UNIVERSITÀ DI PISA

Appunti di Reti di Calcolatori
Laboratorio A
A.A. 2020/2021

Lezioni prof.ssa L. M. E. Ricci - A. Michienzi, PhD
December 10

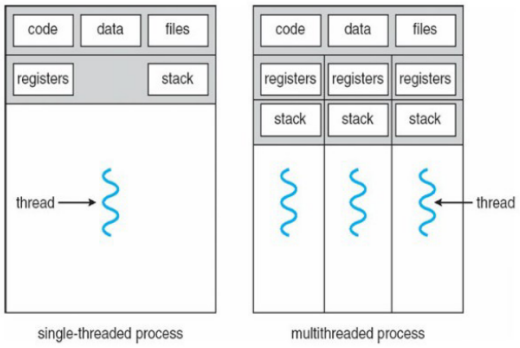
A cura di L. G. Pinta

Tali appunti sono stati pubblicati solo a fini orientativi e non rappresentano in alcun modo materiale sostitutivo al materiale consigliato o alla frequenzatione del corso e delle relative esercitazioni. Si consigliano comunque solide conoscenze di *Sistemi Operativi e Programmazione di Sistema, Architettura degli Elaboratori e Programmazione II.*

Contents

1 Introduzione ai threads

Un **thread** o *light weight process* è un *flusso di esecuione* all'interno di un processo. Viene definito *light weight process* poiché consente una **commutazione di contesto** (***context switch***) meno onerosa rispetto ad un processo.



Con il termine **multitasking** ci riferiamo a *processi* o *thread*:

- a livello di processo è controllato esclusivamente dal SO
- a livello thread è controllato, almeno in parte, dal programmatore

I vantaggi principali di thread multitasking rispetto a process multitasking risiede nel fatto che i thread condividono lo stesso spazio degli indirizzi e generalmente sono meno costosi in termini di context switch e comunicazioni tra threads.

A runtime i thread comportano diversi vantaggi in relazione al tipo di architettura:

- **Single core**: multiplexing, interleaving (tramite meccanismi di time sharing delle risorse)
- **Multicore**: più flussi di esecuzione eseguiti in parallelo consentono simultaneità di esecuzione

1.1 Multithreading

La gestione di funzionalità che richiedono **esecuzione simultanea** è realizzabile tramite una *decomposizione* del programma in thread. Ciò implica una **modularizzazione** della struttura dell'applicazione con conseguente aumento della *responsiveness*. Ciò consente di catturare la struttura dell'applicazione semplificandola in componenti interagenti e ogni componente viene gestita da thread distinti.

L'utilizzo del multithrading tuttavia comporta la gestione di alcune peculiarità:

- difficoltà nel debugging e nella manutenzione rispetto ad un software single-threaded
- *race conditions e sincronizzazioni*
- *deadlock, livelock, starvation*

1.2 Creazione e attivazione di thread

Quando si manda in esecuzione un programma Java la JVM crea un thread che invoca il metodo main del programma. Dunque esiste sempre almeno un thread per ogni programma indicato generalmente come *main thread*.

Successivamente altri thread sono attivati automaticamente da Java quali gestore eventi, interfaccia, garbage collector, etc.

La creazione esplicità di un thread segue due metodologie principali:

- **Metodo 1**: implements Runnable
- **Metodo 2**: extends Threads

1.3 Metodo 1: implements Runnable

Si implementa tramite i seguenti passi:

1. Definire un task (codice che il thread deve eseguire) tramite la definizione di una classe C che implementi l'interfaccia Runnable.
2. Creare un'istanza R di tale classe C
3. Creare un oggetto thread passandogli l'istanza R

L'interfaccia *runnable* appartiene al package ***java.language*** e contiene solo la segnatura del metodo *void run()*. Il task che il thread andrà ad eseguire andrà implementato all'interno di tale metodo *void run()*.

Un istanza della classe che implementa Runnable è un task ovvero un frammento di codice che può essere eseguito in un thread. La creazione dei task non implica la creazione di un thread che lo esegua. Lo stesso task può essere eseguito da più thread.

```
// Task.java

public class Task implements Runnable {

    public Task(){}

    public void run () {
        System.out.println("I'm the thread " +
            Thread.currentThread().getName());
    }
}
```

Poiché più thread possono eseguire lo stesso codice (*task*) è possibile identificare il thread che sta eseguendo una determinata istanza del task tramite il metodo *public static native Thread.currentThread()*.

```
// Main.java

public class Main {

    public static void main(String[] args){

        Task t = new Task(); // Istanziazione task
        Thread thread1 = new Thread(t); //Istanziazione thread
        thread.start(); //Avvio thread

    }

}
```

Se erroneamente, piùttosto che invocare *thread.start()* avessimo invocato thread.run() non avremmo avviato alcun thread poiché ogni metodo *run()* viene eseguito all'**interno del flusso del thread attivato** per l'esecuzione del programma principale.
L'invocazione del metodo run() rende il flusso di esecuzione sequenziale (*il controllo dal main passa al metodo run() e tornerà unicamente al main al termine dell'esecuzione del metodo run()*). Solo il metodo *start()* comporta l'avvio di un nuovo thread.

L'invocazione del metodo *start()* provoca l'esecuzione del metodo *run()*, che, a sua volta, provoca l'esecuzione del metodo *run()* di Runnable. Il metodo *start()*:

- segnala allo schedulatore (*tramite la JVM*) che il thread può essere attivato (*invoca un metodo nativo*)
- l'ambiente del thread viene inizializzato
- restituisce il controllo al chiamante senza attendere che il thread attivato inizi la sua esecuzione

1.4 Metodo 2: extends Thread

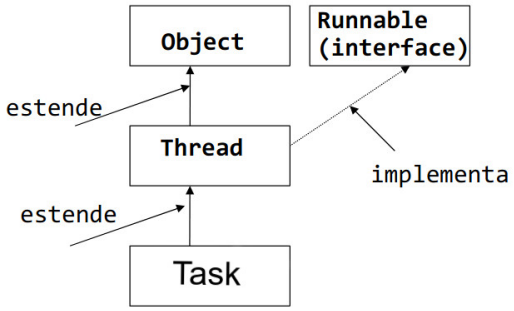
Si implementa tramite i seguenti passi:

- creare una classe C che estenda Thread
- effettuare l'overriding del metodo run() definito di quella classe
- istanziare un oggetto di quella classe: questo oggetto è un thread il cui comportamento è quello definito nel metodo run ridefinito

- invocare il metodo start() sull'oggetto istanziato

Ricordiamo che l'overriding consiste nel definire e implementare un metodo della sottoclasse con stesso nome e segnatura del metodo della superclasse.

```
// Task.java
public class Task extends Thread {

    public void run(){
    //Task here
        System.out.println("I'm thread " +
            Thread.currentTHread().getName());

    }

}

// Main.java
public class Main {

    public static void main(String[] args){

    Task t = new Task();
    t.start();


    }

}
```



Viene effettuato l'overriding del metodo run() all'interno della classe Task che estende Thread. Il comportamento del thread è definito dal metodo run() di Task.

1.5 Vantaggi e svantaggi dei due metodi

In Java una classe può estendere una sola altra classe secondo l'ereditarità singola: se si estende la classe Thread, la classe i cui oggetti devono essere eseguiti come thread non può estendere altre classi.
Ciò rappresenta una limitazione all'utilizzo del metodo 2: impedirebbe la gestione di eventi di interfaccia poiché la classe che getisce un evento deve estendere una classe C predefinita di Java dunque se il gestore deve essere eseguito in un thread separato, occorrerebbe definire una classe che estenda sia C che Thread, ma non è permesso poiché Java adotta l'ereditarietà singola.
Viene dunque generalmente utilizzato il metodo 1.

1.6 Thread demoni

Sono thread a **bassa priorità** adatti a **jobs non-critici** da eseguire in background. Sono generalmente utilizzati per servizi in background utili fino a che il programma è in esecuzione (*ex: garbage collector*) ma possono anche essere user-defined.
Non appena tutti i thread non demoni del programma sono terminati, la JVM termina il programma, forzando la terminazione dei thread demoni.Un esempio di thread daemon tramite l'implementazione del primo metodo è:

```
// JavaDaemonThread
public class JavaDaemonThread {

    public static void main(String[] args) throws InterruptedException {

    //primo metodo creazione thread
    //dunque la classe Task deve estendere Runnable

    Thread daemon = new Thread(new Task(),"daemon-thread");
    daemon.setDaemon(true);
    daemon.start();

    }

}
```

E' possibile assegnare ad un thread un nome specifico indicandolo come parametro nel costruttore. In questo caso il nome del thread è "*daemon-thread*".
Un programma Java termina quando terminano tutti i thread non demoni. Se il thread main (che esegue il metodo main()) termina, i restanti thread ancora attivi e non demoni continuano la loro esecuzione fino alla terminazione. Se un thread usa l'istruzione **System.exit()** per terminare, allora tutti i threads terminano la loro esecuzione.

1.7 Gestione delle interruzioni

Java mette a disposizione un meccanismo per interrompere un thread e diversi meccanismi per intercettare l'interruzione in relazione a:

- lo stato del thread (running, blocked)
- se il thread è sospeso l'interruzione solleva una InterruptedException
- se in esecuzione, può testare un flag che segnala se è stata inviata una interruzione

In generale è il thread che decide in autonomia come rispondere alla interruzione.

1.8 Thread interrupt

Il metodo ***interrupt()*** viene utilizzato per lanciare un eccezione ad un determinato thread:

- imposta a true un valore booleano nel descrittore del thread
- flag vale true se vi sono interrupts pendenti

Tale flag è utilizzabile dal thread tramite due metodi:

- *public static boolean Interrupted():* metodo statico che si invoca con il nome della classe *Thread.Interrupted*. Se il flag è true, lo resetta a false.
- *public boolean isInterrupted()*: deve essere invocato su un'istanza di oggetto di tipo thread (*daemon.isInterrupted()*). Non cambia il valore del flag se già true.

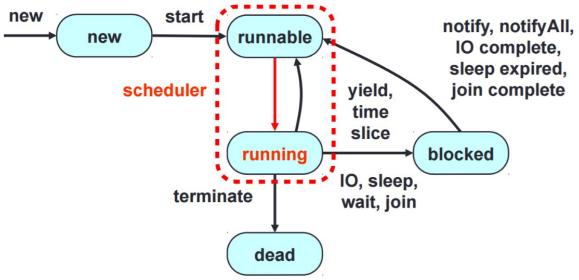
Entrambi i metodi restitusiscono un valore booleano che segnala se il thread ha ricevuto un'interruzione.

La classe thread fornisce una serie di metodi per l'interruzione, la sospensione e l'attesa della terminazione di un thread. Tuttavia **non contiene** metodi per la sincronizzazione tra thread (vedi java.lang.object). Inoltre è dotata di costruttori diversi differenti per i parametri utilizzati (nome thread, gruppo thread, etc).

Un thread viene posto nello stato i blocked tramite l'invocazione del metodo void sleep(long M) che sospense l'esecuzione del thread per M millisecondi. Durante la sleep il thread può essere interrotto. Tale metodo, essendo statico, non può essere invocato su una istanza di un thread.

Vi sono inoltre una serie di metodi get() e set() che consentono di reperire le caratteristiche di un thread. La classe Thread salva alcune informazioni:

- **ID**: identificatore thread
- **nome**: nome thread
- **priorità**: valore da 1 a 10 (1 priorità bassa)



- **stato**: uno dei possibili stati (new, runnable, blocked, waiting, time waiting o terminated).

Nel diagramma degli stati si evidenzia come, a partire dall'invocazione di specifici metodi Java, viene istanziato e messo in esecuzione. D'interesse è la transizione dallo stato running allo stato blocked: viene eseguita in caso di operazioni di IO (come scrittura su socket), sleep, wait (permette sospensione condizionata del thread in relazione ad un evento. Condizioni inverse per la transizione da blocked a runnable (notify (simile ad una signal), notifyAll (simil broadcast), IO complete, sleep expired, join complete). Si passa da running a runnable nel caso al termine del time slice.

1.9 Thread State Monitoring

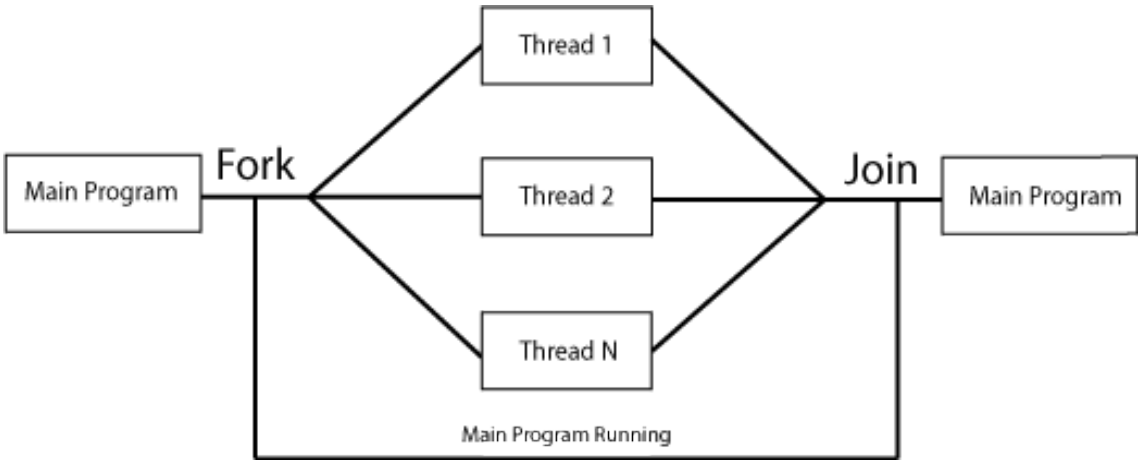
Si riporta un esempio di programma Java in grado di monitorare lo stato di più thread nel caso in cui questo vari durante la sua esecuzione, memorizzando nome, priorità, stato precedente, nuovo stato in un file "log.txt". Il monitor termina quanto tutti i thread sono terminati.

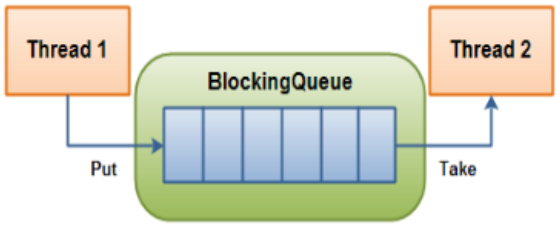
```
public static void main(String[] args) throws Exception
{
    Thread threads[]=new Thread[10];
    Thread.State status[]=new Thread.State[10];
    for (int i=0; i<10; i++){
        threads[i]=new Thread(new Calculator(i));
        if ((i%2)==0){
            threads[i].setPriority(Thread.MAX_PRIORITY);
        }else {
            threads[i].setPriority(Thread.MIN_PRIORITY);
        }
        threads[i].setName("Thread "+i);
    }


    FileWriter file = new FileWriter("log.txt");
```

```
PrintWriter pw = new PrintWriter(file);
pw.printf("**********************************\n");
for (int i=0; i<10; i++){
       pw.println("Status of Thread"+i+":"+threads[i].getState());
       status[i]=threads[i].getState();
    }
for (int i=0; i<10; i++){
       threads[i].start();
}

boolean finish=false;
while (!finish) {
   for (int i=0; i<10; i++){
       if (threads[i].getState()!=status[i]) {
           pw.printf("Id %d -
               %s\n",threads[i].getId(),threads[i].getName());
           pw.printf("Priority: %d\n",threads[i].getPriority());
           pw.printf("Old State: %s\n",status[i]);
           pw.printf("New State: %s\n",threads[i].getState());
           pw.printf("**********************************\n");
           pw.flush();
           status[i]=threads[i].getState();
           }
       }
finish=true;
for (int i=0; i<10; i++){
       finish=finish && (threads[i].getState()==
            Thread.State.TERMINATED);
       }
    }
  }
}
```

1.10 Thread Join

La classe Thread fornisce un metodo **join()** che invocato sull'istanza di un thread t determina la sospensione del chiamante che rimane in attesa della terminazione del thread t.
E' possibile specificare un *timeout di attesa*: finita l'attesa, se il thread non è ancora terminato si riprenderà l'esecuzione a partire dalle istruzione successive a **t.join(maxTime)**. Se il thread sospeso sulla join() riceve un'interruzione viene sollevata una eccezione dunque si utilizza in un **blocco try-catch**.





1.11 Blocking Queue: interazione tra threads

Java definisce ***BlockingQueue*** (java.util.concurrent package) che consente la definizione di una **coda sincronizzata thread-safe** per quanto riguarda inserimenti e rimozioni. Il produttore può inserire elementi nella coda fino a che la dimensione della coda non raggiunge un limite, dopo di che si blocca e rimane bloccato fino a che un consumatore non rimuove un elemento. Il consumatore può rimuovere elementi della coda, ma se tenta di eliminare un elemento dalla coda si blocca fino a che il produttore inserisce un elemento nella coda.
Sono definiti 4 metodi per operare sulla BlockingQueue:

	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

BlockingQueue è un'interfaccia, alcune implementazioni:

- **ArrayBlockingQueue**: coda di dimensione limitata. Memorizza gli ele-

menti in un array il cui upper bound è definito a tempo di inizializzazione

- **LinkedBlockingQueue**: mantiene gli elementi in una struttura linkata che può avere un upper bound che se non specificato è la costante Java intera MAX VALUE.
- **SynchronousQueue**: non possiede capacità interna e le operazioni di inserzione devono attendere per una corrispondente operazione di rimozione e viceversa (fuorviante chiamarla coda).

```
// BlockingQueueExample
import java.util.concurrent.*;
    public class BlockingQueueExample {
        public static void main(String[] args) throws Exception {
            BlockingQueue queue = new ArrayBlockingQueue(1024);

            Producer producer = new Producer(queue);
            Consumer consumer = new Consumer(queue);

            new Thread(producer).start();
            new Thread(consumer).start();
            Thread.sleep(4000);
    }
}
```

```
// Producer
import java.util.concurrent.*;
    public class Producer extends Thread{
        protected BlockingQueue queue = null;

            public Producer(BlockingQueue queue) {
                this.queue = queue;
                }

        public void run() {
                try {
                    queue.put("1");
                    Thread.sleep(1000);
                    queue.put("2");
                    Thread.sleep(1000);
                    queue.put("3");
            } catch (InterruptedException e) {
        e.printStackTrace();
        }
}
```

```
// Consumer
```

```
import java.util.concurrent.*;
    public class Consumer extends Thread {
        protected BlockingQueue queue = null;

        public Consumer(BlockingQueue queue) {
            this.queue = queue;
            }

    public void run() {
        try {
                System.out.println(queue.take()); //la take potrebbe
                    sospendere il thread
                System.out.println(queue.take()); //dunque si fa uso del
                    try-catch
                System.out.println(queue.take());
        } catch (InterruptedException e) {
                    e.printStackTrace();
            }
    }
```

2 Thread Pooling

Istanziare un thread per ogni task comporta alcuni svantaggi:

- **Thread life cycle overhead**: la creazione/distruzione dei thread richiede un'interazione tra JVM e SO. Il ciclo di vita varia a seconda della piattaforma dunque non è mai trascurabile. In caso di richieste frequenti e 'lightweight' ciò può impattare negativamente sulle prestazioni dell'applicazione poiché costa più la gestione del thread che l'esecuzione del task.
- **Resource consumption**: molti threads idle quando il numero supera il numero di processori disponibili, comportando un alta occupazione di risorse (memoria). Utilizzo elevato del garbage collector e dello scheduler.
- **Stability**: limitazione al numero di threads imposto dalla JVM o dal SO.

L'idea alla base del thread pool consiste nel porre un **limite** oltre il quale non risulta conveniente creare ulteriori threads in modo da:

- sfruttare al meglio i processori disponibili
- evitare una contesa delle risorse disponibili da troppi threads
- diminuire il costo per l'attivazione/terminazione threads

L'utente struttura l'applicazione mediante un insieme di task: il thread pool è una **struttura dati** la cui dimensione massima può essere prefissata *(che contiene riferimenti ad un insieme di threads)*. I thread pool possono essere riutilizzati per l'esecuzione di più **task diversi**: la sottomissione di un task al

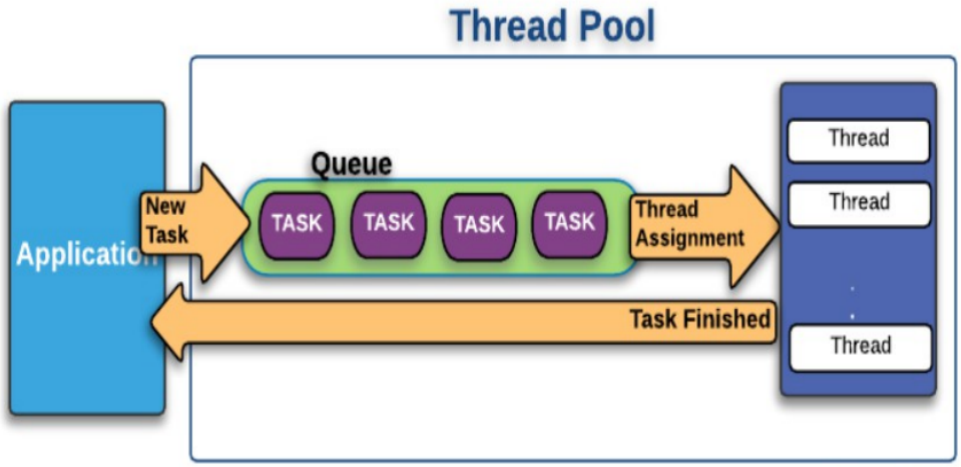


Figure 1: Schema generale

pool viene **disaccopiata** dall'esecuzione del thread infatti l'esecuzione del task può essere ritardata se non vi sono risorse attualmente disponibili.

2.1 Concetti generali e implementazione

L'utente, al momento della sottomissione:

- crea un pool e stabilisce una politica di gestione del thread pool
- l'attivazione di un thread può avvenire alla creazione del pool oppure on demand, all'arrivo di un nuovo task, etc.
- se è quando è opportuno può terminare l'esecuzione di un thread (in caso di un numero non sufficiente di tasks da eseguire)
- l'utente può sottomettere i task per l'esecuzione del thread pool

Il supporto, *al momento della sottomissione del task*, può:

- utilizzare un thread attivato in precedenza, inattivo in quel momento
- creare un nuovo thread
- memorizzare il task in una struttura dati (coda) in attesa dell'esecuzione
- respingere la richiesta di esecuzione del task

Il numero di threads attivi nel pool può variare dinamicamente.
Alcune interfacce definisco servizi generici di esecuzione: la classe **Executor** che opera come una **Factory** in grado di generare oggetti di tipo ExecutorService con comportamenti predefiniti. I tasks devono essere incapsulati in oggetti di tipo Runnable e passati a questi esecutori, mediante invocazione del metodo **execute()**.

```
//Main

public class Main {
   public static void main(String[] args) throws Exception
        {
         Server server=new Server();
         for (int i=0; i<10; i++){
            Task task=new Task("Task "+i); //Sottometto 10 task
            server.executeTask(task);
           }
         server.endServer();
        }
}


//Server

import java.util.concurrent.*;
    public class Server {
       private ThreadPoolExecutor executor;
        public Server( ){
            executor=(ThreadPoolExecutor)
                 Executors.newCachedThreadPool(); //Creazione thread pool
                 con una politica specifica
        }

        public void executeTask(Task task){
           System.out.printf("Server: A new task has arrived\n");
           executor.execute(task); //Sottomissione task
           System.out.printf("Server:Pool
                Size:%d\n",executor.getPoolSize());
           System.out.printf("Server:Active
                Count:%d\n",executor.getActiveCount());
           System.out.printf("Server:Completed Tasks:%d\n",
           executor.getCompletedTaskCount());

        public void endServer() {
                    executor.shutdown(); //Terminazione server

        }
    }
```

La primitiva **NewCachedThreadPool** crea un pool con un comportamento predefinito: *alla sottomissione di un nuovo task, se tutti i thread del pool sono occupati nell'esecuione di altri task allora ne crea uno nuovo.*
Se disponibile un thread, viene riutilizzato per il nuovo task tuttavia se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina.
Ciò implementa l'**elastiticità**: un pool che può espandersi all'infinito ma si contrae quando la domanda di esecuzione di task diminuisce. E' possibile aumentare il riuso distanziando appropriatamente la sottomissione di task successivi in modo da riutilizzare lo stesso thread (ex: ogni 5 secondi una nuova sottomissione di task).
La primitiva **newFixedThreadPool(int N)** crea un pool in cui vengono creati N thread, al momento della inizializzazione del pool, riutilizzati per l'esecuzione di più task.
Alla sottomissione di un task T, se tutti i thread sono occupati nell'esecuzione di altri tasks, T viene inserito in una coda, gestita automaticamente dall'ExecutorService. Tale coda è illimitata e se almeno un thread è inattivo verrà utilizzato quel thread per l'esecuzione del task T.

2.2 Thread Pool Executor

E' definito con 5 parametri in ingresso: i primi 4 controllano la gestione del thread pool, l'ultimo è uno struttura dati per la memorizzazione di eventuali task in attesa di esecuzione.

