contex-switching sostituendo la `notifyAll` con la `notify`. Tale meccanismo può essere usato per migliorare le performance quando si è certi che almeno un thread è in atteso per eseguire un lavoro.

Alla condizione di *wait* è possibile associare un timer molto utile per migliorare la longevità del sistema in quanto tende a risolvere in modo automatico i deadlock.

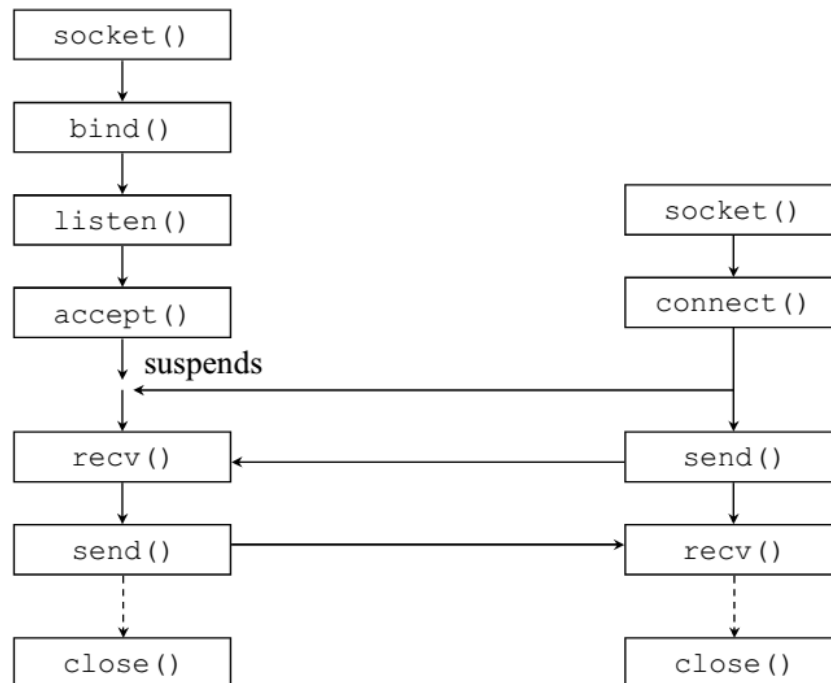


Figura 4: Flusso di esecuzione di una comunicazione orientata alla connessione

## 2 Core Communication Facilities

### 2.1 Socket

Nella programmazione di rete esistono diversi tipi di protocolli tra cui l'*unicast TCP*, l'*unicast UDP* e il *multicast IP*. Per ogni protocollo esiste un RFC che descrive come questi protocolli lavorano in pratica, tuttavia, il problema principale è capire come un programmatore possa trarre vantaggio da questi protocolli. La risposta a questa domanda sono i **Socket**, comparsi per la prima volta nel 1982 nel sistema Unix BSD oggi sono disponibili per qualsiasi sistema. I *socket* forniscono un'astrazione comune per la comunicazione interprocesso sia essa di tipo *connection-oriented (TCP)* sia di tipo *connection-less (UDP)*.

#### 2.1.1 I socket in C

Il flusso di esecuzione di una comunicazione tra due processi segue lo schema in Figura 4 in questo caso si tratta di una comunicazione orientata alla connessione infatti vediamo come il processo *client* (quello a destra) effettui una `connect()` prima di inviare o ricevere dei dati. Vediamo ora due esempi di un client e di un server TCP rispettivamente nel Listato 15 e nel Listato 16