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements ExecutorService{

    public ThreadPoolExecutor(int CorePoolSize,
                              int MaximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue <Runnable> workqueue);

    }
```

- **CarePoolSize**: dimensione minima del pool, definisce il core del pool.
- **MaxPoolSize**: indica la dimensione massima del pool. Non si avranno mai più di MaxPoolSize thread nel pool anche se vi sono task da eseguire e tutti i threads sono occupati nell'elaborazione di altri task.

Generalmente i thread del core possono venire creati secondo varie modalità:

- **PreStartAllCoreThreads()**: al momento della creazione del pool
- **on demand**: alla sottomissione di un task si crea un nuovo thread anche se qualche thread già creato dal core è inattivo (obiettivo:riempire il pool il prima possibile).

- quando sono stati creati tutti gli n thread del core, questi saranno tutti ed n sempre attivi

Se tutti i thread del sono sono stati già creati e viene sottomesso un nuovo task:

- se un thread del core è inattivo, il task gli viene assegnato altrimenti se la coda passata ultimo parametro del costruttore non è piena, il task viene inserito nella coda: i task verranno poi prelevati dalla coda ed inviati ai thread disponibili
- se coda piena e tutti i thread del core stanno eseguendo un task allora si crea un nuovo thread attivando così k thread t.c. corePoolSize <= k <= MaxPoolSize
- se coda piena e sono attivi MaxPoolSize thread allora il task viene respinto

E' possibile scegliere diversi tipi di coda (*tipi derivati da BlockingQueue*). Il tipo di coda scelto influisce sullo scheduling.

Supponendo che un thread termini dopo l'esecuzione di un task e che il pool contenga k threads:

- se **k $\leq$ core**: il thread si mette in attesa di nuovi task da eseguire (attesa è indefinita)
- se **k $>$ core ed il thread non appartiene al core**: si considera il timeout T definito al momento alla costruzione del thread pool. Se nessun viene sottomesso entro T, il thred termina la sua esecuzione riducendo il numero di threads del pool. Per definire un timeout occorre specificare un valore e l'unità di misura utilizzata (*ex. TimeUnit.MILLISECONDS*).

Tra i diversi tipi di coda vi sono:

- **SynchronousQueue**: dimensione uguale a 0. Ogni nuovo task T viene eseguito immediatamente o respinto. Alternativamente viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (*numero di threads $\leq$ MaxPoolSize*).
- **LinkedBlockingQueue**: dimensione limitata. E' sempre possibile accodare un nuovo task, nel caso in cui tutti i threads siano attivi nell'esecuzione di altri task. Limite imposto è che la dimensione del pool non può superare core.
- **ArrayBlockingQueue**: Dimensione limitata stabilita dal programmatore.

Il comportamento del newFixedThreadPool o del newCachedThreadPool si può ottenere come istanza del metodo più generale **ThreadPoolExecutor**:

```
//newFixedThreadPool
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads, 0L,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());
   }
```

Il ***newFixedThreadPool*** avendo n thread sempre attivi allora i primi due parametri saranno uguali (***dimensione core = dimensione massima)***, con 0L non ucciderà mai i thread. Infine fa uso di una *LinkedBLockingQueue<Runnable>* poiché è sempre possibile accodare nuovi task.

```
//newCachedThreadPool
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L,
        TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
   }
```

Il ***newCachedThreadPool*** lo ottengo similmente ponendo a 0 i thread del pool poiché vogliamo che al termine di un task un thread muoia.Con MAX VALUE non poniamo alcun limite al numero di thread del pool e inoltre dopo 60L (*60 secondi*) viene ucciso per convenzione. Infine si fa uso di una *SynchronousQueue<Runnable>* poiché i task non vanno in coda ma bensì ad ogni nuovo task avvio un thread per quello specifico task, evitando accodamento.

2.3 Executor Lifecycle

La JVM termina la sua esecuzione quando tutti i thread (**non demoni**) terminano la loro esecuzione. E' tuttavia necessario analizzare il concetto di **terminazione**, nel caso si utilizzi un *Executor Service* poiché i task vengono eseguiti in modo **asincrono** rispetto alla loro sottomissione dunque in un certo istante, alcuni task sottomessi precedentemente possono essere completati, altri in esecuzione, altri in coda. Un thread pool può rimanere attivo anche quando ha terminato l'esecuzione di un task.
Poiché alcuni threads possono essere sempre attivi, Java mette a disposizione dell'utente metodi che consentono di terminare l'esecuzione del pool. La terminazione può avvenire:

- **shutdown() - *modo graduale***: nessun task viene accettato dopo che la shutdown è stata invocata e tutti i task sottomessi precedentemente e non ancora terminati vengono eseguiti, compresi quelli in coda. Successivamente tutti i threads del pool terminano la loro esecuzione.

- **shutdownNow() - *modo istantaneo***: non accetta ulteriori task ed elimina tasks non ancora iniziati (*dalla coda*). Restituisce una lista dei task eliminati dalla coda e tenta di terminare l'esecuzione dei thread che stanno eseguendo i task.

Nello specifico, ***shutdownNow()*** non garantisce la terminazione immediata dei threads del pool: invia una **interruzione** al thread in esecuzione nel pool ma se un thread non risponde all'interruzione non termina.
Il ciclo di vita di un execution service è dunque **running, shutting down** e infine **terminated**. Un pool viene creato in running, quando viene invocata una shutdown() o shutdownNow() passa allo stato shutting down e quando tutti i thread sono terminati passa nello stato terminated.
I thread sottomessi per l'esecuzione ad un pool in stato di shuttind down o terminated possono essere gestiti da un **rejected execution handler** che può semplicemente scartarli, sollevare una eccezione o adottare politiche più complesse e mirate alla gestione.

2.4 Callable e Future

Un oggetto di tipo **Runnable** incapsula un'attività che viene eseguita in *modo asincrono*. La Runnable è un metodo asincrono, senza parametri e che non restituisce un valore di ritorno.
Per definire un task che restituisca un valore di ritorno si deve ricorrere all'**interfaccia Callable** che può restituire un risultato e sollevare eccezioni.
Ricordiamo che per le Callable non è consentito il ritorno di un tipo primitivo (*int*) ma bensì tipi definiti come oggetti (*Integer*).
Immaginiamo di definire un thread che esegue un calcolo: il risultato vorrei gestirlo in modo asincrono in fasi successive alla sua computazione dunque definisco quindi un thread Callable (*il task*) ed il risultato sarà memorizzato in un oggetto di tipo ***Future***. La classe *FutureTask* fornisce una implementazione dell'interfaccia Future.

```
public interface Callable <V>{
      V call() throws Exception;
   }
```

L'interfaccia Callable contiene solo **call()** *(analogo al metodo run() di Runnable).* Il codice del task è implementato nel metodo call() e può restituire un valore e sollevare eccezioni. Il parametri di tipo $<\mathbf{V}>$ indica il tipo del valore restituito.

```
import java.util.concurrent.*;

public class Task implements Callable<Integer>{

   private Integer age;
   public Task(Integer age){
       this.age = age;
   }

   public Integer call(){

       Integer year_of_birthday = <calculate year>;
```

```
        return year_of_birthday;
    }

}
```

Il risultato restituito sarà nella forma:

```
//....
ExecutorService pool = Executors.newCachedThreadPool();
//....
Task t1 = new Task(10); //Istanzio nuovo task
Future<Integer> year = pool.submit(t1); //Sottometto al pool
```

Dunque il valore restituito dalla Callable, acceduto mediante un oggetto di tipo <Future>, rappresenta il risultato della computazione. Se si usano i threads pools si sottomette direttamente l'oggetto di tipo Callable al pool tramite il metodo **submit()** e la sottomissione restituisce un oggetto di tipo <**Future**>.

3 Sincronizzazione e risorse condivise

Scrivere programmi con i threads che si sincronizzano tra loro è un'attività complessa e pericolosa in confronto all'uso di nuove tecnologie tuttavia le locks sono ancora utilizzate per scrivere programmi concorrenti e sono solitamente alla base di costrutti ad alto livello (anche in Java). Le locks sono di fatto una formalizzazione dei meccanismi hardware sottostanti i cui vantaggi sicuramente risiedono nell'assenza di vincoli sul loro utilizzo ma ne implicano anche le difficoltà di implementazione e manutenzione.

Uno scenario tipico di un programma concorrente è quando un insieme di thread condividono una risorsa e l'accesso non controllato a tale risorsa può provocare situazioni di errore ed incosistenze (*come le race conditions*). Definiamo sezione critica un blocco di codice che viene eseguito un thread per volta e all'interno del quale si effettua l'accesso ad una risorsa condivisa. I meccanismi Java di sincronizzazione per l'implementazione delle **sezioni critiche** sono:

- **interfaccia Lock** (e relative implementazioni)
- concetto di **monitor**

Caratteristiche tipiche delle **race conditions** sono il **non determinismo** dovuto ad una commutazione di contesto prima della terminazione di un certo metodo che modifica lo stato condiviso e può generare un *risultato incosistente*. Chiaramente l'incosistenza non si presenta necessariamente ad ogni esecuzione e dunque è un comportamento dipendente dal tempo (*interleaving, context switch, etc.*).

3.1 Classi thread-safe

L'esecuzione concorrente dei metodi definiti in una classe **thread safe** non provoca comportamenti anomali o race conditions. Per rendere una classe thread safe è necessario garantire che le istruzioni contenute all'interno dei metodi che utilizzano risorse condivise vengano eseguite in modo **atomico / indivisibile / in mutua esclusione**. Java offre diversi meccanismi per la sincronizzazione di threads:

- **meccanismi a basso livello**: lock e variabili condizione associate
- **meccanismi ad alto livello**: syncronized() oppure wait(), notify(), notifyAll() ed i monitors.

3.2 Sincronizzazione: Lock

In Java una **lock** è un oggetto che può trovarsi in due stati diversi **locked** o **unlocked**. Lo stato viene impostati dai relativi metodi *lock() e unlock()*. Un solo thread alla volta può impostare lo stato a locked cioè ottenere la lock mentre gli altri thread che tentano di acquisire una lock si bloccano. Se un thread tenta di acquisire una lock rimane bloccato fintanto che la lock è detenuta da un altro thread mentre se un thread rilascia una lock uno dei thread in attesa la acquisisce.

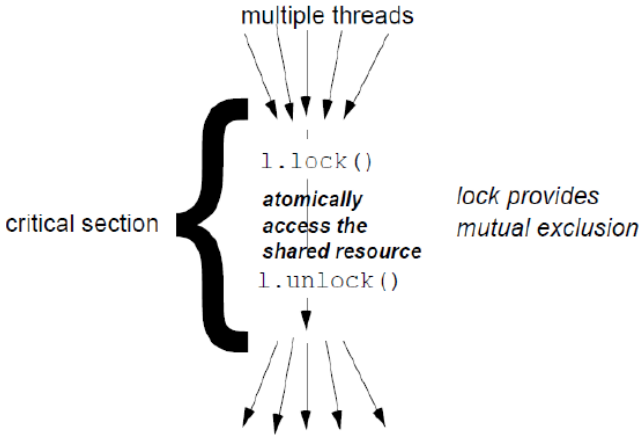


Figure 2: Schema generale

L'interfaccia in Java per le lock è *java.util.concurrent.locks.Lock* con relativa implementazione *java.util.concurrent.locks.ReentrantLock* (vedi API). Si riporta un esempio in Java in cui si fa uso della lock:

```
import java.util.concurrent.locks.*;
```

```
public class Account {
    private double balance;
    private final Lock accountLock = new ReentrantLock();

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public void addAmount(double amount) {

            accountLock.lock();
            double tmp=balance;

            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            tmp+=amount;
            balance=tmp;
            accountLock.unlock();
    }

    public void subtractAmount(double amount){

        accountLock.lock();
        double tmp=balance;
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
                    e.printStackTrace();
            }

        tmp-=amount;
        balance=tmp;
        accountLock.unlock();
    }
}
```

Si possono verificare delle situazioni di **deadlock** quando un thread A acquisice una lock X e il thread B acquisice una Y. Se rispettivamente A tenta di acquisire Y e B tenta di acquisire X entrambi i thread rimarranno bloccati all'infinito. L'interfaccia *Lock* e la classe *ReentrantLock* che implementa include un altro metodo utilizzato per ottenere il controllo della lock: il metodo **try-**

Lock() (*non bloccante*) tenta di acquisire la lock e se essa è già posseduta da un altro thread, il metodo termina immediatamente e restituise il controllo al chiamante. Restituisce invece un booleano con valore true se è riuscito ad acquisire la lock, false altrimenti.

Inevitabilmente l'uso delle lock introduce un overhead che causano una **performance penalty** dovuta a più fattori quali *contention, bookkeeping, scheduling, blocking o unblocking*. Tale penalità nelle performance sono caratteristiche di tutti i costrutti ad alto livello di Java basati su lock.

Le **reentrant locks** (*o recursive lock*) utilizzano un contantore che viene incrementato ogni volta che un thread acquisce la lock e decrementanto ogni volta che un thread rilascia la lock. Tale lock è definitivamente rilasciata quando il contatore diventa 0 e un thread può acquisire più volte la lock su uno stesso oggetto senza bloccarsi.

Un altro tipo di lock sono le **read/write locks**: la struttura dati condivisa può essere letta e acceduta da più thread tuttavia vi può essere solo uno scrittore (che modifica la struttura dati) o più lettori contemporaneamente (senza alcun thread scrittore).
Java implementa le **read/write locks** come *interfaccia ReadWriteLock* che mantiene una coppia di lock associate, una per le *operazioni di lettura* e una per le *scritture*: chiaramente la read lock può essere acquisita da più lettori purché non vi siano scrittori mentre la write lock è esclusiva. Vediamo un esempio di read/write lock:

```
import java.util.concurrent.locks.*;

public class SharedLocks extends Thread {

    int a =1000, b=0;
    private ReentrantReadWriteLock readWriteLock = new
        ReentrantReadWriteLock();
    private Lock read = readWriteLock.readLock();
    private Lock write = readWriteLock.writeLock();

    public int getsum (){
        int result;
        read.lock();
        result=a+b;
        read.unlock();
        return result;
    }

    public void transfer (int x){

        write.lock(); //Recinto di mutua esclusione
        a = a-x;
        b = b+x;
        write.unlock();
    }
```

```
}
```

3.3 Problema produttore-consumatore

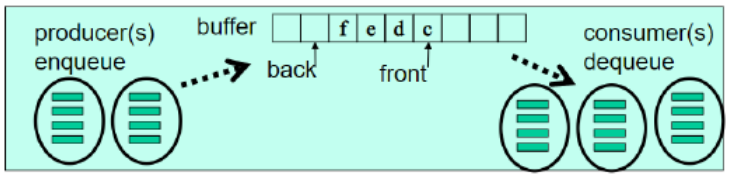


Figure 3: Produttore - Consumatore

E' un classico problema che descrive due o più thread che condivisono un buffer di dimensione fissata, usato come coda. Il **produttore P** produce un nuovo valore, lo inserisce nel buffer e torna a produrre valori. Il **consumatore C** consuma il valore (*rimuovendolo dal buffer*) e torna a richiedere valori. Necessita garantire che il produttore non provi ad aggiungere un dato nella coda se è piena ed il consumatore non provi a rimuovere un dato da una coda vuota.

Si evince che vi sono due tipi di sincronizzazione:

- **implicita**: la mutua esclusione sull'oggetto condivisa è garantita dall'uso di lock
- **esplicita**: occorrono meccanismi per attivare/riattivare threads al verificarsi di determinate condizioni

Un ipotesi non banale è l'uso di un buffer a **dimensione finita** (tramite un ArrayList o vettore di dimensione limitata) e anche se utilizzassimo una coda di dimensione illimitata sarebbe comunque necessaria la sincronizzazione per la coda vuota. L'idea di ottimizzazione utilizzata da Java è quella di *limitare la concorrenza a determinate parti di una struttura dati poiché è possibile che più thread accedano (in lettura o scrittura) a parti diverse della stessa struttura dati condivisa.*

3.4 Variabili condizione

Dunque i meccanismi forniti da Java devono consentire di definire un insieme di **condizioni sullo stato** dello'oggetto condiviso e consetire la **sospensione o riattivazione** dei threads sulla base del valore di queste condizioni.
Una prima soluzione è fornita dalla combinazione degli strumenti di basso livello visti finora: l'uso di variabili condizione e metodi per la sospensione su queste variabili in combinazione con la definizione di code associate alle variabili in cui memorizzare i threads sospesi consente l'implementazioni di tale meccanismo. Altre soluzioni verrano da meccanismi di *monitoring* ad alto livello. L'idea delle variabili condizione è data da tale schema:

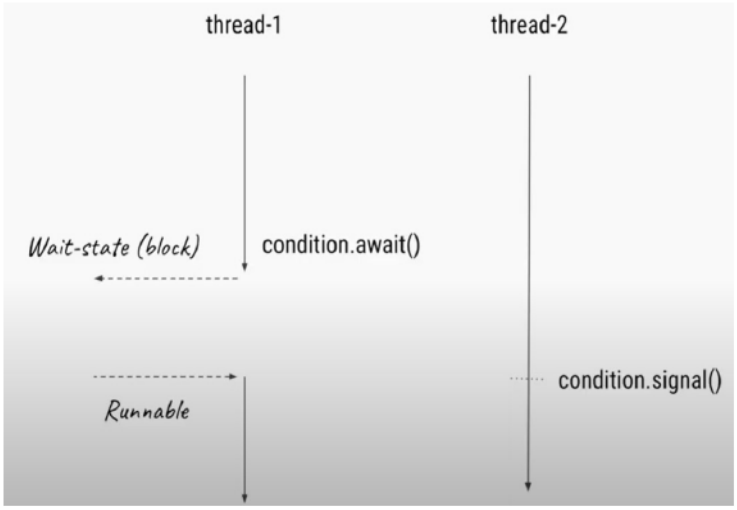


Figure 4: Schema variabili condizione

Le **variabili condizione** sono associate ad una lock e permettono ai thread di controllare se una *condizione* sullo stato della risorsa è verificata o meno. Se la condizione è falsa, rilasciano la *lock()*, si sospendono ed inseriscono il thread in una coda di attesa per quella condizione. Inoltre risvegliano un thread in attesa quando la condizione risulta verificata.

Solo dopo aver acquisito la lock su un oggetto è possibile sospendersi su una variabile d condizione, altrimenti viene generata una IllegalMonitorException. La JVM mantiene più code:

- una per i threads in attesa di acquisire la lock
- una per ogni variabile condizione

```
//Schema generale

private Lock lock = new ReentrantLock();
private Condition conditionMet = lock.newCondition();

public void method1() throws InterruptedException{
    lock.lock();
    try {
    conditionMet.await(); //sospensione
    //l'esecuzione riprende da questo punto
    //operazioni che dipendevano dalla verifica della condizione
    } finally {
        lock.unlock();
    }
```

```
}

public void method2() {
    lock.lock();
    try {
        //operazioni che rendono valida la condizione
        conditionMet.signal();
        }finally {
            lock.unlock();
        }
}
```

L'**interfaccia Condition** definisce i metodi per sospendere un thread e per risvegliarlo. Le condizioni sono istanze di una classe che implementa questa interfaccia. Vediamo una soluzione del problema produttore-consumatore con le variabili condizione:

```
    public class Messagesystem {
        public static void main(String[] args) {
            MessageQueue queue = new MessageQueue(10);
                new Producer(queue).start();
                new Producer(queue).start();
                new Producer(queue).start();
                new Consumer(queue).start();
                new Consumer(queue).start();
                new Consumer(queue).start();
            }
    }
```

```
// Produttore
import java.util.concurrent.locks.*;

    public class Producer extends Thread {

        private int count = 0;
        private MessageQueue queue = null;

            public Producer(MessageQueue queue){
                this.queue = queue;
            }

    public void run(){
        for(int i=0;i<10;i++){
            queue.produce ("MSG#"+count+Thread.currentThread());
            count++;
        }
    }
}
```

```java
//Consumatore
public class Consumer extends Thread {

    private MessageQueue queue = null;

    public Consumer(MessageQueue queue){
         this.queue = queue;
    }

    public void run(){

        for(int i=0;i<10;i++){
                Object o=queue.consume();
                int x = (int)(Math.random() * 10000);
            try{
                Thread.sleep(x);
            }catch (Exception e){};
      }
    }
 }


//Implementazione coda
import java.util.concurrent.locks.*;
public class MessageQueue {

    final Lock lockcoda;
    final Condition notFull;
    final Condition notEmpty;
    int putptr, takeptr, count;
    final Object[] items;

    public MessageQueue(int size){

            lockcoda = new ReentrantLock();
            notFull = lockcoda.newCondition();
            notEmpty = lockcoda.newCondition();

            items = new Object[size];
            count=0;putptr=0;
            takeptr=0;
    }


    //Metodo per inserire in coda
    public void produce(Object x) throws InterruptedException {
            lockcoda.lock();
            try{
```

```
            while (count == items.length)
            notFull.await();

            // gestione puntatori coda
            items[putptr] = x;
            putptr++;
            ++count;

            if (putptr == items.length) putptr = 0;
            System.out.println("Message Produced"+x);
            notEmpty.signal();
        }finally {
                lockcoda.unlock();
        }
    }


    // Metodo per eliminare dalla coda
    public Object consume() throws InterrupedExeceptions {

    lockcoda.lock();
        try{
                while (count == 0)
                notEmpty.await();
        }

        \\ gestione puntatori coda
        Object data = items[takeptr];
        takeptr=takeptr+1;
        --count;

        if (takeptr == items.length) {takeptr = 0};
        notFull.signal();
        System.out.println("Message Consumed"+x);
        return x;}
        finally
        {lockcoda.unlock(); }
    }
}
```

Come si nota dall'esempio, si fa uso di un *while(condizione) wait()* a causa delle **signal spurie**: tra il momento in cui ad un thread arriva una notifica ed il momento in cui riacquisisce la lock, la condizione può diventare dinuovo falsa. In figura l'esempio del controllo di una condizione tramite (*l'errato*) uso dell'if. La prassi in uso è quella di ricontrollare la condizione dopo aver acquisito la lock e dunque inserire la *wait* in un *ciclo while*.

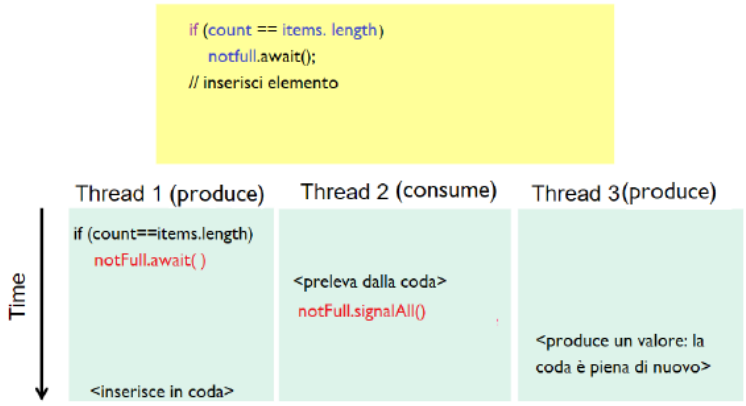


Figure 5: Signal spurie con if

3.5 Granularità delle lock

L'accesso concorrente a strutture dati deve garantire la **thread safeness** ovvero l'imperativo di mantenere consistente la struttura dati condivisa quando più thread vi accedono concorrentemente. E' possibile gestire l'accesso ad una struttura dati condivisa in relazione alla granularità delle restrizioni, in particolare tramite l'uso di:

- **Linked list**: per ogni elemento vi sono puntatori all'elemento succesivo e precedente. In questo modo inserzione ed eliminazione riguarderanno i singoli elementi della lista.

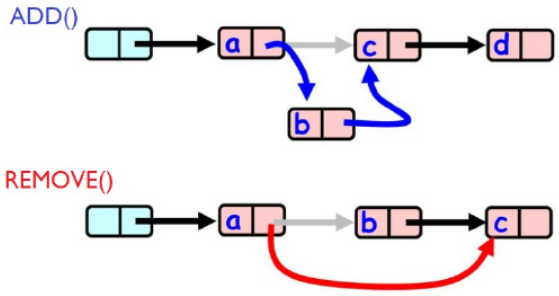


Figure 6: Linked list scheme

- **Coarse grain lock**: una singola lock per tutta la struttura. E' indubbiamente inefficiente poiché nessun thread può accedere alla struttura mentre un altro thread la sta modificando creando un *bottleneck*.

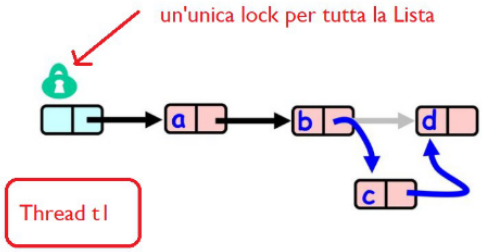


Figure 7: Linked list scheme

- **Hand-over-hand locking** (*o Fine Grain Locks*): si implementa la mutua esclusione solo su piccole porzioni della lista, permettendo ad altri thread l'accesso ad elementi diversi della struttura (*sotto ipotesi di strutture ampie, composte da più componenti a cui thread differenti necessitano di lavorare contemporamente*). Tale struttura consente di acquisire le lock sulle diverse parti della struttura **dinamicamente**.

3.5.1 Fine Grain Locks: implementation

La struttura di *Linked list* accessibile tramite l'***hand-over-hand locking*** in porzioni di ristretta dimensione comporta tuttavia una gestione dei riferimenti degli oggetti (*diremmo, in parallelo con C, i puntatori agli elementi consecutivi in lista*) al fine di evitare incosistenze nei riferimenti agli oggetti e alle modifiche su questi. Poniamoci nel seguente scenario di una rimozione concorrente:

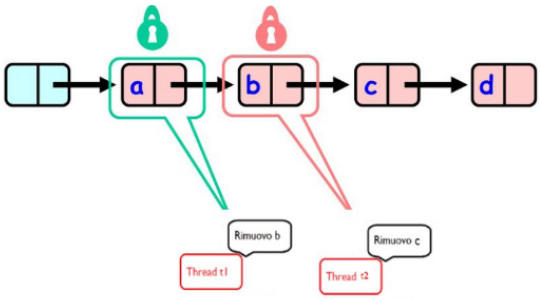


Figure 8: Linked list scheme

In questo contesto in cui il *thread t1* e il thread t2 accedono concorrentemente alla lista e sotto l'ipotesi che ogni thread acquisisca la lock solo sull'elemento che sta visitando la rimozione da parte del *thread t2* porterebbe ad un **fallimento** della rimozione poiché per rimuovere il **nodo c** il *thread t2* assegna al predecessore di *c* (*quindi b*) il puntatore a *d* ma contemporaneamente *il thread t1* elimina b facendo puntare direttamente a *c* sganciando *b* dalla lista. La

rimozione da parte di *t2* del nodo *c* non avrebbe alcun effetto sulla lista (*ciò consente di mantenere i riferimenti agli oggetti ma non di implementare una corretta politica di rimozione*).
Una soluzione a tale problema è data dall'acquisizione sia da parte del *thread t1* che del *thread t2* di due lock su due nodi diversi: la lock del nodo da eliminare e la lock del nodo predecessore al nodo da eliminare. Ciò consente al thread *t1*

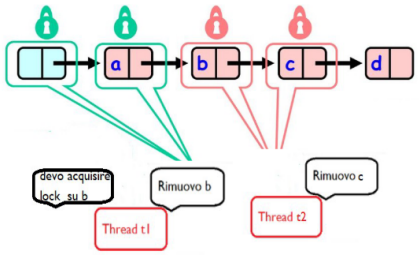


Figure 9: t2 è in grado di rimuovere il nodo c, poiché detiene anche la lock di b

di attendere che la lock su *b* sia libera prima di poter effettuare la propria operazione di rimozione, in attesa che il thread *t2* elimini *c*, ponga come successore di *b* il nodo *d* e infine rilasciando la lock consenta a *t1* di eliminare *b* ponendo *d* come successore di *a*.

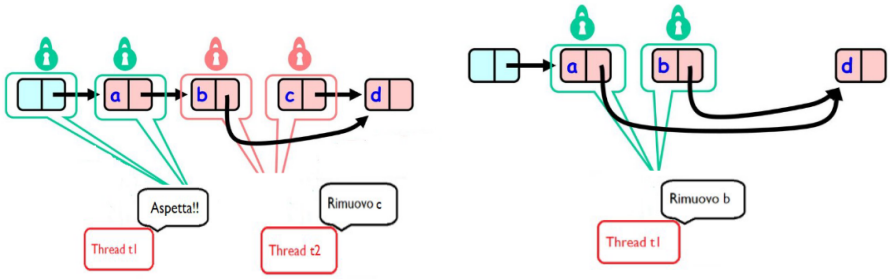


Figure 10: t1 aspetta durante la rimozione di c da t2 ed infine acquisice la lock su b

3.6 High level lock

Ciascon oggetto Java ha associato una lock implicita (allocata dalla JVM) e una coda associata (*gestita dalla JVM*). E' stato l'unico meccanismo di sync fino a Java 5. Una lock intrinseca è utilizzata nel seguente modo:

```
public void somemethod ( ){
    synchronized (object) {
```

```
    // a thread that is executing this code section
    // has acquired the object intrinsic lock
    // only a single thread may execute this
    // code section at any given time
    }
}
```

Un thread acquisisce la lock su un object quando entra nel blocco sincronizzato e la rilascia quando termina il blocco. Può delimitarsi a blocchi di codice o a singoli metodi (*public synchronized void method1()*). Quando il metodo synchronized viene invocato tenta di acquisire la lock intrinsenca associata all'istanza dell'oggetto su cui esso è invocato. Nell'accezione classica se un oggetto è **bloccato** (*lock acquisita da un blocco o un altro metodo sync*) il thread si sospende sulla coda associata all'oggetto finché non viene rilasciata. Generalmente la lock viene rilasciata al ritorno del metodo o in casi di eccezione (UncaughtException). Tra le regole di utilizzo:

- I costruttori non possono essere dichiarati synchronized
- Solo il thread che crea l'oggetto deve avere accesso ad esso mentre l'oggetto viene creato
- Non ha senso specificare synchronized nelle interfacce
- Se una sottoclasse sovrascrive un metodo synchronized della superclasse, il metodo della sottoclasse deve essere reso synchronized (*non lo è di default*).

Tale struttura delle lock consente di sincronizzare solo parti di un metodo piuttosto che l'intero metodo, diminuendo la concorrenza e ottenendo sezioni critiche di dimensione minore all'interno di metodi.