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>

```

```

9
10 int main (int argc, char * argv[]) {
11     int sock, port;
12     char *host, *msg;
13     struct sockaddr_in sa;
14
15     /*parse command line arguments*/
16     if (argc < 3) {
17         printf ("Usage: client <host> <port> [<message>]\n");
18         exit(-1);
19     }
20     host = argv[1];
21     port = atoi(argv[2]);
22     if (argc == 3) msg = "Let me try...";
23     else msg = argv[3];
24     /*set the destination address*/
25     memset(&sa, 0, sizeof(struct sockaddr_in));
26     sa.sin_family = AF_INET;
27     sa.sin_port = htons(port);
28     sa.sin_addr.s_addr = inet_addr(host);
29     /*create the socket & connect it*/
30     sock=socket(AF_INET, SOCK_STREAM, 0);
31     if (sock < 0) {
32         perror("Creating the socket");
33         exit(-1);
34     }
35     if (connect(sock, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
36         perror("Binding");
37         exit(-1);
38     }
39     /*Send the message*/
40     if (send(sock, msg, strlen(msg), 0) < 0) {
41         perror("Writeing");
42         exit(-1);
43     }
44     /*Close the connection*/
45     close(sock);
46     return 0;
47 }

```

Listing 15: Client TCP

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
9
10 #define BUFFLEN 1024
11
12 int main (int argc, char * argv[]) {
13     int sock, port, s, n;

```

```

14  char buf[BUFFLEN];
15  struct sockaddr_in sa;
16
17  /*parse command line arguments*/
18  if (argc < 2) {
19      printf ("Usage: _server_<port>\n");
20      exit(-1);
21  }
22  port = atof(argv[1]);
23  /*set the destination address*/
24  memset(&sa, 0, sizeof(struct sockaddr_in));
25  sa.sin_family = AF_INET;
26  sa.sin_port = htons(port);
27  sa.sin_addr.s_addr = htonl(INADDR_ANY);
28  /*create the socket & connect it*/
29  sock=socket(AF_INET, SOCK_STREAM, 0);
30  if (sock < 0) {
31      perror("Creating _the_socket");
32      exit(-1);
33  }
34  if (bind(sock, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
35      perror("Binding");
36      exit(-1);
37  }
38  if (listen(sock, 0) < 0) {
39      perror("Listening");
40      exit(-1);
41  }
42  /*Accepting incoming request (main loop)*/
43  while (1) {
44      /*Accept new connection*/
45      s = accept(sock, NULL, NULL);
46      if (s < 0) {
47          perror("Accepting");
48          exit(-1);
49      }
50      /*Recive incoming data*/
51      while (1) {
52          memset(buf, 0, sizeof(buf));
53          if ((n = read(s, buf, BUFFLEN-1)) < 0)
54              perror("Receiving _data");
55          else if ( n == 0) break;
56          else {
57              printf("%s\n",buf);
58              if(buf[n-1] == 4) break;
59          }
60      }
61      close(s);
62  }
63  close(s);
64  close(sock);
65  return 0;
66 }

```

Listing 16: Server TCP

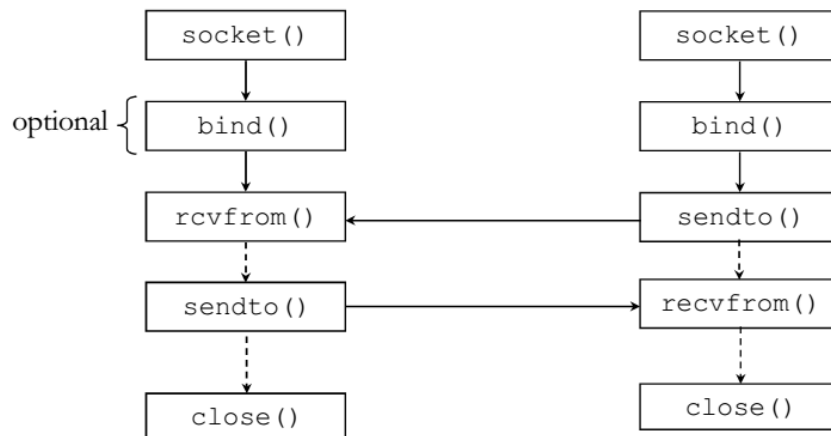


Figura 5: Flusso di esecuzione di una comunicazione senza connessione

Abbiamo visto come funziona una comunicazione orientata alla connessione vediamo ora come funziona invece una comunicazione di tipo *connection-less* mostrata in Figura5. Come vediamo in questo caso non è necessario effettuare l'**accept** ne la **connect** in quanto i messaggi sono inviati direttamente, questo meccanismo è più rapido del precedente ma risulta essere meno sicuro.