```
public class test {
    private longc1 = 0;
    private longc2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {

            synchronized(lock1) {c1++;}
    }
    public void inc2() {
        synchronized(lock2) {c2++;}

    }
}
```

La **sincronizzazione tra metodi statici** consente di acquisire una lock intrinsenca associata alla **classe** invece che all'istanza di un oggetto. Nell'esempio i metodi vengono eseguiti in mutua esclusione:

```
public class SomeClass {

    public static synchronized void methodOne() {
    // Do work
    }
    public void methodTwo() {
        synchronized (SomeClass.class) {
        // Do work
        }
    }
}
```

3.7 Monitor (Hoare-Hansen)

Un **monitor** è un meccanismo linguistico ad alto livello per la sincronizzazione: classe di oggetti utilizzabili concorrentemente in modo safe. E' in grado di incapsulare una struttura condivisa è le operazioni su di essa. Ogni risorsa è un oggetto passivo: le sue operazioni vengono invocate da entità attive (threads). Garantisce la mutua esclusione sulla struttura tramite la lock implicita associata alla risorsa (*un solo thread per volta si trova all'interno del monitor)*. La sincronizzazione sullo stato della risorsa è garantita esplicitamente, fornendo meccanismi per la sospensione/risveglio sullo stato dell'oggetto condiviso. Il monitor è dunque un oggetto con un insieme di metodi synchronized che incapsula lo stato di una risorsa condivisa.

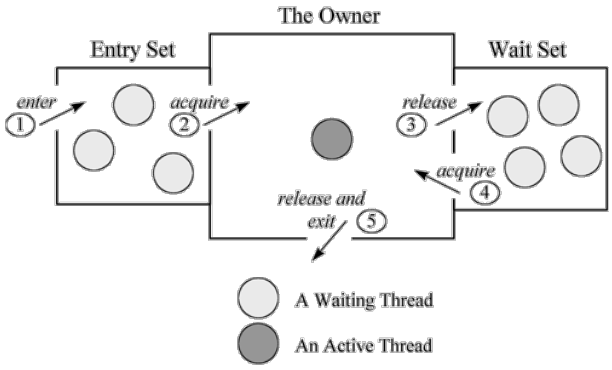


Figure 20-1. A Java monitor.

Un monitor ha due code gestite in modo implicito:

- **Entry Set**: contiene i thread in attesa di acquisire la lock. Inserzione/estrazione associate ad invocazione/terminazione del metodo.
- **Wait Set**: contiene i thread che hanno eseguito una qait e sono in attesa di una notify. Inserzione/estrazione in questa coda in seguito ad invocazione esplicita di wait(), notify().

Dunque rispettivamente **entry set** e **wait set** sono due code: la prima è quella per i thread che attendono di detenere il monitor, la seconda è quella per i thread che già precedentemente hanno ottenuto il monitor e lo hanno rilasciato in attesa del verificarsi una certa condizione sullo stato del monitor. Anche se non si evince dallo schema, al risveglio al wait set i thread vengono rimessi nell'entry set poiché è necessario nuovamente riacquisire il monitor.

3.7.1 Metodi

Tali metodi sono invocati su un oggetto e dunque appartengono alla classe Object. E' necessario per invocarli aver già acquisito precedentemente la lock: vanno invocati all'interno di un metodo o di un blocco sincronizzato, altrimenti viene sollevata l'eccezione **IllegalMonitorException**. Se non compare riferimento esplicito all'oggetto allora il riferimento implicito è *this*.

- **void wait()**: sospende il thread in attesa che si sia verificata una condizione. Permette di attendere fuori dal monitor un cambiamento su una condizione tramite attesa passiva evitando il controllo ripetuto di una condizione (*polling*).La wait rilascia la lock sull'oggetto acquisita eseguendo il metodo sincronizzato, prima di sospendere il thread (*contrariamente a sleep() e yield()*) e conseguentemente ad una notifica, può riacquisire la lock.
- **void wait(long timeout)**: sospende per al massimo timeout millisecondi
- **void notify()**: risveglia uno dei thread nella coda di attesa sulla lock di quell'oggetto. Se invocato quando non vi è alcun thread sospeso la notifica viene persa. **notifyAll()**: risveglia tutti i threads in attesa, che passano in stato di pronto. I thread risvegliati competono per l'acquisizione della lock e dunque verranno eseguiti uno alla volta, quando riusciranno a riacquisire la lock sull'oggetto.

3.8 Problema lettori-scrittori

E' un classico problema che contempla due entità: i **lettori** che escludono gli scrittori ma non gli altri lettori e gli **scrittori** che escludono sia i lettori che gli scrittori.
Tale problema è un'astrazione del problema dell'accesso ad una base di dati: un insieme di threads possono leggere dati in modo concorrente per assicurare la consistenza dei dati infatti le scritture devono essere eseguite in mutua esclusione.

Analizziamo una soluzione con Monitor senza l'uso di lock esplicite, ReadWriteLock o condition variables: in questo caso, e nel caso generale, il Monitor funge da arbitro della risorsa senza necessariamente incapsularla (*possibile estendendo i metodi per l'interazione con la risorsa*). Nell'esempio c'è una sola coda associata al monitor:

```
//Main - istanzia lettori e scrittori passando il Monitor
public class ReadersWriters {
    public static void main(String args[]){

        RWMonitor RWM = new RWMonitor();

        for (int i=1; i<10; i++){
            Reader r = new Reader(RWM,i);
            Writer w = new Writer(RWM,i);
            r.start();
            w.start();
        }
    }
}


//Lettore
public class Reader extends Thread {

    RWMonitor RWM;
    int i;
    public Reader (RWMonitor RWM, int i){
        this.RWM=RWM; this.i=i;
    }

    public void run(){
         while (true){

                RWM.StartRead();
            try{
                Thread.sleep((int)Math.random() * 1000);
                System.out.println("Lettore"+i+"sta leggendo");
            }
            catch (Exception e){};

            RWM.EndRead(); //Rilascia la risorsa
        }
    }
 }

//Scrittore
public class Writer extends Thread {
```

```
    RWMonitor RWM; int i;
    public Writer (RWMonitor RWM, int i){
        this.RWM=RWM; this.i=i;
    }

        public void run(){
            while (true){

                        RWM.StartWrite(); //Blocca lo scrittore se non pu
                            scrivere (lettori o altro scrittore)
                    try{
                        Thread.sleep((int)Math.random() * 1000);
                        System.out.println("Scittore"+i+"sta scrivendo");
                }
                catch (Exception e){};
                RWM.EndWrite();
            }
        }
}

//Monitor
class RWMonitor {

    int readers = 0; //Numero lettori
    boolean writing = false; //Se c' uno scrittore

    synchronized void StartRead() {
        while (writing)
            try {
                    wait(); //Lettore si autosospende nella sola coda
                        presente nel Monitor
                }catch (InterruptedException e) {}

        //Al risveglio, comincia a leggere incrementando il numero di
            lettori
        readers = readers + 1;
    }

    synchronized void EndRead() {

        readers = readers - 1;
        if (readers == 0) notifyAll(); //Potrebbe esserci degli
            scrittori sospesi in attesa di avere la risorsa
            //esclusivamente
    }

    synchronized void StartWrite() {

        while (writing || (readers != 0)) //Se nessuno scrittore e
            nessun lettore allora scrivo
```

```
            try {
                wait();
            }catch (InterruptedException e) {}

        writing = true;//Risvegliato, comincio a scrivere
    }

    synchronized void EndWrite() {

        writing = false;
        notifyAll();
    }
}
```

Un lettore dunque può accedere alla risorsa se vi sono altri lettori e tutti i lettori possono accedere continuamente alla risorsa e non dare la possibilità di accesso agli scrittori causando un evindente scenario di **starvation** nei confronti degli scrittori: se uno scrittore esegue una *notifyAll()* che sveglia sia i lettori che gli scrittori il comportamento è equo (per entrambe le entità) se è equa la strategia di schedulazione di Java

```
    public synchronized void act() throws InterruptedException{
        while(!cond) wait();
        //modify monitor data
        notifyAll();

    }
```

La prassi d'uso per l'uitlizzo delle *wait-notify* è, come precedentemente accennato, di testare sempre la condizione all'interno di un *while* poiché la coda di attesa è unica per tutte le condizioni e dunque il thread potrebbe essere stato risvegliato in seguito al verificarsi di un'altra condizione.
Uno scenario possibile è che la condizione su cui il thread T è in attesa si è verificata tuttavia un altro thread invalida nuovamente tale condizione dopo che il thread T è stato risvegliato. Ciò rende necessario il testing della condizione nella guardia del *while*.

Finora abbiamo visto l'uso della wait() in un Monitor tuttavia può essere utilizzata anche in un blocco **synchronized**.
Il seguente snippet consente l'attesa del verificarsi di una condizione su un oggetto diverso da *this*:

```
synchronized (obj)
    while (!condition){
            try {
                obj.wait ();
        }catch (InterruptedException ex){...}
    }
```

Il secondo snippet permette di segnalare una certa condizione:

```
synchronized(obj){
    condition=.....;
    obj.notifyAll()
}
```

In conclusione, le **lock implicite** comportano alcuni vantaggi quali: costrutti strutturati (*evitano errori dovuti alla complessità del programma*), concorrenza (*deadlocks, mancato rilascio, etc*) e maggior manutenibilità del software. Tra i principali svantaggi tuttavia sono poco flessibili. Dualmente, le lock esplicite comportano alcuni vantaggi quali un numero maggiore di funzioni disponibili e maggiore flessibilità (*come l'uso di tryLock() o nel contesto di più lettori ed un singolo scrittore*) garantendo maggiori performance. Tra gli evidenti svantaggi tuttavia vi è la poca leggibilità del codice se non usate in modo strutturato.

3.9 Java Concurrency Framework

Il JCF offre tre principali package di ampio utilizzo:

- ***java.util.concurrent***: fornisce strumenti per l'esecuzione asincrona di task (*Executors, Thread pools and future*), strutture dati per disegnate per ambienti concorrenti (*queue, blocking queues, concurrent hash map, etc*), tools per coordinamento dei thread (*semaphore, latch, barriers, exchanger, etc*)
- ***java.util.concurrent.atomic***: meccanismi per implementare l'atomicità di variabili di tipo semplice (*AtomicBoolean, AtomicInteger*)
- ***java.util.concurrent.locks***: meccanismi di sync più flessibili (*read-write locks, locks and conditions*)

3.9.1 Lock Free Atomic Operations

Il JCF Include il package *java.util.concurrent.atomic* contenente delle classi che incapsulano variabili di tipo primitivo e garantiscono l'atomicità di operazioni (*incremento, decremento, etc*). Sono operazioni atomiche che non fanno uso di locks e consentono di operare inoltre su array senza usare sincronizzazioni esplicite o lock. Un banale esempio:

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
public class AtomicIntExample {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        AtomicInteger atomicInt = new AtomicInteger();
        CounterRunnable runnableTask = new CounterRunnable(atomicInt);
```

```
        for(int i = 0; i < 10; i++){
                executor.submit(runnableTask);
        }

        executor.shutdown();
    }
}

//Classe CounterRunnable
class CounterRunnable implements Runnable{

    AtomicInteger atomicInt;

    CounterRunnable(AtomicInteger atomicInt){
        this.atomicInt = atomicInt;
    }

    @Override
    public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());
        //Operazione atomica che incrementa il valore del contatore
    }
}
```

3.9.2 Collection Framework

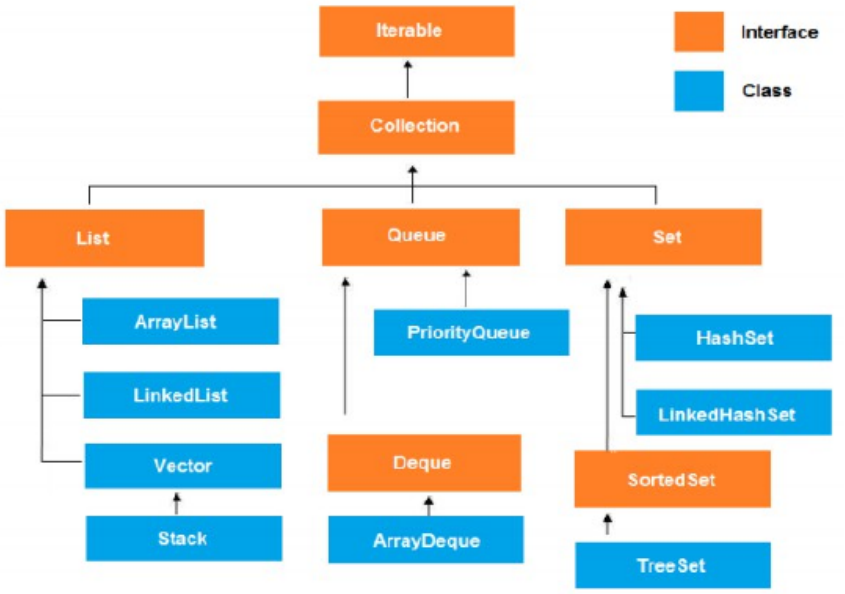
Sono un insieme di classi (*contenute nel package java.util*) che consentono di lavorare con gruppi di oggetti ovvero **collezioni di oggetti**.

Un'ulteriore interfaccia è data da Map, avente diverse implementazioni (***HashMap, implementazione di Map che realizza una struttura dati dizionario che associa termini chiave univoci a valori)***). Un secondo strumento utile sono le **Colletions** che contiene metodi utili per l'elaborazioni di collezioni di qualunque tipo (*ordering, permutation, equals, etc*): particolarmente rilevante è la possibilità di aggiungere un **wrapper** di sincronizzazione ad una collezione, rendendola thread-safe.

In generale si possono distinguere tre tipi di collezioni:

- collezioni che non offrono alcun tipo di supporto per il multithreading
- syncrhonized collections
- concurrent collections

La collezione **Vector** è un contenitore elastico, estensibile ed accorciabile ma non generico che garantisce la thread-safeness. Contrariamente, la collezione **ArrayList** è un vettore di dimensione variabile che non fornisce thread-safety di default e dunque privo di meccanismi di sincronizzazione ma più efficiente in termini di performance penalty.



Nell'ArrayList l'implementazione della *add()* non è atomica e dunque esecuzioni concorrenti della add possono portare ad interleaving scorretti. Il metodo **SynchronizedColletion** consente di rendere una collezione sincronizzata implementando una lock intrinseca sull'intera collezione (*tramite metodi wrapper*). Ogni metodo si sincronizza sulla stessa lock comportando una degradazione delle performance poiché un singolo thread per volta può operare sull'intera collezione. Esistono diversi metodi che implementano un *wrapped* con codice di sincronizzazione (*synchronizedList, synchronizedMap, synchronizedSet, etc*). Vediamo un esempio in cyu su fa uso di **ArrayList** e **SyncrhonizedArrayList**:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

public class VectorArrayList {

    public static void addElements(List<Integer> list){
        for (int i=0; i< 1000000; i++) {list.add(i);}
    }

    public static void main (String args[]){

        final long start1 = System.nanoTime();
        addElements(new ArrayList<Integer>());
```

```
        ArrayList final long end1 = System.nanoTime();
        SynchronizedArrayList final long start2 = System.nanoTime();

        addElements(Collections.synchronizedList(new
            ArrayList<Integer>()));
        final long end2=System.nanoTime();

        System.out.println("ArrayList time "+(end1-start1));
        System.out.println("SynchronizedArrayList time "+(end2-start2));
    }
}
```

Nonostante l'efficacia, l'uso delle SynchronizedCollections non risolve granché poiché la thread safety garantisce che **le invocazioni delle singole operazioni della collezione siano thread safe, non gruppi di operazioni**. Ciò è dovuto alla combinazione delle operazioni atomiche che non necessariamente è atomica: *isEmpty()* e *remove()* sono entrambe atomiche, ma la loro combinazione non lo è.
Per rendere atomica una operazione composta da più operazioni individuali è richiesta una sincronizzazione esplicita da parte del programmatore:

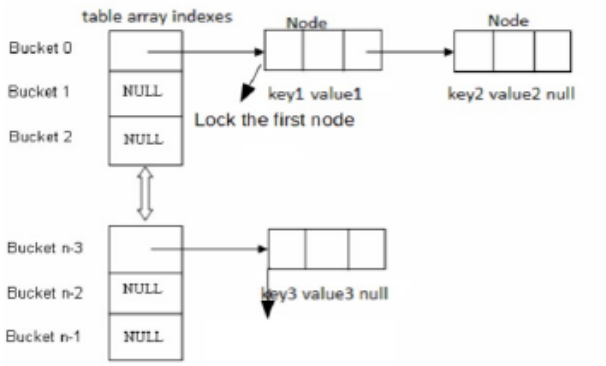
```
    synchronized(synchList) {
        if(!synchList.isEmpty())
            synchList.remove(0);
    }
```

E' un classico esempio di **blocchi sincronizzati**: il thread che l'esegue l'operazione composta acquiscela lock sulla struttura *synchList* più di una volta (sia quando esegue il blocco sincronizzato che quando esegue i metodi della collezione) ma il comportamento è garantito corretto dalle lock rientranti.

3.9.3 Concurrent Collections

Sono delle alternative thread-safe rispetto alle Synchronized Collections: l'idea è di non avere una sola lock per tutta la collezione ma di migliorare la performance consentendo l'**overlapping** di operazioni di scrittura su parti diverse della struttura, eseguendo letture in parallelo a modifiche della struttura. Inevitabilmente ciò comporta un costo di consistenza (*weak consistency*): ad esempio, l'uso di operazioni quali *size()* o *empty* potrebbe non restituire l'ultima modifica sulla collezione.

Vediamo come si applica sulla **HashMap** che di default è dotata di 16 buckets (se all'istanziazione nessun parametro, di default 16) che vengono incrementati se il load factor è alto. La ConcurrentHashMap fa uso del **lock striping** ovvero di non associare un unica lock a tutta l'HashMap ma bensì di associare una lock diversa per ogni bucket diverso. Ciò rende parallele le



operazioni parallele se eseguite su buckets distinti.

Le concurrent collections non permettono al programmatore di sincronizzare sequenze di operazioni su singoli elementi della struttura tuttavia offrono un insieme di **operazioni atomiche** che sono sequenze di operazioni di uso comune e vengono definite come una operazione unica. La JVM traduce la singola operazione *ad alto livello* in una seqeunza di operazioni di più basso livello garantendo la sincronizzazione corretta su tale operazione.
Una problematica presente ma meno evidente nell'uso delle Concurrent Collections sono l'insorgenza di **race conditions** in cui nonostante le "appropriate" verifiche è possibile generare scenari di inconsistenza. Per lo specifico esempio (*presente nelle slide lez. 3, pagina 44*) riguardante le HashMap si pone come soluzione l'uso (*e relativa implementazione*) dell'interfaccia ConcurrentMap:

```
    public interface ConcurrentMap<K,V> extends Map<K,V> {
        V putIfAbsent(K key, V value);
        // Insert into map only if no value is mapped from K
        // returns the previous value associated to the key
        // or null if there is no mapping for that key
        boolean remove(K key, V value);
        // Remove only if K is mapped to V
        boolean replace(K key, V oldValue, V newValue);
        // Replace value only if K is mapped to oldValue
        V replace(K key, V newValue);
        // Replace value only if K is mapped to some value
}
```

Tale interfaccia offre nuove funzioni per garantire atomicità ad operazioni composte tramite l'uso di funzioni del tipo **query-then-update** e **test-and-set**.

```
    import java.util.concurrent.ConcurrentHashMap;
public class PutIfAbsentExample {
public static void main(String[] args) throws Exception
{ ConcurrentHashMap<String, String> chm =
new ConcurrentHashMap<String,
```

```
 String>();
chm.put("Roma", "Italia");
chm.put("Berlino", "Germania");
chm.put("Parigi", "Francia");
System.out.println("La mappa iniziale : " + chm);
Thread PutThread = new RaceThread(chm);
PutThread.start();
Thread.sleep(10000);
String returnedValue = chm.putIfAbsent ("Londra", "Great Britain");
if (returnedValue==null) {System.out.println("Sono il main ed ho
inserito il valore" + "Londra=Great Britain");};
System.out.println("Sono il Main e la mappa ora contiene "+chm); } }

public class RaceThread extends Thread {
ConcurrentHashMap<String, String> chm;
public RaceThread (ConcurrentHashMap<String, String> chm)
{this.chm=chm; };
public void run()
{
Object returnedValue = chm.putIfAbsent ("Londra", "UK");
if (returnedValue==null)
{System.out.println("Sono il Thread ed ho inserito il
valore" + "Londra=Great UK");};
System.out.println("Sono il Thread e la mappa ora contiene "
+ chm);
}
}
```

3.9.4 Iterators

Gli **iteratori** sono oggetti di supporto usati per accedere agli elementi di una collezione, uno alla volta ed in sequenza. Sono associati ad un oggetto collezione e deve conoscere la rappresentazione interna della classe che implementa la collezione al fine di potervi operare (*se tabella has, tree, etc*). L'interfaccia Collection contiene il metodo **iterator()** che restituisce un iteratore per una collezione: chiaramente diverse implementazioni di Collection implementano a loro volta il metodo *iterator()* in modo diverso. Comodo è l'uso dell'interfaccia **Iterator** che prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo.

```
    //collezione di oggetti di tpo T che vogliamo scandire
    Collection<T> c = ...