### 2.1.2 I socket in Java

In Java l'approccio ai *socket* è simile al C risulta tuttavia molto più semplice. Esistono cinque classi principali fornite dal *package java.net* queste classi sono:

- **InetAddress**: fornisce i metodi per ottenere l'indirizzo di un host dato il suo *hostname* e vice versa
- **ServerSocket**: utilizzata dal server per accettare le connessioni in ingresso.
- **Socket**: la classe principale utilizzata per la comunicazione
- **DatagramPacket**: classe utilizzata per inviare messaggi attraverso un **DatagramSocket**
- **DatagramSocket**: classe utilizzata per l'invio e la ricezione di messaggi senza l'ausilio di una connessione

Un esempio di un client e di un server TCP in Java è mostrato rispettivamente nel Listato 17 e nel Listato 18. Vediamo come in java la comunicazione avvenga tramite input e output stream associati ad una socket.

```

1 import java.io.*;
2 import java.net.*;
3
4 public class TCPClient {
5     public static void main(String args[]) throws Exception {
6         int port;
7         String host, msg;
8         if(args.length!=3) {
9             System.err.println("Usage: java TCPClient <host> <port> <msg>");

```

```

10         System.exit(-1);
11     }
12     host = args[0];
13     port = Integer.parseInt(args[1]);
14     msg = args[2];
15
16     Socket s = new Socket(host, port);
17     PrintWriter p = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
18     for(int i=0; i<10; i++) {
19         System.out.println("Sending:␣"+msg);
20         p.println(msg);
21         p.flush();
22         Thread.sleep(1000);
23     }
24     p.close();
25     s.close();
26 }
27 }

```

Listing 17: Client TCP in Java

```

1  import java.net.*;
2  import java.io.*;
3
4  public class TCPServer {
5      public static void main(String[] args) throws java.io.IOException {
6          int port;
7          ServerSocket sock;
8          Socket s;
9          BufferedReader in;
10         String line;
11         // Parse command line arguments
12         if(args.length!=1) {
13             System.err.println("Usage:␣java␣Server␣<port>");
14             System.exit(-1);
15         }
16         port = Integer.parseInt(args[0]);
17         // Create the server socket
18         sock = new ServerSocket(port);
19         // Accept incoming requests (main loop)
20         while(true) {
21             // Accept a new connection
22             s = sock.accept();
23             // Receive incoming data
24             in = new BufferedReader(new InputStreamReader(s.getInputStream()));
25             while((line = in.readLine())!=null) {
26                 System.out.println(line);
27             }
28         }
29     }
30 }

```

Listing 18: Server TCP in Java

Le classi `ObjectInputStream` e `ObjectOutputStream` permettono di leggere e scrivere su di uno stream serializzato; per essere trasmesso su questo tipo di stream un oggetto deve essere

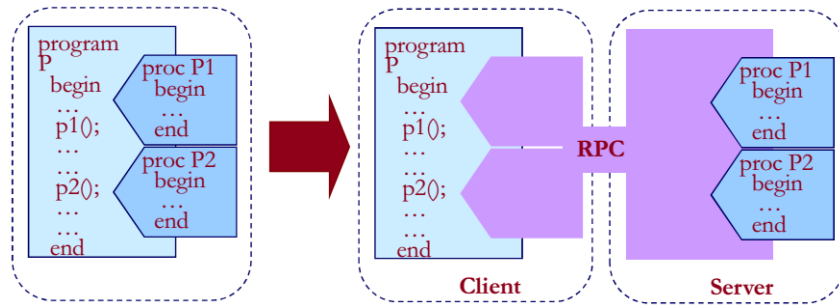


Figura 6: Modello di chiamata a procedura remota

serializzabile ovvero deve implementare l'interfaccia **Serializable**. La serializzazione permette una *copia profonda* di un oggetto. Un programmatore può impedire la serializzazione di un oggetto dichiarandolo di tipo **transient**.

### 2.1.3 IP Multicast

Il **multicast IP** è un protocollo di rete che permette la consegna di pacchetti UDP a molti destinatari contemporaneamente. Per implementare tale protocollo solitamente si riservano degli indirizzi IP di classe D per i gruppi *multicast*.

I componenti interessati a ricevere dei messaggi inviati in un gruppo devono effettuare una *join* a tale gruppo, non è necessario, invece, unirsi al gruppo per inviare un messaggio indirizzato ad un determinato gruppo.

Java fornisce una sottoclasse della classe **DatagramSocket** denominata **MulticastSocket** per implementare in modo semplice il multicast; questa classe aggiunge due metodi alla classe **DatagramSocket** che sono il **joinGroup** e il **leaveGroup**.

## 2.2 Remote procedure call

Con lo sviluppo della programmazione il problema che si è presentato è che l'interazione tra client e server avveniva sempre tramite le primitive di I/O del sistema, questo rendeva le applicazioni difficili da sviluppare. L'idea di *Sun Microsystems* è stata quella di permettere l'accesso remoto attraverso un meccanismo ben conosciuto ovvero quello delle chiamate a procedura come mostrato in Figura 6. Tuttavia questo modello comporta alcuni problemi soprattutto per quanto riguarda il passaggio dei parametri. Il primo problema dovuto al passaggio dei parametri è quello della conversione di strutture dati complesse come oggetti o **struct** in uno stream di byte sequenziale, tale problema è chiamato *serializzazione*. Il secondo problema che si presenta è il fatto che un sistema host potrebbe usare una rappresentazione dei dati differente ed è quindi necessaria una conversione, tale problema è chiamato *marshalling*.

I middleware forniscono supporto automatico per risolvere questi problemi, il marshaling e la serializzazione sono implementati automaticamente e entrano a far parte dello stub; inoltre permettono un'indipendenza dal linguaggio in quanto le procedure vengono rappresentate tramite un *Interface Definition Language* (IDL).

L'*interface definition language* alza il livello di astrazione della definizione del servizio separando l'*interfaccia* del servizio dalla sua *implementazione*. I vantaggi di questa tecnica è quello di definire un servizio indipendentemente dal linguaggio utilizzato per implementarlo, inoltre, l'utilizzo di un IDL definito formalmente permette la generazione automatica delle interfacce nel linguaggio target.

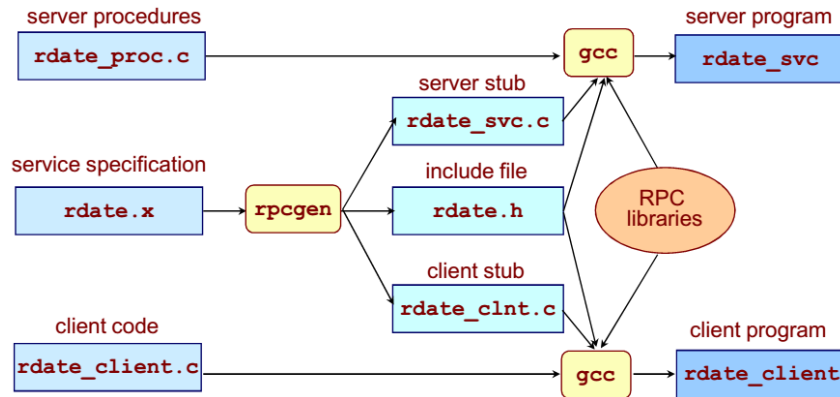


Figura 7: Funzionamento delle Sun RPC

Sun Microsystems RPC anche chiamata ONC RPC è lo standard *de facto* per internet, il formato dei dati è specificato da un XDR (*eXternal Data Representation*) che, nato come un linguaggio di rappresentazione dei dati, attualmente si è evoluto in un vero e proprio IDL. Il livello di trasporto può utilizzare sia il TCP che l'UDP, il passaggio di parametri è consentito solo tramite copia ed un solo valore di input e uno di output sono permessi.