    //iteratore specifico per la collezione c
    Iterator<T> it = c.iterator();

    // finch  non abbiamo raggiunto l'ultimo elemento
    whie(it.hasNext()){
```

```
    T element = it.next();
    //uso oggetto estratto (anche rimuovendolo)
}
```

UN interatore non ha alcuna funzione di reset: una volta iniziata la scansione non è possibile far tornare indietro l'iteratore e una volta finita la scansione è necessario crearne uno nuovo.

Uno scenario plausibile è quello in cui un thread modifica una collezione mentre un alto thread sta iterando sulla stessa collezione. Distinguiamo dunque due tipologie di iteratori in relazione al modo di "*comportarsi*" in uno scenario simile:

- **Fail-fast iterators**: sollevano una ConcurrentModificationException se c'è una modifica strutturale alla collezione su cui l'iteratore sta operando (*ne sono esempi iteratori su ArrayList, HashMap, etc*).
- **Fail-safe iterators**: sono thread-safe e dunque non sollevano eccezioni ma creano un clone (*non sempre*) della collection e lavorano su quello (*ne sono esempi iteratori su ConcurrentHashMap, CopyOnWriteArrayList, etc*). Inevitabilmente genera **weak-consistency**.

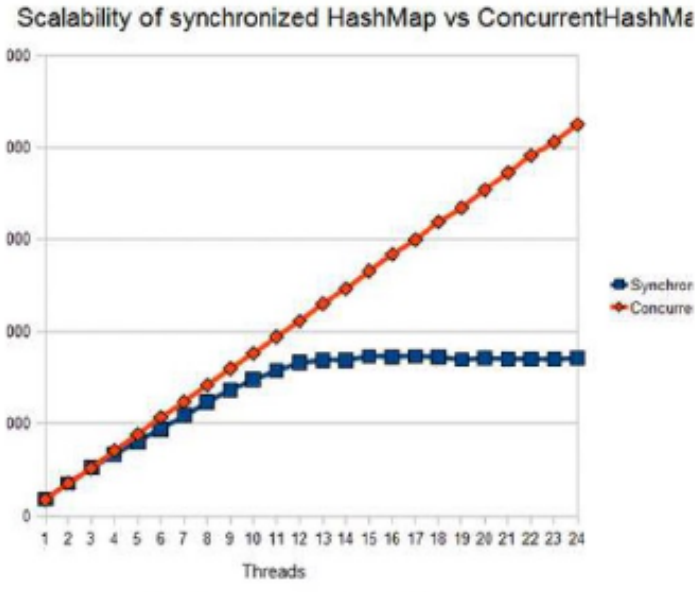
Dunque un iteratore se non clona la collezione può catturare le modifiche effettuate sulla collezione dopo la sua creazione. Se clona le modifiche possono non essere visibili all'iteratore. Se clona inevitabilmente alcuni metodi possono restituire un valore approssimato (***weakly consistent behaviour*** **quali size() e isEmpty()**). JavaDoc riporta che : *"The view's iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction."*

Sotto riportato il grado di performance delle synchronized HashMap rispetto alle ConcurrentHashMap. Sull'asse delle x è riportato il throughput ovvero il totale di accessi alla map eseguite da tutti i threads insieme in due secondi.

4 Java I/O

Le operazioni di I/O consentono di reperire informazioni da una sorgente esterna o inviare ad una sorgente esterna come

- file system
- connessioni di rete
- keyboard
- in-memory buffers (array): vista di un buffer di memoria come sorgente o destinazione esterna. Per ottimizzare l'accesso ai dati si legge il file in un buffer della memoria centrale. Per astrazione l'interfaccia verso il modulo che gestisce i dati deve rimanere la stessa aldilà di ogni implementazione.



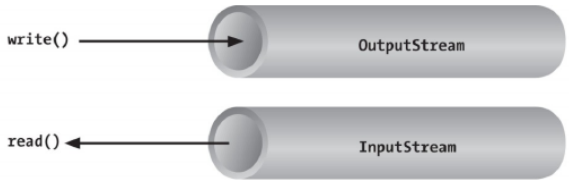
Java I/O definisce un insieme di astrazioni per gestire le operazioni di input-output che rappresentano una delle parti più complesse dell'intero linguaggio poiché deve astrarre dai diversi tipi di device di input-output evitando al programmatore la complessità di adattarsi ad ogni singola specifica implementazione. In Java la prima astrazione definita è basato sul concetto di **stream *o flusso***. Durante il corso i package rilevanti saranno principalmente:

- **Java IO API**: package standard di I/O contenuto in *java.io*, introdotto già da JDK 1.0 e basato su stream (*lavora su bytes e caratteri*).
- **JAVA NIO API (New IO)**: funzionalità simili a java.io, ma con basato su canali e con comportamento non bloccante garantendo migliori performance in alcuni scenari (*a partire da JDK 1.4*).
- **Java NIO.2**: ulteriori miglioramenti rispetto a Java NIO

4.1 Java Stream

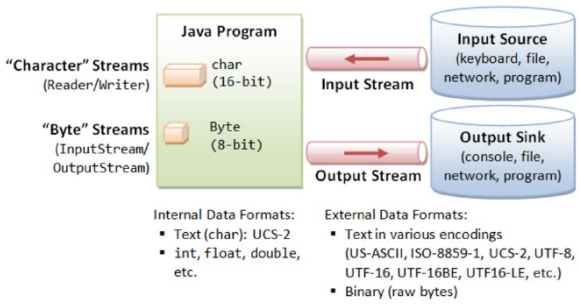
Uno stream rappresenta una connessione tra un programma Java ed un dispositivo esterno *(file, buffer di memoria, connesione di rete,...)* il cui flusso di informazione è di lunghezza illimitata.

Astraendo, uno stream è un *"tubo"* tra sorgente e destinazione (dal programma ad un dispositivo e viceversa). L'applicazione inserisce dati o li legge ad/da un capo dallo stream. L'accesso ai dati dello stream è sequenziale: il primo byte che viene mandato sullo stream è il primo che viene letto all'altro capo dello stream, mantenendo un ordinamento FIFO. Gli stream sono **one**



way: read-only oppure write-only infatti se un programma ha bisogno di dati in input ed output è necessario aprire due stream, in input ed output. Gli stream così definiti sono **bloccanti** poiché quando un'applicazione legge un dato dallo stream (*o lo scrive*) si blocca finché l'operazione non è completata.
Nota positiva è che non è richiesta una corrispondenza stretta tra letture/scritture (*possibile effettuare 100bytes di scrittura e nulla esclude che all'altro capo dello stream la lettura avvenga in due read da 20 e 80 bytes*).

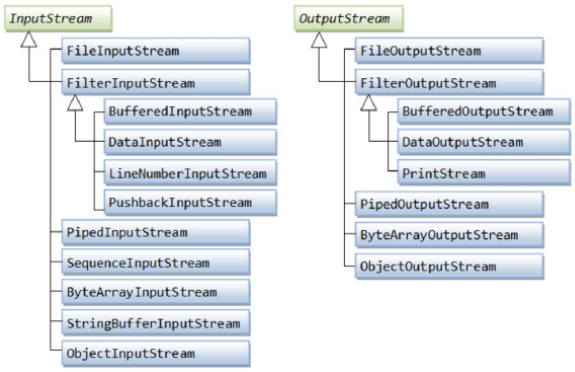
Vi sono 4 classi astratte fondamentali di Java Stream: **InputStream, OutputStream, Reader, Writer**. La classe **Input/OutputStream** consente di leggere e scrivere bytes che vengono copiati byte a byte e dunque non sono strutturati e non viene effettuata nessuna traduzione (non interpretati secondo nessuno standard). Rappresentano l'ideale per leggere/scrivere raw data in binario (come immagini, codifica di un video, etc). Dualmente, **Readers/Writers** leggono o scrivono caratteri.



4.1.1 Caratteristiche generali

Le classi astratte **Input/OutputStream** forniscono operazioni di base e possono essere associate ad ogni tipo di device di input/output. Vengono fornite diverse implementazioni per differenti tipi di I/O (*System.out/in per console, FileInputStream per files, etc).* Vediamo alcuni metodi caratteristici di **InputStream**:

- ***int read()***: tale metodo legge **un byte** dallo stream come un intero *unsigned* tra 0 e 255. Generalmente restituisce -1 se viene indivuduata la



fine dello stream o solleva un **IOException** se c'è un errore di I/O. E' un metodo bloccante.

- ***public int read(byte[] byte)***: riempe il vettore passato in input con tutti i dati disponibili sullo stream fino alla capacità massima del vettore e restituisce il numero di byte letti.
- ***skip(long n), available(), close() (vedi API)***: ad esempio, *available* consente di verificare i byte disponibili in input.

Diverse classi concrete implementano *InputStream* in relazione al tipo di device.

4.1.2 Descrittore di file

Così come un indirizzo è diverso dalla località a cui si riferisce, un descrittore di file non è il file stesso. Un'istanza della classe **File** descrive un path per l'individuazione del file o della directory fornendo alcuni metodi specifici per verificare l'esistenza del path, restituire meta-informazioni sul file, etc. Quando si vuole stabilire una connessione (*stream*) con un file si possono passare diversi parametri (*un oggetto di tipo File, una stringa, etc*). Vediamo un uso della classe *File*:

```
public class ListFiles {
   public static void main(String[] args) {

      File dir = new File("."); // current working directory

      if (dir.isDirectory()) {
         // List only files that meet the filtering criteria
         String[] files = dir.list();
         for (String file : files) {
                if (file.endsWith(".java"))
                System.out.println(file);
            }
      }
```

```
    }
}
```

Un'applicazione pratica è quella riportata di seguito: tale snippet consente di copiare il contenuto di un file (immagine) da un file ad un nuovo file tramite una copia byte a byte.

```
import java.io.*;
public class CopyImage {

    public static void main(String[] args){

        InputStream is = null;
        OutputStream os = null;

        try {
            is = new FileInputStream(new File("immagine.jpg"));
            os = new FileOutputStream(new File("immaginenew.jpg"));
            byte[] buffer = new byte[8192]; int length;

            while ((length = is.read(buffer)) > 0) { //finch vi sono byte
                da leggere vado a scriverli su os (in output)
               os.write(buffer, 0, length); }
            }

        catch(Exception e){   e.printStackTrace();     }
        finally {
            if (is!=null)
            try {is.close();} catch(Exception e){   e.printStackTrace();  }
            if (os!=null)
            try {os.close();} catch(Exception e){   e.printStackTrace();  }
            }
        }
    }
}
```

4.2 Try with resources

L'utilizzo classico del costrutto *try-finally* per la gestione di stream e connessioni varie che generalmente necessitano di essere chiuse comporta alcune problematiche. Vediamolo tramite un esempio:

```
private static void printFile() throws IOException {
InputStream input = null;
try {
    input = new FileInputStream("file.txt");
    int data = input.read(); //pu sollevare eccezioni
```

```
        while(data != -1){
            System.out.print((char) data);
            data = input.read(); //pu sollevare eccezioni
        }
    } finally {
        if(input != null) input.close(); //pu sollevare eccezioni

    }
}
```

Le istruzioni riportate nello snippet possono sollevare eccezioni tuttavia il costrutto try-finally consente la propagazione di una singola eccezione, generalmente l'eccezzione nel finally viene sollevata nello stack delle chiamate e quella nel try viene soppressa. Ciò è controintuitivo poiché sicuramente si avrà maggiore necessità di gestire efficacemente l'eccezione sollevata alla lettura piuttosto che un'eccezione dovuta alla chiusura.
Da Java 7 è stato introdotto il costrutto **try with resources** che consente la chiusura sistematica delle risorse e/o connessioni aperte da un programma occupandosi automaticamente di chiudere le risorse precedentemente aperte. Si utilizza in un blocco try con uno o più argomenti tra paretensi: gli argomenti sono le risorse che Java garantisce di chiudere al termine del blocco.
Quando si verificano delle eccezioni sia nel blocco *try-with-resources* sia durante la chiusura, la JVM sopprime l'eccezione generata dalla chiusura automatica consentendo le **suppressed exceptions** e propagata quella nel blocco try. Possono comunque essere inseriti blocchi catch e finally che vengono eseguiti comunque dopo la chiusura della risorsa. Vediamone un esempio:

```
import java.io.*;
public class trywithresources{
    public static void main (String args[])throws IOException {
        try(FileInputStream input = new FileInputStream(new
            File("immagine.jpg")); //try-with-resources con risorse
            passate come parametri
        BufferedInputStream bufferedInput = new
            BufferedInputStream(input)){ // parametri

           int data = bufferedInput.read();
               while(data != -1){
                   System.out.print((char) data);
                   data = bufferedInput.read();
                }
           }
    }
}
```

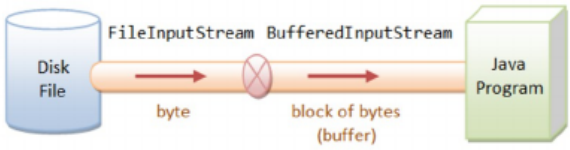
4.3 Filter streams

Rispettivamente **InputStream e OutputStream** operano su dati grezzi (*row bytes*). Esistono delle classi filtro in grado effettuare delle trasformazioni sui dati a basso livello. Alcuni tipi di filtro sono:

- Filter Stream: crittografia, compressione, bufering, traduzione dei dati in formato ad alto livello
- Readers Writes: orientati unicamente al testo e permettono di decodificare i bytes in caratteri.

Più filtri sono concatenabili: un filtro può ricevere dati da uno stream o da un altro filtro a sua volta e passarli ad un programma o ad un altro filtro.

I **BufferedInputStream** o **BufferedOutputStream** sono utilizzati rispettivamente per la bufferizzazione di input e output in cui i dati vengono scritti e letti in blocchi di bytes invece che un solo blocco per volta garantendo migliori performance in termini di tempo di lettura bytes/trasferimento.



Tale schema rispecchia in larga parte le seguenti due righe di codice:

```
FileOutputStream outputFile = new
    FileOutputStream("primitives.data");
BufferedOutputStream bufferedOutput = new
    BufferedOutputStream(outputFile);
```

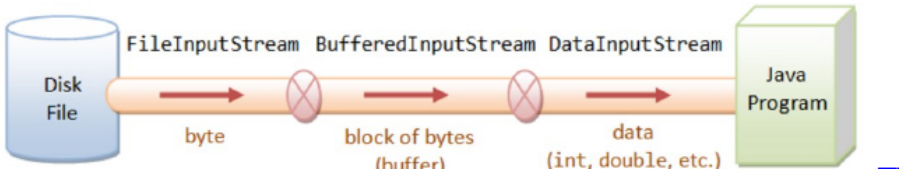
Un esempio pratico dell'applicazione di più filtri concatenati è dato dal seguente snippet:

```
import java.io.*;
public class TestDataIOStream {
public static void main(String[] args) {
String filename = "data-out.dat"; String message = "Hi,$%&!";
// Write primitives to an output file
try (
DataOutputStream out = new DataOutputStream(
                    new BufferedOutputStream(
                    new FileOutputStream(filename)))) {
                            out.writeByte(127);
                            out.writeShort(-1);
                            out.writeInt(43981);
```

```
                        out.writeLong(305419896);
                        out.writeFloat(11.22f);
                        out.writeDouble(55.66);
                        out.writeBoolean(true);
                        out.writeBoolean(false);

                        for (int i = 0; i < message.length(); ++i) {
                                out.writeChar(message.charAt(i));
                            }

                        out.writeChars(message);
                        out.writeBytes(message);
                        out.flush();

                    } catch (IOException ex) {
                        ex.printStackTrace();
                    }


// Read raw bytes and print in Hex
try (BufferedInputStream in = new BufferedInputStream(
                          new FileInputStream(filename))) {

                                int inByte;

                                while ((inByte = in.read()) != -1) {
                                    System.out.printf("%02X ", inByte);
                                    System.out.println();// Print Hex
                                        codes
                                }
       System.out.printf("%n%n");
} catch (IOException ex) { ex.printStackTrace();}

// Read primitives
try (DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                        new FileInputStream(filename)))) {

               System.out.println("byte: " + in.readByte());
               System.out.println("short: " + in.readShort());
               System.out.println("int: " + in.readInt());
               System.out.println("long: " + in.readLong());
               System.out.println("float: " + in.readFloat());
               System.out.println("double: " + in.readDouble());
               System.out.println("boolean: " + in.readBoolean());


               System.out.print("char:");
               for (int i = 0; i < message.length(); ++i) {
```

```
                System.out.print(in.readChar()); }
            System.out.println();

            System.out.print("chars: ");
            for (int i = 0; i < message.length(); ++i) {
                System.out.print(in.readChar()); }
            System.out.println();
            System.out.println();

            System.out.print("bytes:");
            for (int i = 0; i < message.length(); ++i) {
                System.out.print((char)in.readByte()); }
            System.out.println();

        } catch (IOException ex) { ex.printStackTrace(); }
    }
}
```

In figura lo schema generale rispetto allo snippet:



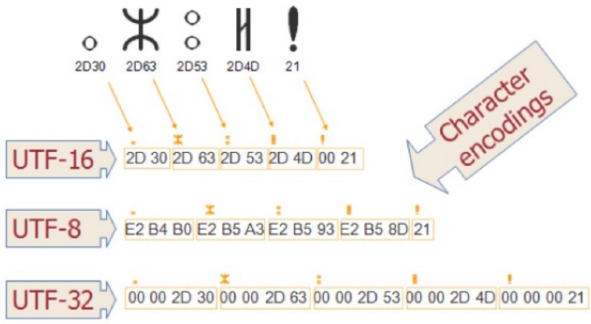
4.4 Serializzazione di oggetti

4.4.1 Encoding

Prima di affrontare il concetto di serializzazione è utile avere alcune nozioni di codifica. Inizialmente la codifica ASCII era l'unica codifica disponibile per la rappresentazione delle lettere e dei numeri ma grazie alla rapida evoluzione di internet si è reso necessario espandere a diversi idiomi e linguaggi la possibilità di rappresentazione.
Definiamo come **code character set** un insieme in cui ad ogni carattere viene assegnato un valore numerico o code point. Con **encoding** s'intende come i numeri che appartengono ad un certo character set sono rappresentanti (*8 bit, una parola, una sequenza di bytes, etc*). Tra i coded character set più diffusi:

- ASCII: 128 caratteri, 7 bits encoding all'interno di un byte.
- ISO 8859-1: caratteri identici as ASCII da 0 a 128, altri caratteri da 160 a 255, 8 bit encoding, utilizzato da sistemi UNIX-based.
- Windows 1252: come ASCII da 0 a 128, come ISO 8859 da 160 a 255 + altri caratteri, 8 bit encoding, utilizzato in Windows.

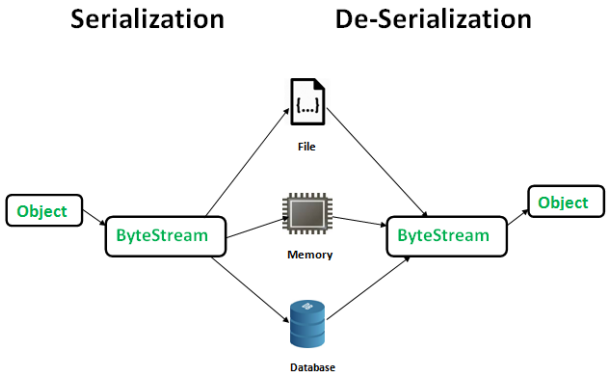
- UNICODE: gestito dall'Unicode Consortium (*standard de facto*).



Java, tra le altre cose, consente l'utilizzo di un **InputStreamReader** in cui è possibile di specificare per parametro diversi encoding. Di default il charset è settato dalla JVM al momento di start-up e varia in relazione al sistema operativo.

4.4.2 Serializzazione

Gli oggetti esistono in memoria fino a che la JVM è in esecuzione. Per assicurarne la persistenza al di fuori della JVM occorre creare rappresentare l'oggetto indipendentemente dalla JVM tramite meccanismi di serializzazione. E' noto che ogni oggetto è caratterizzato da uno **stato** (*il valore degli attributi*) e da un **comportamento** (*specificato dai metodi*). Lo stato di fatto vive con l'istanza dell'oggetto: la serializzazione si occupa del **flattening** dello stato dell'oggetto mentre la deserializzazione ricostruisce lo stato dell'oggetto.
L'oggetto serializzato può quindi essere scritto su un qualsiasi stream di output. Si chiama serializzazione poiché solitamente un oggetto può essere visto



come un **grafo** (*anche composto da più oggetti*): ciò che fa la serializzazione è

scomporlo in sequenze di byte associate ad un descrittore in grado di ricostruire l'oggetto. Solitamente si fa uso della serializzazione per inviare oggetti su uno stream che rappresenta una connessione TCP, pacchetti UDP o anche come parametri di metodi invocati tramite RMI (***Remote Method Invocation - su JVM remote***).

4.4.3 How to do

Per rendere un oggetto "persistente", l'oggetto deve implementare l'interfaccia ***Serializable***. L'interfaccia Serializable è una **marker interface** ovvero non ha nessun metodo, solo informazione su un oggetto per il compilatore e la JVM. Vi è comunque un controllo limitato sul meccanismo di linearizzazione dei dati: tutti i tipi di dato primitivi sono serializzabili mentre alcuni oggetti, se implementano Serializable, sono serializzabili, altri no.
Una seconda interfaccia è **Externizable** consente di definire un proprio protocollo di serializzazione, ottimizzando la rappresentazione serializzata dell'oggetto (*non lo vedremo durante il corso*).

```
import java.io.Serializable; //serialization
import java.util.Date;
import java.util.Calendar;

public class PersistentTime implements Serializable{ //serialization

        private static final long serialVersionUID = 1; //serialization
        private Date time;

        public PersistentTime(){
                time = Calendar.getInstance().getTime();
        }

    public Date getTime(){
        return time;
        }
}
```

Regola 1: *per serializzare un oggetto persistente la classe di cui l'oggetto è istanza deve implementare l'interfaccia Serializable oppure ereditare l'implementazione dalla sua gerarchia di classi.*

Per rendere dunque un oggetto persistente lo vado a scrivere su file (o comunque su un otuput persistente) come una sequenza di byte che poi, in un secondo momento, potrò rileggere, reinterpretare ed eventualmente manipolare.

```
import java.io.*;
public class FlattenTime{
    public static void main(String [] args){
```

```
        String filename = "time.ser";
        if(args.length > 0) { filename = args[0];  }

        PersistentTime time = new PersistentTime();
        try(FileOutputStream fos = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(fos);
                //Filtro che implementa la serializzazione
        ){
               out.writeObject(time); //Serializza l'oggetto e lo scrive
                   su file
        }
        catch(IOException ex) {ex.printStackTrace();
        }
    }
}
```

La serializzazione vera e propria è gestita dalla classe **ObjectOutputStream**: tale stream deve essere concatenato con uno stream di bytes, che può essere un FileOutputStream, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria, etc.
La serializzazione è un **processo ricorsivo** poiché l'oggetto serializzato può contenere altri oggetti quando un oggetto viene serializzato, si percorre la gerarchia delle superclassi e si salva lo stato di ogni classe, fino a che non si trova la prima classe non serializzabile.

4.4.4 Deserializzazione

Rispetto allo snippet di prima, attuaiamo una deserializzazione della data e dell'ora:

```
public class InflateTime{
    public static void main(String [] args){

        String filename = "time.ser";
        if(args.length > 0) { filename = args[0];  }

        PersistentTime time = null;
        FileInputStream fis = null;
        ObjectInputStream in = null;

        try(FileInputStream fis = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(fis);){

                time = (PersistentTime)in.readObject();

            }catch(IOException ex){ ex.printStackTrace(); }
             catch(ClassNotFoundException ex) { ex.printStackTrace(); }
```

```
        // print out restored time
        System.out.println("Flattened time: " + time.getTime());
        System.out.println();
        // print out the current time
        System.out.println("Current time: "+
        Calendar.getInstance().getTime());
    }
}
```

ClassNotFoundException: l'applicazione tenta di caricare una classe, ma non trova nessuna definizione di una classe con quel nome.
Il metodo ***readObject()*** legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale. Inoltre readObject può leggere qualsiasi tipo di oggetto, è necessario tuttavia effettuare un cast al tipo corretto dell'oggetto.
La JVM determina, mediante informazione memorizzata nell'oggetto serializzato, il tipo della classe dell'oggetto e tenta di caricare quella classe o una classe compatibile: se non la trova viene sollevata una **ClassNotFoundException** ed il processo di deserializzazione viene abortito altrimenti, viene creato un nuovo oggetto sullo heap. Lo stato di tutti gli oggetti serializzati viene ricostruito cercando i valori nello stream, senza invocare il costruttore (*uso di Reflection, non la vedremo*): di fatto si percorre l'albero delle superclassi fino alla prima superclasse non-serializzabile e per quella classe viene invocato il costruttore.

4.4.5 Non serializzabilità

Alcuni oggetti non sono serializzabili, in particolare per oggetti contenenti riferimenti specifici alla JVM o al SO (*Java native class*). Un esempio sono

- *Thread, OutputStream, Socket, File* che non possono essere ricreati, perché contengono riferimenti specifici al particolare ambiente di esecuzione
- le variabili marcarte come ***transient*** *(ad esempio variabili che non devono essere scritte per questioni di privacy, es. numero carta di credito*).
- variabili statiche che sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando

Tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile e non transient si solleva una **notSerializableException**.

Regola 2: *per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come transient (la variabile transient non viene considerata ma l'oggetto viene serializzato).*

Durante la deserializzazione non vi è garanzia che la ricostruzione della classe sia analoga a quella serializzata: si tenta di rendere la classe di partenza e arrivo

compatibili.

La serializzazione è un meccanismo computazionalmente pesante poiché Java utilizza diversi meccanismi in combinazione con i meccanismi di serializzazione quali **chaching, controllo delle versioni e performance**.

4.4.6 Caching

Ogni volta che un oggetto viene serializzato e inviato ad una *ObjectOutputStream*, un suo riferimento viene memorizzato in una **identity hash table**: se l'oggetto viene scritto nuovamente sull'*OutputStream* non viene nuovamente serializzato ma bensì inserito un puntatore all'oggetto precedente.
Analogamente quando si legge da uno stream l'oggetto viene memorizzato in un identity hash table la prima volta per cui letture future faranno riferimento allo stesso oggetto.
Evidente è la possibilità di incosistenze quando lo stato dell'oggetto viene modificato poiché nonostante l'oggetto venga modifcato il riferimento all'oggetto rimane lo stesso, perdendo la modifica effettuata.

4.4.7 Controllo delle versioni

Per deserializzare un oggetto correttamente occorre conoscere i byte che rappresentano l'oggetto serializzato ed il codice della classe che descrive la specifica dell'oggetto. La deserializzazione può avvenire in ambienti diversi come, ad esempio, tramite l'uso di un diverso compilatore o a distanza di tempo rispetto al momento in cui è stata effettuata la serializzazione o su una macchina diversa in cui l'oggetto serializzato è arrivato tramite una connessione di rete. Data la seguente classe:

```
    public class Employee {
        private String id;
        private String name;
        private int age;
}
```

Supponiamo di serializzare i dati di un insieme di *Employee* e successivamente modifichiamo tale classe come segue:

```
    public class Employee {
        private String id;
        private String name;
        private Date dateOfBirth;
}
```

E' possibile utilizzare la classe modificaa per deserializzare un oggetto che è stato serializzato con la prima classe precedente alla modifica? Si, tramite la costruzione di una classe compatibile resa tale tramite l'uso del **serialVer-**

sionUID (o **SUID/UUID**). Il **serialVersionUID** è un identificatore unico che identifica una classe e che viene utilizzato per verificare che ci sia compatibilità tra le classi usate per la serializzazione e la deserializzazione in fase di deserializzazione. Può essere generato in due modi distinti:

- **identificatore implicito**: viene generato dal compilatore come un hash crittografico a 64 bit (*SHA*) a partire dalla struttura della classe durante la serializzazione (*coinvolgendo nome della classe, nomi delle interfacce, metodi e campi*). In alcuni casi compilatori diveris possono generare identiifcatori diversi per la stessa classe. Tale metodo è sconsigliabile poiché non assicura la backward compatibilità.
- **identificatore esplicito**: il programmatore gestisce esplicitamente la compatibilità delle classi, gestendo i loro identificatori. Possono essere generati dichiarandoli a scelta dal programmatore, tramite il supporto dell'IDE in uso o con un generatore a linea di comando. Dunque classi compatibili avranno analogo serialVersionUID, diversamente classi non compatibili avranno identificatore diverso.

La classe usata per la serializzazione può essere modificata ma essere sempre **backward-compatible** pur avendo aggiunto attributi e/o oggetti, trasformato attributi da transient a non-transient, cambiare variabili di istanza in static, etc. In fase di deserializzaione il meccanismo di default semplicemente imposta i valori dei campi mancanti con valori di default.
Alcune modifiche che rendono una classe non backward-compatible sono, ad esempio, rimuovere attributi o trasformare attributi non-transient in transient.
Per dichiarare un identificatore esplicito si fa uso della variabile privata *serialVersionUID*:

```
private static final long serialVersionUID = 42L;
```

Java generalmente suggerisce di indicare esplicitamente un identificatore: *"the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization"*. Spesso si ottiene una eccezione anche se le classi utilizzate in fase di serializzazione e deserializzazione sono in realtà le stesse.
La serializzazione registra un descrittore dell'oggetto di dimensione significativa composto da:

- magic data composto da *STREAM MAGIC* e *STREAM VERSION* che contiene la versione della JVM.
- metadati: descrivono la classe associata all'istanza dell'oggetot serializzato includendo il nome della classe, il serialVersionUID, il numero di campi e altri flag.
- metadati di altre eventuali superclassi fino a raggiungere nella gerarchia *java.lang.Object*

- i valori associati all'oggetto istanza della classe, partendo dalla super classe a più alto livello
- dati degli oggetti eventualmente riferiti dall'oggetto istanza della classe, iniziando dai metadati e poi registrando i valori

Tra i principali svantaggi della serializzazione in Java vi è la dimensione che raggiunge l'oggetto serializzato rispetto all'oggetto originario a causa della quantità di informazioni che devono essere memorizzate per deserializzarlo correttamente. Vi è inoltre una limitata interoperabilità poiché la serializzazione è applicabile solo quando entrambe le applicazoni sono scritte in Java (*a differenza di JSON, largamente più usato*).

4.5 Java NIO - NewIO

E' una libreria a basso livello degli stream di Java e di conseguenza, come vedremo, impone di gestire la complessità che deriva dalla gestione di costrutti a basso livello. L'idea di base è quella di adottare una programmazione ad eventi: tramite i canali messi a disposizione per la comunicazione posso gestire la comunicazione in relazione alle attività delle componenti coinvolte nella comunicazione, se sono pronti a ricevere/inviare comunicazioni di vario genere. Consente operazioni block-oriented I/O ovvero ogni operazione produce e consuma **blocchi di dati**. Tra i principali obiettivi che si prefigge Java NIO vi è di incrementare le performance dell'I/O senza dover scrivere codice nativo oltre a poter fornire un input ad alte prestazioni da file, network socket o piped I/O, aumentando l'espressività delle applicazioni. Un vantaggio esplicito è chiaramente l'incremento delle performance definendo primitive *più vicine* al livello di sistema operativo. Tra gli inevitabili svantaggi vi è la dipendenza della piattaforma su cui si eseguono le applicazioni oltre che l'uso di primitive a più basso livello di astrazione impongono una perdità di semplicità ed elegenza rispetto all'uso di stream-based I/O.

Distinguiamo principalmente due version:

- NIO (Java 1.4): contiene entità quali *Buffer,Channel, Selectors.*
- NIO.2 (Java 1.7): contiene funzionalità interessanti quali new File System API, asynchronous I/O, update.

Nel corso delle note vedremo principalmente Java NIO e solo alcune funzionalità di NIO.2.

Tra le componenti principali di Java NIO vi sono **canali e buffers**: un canale è analogo ad uno stream in JAVA.IO, necessariamente tutti i dati da e verso dispositivi devono passare da un canale. Tutti i dati inviati o letti da un canale devono essere memorizzati in un buffer (in paragone con gli stream che invece leggono/scrivono direttamente da uno stream).

Una terza componente che consente di implementare l'event-driven programming è il **Selector** che è un oggetto in grado di monitorare un insieme di canali oltre che intercettare eventi provenienti da diversi canali (in relazione all'arrivo

di dati, richiesta apertura connessione, etc) che è in grado di monitare più canali con un unico thread.

I **Buffer** sono implementati nella classe *java.nio.buffer* e contengono dati appena letti o che devono essere scritti su un oggetto di tipo Channel. Un buffer può essere visto come un array (*di diversi tipi*) congiunto ad una serie di puntatori per tenere traccia di *read* e *write* fatte dal programma e dal sistema operativo sul buffer. Non essendo una struttura thread-safe è necessario garantirla esplicitamente tramite metologie precedentemente viste.

Un **Channel** invece è un collegamento da/verso i dispositivi esterni in modo bidirezionale. Differentemente dagli stream non si scrive/legge mai direttamente da un canale ma le interazioni con i canali possono essere di trasferimento dal canale nel buffer (*quindi il programma legge il buffer*) o quando il programma scrive nel buffer (*quindi nel caso di trasferimento dati dal buffer al canale*).

Vediamo alcuni snippet iniziali per farci un'idea di come istanziare le strutture e le relative interazioni tramite una **lettura dal canale**:

```
//Il canale  associato ad un FileInputStream
    FileInputStream fin = new FileInputStream( "example.txt" );
    FileChannel fc = fin.getChannel();

//Creazione di un ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate( 1024 );

//Lettura dal canale al Buffer
    fc.read( buffer );
```

Non è necessario specificare quanti byte il sistema operativo deve leggere nel Buffer: quando la *read* termina ci saranno alcuni byte nel canale (*che tramite opportuni metodi possiamo evidenziare*) dunque si evince la necessità di avere un insieme di variabili di stato interne all'oggetto *Buffer* in grado di evidenziare lo stato dell'oggetto Buffer ad ogni manipolazione/interazione.
Vediamo come **scrivere sul canale**:

```
//il canale  associato ad un FileOutputStream
    FileOutputStream fout = new FileOutputStream( "example.txt" );
    FileChannel fc = fout.getChannel();

//creazione del Buffer per scrivere sul canale
    ByteBuffer buffer = ByteBuffer.allocate( 1024 );

//copia del messaggio nel Buffer
    for (int i=0; i<message.length; ++i) {
     buffer.put( message[i] );
    }

//per indicare quale porzione del Buffer  significativa occorre
    modificare le
//variabili interne di stato (vedi successivamente), quindi si scrive
```

```
 sul canale
buffer.flip();
fc.write( buffer );
```

4.5.1 Buffer

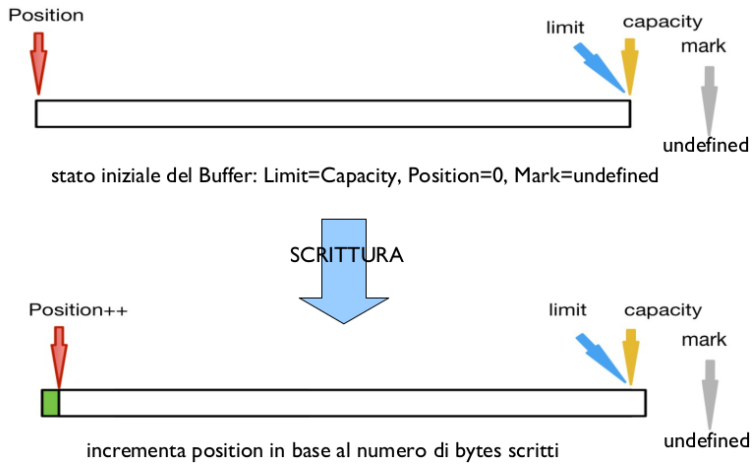
Il Buffer è dunque un interfaccia tra il programma ed il dispositivo di I/O: se effettuo una lettura allora il SO avrà prelevato precedentemente alcuni dati dal dispositivo I/O (tramite una *read*) per immetterli nel Buffer ed il programma è in grado di leggerli ed elaborarli (tramite una *get* sul Buffer). Se invece sono in scrittura è il programma che manda alcuni dati sul Buffer (*per poi mandarli al dispositivo I/O*) e tali dati vengono immessi sul Buffer tramite una *put*. Per richiedere al SO di scrivere i dati presenti sul Buffer si usa una *write*. Nel caso specifico di un *ByteBuffer* è possibile allocarlo tramite i 3 metodi distinti (*vedi API*) *allocate(), wrap() e allocateDirect()*.

4.5.2 Variabili di stato del Buffer

Le variabili di stato consentono di determinare lo stato del Buffer in relazione a letture/scritture. Generalmente è sconsigliata la modifica esplicita delle variabili se non tramite metodi predefiniti e safe in termini di congruità delle operazioni. Vi sono 4 variabili di stato:

- **Capacity**: indica il massimo numero di elementi nel Buffer. Viene definita al momento della creazione del Buffer e non può essere modificata. Genera una *BufferOverflowException* se si tenta di leggere/scrivere in una posizione maggiore di Capacity.
- **Limit**: indica il limite della porzione del Buffer che può essere letta/scritta. Per le scritture generalmente limit=capacity mentre per le lettura delimita la porzione del Buffer contenente dati significativi.
- Position: è come un file pointer per un file ad accesso sequenziale infatti indica la posizione in cui bisgona scrivere o da cui bisogna leggere. Anche tale variabile è aggiornata implicitamente dalle operazioni di lettura/scrittura sul Buffer-
- **Mark**: è un mercatore in grado di memorizzare il puntatore alla posizione corrente. Il puntatore può essere quindin resettato a quella posizione per poi rivisitarla. E' settata inizialmente ad *undefined* e tentativi di resettare un mark undefined causano *InvalidMarkException*.

Valgono sempre le seguenti relazioni:
$0 \leq mark \leq position \leq limit \leq capacity$
Lo stato iniziale del Buffer è riportato in figura: per ogni scrittura effettuata incremento position di ogni byte scritto (intero 4 byte, position+4).



Se effettuo una operazione di **mark**, faccio puntare *mark* nella stessa posizione a cui punta *position* (*ricordando la posizione corrente*).

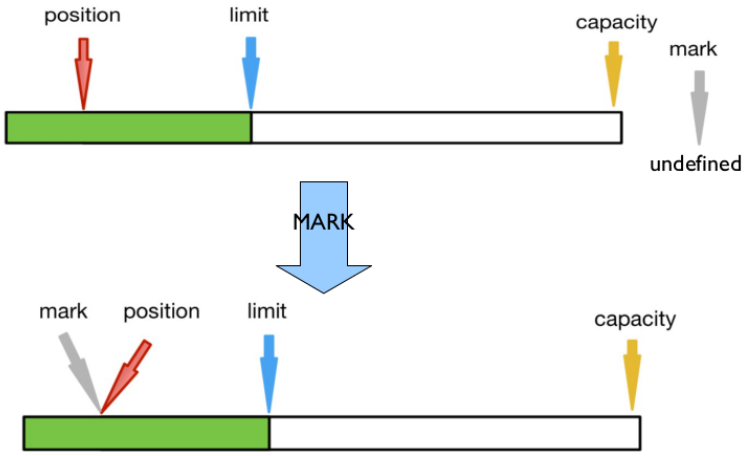
Nel caso effettuassi una **reset** invece farei in modo di far puntare *position* = *mark*, tornando alla posizione precedentemente memorizzata in *mark*.

Spieghiamo l'operazione di flipping tramite lo scenario in figura: ipotizziamo di aver già scrritto una certa porzione del buffer (parte verde) e marcato una certa posizione. L'operazione di **flip** consiste nel portare limit alla posizione position e resetta position consentendo dunque di cambiare *"modalità"* da scrittura (*scenario inizialmente descritto*) a lettura dei dati appena scritti (*parte verde del Buffer*). Dunque l'operazione di flip consente di predisporre il buffer alla lettura dopo una scrittura. Ad, esempio, due chiamate consecutive del metodo *flip()* hanno l'effetto di otenere in ogni caso un buffer di dimensione zero.

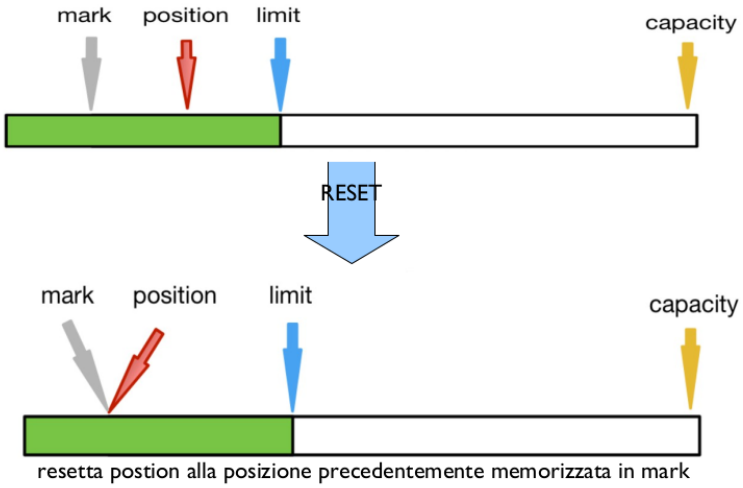
Analogamente alla scrittura, nel caso di una **lettura** incrementa position in base al numero di bytes letti.

Si effettua una **clear** per ritornare in writing mode e permette di non eliminare i dati dal buffer ma unicamente resettare le variabili di stato, resettando i puntatori.

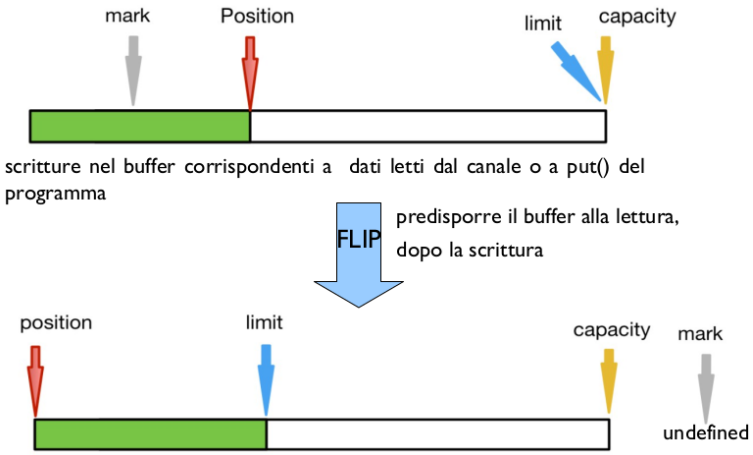
Effettuare una **rewind** permette di riportare position a 0 ma tuttavia lascia limit inviariato: è generalmene utile per rileggere dati già letti.

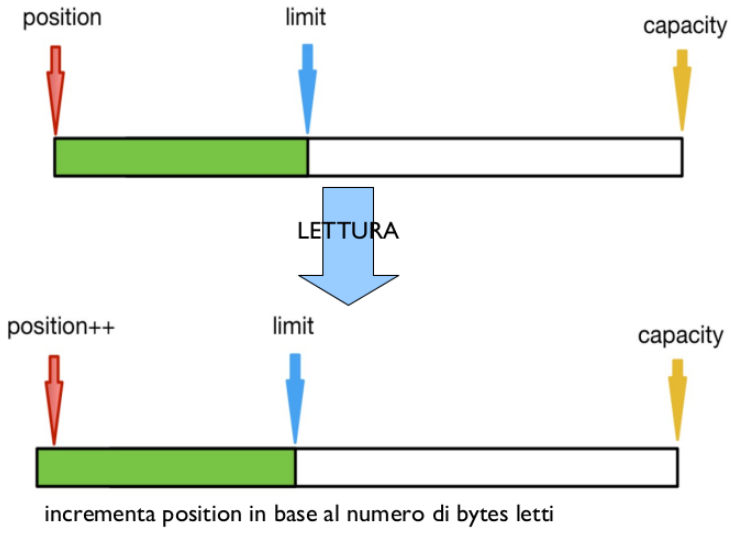


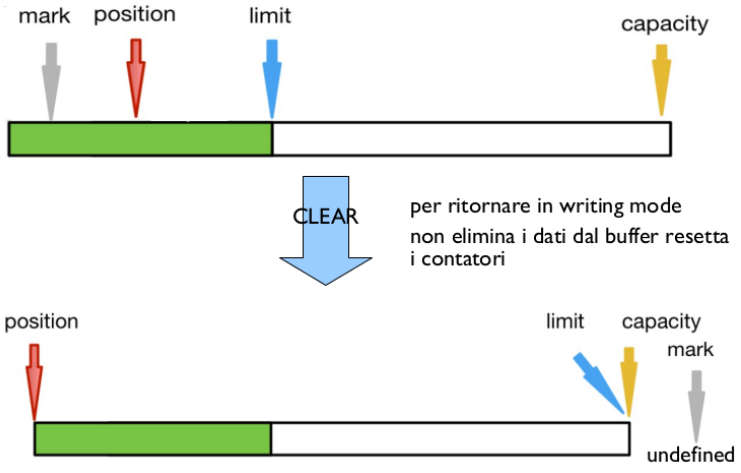
ricorda la position corrente, per poi eventualmente riportare il puntatore a questa posizione



resetta postion alla posizione precedentemente memorizzata in mark


mark
Position
limit
capacity
scritture nel buffer corrispondenti a dati letti dal canale o a put() del programma
FLIP
predisporre il buffer alla lettura, dopo la scrittura
position
limit
capacity
mark
undefined


position
limit
capacity
LETTURA
position++
limit
capacity
incrementa position in base al numero di bytes letti


mark
position
limit
capacity
CLEAR
per ritornare in writing mode
non elimina i dati dal buffer resetta
i contatori
position
limit
capacity
mark
undefined

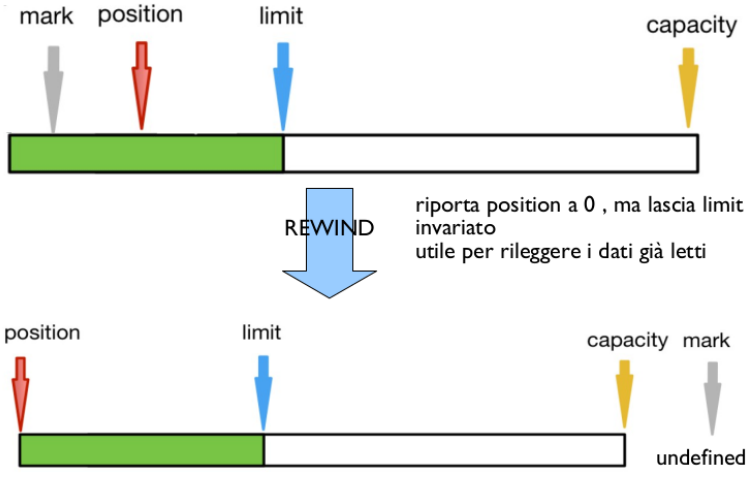

mark
position
limit
capacity
REWIND
riporta position a 0 , ma lascia limit
invariato
utile per rileggere i dati già letti
position
limit
capacity
mark
undefined

Figure 11: Si effettua una compact se il contenuto del buffer non è stato completamente letto e si inizia una nuova scrittura consentendo di copiare i bytes non ancora letti all'inizio del buffer.

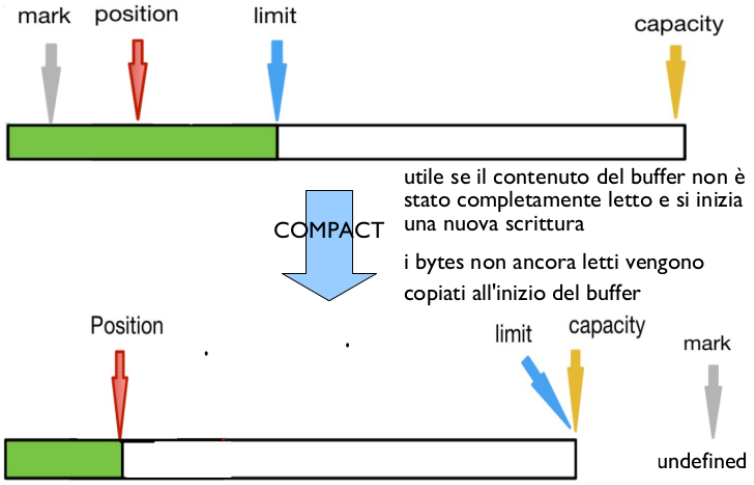
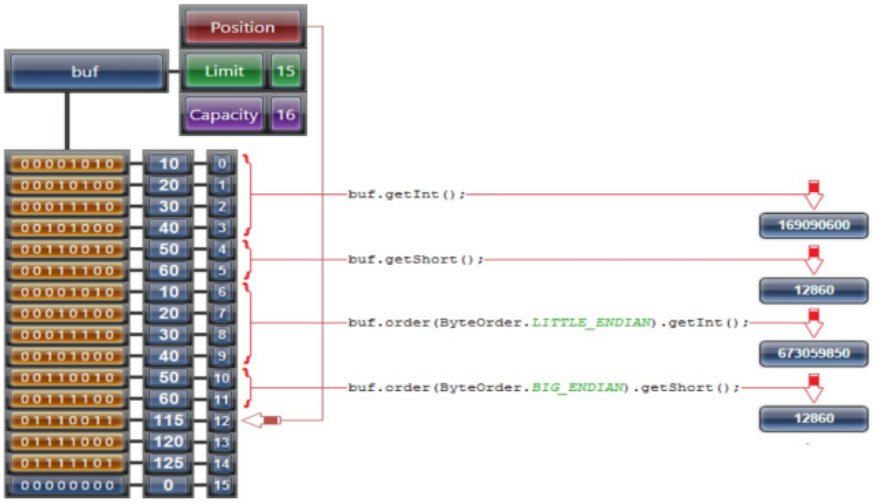


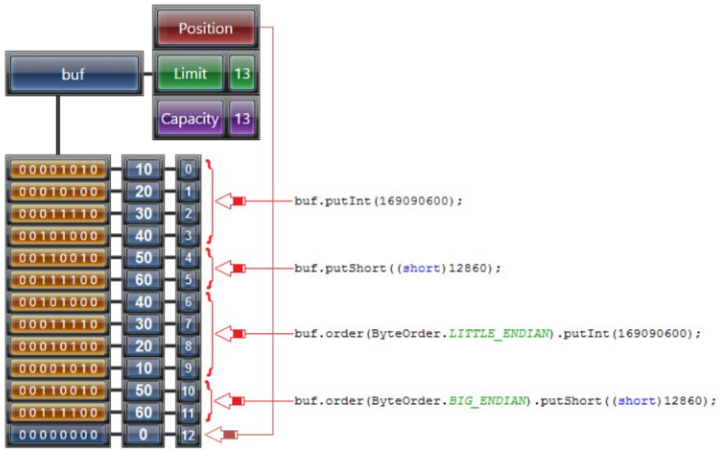
Figure 12: Altri metodi utili sono ***remaining()*** che restituisce il numero di elementi nel buffer compresi tra *position e limit* e ***hasRemaining()*** che restituisce true se remaining() è maggiore di 0.

4.5.3 Lettura e scrittura dati

Operare sui dati a livello di byte sarebbe troppo limitante per applicazioni complesse dunque si forniscono alcuni metodi per operare su tipi di dato primitivo. Ricordiamo che un intero è rappresentato da *4 byte*, uno short da *2 byte*. Il metodo in figura ***order*** consente di specificare anche l'ordine di lettura. L'invocazione di *byteBuffer.getInt()* estrae quattro bytes dal buffer iniziando dalla posizione corrente e li combina per comporre un intero a *32 bits*.



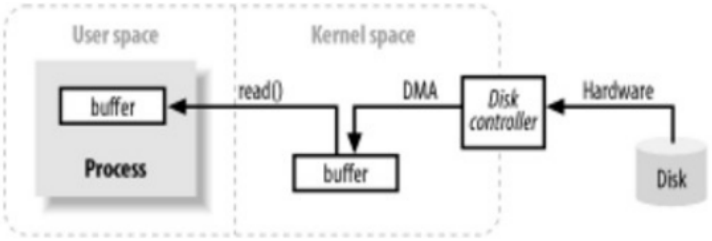
Analogamente alla lettura, vi è anche la scrittura che inserisce nel buffer un valore (*in relazione al tipo di dato primitivo*):



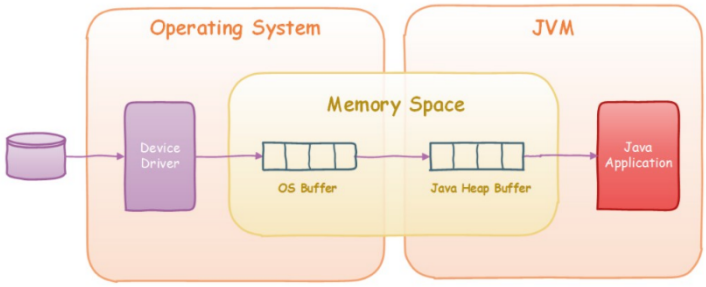
Riportiamo di seguito uno snippet che consente di analizzare le variabili di stato (*riportiamo in verde il valore delle variabili di stato*:

```
import java.nio.*;
public class Buffers {
public static void main (String args[])
{ByteBuffer byteBuffer1 = ByteBuffer.allocate(10);
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
byteBuffer1.putChar('a');
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=2 lim=10 cap=10]
byteBuffer1.putInt(1);
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=6 lim=10 cap=10]
byteBuffer1.flip();
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=0 lim=6 cap=10]