Esiste un altro standard per le RPC, questo standard è il *Distributed Computing Environment* il quale è un insieme di specifiche e riferimenti ad implementazioni, questo standard fornisce specifiche per i servizi di alto livello come *directory service* o *distributed time service*; la sicurezza inoltre è fornita attraverso *Kerberos*. Il sistema di RPC di Microsoft DCOM e .Net sono basati su DCE.

Il funzionamento delle Sun RPC è mostrato in Figura 7 nel quale partendo da il file di specifica `rdate.x` mostrato nel Listato 19 e mediante l'invocazione del programma `rpcgen` si vengono a creare i file necessari per la stesura dei programmi client e server.

```

1 program RDATE_PROG {
2     version RDATE_VERS {
3         long BIN_DATE(void) = 1; /* procedure number */
4         string STR_DATE(long) = 2; /* procedure number */
5     } = 1; /* version number */
6 } = 0x20000001; /* program number */

```

Listing 19: Esempio di file `rdate.x`

Come detto il passaggio di parametri è consentito solo tramite copia, per molti linguaggi questo non è un problema in quanto non è supportato nemmeno a livello di linguaggio in altri contesti tuttavia è possibile utilizzare delle chiamate per *valore/risultato* al posto della chiamata per riferimento.

### 2.3 Remote method invocation

La *Remote Method Invocation* (RMI) sfrutta la stessa idea della RPC ma si basa su costruzioni di programmazione differenti, lo scopo è quello di ottenere i vantaggi di una programmazione Object-Oriented solamente in un contesto distribuito. Un'importante differenza rispetto alle RPC è che nelle RMI è possibile il passaggio di oggetti per riferimento, questo richiede il mantenimento delle relazioni con gli alias. In molti casi le RMI sono sviluppate su un livello di RPC. Nelle RPC l'IDL separa l'interfaccia dall'implementazione, questa separazione è alla base dei principi di programmazione OO, è naturale posizionare l'interfaccia di un oggetto su di un host



e l'implementazione su di un altro.

### 2.3.1 Java RMI

Java RMI è il più semplice tra i moderni sistemi ad oggetti distribuiti più diffusi, esso si concentra solamente sulle chiamate di metodi remote il resto dei servizi è fornito da altri componenti della famiglia Java, permette di effettuare il download delle classi, di avere dei proxies dinamici. Il concetto di interfaccia Java assume un nuovo ruolo con la distribuzione, un *oggetto remoto* deve implementare un'interfaccia che estende `java.rmi.Remote` come nell'esempio del Listato 20

```
1 import java.rmi.*
2 public interface AccountServer extends Remote {
3     Account getAccount (int num) throws RemoteException;
4 }
```

Listing 20: Esempio di interfaccia remota

Implementare un interfaccia remota non è sufficiente per accettare delle chiamate remote, per fare ciò è necessario *esportare* l'oggetto. Questo può essere fatto automaticamente nel costruttore se la classe deriva da `java.rmi.server.RemoteObject` come nel caso di `UnicastRemoteObject` oppure staticamente invocando il metodo `UnicastRemoteObject.exportObject`.

Per ottenere un riferimento ad un oggetto remoto esistono due modi, attraverso il passaggio di parametri o come valore di ritorno oppure interrogando esplicitamente un servizio di *lookup* denominato `rmiregistry`. In entrambi i casi il riferimento all'oggetto remoto contiene lo stub del client ovvero implementa l'interfaccia remota, istanzia il `RemoteStub`. Una volta acquisito il riferimento remoto esso è indistinguibile da uno locale.

RMI non maschera completamente la distribuzione, notiamo la presenza di eccezioni ed interfacce remote, questo è il risultato di una precisa scelta progettuale per distinguere un'interazione locale da una remota. RMI inoltre è uno strumento poco potente rispetto agli altri sistemi ad oggetti distribuiti, tuttavia altri componenti Java sono costruiti su di esso come JavaEE.



**ORB Interface:** è l'interfaccia standard per accedere ad un servizio del core, questo disaccoppia client e server da una specifica implementazione dell'ORB.

**IDL Stub e Skeleton:** costruito dalle interfacce degli oggetti remoti tramite il compilatore IDL, insieme all'ORB permette il dispacciamento delle richieste al giusto oggetto remoto.

**Dynamic Invokation Interface:** (DII) interfaccia standard utilizzata dal client per accedere ai servizi forniti da un oggetto remoto la cui interfaccia non era conosciuta a tempo di compilazione.

**Dynamic Skeleton Interface:** (DSI) interfaccia standard utilizzata per implementare un oggetto remoto.

**Object Adapter:** è il componente che gestisce la comunicazione tra l'oggetto remoto e l'ORB, esso incorpora i meccanismi e le policies principali per implementare le seguenti operazioni:

- Registrare, attivare e disattivare i servant
- Creare e interpretare i riferimenti agli oggetti
- mediare l'invocazione dei servizi

**GIOP e IIOP:** sono i protocolli utilizzati per connettere il client e il server sono standardizzati dal OMG.

Il *Portable Object Adapter* (POA) come abbiamo visto, è un componente situato tra l'ORB e i servant, il client effettua una richiesta utilizzando un riferimento all'oggetto remoto, la richiesta viene ricevuta dall'ORB che inoltra la richiesta al POA che ospita l'oggetto. Il POA spaccia la richiesta al servant di competenza il quale esegue l'operazione e restituisce il risultato al POA che a sua volta la inoltra all'ORB che la restituisce al client. Il POA deve soddisfare tre requisiti:

- creare riferimenti agli oggetti per permettere al client di raggiungere l'oggetto remoto.
- assicurarsi che ogni oggetto target sia riferito ad un servant.
- ricevere le richieste dal parte client dell'ORB e indirizzarle verso il servant adatto.

Oltre alla parte principale di CORBA esistono una serie di servizi che forniscono una serie di funzionalità a supporto dell'integrazione e dell'interoperabilità degli oggetti distribuiti. Questi oggetti sono chiamati *CORBA Service* o *Object Service* e sono definiti come standard in CORBA da interfacce specificate in IDL. I principali servizi sono il servizio di *Naming* e di *Trading Object* che permettono al server di esporre i suoi servant. I servizi *Event* e *Notification* supportano la comunicazione asincrona multipla, il servizio *Transaction* fornisce un supporto ai sistemi transazionali.

Il sistema CORBA, infine, fornisce delle *horizontal facilities* e delle *vertical facilities*, le prime si posizionano tra il middleware CORBA e i potenziali servizi utilizzati nel dominio di applicazione come una *printing facility* o una *mobile agent facility*. Le *Vertical Facilities* definiscono delle interfacce standard per quegli oggetti che ogni settore del dominio vuole condividere.

### 3.1 CORBA e Java

Java include di default un'implementazione base, ma completamente funzionante di CORBA, un IDL per il compilatore Java e un Naming Service.

Lo scopo dell'IDL è quello di permettere la definizione delle interfacce degli oggetti in un modo

indipendente dal linguaggio di programmazione utilizzato per l'implementazione. Per chiamare una funzione di un oggetto CORBA tutto quello che è necessario al client è l'IDL dell'oggetto. l'IDL supporta *multiple inheritance* e *genericity*, esso supporta parametri sia in input che in output sia bidirezionali, le operazioni possono sollevare delle *eccezioni* definite nell'IDL, infine, esso fornisce una serie di tipi di dato come `string`, `boolean`, `int`, `long`, `float` e `double` ma permette anche la definizione di tipi complessi tramite `struct`, `sequence`, `array`, `typedef`, `enum` e `union`. Un esempio di file IDL è quello del Listato 21

```
1 module Finance {
2     struct AccountDetails {
3         string name;
4         string address;
5         long number;
6         double balance;
7     };
8
9     exception InsufficientFunds {};
10
11     interface Account {
12         void deposit (in double amount);
13         void withdraw(in double amount) raises (InsufficientFunds);
14         readonly attribute AccountDetails details;
15     };
16 };
```