System.out.println(byteBuffer1.getChar());
System.out.println(byteBuffer1);
// a
// java.nio.HeapByteBuffer[pos=2 lim=6 cap=10]
byteBuffer1.compact();
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=4 lim=10 cap=10]
byteBuffer1.putInt(2);
System.out.println(byteBuffer1);
// java.nio.HeapByteBuffer[pos=8 lim=10 cap=10]
byteBuffer1.flip();
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
System.out.println(byteBuffer1.getInt());
 System.out.println(byteBuffer1);
// 1
// 2
// java.nio.HeapByteBuffer[pos=8 lim=8 cap=10]
byteBuffer1.rewind();
// rewind prepara a rileggere i dati che sono nel buffer, ovvero resetta
position a 0 e non modifica limit
// java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
System.out.println(byteBuffer1.getInt());
// 1
byteBuffer1.mark();
System.out.println(byteBuffer1.getInt());
// 2
System.out.println(byteBuffer1);
//position:8;limit:8;capacity:10
byteBuffer1.reset();
System.out.println(byteBuffer1);
//position:4;limit:8;capacity:10
byteBuffer1.clear();
```

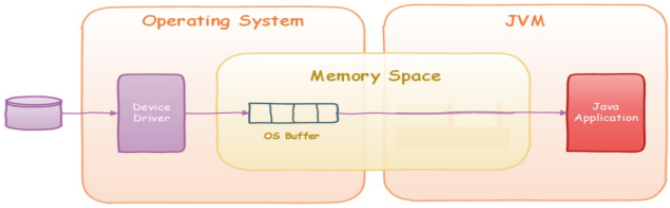
```
System.out.println(byteBuffer1);
//position:0;limit:10;capacity:10]]>
```



In figura riportiamo le interazioni tra **JVM** e **SO**: la JVM esegue una *read()* e provoca una system call in cui il kernel invia un comando al disk controller che a sua volta, tramite DMA (*senza supervisione o controllo della CPU* scrive direttamente un blocco di dati nel kernel space. In seguito i dati vengono copiati dal kernel space allo user spacce (*all'interno della JVM*. La gestione ottimizzata dei buffer comporta un notevole miglioramente delle performance e si può ottenere evitando la doppia copia dei dati (copia da dispositivo a disco e da disco a buffer processo).
I buffer classici, solitamente chiamati **Non-direct buffers** sono riportati in figura:

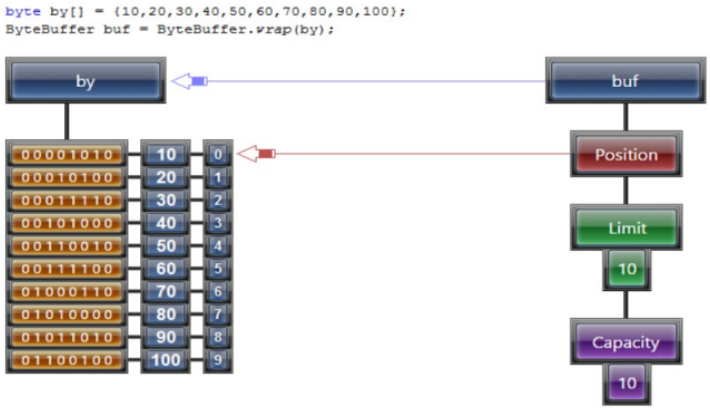


Tramite *ByteBuffer buf = ByteBuffer.allocate(10)* si crea sullo heap un oggetto Buffer che incapsula una struttura per memorizzare gli elementi e le variabili di stato. Genera dunque una doppia copia dei dati.



Tramite *ByteBuffer buf = ByteBuffer.allocateDirect(1024)* invece trasferisce

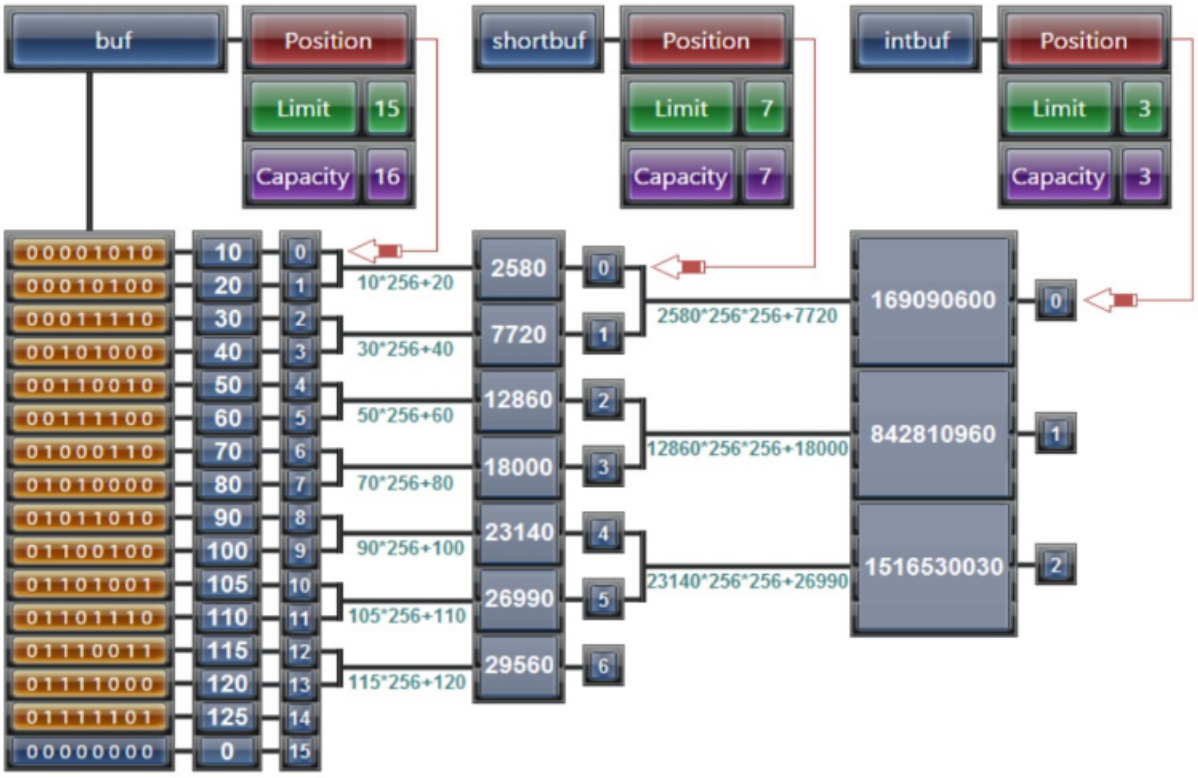
i dati tra il programma e il sistema operativo mediante accesso diretto alla kernel memory da parte della JVM e dunque evitando una copia di troppo dei dati. Tale approccio tuttavia comporta degli svantaggi tra cui un maggiore costo di allocazione/deallocazione (*delegare l'allocazione di memoria al SO potrebbe essere più costoso che allocare memoria sullo heap*) e impossibilità da parte del **Garbage Collector** di recuperare memoria poiché il buffer non è allocato sullo heap.



In figura *ByteBuffer.wrap()* è un metodo statico che crea un oggetto di tipo Buffer ma non alloca memoria per gli elementi. Poiché è un metodo *wrapping* utilizza un vettore precedentemente allocato come **backing storage** infatti l'oggetto Buffer è distinto dalla memoria utilizzata per i suoi elementi tuttavia ogni modifica del buffer è visibile nell'array e viceversa.

Java.NIO dispone anche di **view bufers** che sono buffer che consentono di caricare dati che non sono di tipo byte in un ByteBuffer prima di scriverlo su file o di accedere ai dati letti da un file come valori diversi da *byte grezzi* o *raw bytes*.
Generalmente si applicano in casi in cui si vuole leggere una sequenza di byte non come un intero ma come un float. Lo schema in figura si spera sia chiarificatore:

```
ByteBuffer buf = ByteBuffer.allocate(16);
buf.put(new byte[]{10,20,30,40,50,60,70,80,90,100,105,110,115,120,125});
buf.flip();
ShortBuffer shortbuf = buf.asShortBuffer();
IntBuffer intbuf = buf.asIntBuffer();
```



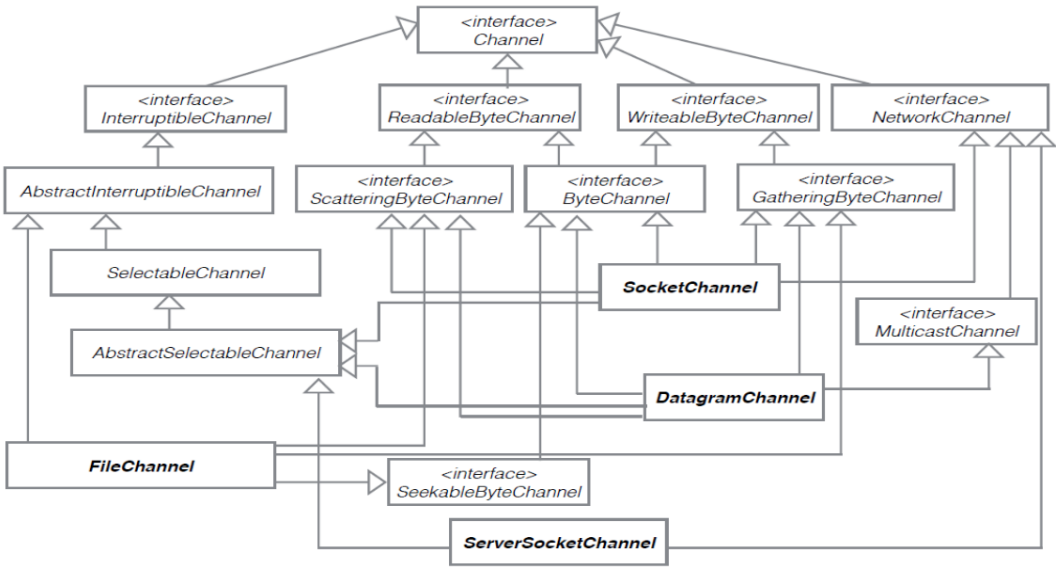
4.5.4 Channel

Un **Channel** è un oggetto che è connesso un descrittore di file/socket gestiti dal SO. L'API per i Channel utilizza principalmente interfacce Java le cui implementazioni utilizzano principalmente codice nativo.

Channel è una interfaccia che è radice una gerarchia di interfacce:

- FileChannel: legger/scrive dati su un File
- DatagramChannel: legge/scrive dati sulla rete via UDP
- SocketChannel: legge/scrive dati sulla rete via TCP
- ServerSocketChannel: attende richieste di connessioni TCP e crea un
- SocketChannel per ogni connessione creata.

Gli ultimi tre possono essere **non bloccanti**.



I **Channel** sono bidirezionali infatti lo stesso Channel può leggere dal dispositivo e scrivere sul dispositivo e dunque più vicini all'implementazione reale del SO. Come già accennato non si scrive/legge direttamente su un canale ma si passa da un buffer. I Channel non bloccanti sono utili per comunicazioni in cui i dati arrivano in modo incrementale (*tipiche dei collegamenti di rete* mentre i bloccanti sono utili in caso di lettura da file. E' inoltre possibile effettuare un trasferimento da Channel a Channel evitando l'uso di un Buffer intermedio ma solo nel caso in cui uno dei due è un FileChannel.

Oggetti di tipo *FileChannel* possono essere creati direttamente utilizzando FileChannel.open (di JAVA.NIO.2) dichiarando il tipo di accesso al channel (READ/WRITE) mentre in JAVA.NIO esisteva una operazione più complessa.

I **FileChannel** sono bloccanti e thread-safe infatti alcune operazioni (come le read) possono essere eseguite parallelamente mentre altre vengono automaticamente serializzate.

Vediamo uno snippet di codice che effettua la copia di un file:

```
importimportimportimportimportjava.nio.ByteBuffer;
java.nio.channels.ReadableByteChannel;
java.nio.channels.WritableByteChannel;
java.nio.channels.Channels;
java.io.*;

public class ChannelCopy{
    public static void main (String [] argv) throws IOException{
        ReadableByteChannel source = Channels.newChannel(new
            FileInputStream("in.txt"));
        WritableByteChannel dest = Channels.newChannel (new
            FileOutputStream("out.txt"));
        channelCopy1 (source, dest);
        source.close();
        dest.close();
}

private static void channelCopy1 (ReadableByteChannel src,
     WritableByteChannel dest) throws IOException{

    ByteBuffer buffer = ByteBuffer.allocateDirect (16 * 1024);
    while (src.read (buffer) != -1) {
        // prepararsi a leggere i byte che sono stati inseriti nel buffer
        buffer.flip();

        // scittura nel file destinazione; pu essere bloccante
        dest.write (buffer);

        // non   detto che tutti i byte siano trasferiti, dipende da
            quanti
        // bytes la write ha scaricato sul file di output
```

```
        // compatta i bytes rimanenti all'inizio del buffer
        // se il buffer  stato completamente scaricato, si comporta come
            clear(
        buffer.compact();
        }
    // quando si raggiunge l'EOF,  possibile che alcuni byte debbano
        essere ancor
    // scritti nel file di output
    buffer.flip();
    while (buffer.hasRemaining()) { dest.write (buffer); }
}}
```

Per lo snippet soprastante:

- read(): può non riempire l'intero buffer, limit indica la porzione di buffer riempita dai dati letti dal canale e restituisce -1 quando i dati sono finiti (EOF)
- flip(): converte il buffer da modalità scrittura a modalità lettura
- write(): preleva alcun dati dal buffer e li scarica sul canale. Non necessariamente scrive tutti i dati presenti nel Buffer
- hasRemaining(): verifica se esistono elementi nel buffer nelle posizioni comprese tra position e limit

4.5.5 Stream vs Buffer

Dato il seguente testo in un file:

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

Supponiamo che un server debba elaborare le linee di codice precedenti provenienti da una connessione. La soluzione con l'uso dello **Stream** sarebbe:

```
    InputStream input = ... ; // get the InputStream from the client
        socket
    BufferedReader reader = new BufferedReader(new
        InputStreamReader(input));
    String nameLine = reader.readLine();
    String ageLine = reader.readLine();
    String emailLine = reader.readLine();
    String phoneLine = reader.readLine();
```

La *reader.readLine()* quando restituisce il controllo al chiamante, una linea di testo è stata letta e si blocca fino a che la linea è stata completamente letta.

Ad ogni passo il programma sa quali dati sono stati letti e dopo aver letto dei dati non si può tornare indietro nello stream.

La soluzione con l'uso dei **Buffer**:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Quando la *read* restituisce il controllo non è detto che siano stati letti tutti i byte necessari per comporre una linea di testo (*potrebbero essere stati letti solo i dati relativi a "Nome: An"*). Nel caso dei Channel dunque l'applicazione deve verificare se sono stati letti abbastanza dati verificando ripetutamente tale condizione come si vede nello snippet:

```
ByteBuffer buffer = ByteBuffer.allocate(48);

int bytesReader = inChannel.read(buffer);

while(!bufferFull(bytesRead)){
    bytesRead = inChannel.read(buffer);
}
```

Se il buffer è pieno allora si procedere all'elaborazione mentre, in caso contrario, si può decidere di elaborare o meno i dati letti controllando iterativamente lo stato del buffer.

4.6 JSON: JavaScript Object Notation

Estendendo il concetto di serializzazione è evidente come non vincolare chi scrive/legge ad usare lo stesso linguaggio ne aumenti l'interoperabilità delle operazioni. Tra i formati più diffusi per la serializzazione dei dati tra linguaggi/macchine diversi vi sono sicuramente **JSON** e **XML**.
JSON è un formato nativo di Javascript che ha il vantaggio di essere espresso con una sintassi semplice e facilmente riproducibile. E' utilizzato per l'interscambio di dati indipendentemente dalla piattaforma poiché è testo scritto in notazione JSON. Si basa su due strutture:

- **coppie (chiave:valore)** in cui i tipi di dato ammissibili per i valori sono *String, Number, object, Array, Bollean o null.*
- **liste ordinate di valori**

Come vedremo, una risorsa JSON ha una struttura ad albero in cui le due principali strutture dati che appaiono sono *Array* e *Object*.

Un **array** in JSON è una raccolta ordinata di valori delimitato da parentesi quadre ed i valori separati da virgola. Tali strutture possono essere annidate.

```
["Ford", "BMW", "Fiat"]
```

Un **oggetto** in JSON è invece una serie non ordinata di coppie (nome,valore), delimitato da parentesi graffe e coppie separate da virgole. UN JSON object può contenere un altro JSON Object.

```
    {
        "name": "John",
        "age": 30,
        "cars": {
            "car1": "Ford",
            "car2": "BMW",
            "car3": "Fiat"
        }
}
```

Per la serializzazione e la deserializzazione si tratta una libreria esterna chiamata **Jackson** che consente di serializzare/deserializzare oggetti da Java a JSON e viceversa.

Il metodo principale utilizzato da Jackson è ***ObjectMapper*** che prende in ingresso un file o una stringa JSON e crea un oggetto o un grafo di oggetti. I metodi *writeValue() e readValue()* consentono di convertire oggetti Java a/da JSON quando si conosce la classe a cui si vuole associare il contenuto JSON mentre *readTree()* si utilizza quando non si conosce il tipo esatto di oggetto e il parsing restiuisce oggetti *JsonNode*. Gli esempi riportati sotto si sperano siano chiarificatori. Ipotizziamo di avere la struttura dati sotto riportata:

```
{
    "name": "Italia",
    "population": 54000000,
    "regions": ["Toscana", "Sicilia", "Veneto"]
}
```

Vediamo lo snippet che ci consente di serializzare l'oggetto da Java a JSON:

```
//Classe che rappresenta la nazione e relativi metodi
import java.util.ArrayList;
public class Country {
    private String name;
    private int population;
    private final ArrayList<String> regions = new ArrayList<String>();

    public String getName(){ return name; }

    public void setName(String name){ this.name = name; }

    public void setPopulation(int population){
        this.population =population;
    }
```

```
    public int getPopulation(){ return population; };

    public void addRegion(String region){
        this.regions.add(region);
    }

    public ArrayList<String> getRegions()
    { return regions; };
}

//######## JAVA TO JSON

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JSONFileCreator2{

    public static void main(String[] args) {

    ObjectMapper objectMapper = new ObjectMapper();

    Country countryObj = new Country();
    countryObj.setName("Italia"); countryObj.setPopulation(54000000);
    countryObj.addRegion("Toscana"); countryObj.addRegion("Sicilia");
    countryObj.addRegion("Veneto");

    try {
        File file=new File("RegionFileJackson.json");
        file.createNewFile();

        System.out.println("Writing JSON object to file");
        System.out.println("-----------------------");

        //Serializza l'oggetto
        objectMapper.writeValue(file, countryObj);
        //FileWriter fileWriter = new FileWriter(file);
        // in alternativa
        //objectMapper.writeValue(fileWriter, countryObj);
        //fileWriter.close();

        System.out.println("Writing JSON object to string");
        System.out.println(objectMapper.writeValueAsString(countryObj));
    }
    catch (IOException e) {

    e.printStackTrace();

    }
```

```
}
```

L'output prodotto è:

```
   Writing JSON object to file
   -----------------------
 Writing JSON object to string
 {
   "name":"Italia",
   "population":54000000,
   "regions": ["Toscana","Sicilia","Veneto"]
 }
```

Mentre lo snippet per deserializzare è il seguente:

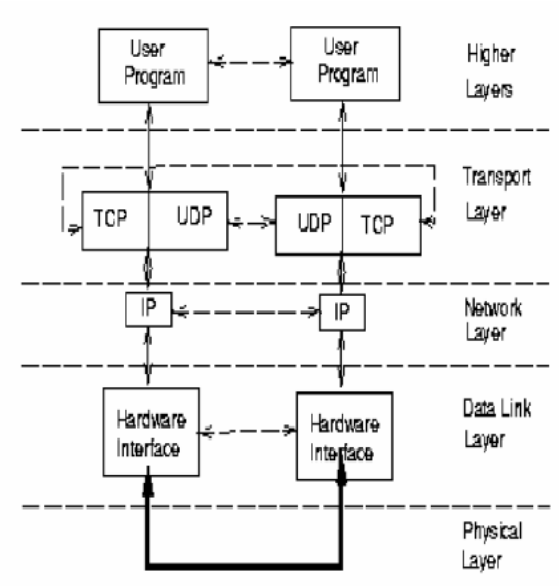
```
//######## JSON TO JAVA

import java.io.File;
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JsonFileReader {
    public static void main(String[] args) {

    ObjectMapper objectMapper = new ObjectMapper();
    File file=new File("RegionFileJackson.json");

    Country newCountry;

    try {
        newCountry = objectMapper.readValue(file, Country.class);
            //Classe Country utilizzata nella serializzazione
        System.out.println("Deserialized object from JSON");
        System.out.println("------------------------");
        System.out.println("Country name " + newCountry.getName() +
            newCountry.getPopulation());
        System.out.println("Country regions " + newCountry.getRegions());
    }
    catch (IOException e) { e.printStackTrace(); }
    }
}
```

5 Network Applications

In una applicazione di rete due o più **processi** (*non thread!*) in esecuzione su host diversi, distribuiti geograficamente sulla rete, comunicano e cooperano per realizzare una funzionalità globale. Il concetto di **cooperazione** consiste nello scambio di informazioni utili per perseguire l'obiettivo globale e dunque implica la **comunicazione** ovvero l'utilizzo di protocolli, l'insieme di regole che i partners devono seguire per comunicare correttmente.

Durante il corso utilizzeremo i protocolli di livello di trasporto di due tipi:

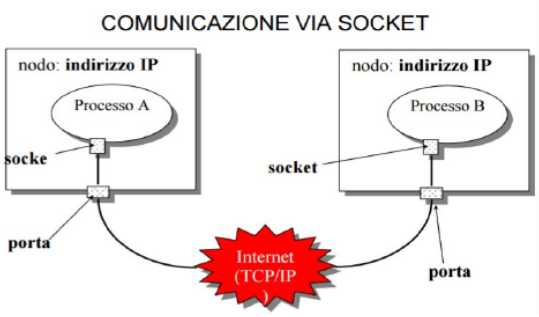
- **connection-oriented**: TCP, Trasmission Control Protocol
- **connectionless**: UDP, User Datagram Protocol



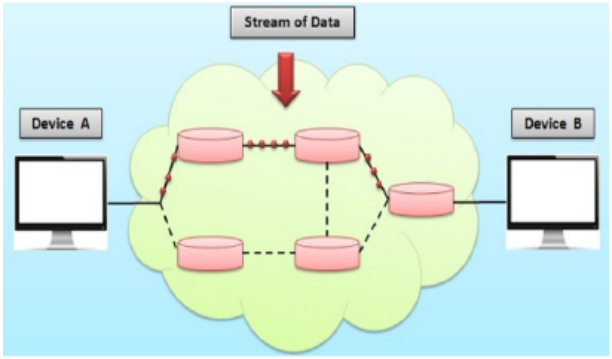
5.1 Socket

Il socket rappresenta uno standard per connetter dispostivi distribuiti, diversi ed eterogenei. Rappresenta di fatto una *presa "standard"* a cui un processo si può collegare per spedire dati ovvero un endpoint sull'host locale di un canale di comunicazione da/verso altri hosts. Il socket, creato dal relativo processo, è associato ad una porta e tramite il protocollo TCP/IP consente la comunicazione tra entità distinte.

Distinguiamo i due tipi di comunicazione tramite socket in:

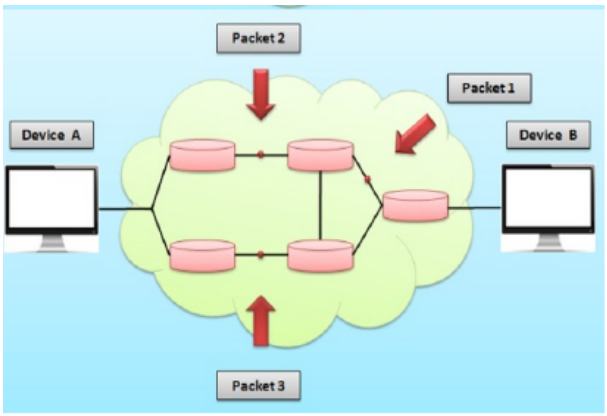


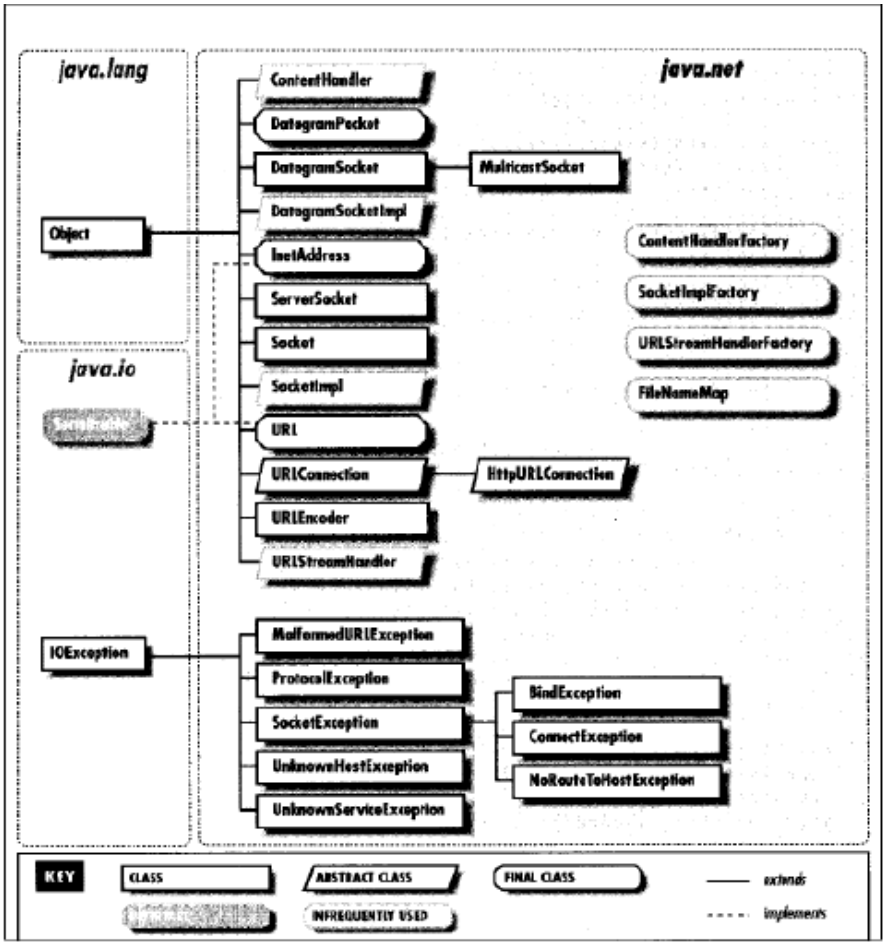
- **Connection Oriented (TCP)**: una connessione stabile (*canale di comunicazione dedicato* tra mittente e destinatario. (*stream socket* in Java). In questo caso l'indirizzo del destinario (*composto da IP+porta*) è specificato al momento dell'apertura della connessione. L'ordinamento è garantito (*ordine di invio-ricezione*) e permette la tramissione di grosse moli di dati.



- **Connectionless (UDP)**: non si stabilisce un canale di comunicazione dedicato e dunque ogni messaggio viene instradato in modo indipendente dagli altri (*datagramsocket* in Java).
  In questo caso l'indirizzo del destinario (*composto da IP+porta*) viene specificato per ogni pacchetto. L'ordinamento dei dati scambiati non è garantito da alcun meccanismo e generalmente la reliability non è essenziale e dunque si privilegia la velocità di tramissione piùttosto che l'affidabilità.

Per i socket **connection oriented** in Java, seguendo la gerarchia delle classi si evince che la connessione è modellata come stream, asimmetrici e fanno uso lato client della classe *Socket* mentre server side si fa uso delle classi *ServerSocket* e *Socket*. Dualmente, i socket **connectionless** fanno uso della classe *datagramSocket* simmetrici sia per il client che per il server. Tali classi sono contenute nel package ***java.net***.


Packet 2
Packet 1
Device A
Device B
Packet 3


java.lang
java.net
java.io
ContentHandler
DatagramPacket
DatagramSocket
MulticastSocket
DatagramSocketImpl
Object
InetAddress
ContentHandlerFactory
ServerSocket
SocketImplFactory
Socket
URLStreamHandlerFactory
SocketImpl
FileNameMap
URL
URLConnection
HttpURLConnection
URLEncoder
URLStreamHandler
IOException
MalformedURLException
ProtocolException
BindException
SocketException
ConnectException
UnknownHostException
NoRouteToHostException
UnknownServiceException
KEY
CLASS
ABSTRACT CLASS
FINAL CLASS
extends
INFREQUENTLY USED
implements

5.2 Indirizzi

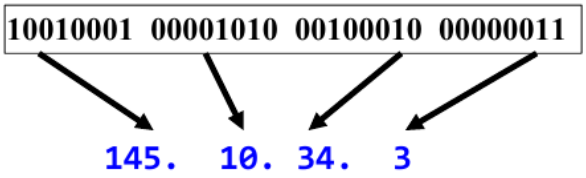
Prima di approcciare l'implementazione di quanto visto, è necessario specificare le motodologie di comunicazione tra processi. Identificare un processo con cui comunicare consiste nell'ottenere le seguenti informazioni:

- la rete all'interno della quale si trova l'host su cui è in esecuzione dil processo
- l'host all'interno della rete
- il processo in esecuzione sull'host

Sia la rete che l'host sono identificati dall'**Internet Protocol** tramite indirizzi IP mentre il **processo** è identificato da una porta, rappresentata da un intero da 0 a 65535. Ogni comunicazione è individuata da una quintupla composta da:

- protocollo (*TCP o UDP*)
- l'indirizzo IP del computer locale (*client example.it, 139.130.118.5*)
- porta locale (*ex: 5101*)
- indirizzo del computer remoto (*server infn.it, 139.130.118.102*)
- porta remota (*ex: 5100*)

Ovvero *tcp,139.130.118.5, 5101, 139.130.118.102, 5100.*
La struttura di un indirizzo IPV4 è la seguente:



- 4 bytes: ognugno interpretato come un numero decimale senza segno
- valore di ogni byte: 0 a 255
- 2 alla 32 indirizzi
- address special blocks in IPV4: 127.0.0.1 (loopback address) e 255.255.255.255 (broadcast)

In IPV6 vi sono 16 bytes per un totale di 2^{128} indirizzi.
Gli indirizzi IP semplificano l'elaborazione effettuata dai routers ma sono poco leggibili per gli utenti della rete. Una soluzione è quella di assegnare un **nome**

un indirizzo IPV6
FE80:0000:0000:0000:02A0:24FF:FE77:4997

simbolico unico ad ogni host della rete, utilizzando uno spazio di nomi gerarchico in cui i livelli della gerarchia sono separati dal punto (*ex: fujih0.cli.di.unipi.it*). Come visto nel modulo di teoria, è il Domain Name Space che traduce nomi in indirizzi IP.

Una porta è di fatto un numero che identifica un servizio in esecuzione su un host: ogni servizio viene incapsulato in un diverso processo. Il numero di porta è compreso tra 1 e 65535 (rispettivamente per ogni protocollo di trasporto) in cui le porte da 1 a 1023 sono riservate per **well-kwnown services** infatti per servizi terzi si usano valori di porta maggiori di 1024.

5.3 Classe InetAddress

In Java rappresenta ad alto livello un indirizzo IP con un oggetot di tipo *InetAddress* contenente due attributi quali *String hostName* che rappresenta il nome simbolico dell'host e *byte[] address* ovvero un vettore di bytes che rappresenta l'indirizzo IP dell'host.
La classe non ha costruttire ma tre metodi statici che definisco una factoru per costruire oggetti di tipo InetAddress:

```
//1
public static InetAddress.getByName (String hostname) throws
    UnKnownHostException
```

Il primo dei tre metodi cerca l'indirizzo IP corrispondente all'host il cui nome è passato come parametro e restituisce un oggetto di tipo InetAddress (composto come sopra visto). Per restituire l'oggetto InetAddress contenenete l'indirizzo IP richiede l'interrogazione del DNS per risolvere il nome dell'host che deve necessariamente essere connesso in rete e può utilizzare alternativamente cache locale. Se non riesce a risolvere il nome dell'host solleva UnknownHostException e attua il **reverse lookup** in cui passo un indirizzo IP come parametro nella forma dotted quad.