Listing 21: Esempio di file IDL

Per un'analisi approfondita sullo sviluppo di una applicazione con CORBA si rimanda a [2] e [5]

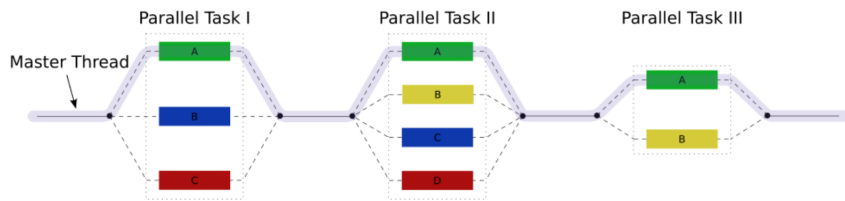


Figura 9: Suddivisione del carico tra diversi threads

## 4 Programmazione concorrente con OpenMP

OpenMP è una API multi-piattaforma per il calcolo parallelo a memoria condivisa, essa è disponibile per tutti i sistemi operativi e per i linguaggi C/C++ e Fortran. Questa API si basa su direttive del compilatore, routine di librerie e variabili d'ambiente.

Più in dettaglio OpenMP è un'implementazione del multi-threading, nel quale un processo *master* effettua una *fork* e genera un determinato numero di thread *slave* sul quale suddividere il lavoro. Questi thread slave vengono eseguiti concorrentemente su differenti core o processori come mostrato in 9.

La maggior parte dei costrutti di OpenMP sono direttive del compilatore o **pragmas**, lo scopo principale di questa API è quello di parallelizzare i cicli e per farlo offre un approccio incrementale al parallelismo. Alcune peculiarità sono:

- **Parallelismo innestato:** le API permettono di costruire del thread all'interno di altri thread.
- **Dynamic thread:** le API permettono di cambiare dinamicamente il numero di thread in esecuzione nelle differenti aree parallelizzate.
- **Input/Output: OpenMP** non specifica nulla riguardo agli I/O paralleli è compito del programmatore perciò assicurare la corretta esecuzione parallela.
- **Memory Consistency:** i threads mantengono una loro *cache* e non è quindi necessario mantenere una consistenza con la memoria principale, tuttavia in caso di una variabile condivisa è responsabilità del programmatore assicurare il corretto uso della variabile.

Vediamo ora nel Listato 22 un esempio di utilizzo delle API OpenMP

```

1  #include <omp.h>
2  #include <iostream>
3
4  using namespace std;
5  int main () {
6      #pragma omp parallel num_threads(3)
7      {
8          cout<<"Hello_World\n"
9      }
10 }
```

Listing 22: Esempio di utilizzo delle OpenMP

Per compilare tale codice è necessario aggiungere l'opzione **-fopenmp** al normale comando di compilazione.

Le direttive OpenMP valide sono identificate dal costrutto

`#pragma omp name [clause, ...]`

dove *name* identifica la direttiva la quale può essere seguita da una o più opzioni. Una delle direttive più importanti è la direttiva **parallel** senza la quale un programma verrebbe eseguito sequenzialmente. Quando un thread raggiunge la direttiva **parallel** esso crea una team di threads ed esso ne diventa il master. A partire dall'inizio della regione parallela il codice viene duplicato e tutti i thread eseguono lo stesso codice. Esiste un *barrier* implicito alla fine della regione parallela dopo il quale solo il thread master continua l'esecuzione. Se un thread termina all'interno di una regione parallela tutti i thread del team terminano ed l'esecuzione da quel punto è indefinita.

Il numero di threads da generare in una regione parallela può essere determinati da questi fattori in ordine di precedenza:

- Valutazione della clausola **if**
- Valore settato dalla clausola **num\_threads**
- L'uso della funzione di libreria **omp\_set\_num\_threads()**
- Il settaggio della variabile d'ambiente **OMP\_NUM\_THREADS**
- Implementazione e numero di CPU

La clausola **if** se valutata vera crea un team di thread altrimenti la regione viene eseguita sequenzialmente.