```
//2
public static InetAddress [ ] getAllByName (String hostname) throws
    UnKnownHostException

//3
public static InetAddress getLocalHost () throws UnKnownHostException
```

Il seguente snippet dato il nome dell'host restituisce l'indirizzo IP associato all'host dopo l'interrogazione al DNS o, prioritariamente, tramite l'ispezione in cache locale:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
public class example {
public static void main(String[] args) throws UnknownHostException
{
// print the IP Address of your machine (inside your local network)
System.out.println(InetAddress.getLocalHost().getHostAddress());
// print the IP Address of a web site
System.out.println(InetAddress.getByName("www.java.com"));
// print all the IP Addresses that are assigned to a certain domain
InetAddress[ ] inetAddresses=InetAddress.getAllByName("www.amazon.com");
for (InetAddress ipAddress : inetAddresses)
{ System.out.println(ipAddress);}
}}
```

Come tutte le classi anche *InetAddress* eredita da *java.lang.Object* ed effettua l'overriding dei 3 metodi base di *Object*:

- ***equals()***: due oggetti InetAddress sono uguali se solo se hanno stesos indirizzo IP ma non necessariamente devono avere stesso hostname
- ***HashCode()***: converte 4 bytes dell'indirizzo IP In un int e coerentemente con equals non considera il valore dell'hostname
- ***toString()***: restituisce nome dell'host/indirizzo dotted quad (*se non esiste il nome restituisce stringa vuota + indirizzo dotted quad*).

5.4 Address Caching

L'accesso al DNS è una operazione potenzialmente molto costosa e il vantaggio del caching, tra i molteplici elencabili, risiede nella permanenza delle traduzioni per i tentativi di risoluzione falliti che vanno in cache. I dati permangono in cache *10 secondi* se la risoluzione non ha avuto successo, spesso il primo tentativo di risoluzione fallisce a causa di un time-out. E' possibile settare un tempo illimitato che tuttavia comporta delle ovvie problematiche in caso di **indirizzi dinamici**.
Il metodo *java.security.Security.setProperty* imposta il numero di secondi in cui un entrata della cache rimane valida (*ad esempio: java.security.Security.setProperty ("networkaddress.cache.ttl","0");*) mentre per i tentativi non andati a buon fine *networkaddress.cache.negative.ttl*.

Nel seguente snippet mettiamo a confronto i tempi di permanenza nel caso in cui il *caching time* sia 0 e 1000:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.security.*;

public class Caching {
```

```
    public static final String CACHINGTIME="0";
    public static void main(String [] args) throws InterruptedException{

        Security.setProperty("networkaddress.cache.ttl",CACHINGTIME);
        long time1 = System.currentTimeMillis();

        for (int i=0; i<1000; i++){
            try {
                System.out.println(
                InetAddress.getByName("www.cnn.com").getHostAddress());

            }catch (UnknownHostException uhe)
                 { System.out.println("UHE");}
        }

        long time2 = System.currentTimeMillis();
        long diff=time2-time1; System.out.println("tempo trascorso
            e'"+diff);
    }
}
```

Per **CACHINGTIME=0** il tempo trascorso è 545, mentre per **CACHINGTIME=1000** il tempo trascorso è 85.

5.5 Client/Server Paradigm

Un servizio è un software in esecuzione su una o più macchine e fornisce l'astrazione di un insieme di operazioni. Dualmente, il client è un software che sfrutta servizi forniti dal server (*nel caso di un web-client un esempio è il browser o nel caso di un email-client può essere un mail-reader*). Di fatto un server è un'istanza di un particolare servizio in esecuzione su un host.

Nello schema in Figura 10 si evidenzia la struttura generale di come gestire una socket TCP in Java: il **welcome-socket** è implementato tramite la classe *ServerSocket*. Il client tenta di collegare, al passo due, il server socket. Il tentativo di apertura da parte del client implica l'uso del **three-way handshake**: se con esito positivo allora il server crea una nuova socket dedicata alla comunicazione con il client che ha appena accettato al fine di liberare la welcome socket per altri eventuali richieste da ulteriori client. La comunicazione vera e propria è attuata tramite la connection socket. La socket verde è la **local socket** (*accetta connessioni + handshake*). In rosso la socket del client, in giallo la socket creata automaticamente dal server per la implementare lo scambio dati.

Dunque esistono due tipi di socket TCP lato server:

- **welcome (passive o listening) socket**: utilizzati dal server per accettare le richieste di connessione
- **connection (active) sockets**: supportano lo streaming di byte tra client e server

Figure 13: Paradigma client-server con layers

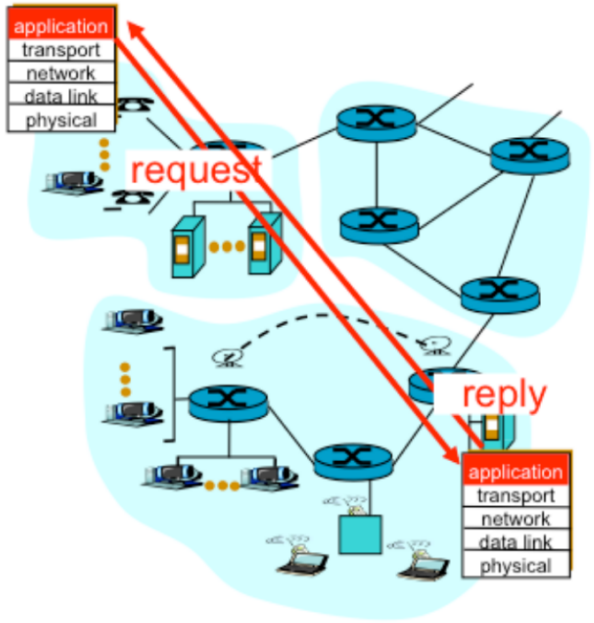
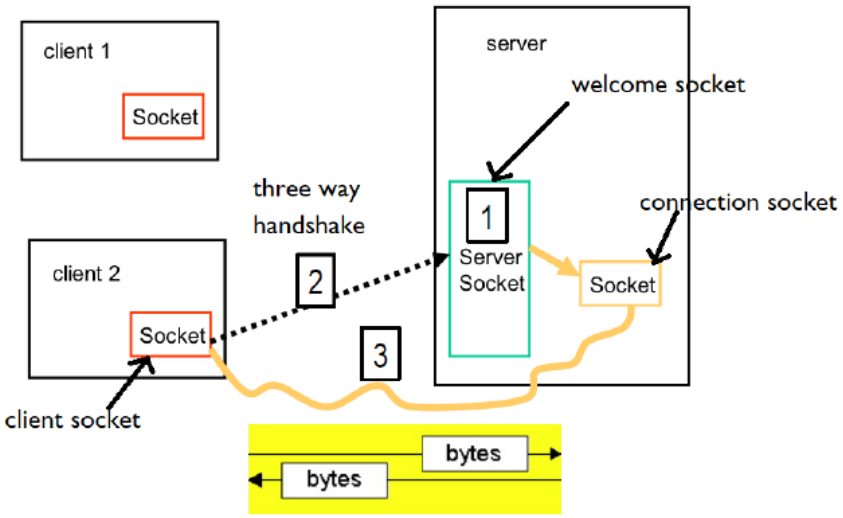


Figure 14: Struttura generale gestione socket TCP



Dunque il client crea un **active socket** per richiedere la connessione e quando il server accettare una richiesta di connessione crea a sua volta un proprio **active socket** che rappresenta il punto terminale della sua connessione con il client.

La comunicazione vera e propria avviene mediante la coppia di active socket presenti nel client e nel server.
Per collegarsi ad un servizio il server pubblica un proprio servizio associato al **listening socket**, creato sulla porta *remote PR*, all'indirizzo *IP SA* (*usa un oggetto di tipo ServerSocket*). Il client che intende usufruire del servizio deve conoscere l'indirizzo IP del server e la *porta remota PR* a cui è associato il servizio (*usa un oggetto di tipo Socket*). La creazione del socket effettuata dal client produce in **modo atomico** la richiesta di connessione al server: se la richiesta viene accettata il server crea un socket dedicato per l'interazione con il client e tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

5.5.1 Java Stream Socket API: Client side

La classe *java.net.Socket* ha i seguenti costruttori:

- *public socket(InetAddress host, int port) throws IOException*: crea un **active socket** e enta di stabilire, tramite esso, una connessione con l'host indivudato da *InetAddress* sulla porta *port*. Se la connessione viene rifiutata lancia invece una eccezione di IO.
- *public socket(String host, int port) throws UnkwnownHostException, IOException*: analoga alla precedente, l'host è individuato tuttavia dal suo nome simbolico (*interroga automaticamente il DNS*).
- *public Socket (String H, int P, InetAddress IA, int LP)*: tenta di creare una connessione verso l'host H sulla porta P, dall'interfaccia locale IA e dalla porta locale LP.

Il seguente snippet funge da **port scanner**, individuando i servizi su un server: il client richiede un servizio tentando di creare un socket su ognuna delle prime 1024 porte di un host e se non viene rilevato alcun servizio il socket non viene creato e viene sollevata un'eccezione.
Il programma effettua 1024 interrogazioni al DNS, una per ogni socket che tenta di creare: è possibile tuttavia ottimizzare il comportamento del programma utilizzando il costruttore *public Socket(InetAddress host, int port) throws IOException* in cui il DNS viene interrogato una sola volta, prima di entrare nel ciclo di scanning dalla ***InetAddress.getByName*** e viene utilizzato l'InetAddress invece del nome dell'host per costruire i socket.

```
import java.net.*;
import java.io.*;

public class PortScanner {

    public static void main(String args[ ]){
        String host;
        try {
```

```
            host = args[0];
        }
        catch (ArrayIndexOutOfBoundsException e) {host="localhost";};
            for (int i = 1; i< 1024; i++){

                try{
                    Socket s = new Socket(host, i);
                    System.out.println("Esiste un servizio sulla porta"+i);
                }
                    catch (UnknownHostException ex)
                           {System.out.println("Host Sconosciuto");
                               break; }
                    catch (IOException ex)
                           {System.out.println("Non esiste un servizio
                               sulla porta"+i);}
            }
        }
}
```

5.5.2 Java Stream Socket API: Server side

La classe *java.net.ServerSocket* ha i seguenti costruttori:

- *public ServerSocket(int port)throws BindException, IOException*
- *public ServerSocket(int port,int length) throws BindException,IOException*: costruisce un listening socket, associandolo alla porta p. Il parametro *length* indica la lunghezza della coda in cui vengono memorizzate le richieste di connessione e se la coda risulta piena ulteriori richieste di connessione sono rifiutate.
- *public ServerSocket(int port,int length,Inetaddress bindAddress)*: permette di colelgare il socket ad uno specifico indirizzo IP locale. Risulta utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi IP (*uno visibile da internet, l'altro interno a livello di rete locale*). Se voglio servire solo le richieste in arrivo dalla rete locale, associo il *connection socket* all'indirizzo IP locale.

Per accettare una nuova connessione dal ***connection socket*** si fa uso del metodo della classe *ServerSocket*:

```
public Socket accept( ) throws IOException
```

Quando il processo server invoca il metodo *accept()*, pone il server in attesa di nuove connessioni. Se non ci sono richieste, il server si blocca (*è possibile l'uso di time-out*). Se invece c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione effettiva tra client e server.

Il seguente snippet effettua un port scanner lato server, ricercando dei servizi attivi sull'host locale:

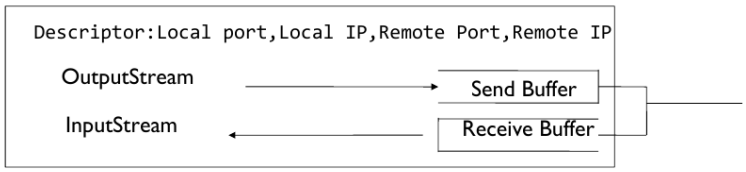
```
import java.net.*;
public class LocalPortScanner {
    public static void main(String args[]){

        for (int port= 1; port<= 1024; port++){
            try{
                ServerSocket server = new ServerSocket(port);
            }catch (BindException ex)
                    {System.out.println(port + "occupata");}
            catch (Exception ex)
                    {System.out.println(ex);}
            }
        }
}
```

5.5.3 Stream Based Communication

Una volta stabilita una connessione tra client e server è possibile iniziare lo scambio di dati: vediamo come modellare una connessione tramite uno stream. In particolare si associa uno stream di input o di output ad una socket:

```
public InputStream getInputStream ( ) throws IOException
public OutputStream getOutputStream ( ) throws IOException
```

Per l'invio di dati client/server leggono/scrivono dallo/sullo stream un byte o una sequenza di byte. I dati sono strutturati o in forma di oggetti e dunque in tal caso è necessario associare dei filtri agli stream. Ogni valore scritto sullo stream di output associato al socket viene copiato nel **Send Buffer** del livello TCP mentre ogni valore letto dallo stream viene prelevato dal **Receive Buffer** dal livello TCP.
Dopo che la richiesta di connessione è accetata, client e server associano stream di bytes di input/output all'active socket poiché gli stream sono unidirezionali. Si avrà uno stream di input e di output per lo stesso socket e la comunicazione avverrà tramite letture/scritture di dati sullo stream, come nello schema in figura. Il seguente snippet rende l'idea del ciclo di vita tipico di un server:



Struttura del Socket TCP

```
    // instantiate the ServerSocket
    ServerSocket servSock = new ServerSocket(port);

    while (! done) // oppure while(true) {
        // accept the incoming connection
        Socket sock = servSock.accept();

        // ServerSocket is connected ... talk via sock
        InputStream in = sock.getInputStream();
        OutputStream out = sock.getOutputStream();

        //client and server communicate via in and out and do their work
        sock.close();
    }

    servSock.close();
```

Nello snippet precednete la fase di *"communicate and work"* può essere eseguire in modo concorrente da più threads. Un thread per ogni cliente che gestisce le interazioni con quel particolare client e dunque lascia al server la possibilità di gestire le richieste in modo più efficiente.
Nonostante i threads siano **processi lightweigth** comunque causano un uso di risorse non indifferente. Tra le soluzioni per garantire l'efficienza nella gestione delle richieste vi è l'uso di *ServerSocketChannels* di Java.NIO o il *Thread pooling*.
Il seguente snippet è un esempio di programma client-server che dato del testo converte i caratteri da minuscolo a maiuscolo (capitalizer service):

```
// ########## SERVER - class Main
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.concurrent.*;

public static void main(String[] args) throws Exception {

    //try-with-resources
    try (ServerSocket listener = new ServerSocket(10000)) {

        System.out.println("The capitalization server is running...");
        ExecutorService pool = Executors.newFixedThreadPool(20);

        while (true) {
            pool.execute(new Capitalizer(listener.accept()));
        }
    }
}
```

```
//### class Capitalizer
private static class Capitalizer implements Runnable {
    private Socket socket;

    Capitalizer(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        System.out.println("Connected: " + socket);
        try (Scanner in = new Scanner(socket.getInputStream());
            PrintWriter out = new
                PrintWriter(socket.getOutputStream(),true)){

            while (in.hasNextLine()) {
                out.println(in.nextLine().toUpperCase());
            }
        } catch (Exception e) {
                System.out.println("Error:" + socket);
        }
    }
    }
}

// ########## CLIENT - class CapitalizeClient
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class CapitalizeClient {
    public static void main(String[] args) throws Exception {

        if (args.length != 1) {
            System.err.println("Pass the server IP as the sole command
                line argument");
            return;
        }

        Scanner scanner=null;
        Scanner in=null;
        try (Socket socket = new Socket(args[0], 10000)) {

            System.out.println("Enter lines of text then EXIT to quit");
            scanner = new Scanner(System.in);

            in = new Scanner(socket.getInputStream());
            PrintWriter out = new
                PrintWriter(socket.getOutputStream(),true);
```

```
            boolean end=false;
            while (!end) {

                String line= scanner.nextLine();
                if (line.contentEquals("exit")) end=true;

                out.println(line);
                System.out.println(in.nextLine());}
            }
        }
        finally {scanner.close(); in.close();}
    }
}
```

Nel caso in cui client e server non siano entrambi scritti in Java è sufficnete al fine di implementare lo scambio di dati rispettare il protocollo ed il formato dei dati scambiati, codificati in un formato interscambiabile (*testo, JSON, XML*).

Un esempio è dato dal servizio *time* offerto dal **NIST - National Institute of Standards and Technology** che consente di aprire una connessione sulla porta 13 verso il servizio *time.nist.gov* ottenendo informazioni su data, ora etc secondo un formato predefinito (*58048 17-10-22 14:01:15 15 0 0 667.9 UTC(NIST) **). Il seguente snippet implementa un client per una richiesta al servizio *time*:

```
public class TimeClient {
    public static void main(String[] args) {

        String hostname = args.length > 0 ? args[0] : "time.nist.gov";
        Socket socket = null;
        try {
                socket = new Socket(hostname, 13);
                socket.setSoTimeout(15000);

                InputStream in = socket.getInputStream();
                StringBuilder time = new StringBuilder();
                InputStreamReader reader = new InputStreamReader(in,
                    "ASCII");

                for (int c = reader.read(); c != -1; c = reader.read()) {
                        time.append((char) c); //caratteri che ricevo
                            appendo alla stringa time
                 }
                System.out.println(time);

        } catch (IOException ex) {
                 System.out.println(ex);
        } finally {
                if (socket != null) {
                try {
```

```
                socket.close();
            } catch (IOException ex) {
                // ignore
                }
            }
        }
    }
}
```

5.5.4 Time Protocol - RFC 868

Riportiamo un esempio dell'implementazione del servizio Time Protocol trattato nell'**RFC 868** (*Request for Comments 868*): può essere implementato sia su TCP che UDP. Il funzionamento è articolato nei seguenti passi:

- **Server**: ascolta sulla porta 37
- **Client**: si connette alla porta 37
- **Server**: invia il tempo come un numero binario a 32 bit
- **Client**: riceve il tempo
- **Client**: chiude la connessione
- **Server**: chiude la connessione

```
// ################## TIME PROTOCOL SERVER
import java.io.*;
import java.net.*;
import java.util.Date;
import java.nio.*;
public class TimeServer {
public final static int PORT = 6000;

public static byte[] longToBytes(long x) {

    ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
    buffer.putLong(x);

    return buffer.array();
}

import java.io.*; import java.net.*; import java.util.Date; import
    java.nio.*;
public static void main(String[] args) {

       // The time protocol sets the epoch at 1900,
       // the Date class at 1970. This number
```

```
        // converts between them.

        long differenceBetweenEpochs = 2208988800L;

        try (ServerSocket server = new ServerSocket(PORT)) {

                while (true) {
                    try (Socket connection = server.accept()) {

                        OutputStream out = connection.getOutputStream();
                        Date now = new Date();

                        long msSince1970 = now.getTime();
                        long secondsSince1970 = msSince1970/1000;
                        long secondsSince1900 = secondsSince1970 +
                            differenceBetweenEpochs;
                        byte[] time = new byte[8];

                        time = longToBytes(secondsSince1900);
                        out.write(time);

                        System.out.println(secondsSince1900);
                        out.flush();
                    } catch (IOException ex) {
                    System.out.println(ex.getMessage());
                    }
                }
        } catch (IOException ex) {
        }
        System.out.println(ex);

}



//    ################## TIME PROTOCOL CLIENT
import java.net.*;
import java.io.*;
import java.nio.*;
public class TimeClient {
    static public long bytesToLong(byte[] bytes) {

        ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES+10);
        buffer.put(bytes);

        buffer.flip();//need flip
        return buffer.getLong();
    }

    public static void main(String[] args) {
```

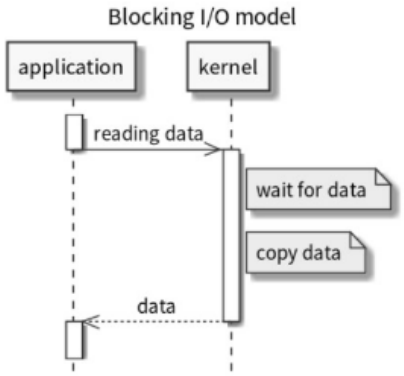
```
        String hostname = "Localhost";
        Socket socket = null;
        byte[] time = new byte[8];

        try {
            socket = new Socket(hostname, 6000);

            InputStream in = socket.getInputStream();
            int x = in.read(time);
            System.out.println(x);

            Long timeL=bytesToLong(time);
            System.out.println(timeL);

        } catch (IOException ex) { System.out.println(ex);}

        finally {
            if (socket != null) {
                try {
                socket.close();
                } catch (IOException ex) {
                // ignore
                }
            }
        }
    }
}
```

5.6 Channel Multiplexing

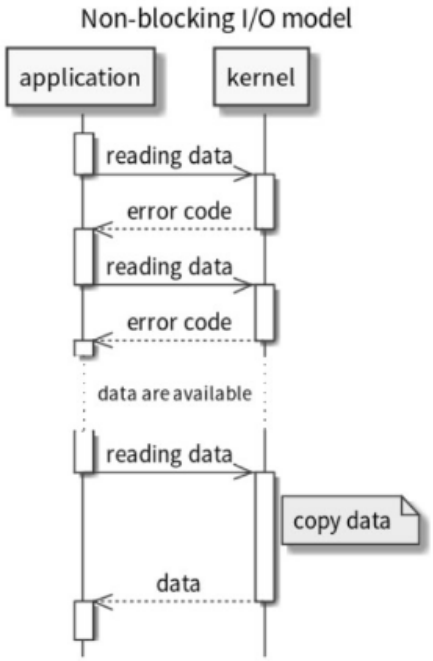
Tra gli obiettivi prefissi da Java NIO vi è la possibilità di sviluppare modalità non bloccanti per l'I/O e l'adozione di multiplexing in grado di sfruttare i vantaggi derivanti dall'uso di più processori.
Come visto precedentemente, per le **operazioni bloccanti su stream** l'applicazione esegue una chiamata di sistema e si blocca fino a che tutti i dati sono ricevuti nel kernel e copiato nel kernel space alla memoria della applicazione. Con l'invocazione del metodo *read()* si blocca (*il thread che esegue il metodo*) fino a quando non è stato letto un byte o un vettore di byte, un intero, etc. Il metodo *accept()* si blocca fino a che non viene stabilita una nuova connessione. Il metodo *write(byte[] buffer)* si blocca fino a che tutto il contenuto del buffer è stato copiato sulla periferica di I/O.

Dualmente, per le operazioni di **non-blocking I/O** la chiamata di sistema restiusice il controllo alla applicazione prima che l'operazione richiesta sia stata *"pienamente soddisfatta"*. Alcuni scenari possibili sono:

- restituiti i dati disponibili o una parte di essi;



- operazione di I/O non possibile (*assenza dati o periferica non raggiungibile*), restituisce codice errore o valore null;



Per completare l'operazione è possbile effettuare system-call ripetute fino a che l'operazione non può più essere effettuata.

E' possibile impelementare il **non-blocking I/O** se si associano i channels ai socket tramite i **SocketChannel**. Un channel associato ad un *socket TCP* consente la combinazione di un socket e canale di comunicazione bidirezionale,

permettendo la scrittura e la lettura da un socket TCP. Estende la classe *AbstractSelectableChannel* e da questa mutua la capacità di passare dalla modalità bloccante a quella non bloccante. Ogni *socketChannel* è associato ad un oggetto Socket della libreria *java.net* e il socket associato può essere reperito tramite invocazione del metodo *socket()*.
L'idea alla base è quella di non bloccarsi in attesa da parte di un client ma di gestire parallelamente più richieste e diverse operazione durante l'attesa del verificarsi di alcuni eventi, secondo il paradigma della programmazione ad eventi.
Un *SelectableChannel* è collegato ad un ***TCP welcome-socket*** (*listening sockets*): ad ogni **ServerSocketChannel** è associato ad un oggetto *ServerSocket* e può essere utilizzato in modalità **blocking** (*come ServerSocket, ma con interfaccia buffer-based*) o **non blocking** (*permette multiplexing di canali*). Vediamo uno snippet per chiarire le idee:

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
ServerSocket socket = serverSocketChannel.socket(); //restituisce socket
    associato al canale

socket.bind(new InetSocketAddress(9999)); //collega socket ad una porta
    locale, host di default localhsot
serverSocketChannel.configureBlocking(false); //i channel di default
    hanno configurazione bloccante a true: in questo modo lo settiamo
    in modalit non-blocking

   while(true){
       SocketChannel socketChannel = serverSocketChannel.accept();
           //restituisce un socketChannel e non un socket (come visto
           precedentemente)

       if(socketChannel != null){
       //do something with socketChannel...
       else //do something useful...
       }
   }
```

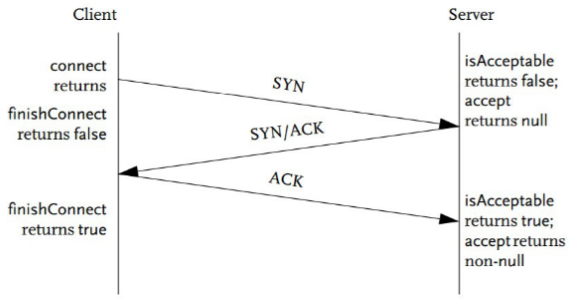
L'oggetto *InetSocketAddress* rappresenta l'indirizzo di un socket descritto da indirizzi IP e numero di porta. E' alternativo rispetto ad *InetAddress*. Uno dei vantaggi dell'uso dei *SocketChannel* è la possibilità di avere una comunicazione bidirezional mente con gli *StreamSocket* era necessario creare uno stream di input e uno di output.
Usando la modalità non bloccante i SocketChannel, come vedremo dopo, consentono il **multiplexing**: immaginiamo di avere 100 canali, registrando un selettore (*ne parleremo approfonditamente dopo*) posso ricevere notifiche quando determinate operazioni vengono eseguite su un determinato canale, attuando il monitoraggio dei canali.
Un ***SocketChannel*** è associato ad un oggetto di tipo *Socket* utilizzato per la comunicazione dei dati tra client e server. La creazione di un *SocketChannel*

può avvenire in maniera implicita se si accetta una connessione su un *ServerSocketChannel* (*come nello snippet sopra*) o esplicitamente, **lato client** quando si apre una connessione verso un server mediante una operazione di *connect()*, come nello snippet sotto riportato:

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("www.google.it", 80))
```

E' significativo settare la modalità **non-blocking** lato client, *ad esempio*, nel caso in cui un'applicazione che deve gestire l'interazione con l'utente tramite GUI e contemporaneamente deve gestire uno o più sockets.

Il metodo *connect* in modalità **non blocking** può restituire il controllo al chiamante prima che venga stabilita la connessione: tramite l'invocazione del metodo (**lato server**) *isAcceptable()* è possibile verificare quando l'operazione di **three-way handshake** è stata ultimata e dunque la connessione è stabilita. Dualmente, **lato client**, l'invocazione del metodo *finishConnect()* è utilizzato



per controllare la terminazione dell'operazione di three-way handshake. Un esempio banale è di seguito riportato:

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("www.google.it", 80));

    while(! socketChannel.finishConnect() ){
            //wait, or do something else...
    }
```

Se l'ultima fase del three-way handshake non è ancora stata completata, quando il client effettua una read tale chiamata restituirà 0 valori nel buffer. La rimozione del while con relativa condizione nello snippet sopra riportato solleva un'eccezione di tipo *java.nio.channels.NotYetConnectedException*.