Lo standard OpenMP definisce un insieme di funzioni per determinare il numero di threads in esecuzione, bloccare l'esecuzione per scopi di sincronizzazione (semafori), definire delle routine. Per controllare il numero di threads da creare, quelli creati e il numero di thread che stiamo eseguendo abbiamo a disposizione delle funzioni di libreria specifiche:

- **void omp\_set\_num\_threads (int num\_threads)** che imposta il numero di thread da utilizzare nelle regioni parallele, questa routine deve essere chiamata in un punto seriale del codice.
- **int omp\_get\_num\_threads (void)** restituisce il numero di threads che attualmente sono nel team e stanno eseguendo la regione parallela.
- **int omp\_get\_thread\_number (void)** restituisce il numero del thread all'interno del team nel quale è stato chiamato, in caso in cui il chiamante sia il thread *master* questa chiamata restituisce il valore 0.

OpenMP si basa sulla un modello a memoria condivisa e molte variabile sono condivise di default, esiste tuttavia la *Data Scope Attribute Clauses* che viene utilizzata per definire esplicitamente qual è lo *scope* di una variabile i possibili valori che si possono assegnare sono:

- **private/firstprivate**
- **shared**
- **default**
- **reduction**

La clausola **private** è privata in ogni thread questo significa che viene dichiarato un nuovo oggetto dello stesso tipo in ogni thread, ogni riferimento all'oggetto originale viene rimpiazzato dai riferimenti al nuovo oggetto, tuttavia si assume che tale oggetto non sia inizializzato. La clausola **firstprivate**, è simile alla clausola **private** ma, al contrario di questa, il valore del nuovo oggetto viene inizializzato al valore globale assunto dalla variabile prima di entrare nel blocco parallelo. La direttiva **shared** dichiara che le variabili nella lista sono condivise tra i diversi threads del team, una variabile dichiarata *shared* esiste solamente in una posizione di memoria e tutti i threads leggono e scrivono su di essa; è compito del programmatore assicurare l'integrità della variabile.

La clausola **default** permette al programmatore di definire lo *scope* di tutte le variabili delle regioni parallele, i valori possibili sono *shared* e *none*, la seconda obbliga il programmatore a definire una clausola per tutte le variabili. La direttiva **reduction** effettua una riduzione sulle variabili che compaiono nella lista, la sintassi prevede anche l'utilizzo di un operatore di riduzione. Una copia privata di ogni variabile viene creata in ogni thread ed alla fine dell'esecuzione parallela la riduzione viene applicata a tutte le copie private e il risultato viene scritto nella variabile globale.

La clausola **critical** specifica una regione di codice che deve essere eseguita da un thread alla volta, nel caso in cui un thread sia già all'interno di una regione critica gli altri threads devono attendere che il thread esca dalla regione, è possibile definire più di una regione critica tramite l'utilizzo di nomi per distinguere le diverse regioni.

Infine, la direttiva **barrier** sincronizza tutti i threads del team, quando un thread raggiunge questa direttiva attende fino a quando tutti i threads non sono arrivati alla *barriera* a questo punto i threads riprendono la loro esecuzione.

Per un approfondimento sulle clausole e su OpenMP si rimanda a [3] e [4] mentre per un tutorial sulla programmazione si rimanda a [1].

## Riferimenti bibliografici

- [1] Blaise Barney. *OpenMP*. 2014. URL: <https://computing.llnl.gov/tutorials/openMP/>.
- [2] Gianpaolo Cugola. *CORBA*. 2013. URL: <http://corsi.dei.polimi.it/distsys/2012-2013/pub/14-corba.pdf>.
- [3] Gianpaolo Cugola. *OpenMP*. 2013. URL: <http://corsi.dei.polimi.it/distsys/pub/18-openmp.pdf>.
- [4] openmp.org, cur. *The OpenMP API*. 2014. URL: <http://openmp.org/openmp-faq.html#OMPAPI>.
- [5] Oracle, cur. *CORBA*. 2013. URL: <http://docs.oracle.com/javase/7/docs/technotes/guides/idl/GShome.html>.