Ricapitoliamo brevemente le varianti viste finora al fine di chiarire le più che confuse idee:

1. **Blocking accept**: si blocca finché non arriva una richiesta di connessione;

2. **Non-blocking accept**: controlla se c'è una richista da accettare (*se c'è accetta*) e ritorna;

3. **Blocking write**: si blocca finché la scrittura dei dati nel buffer non è completata

4. **Non-blocking write**: tenta di scrivere i dati nella socket, ritorna immediatamente, anche se i dati non sono stati completamente scritti

5. **Blocking read**: si blocca in attesa di byte da leggere;

6. **Non-blocking read**: ritorna immediatament e restituisce il numero di byte letti (*anche 0*).

5.6.1 Server models

I criteri di valutazione delle **prestazioni** di un server sono:

1. **scalability**: capacità di servire un alto numero di client che inviano richieste concorrentemente

2. **acceptance latency**: tempo tra l'accettazione di una richiesta da parte di un client e la successiva

3. **reply latency**: tempo richiesto per elaborare una richiesta ed inviare la relativa rispsota

4. **efficiency**: utilizzo delle risorse utilizzate sul server (*RAM, numero di thread, utilizzo della CPU*)

Valutando tali criteri nel caso di **un singolo thread per la gestione di tutti i client** porta a tali considerazioni per ogni metrica:

1. **scalability**: è nulla, in ogni istante un solo client viene servito

2. **acceptance latency**: è alta, il prossimo cliente deve attendere fino a che il primo client termina la connessione

3. **reply latency**: tutte le risorse a disposizione di un unico client

4. **efficiency**: buona, il server utilizza esattamente le risorse necessarie per il servizio dell'utente.

Tale metodo di implementazione è adatto nel caso in cui il tempo di servizio di un singolo utente è garantito rimanere basso.
Nello scenario di avere un **thread per ogni connessione**:

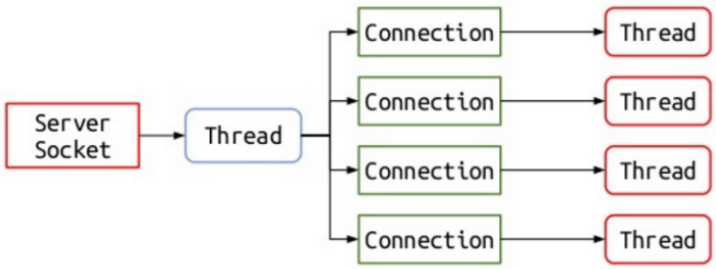
1. **scalability**:possibile servire diversi clienti in maniera concorrente, fino al massimo numero di thread previsti per ogni processo (*ogni thread alloca il proprio stack generando **memory pressure** e inoltre è impossibile predire il numero massimo di client e può dunque essere molto variabile*).

2. **acceptance latency**: tempo tra l'accettazione di una connessione e la successiva è in genere basso rispetto a quello di interarrivo delle richieste

3. **reply latency**: bassa, le risorse del server condivise tra connessioni diverse consentendo un uso ragionevole di CPU e RAM per centinaia di connessioni tuttavia se aumenta il tempo di reply può non essere accettabile

4. **efficiency**: bassa, ogni thread può essere bloccato in attesa di IO ma utilizza risorse come la RAM

Il modello generale di un thread per ogni connessione è riportato di seguito: si attiva un thread per ogni connessione, deattivazione e fine servizio. Ciò consente al server di monitorare un grande numero di comunicazioni che tuttavia impone la gestione di problemi di **scalabilità** poiché il tempo per il ***cambio di contesto*** può aumentare notevolmente con il numero di thread attivi e la maggior parte del tempo verrebbe impiegata in *context switching*. Nello schema, in blu si indica il *thread main*.



Analizziamo il modello in cui vi è un **numero costante di thread in un ThreadPool**:

1. **scalability**:limitata al numero di connessioni che possono essere supportate

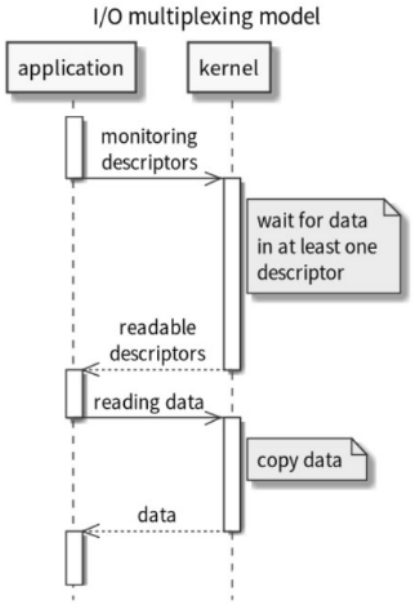
2. **acceptance latency**: bassa fino ad un certo numero di connessioni

3. **reply latency**: bassa fino al numero massimo di thread fissato, degrada se il numero di connessioni è maggiore

4. **efficiency**: trade-off rispetto al modello precedente

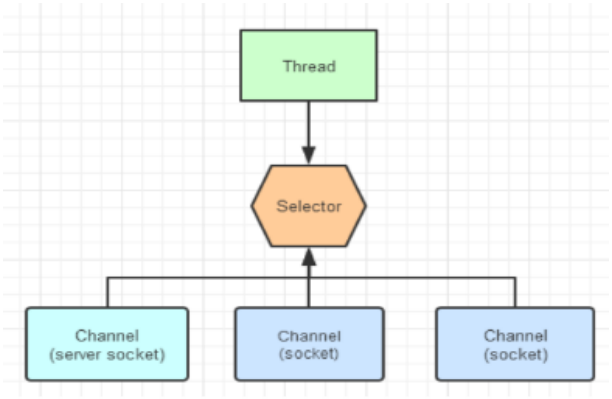
5.6.2 Multiplexed I/O

Il **multiplexed I/O** è anche definito **non blocking I/O con notifiche bloccanti** e l'idea alla base è quella di far registrare all'applicazione dei *"descrittori"* delle operazioni di I/O a cui è interessato. L'applicazione esegue una operazione di **monitoring dei canali** tramite una system call bloccante che restituisce il

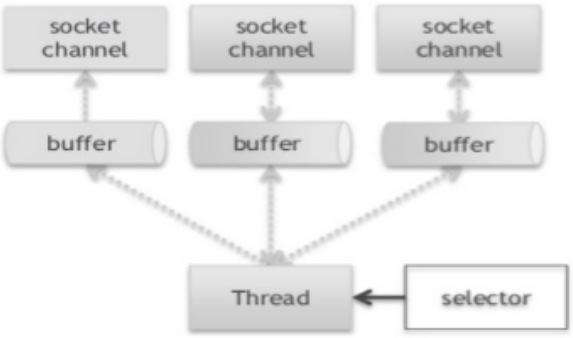


controllo quando almeno un descrittore indica che una operazione di I/O è *"pronta"*, a quel punto si effettua una read non bloccante.

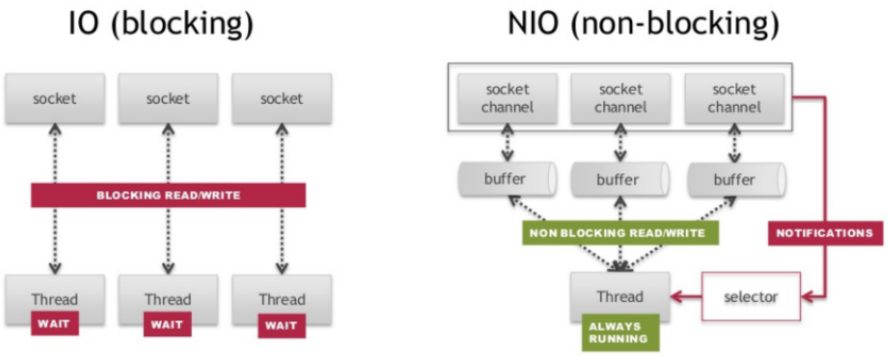
In Java un **selettore** (*classe Selector*) è un componente che esamina uno o più *NIO Channels* e determina quali canali sono pronti per leggere/scrivere. Possono essere gestire più connessioni di rete mediante un unico thread permettendo di ridurre il **thread switching overhead** e l'uso di risorse da thread diversi.



Analizziamo l'uso del *multiplexing* nel caso in cui si abbia **un solo thread** (*o comunque un numero ristretto/limitato di thread*): l'idea è quella di avere un singolo thread che gestisce un numero arbitrario di sockets. Non vi è in questo



scenario un thread per connessione ma un numero ridotto di threads (*numero di thread basso anche con migliaia di sockets o anche un solo thread*) che garantiscono un miglioramente di performance e scalabilità tuttavia impongono un architettura più complessa da capire e soprattutto implementare. In figura i due diversi modelli:



L'**invio di dati** nel modello **NIO non blocking** è dunque così schematizzato:

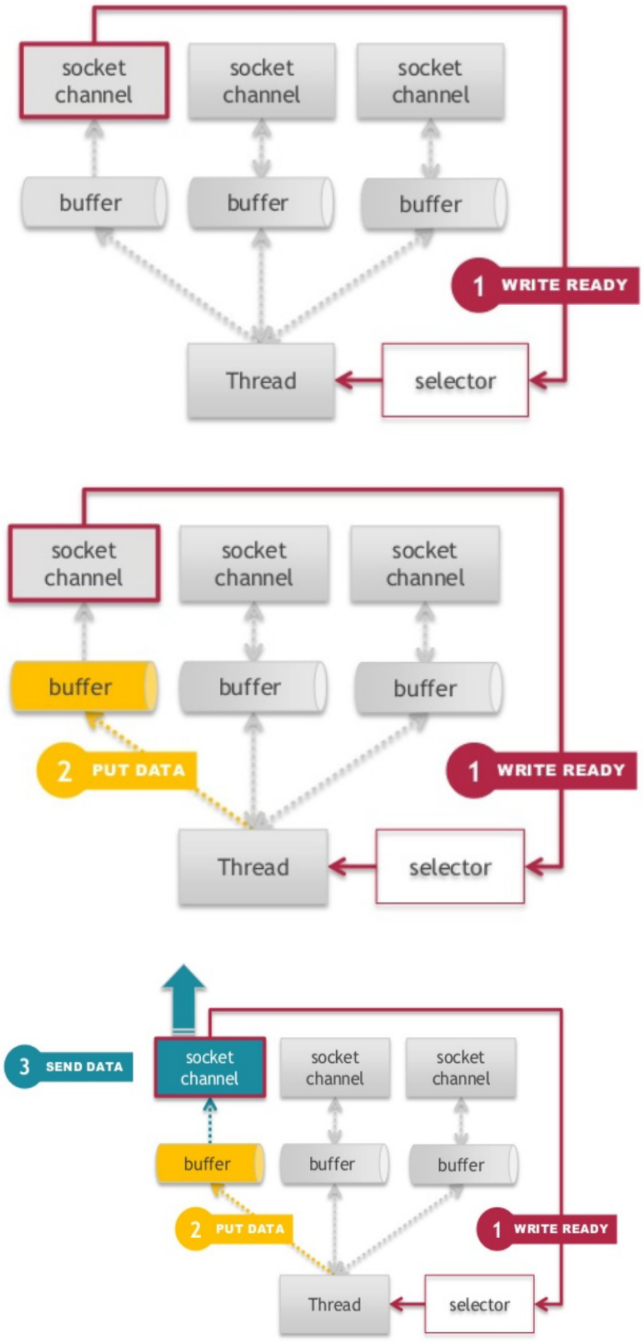
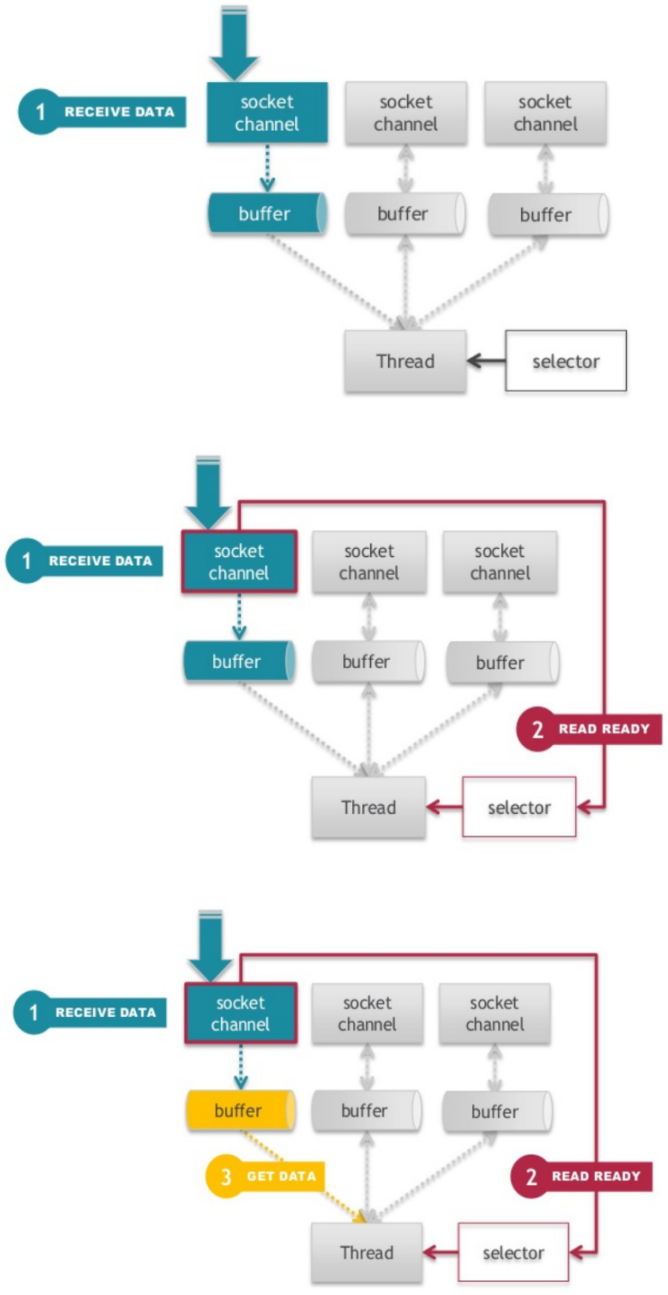
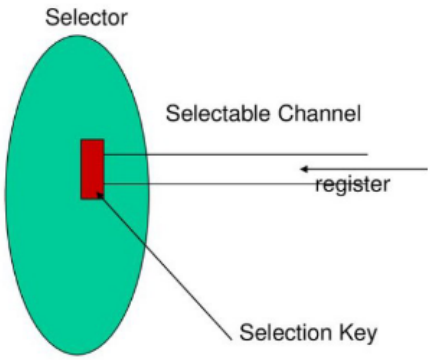

socket channel
socket channel
socket channel
buffer
buffer
buffer
1 WRITE READY
Thread
selector
socket channel
socket channel
socket channel
buffer
buffer
buffer
2 PUT DATA
1 WRITE READY
Thread
selector
3 SEND DATA
socket channel
socket channel
socket channel
buffer
buffer
buffer
2 PUT DATA
1 WRITE READY
Thread
selector

Figure 15: Analogamente, per lo stesso modello, la **ricezione dei dati**:



5.6.3 Selector



Il **Selector** è il componente base che consente il multiplexing secondo la **readiness selection**: permette di selezionare un *SelectableChannel* che è pronto per operazioni di rete (*accept, write, read, connect*) ed è presente un singolo thread che gestisce più eventi che possono avvenire simultaneamente. Tra i *selectable channels* vi sono le classi *ServerSocketChannel, SocketChannel. DatagramChannel, Pipe.SinkChannel, Pipe.SourceChannels* mentre i file I/O sono esclusi.
I canali devono essere **registrati** su un **selettore** per operazioni specifiche:

```
channel.configureBlocking(false);
Selectionkey key = channel.register(selector, ops, attach);
```

I vari parametri passati al metodo *register* verranno dettagliati successivamente. Ogni registrazione di un canale su un selettore restiuisce una **chiave** (***token***) che la rappresenta secondo un oggetto di tipo *SelectionKey* che rimane valido fino a che non viene esplicitamente cancellato. Lo stesso canale può essere registrato con più selettori e chiaramente si otterrà una chiave diversa per ogni registrazione.

5.6.4 SelectionKey

L'oggetto *SelectionKey* è il risultato della registrazione di un canale su un selettore e memorizza cinque informazioni diverse ma correlate tra loro:

1. il **canale** a cui si riferisce
2. il **selettore** a cui si riferisce
3. **interest set**: definisce le operazioni del canale associato a cui si deve fare il controllo di *"readiness"*, la prossima volta che il metodo *select()* verrà invocato per monitorare i canali del selettore

4. **ready set**: dopo la invocazione della select, contiene gli eventi che sono pronti su quel canale (*si può leggere dal canale, scrivere, verificare richiesta di connessione, etc*).

5. **attachment**: spazio di memorizzazione associato a quel canale (*appronditto successivamente*)

L'**interest set** è rappresentato come una **bitmask** che codifica le operazioni per cui si registra un interesse su quel canale. Attualmente sono supportati 4 tipi di operazioni:

1. connect
2. accept
3. read
4. write

A tali 4 operazioni sono associate **4 costanti predefinite** nella classe *SelectionKey*, ognuna corrispondente ad una bitmask:

1. *SelectionKey.OP_CONNECT*
2. *SelectionKey.OP_ACCEPT*
3. *SelectionKey.OP_READ*
4. *SelectionKey.OP_WRITE*

L'**interest set** essendo una bitmask è dunque manipolabile tramite gli operatori Java &, |, ~, ^ che eseguono operazioni bit a bit su operandi interi o booleani. In fase di registrazione del canale con il *Selector* si imposta il valore iniziale dell'interest set:

```
//creazione
Selector selector = Selector.open();
channel.register(selector,SelectionKey.OP_READ | SelectionKey.OP_WRITE);

//reperire interest set e manipolarlo
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept = (interestSet & SelectionKey.OP_ACCEPT) ==
    SelectionKey.OP_ACCEPT
//oppure selectionkey.interestOps(SelectionKey.OP_READ);
```

Il risultato dell'OR tra *SelectionKey.OP_READ* e *SelectionKey.OP_WRITE* sarà una bitmap che avrà un 1 in corrispondenza dell'operazione di lettura e 1 in corrispondenza dell'operazione di scrittura.

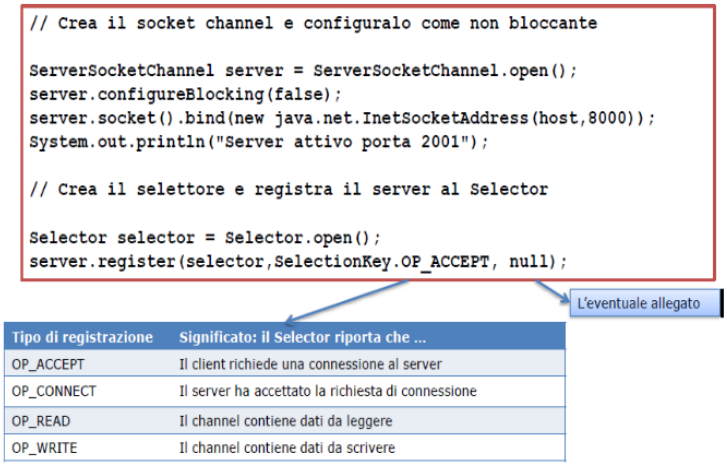
Il **ready set** è aggiornato quando si esegue una operazione di monitoring dei canali, mediante una *select*. Identifica le chiavi per cui il canale è *"pronto"*

per l'esecuzione (*ad esempio un sottoinsieme dell'interest set*). Il ready set è inizializzato a 0 quando la chiave viene creata e non può essere modificato direttamente. Viene restituito dal metodo *readyOps()* invocato su una *SelectionKey*, come nello snippet riportato:

```
if ((key.readyOps() & SelectionKey.OP_READ) != 0){ //op pronte sul canale

    myBuffer.clear( );
    key.channel( ).read (myBuffer);

    doSomethingWithBuffer(myBuffer.flip( ));
}
```

Una shortcuts è rappresentata da *key.isReadable()* che equivale a *ey.readyOps() & SelectionKey.OP_READ) != 0*.

Ricordiamo inoltre che tra le informazioni della *SelectionKey* vi è un ***attachment*** ovverro un riferimento ad un generico oggetto *Object* che è utile quando si vuole accedere ad informazioni relative al canale (*associato d una chiave*) riguardano il suo stato pregresso.
Si rende necessario l'utilizzo perché le **operazioni di lettura o scrittura non bloccanti non possono essere considerate atomiche** dunque non è possibile fare nessuna assunzione sul numero di bytes letti ma l'attachment consente di tenere traccia di quanto è stato fatto in una operazione precedente (*ad esempio, l'attachment può essere utilizzato per accumulare i byte restituiti da una sqeunza di letture non bloccanti o memorizzare il numero di bytes che si devono leggere in totale*).
Riportiamo uno schema (*molto*) generale di riepilogo:



Tipo di registrazione	Significato: il Selector riporta che ...
OP_ACCEPT	Il client richiede una connessione al server
OP_CONNECT	Il server ha accettato la richiesta di connessione
OP_READ	Il channel contiene dati da leggere
OP_WRITE	Il channel contiene dati da scrivere

5.6.5 Multiplexing dei canali: select()

Il metodo ***int selector.select( );*** è bloccante, e seleziona tra i canali registati sul *selector* quelli pronti per almeno una delle operazioni di I/O dell'interest set. All'invocazione il metodo si blocca finché una delle seguenti condizioni non è vera:

1. almeno un canale è pronto
2. il thread che esegue la selezione viene interrotto
3. il selettore viene sbloccato mediante il metodo wakeup() (*è utile invocare il metodo wakeup() su un selettore* ***bloccato*** *perché lo stesso thread che ha attivato il selettore può risvegliarsi*

Al ritorno dalla chiamata restituisce il numero di canali pronti che hanno generato un evento dopo l'ultima invocazione della *select()* e costruisce un insieme contenente le chiavi dei canali pronti.
Una variante dello stesso metodo è data da *int select(long timeout)* che si blocca fino a che non è trascorso il timeout, oppure valgono le condizioni sopra esposte.
Una terza versione è data da *int selectNow()* che è non bloccante, nel caso in cui nessun canale sia pronto restituisce il valore 0.
Ogni oggetto selettore mantiene i seguenti insiemi di chiavi:

- **Key Set**: contiene le *SelectionKeys* dei canali registrati con quel selettore (*restituite dal metodo keys()*)
- **Selected Key Set**: è l'insieme di chiavi precedentemente registrate e per cui una delle operazioni nell'interest set è anche nel ready set della chiave (*ovvero contiene le chiavi dei canali che sono pronti*).
- **Cancelled Key Set**: contiene le chiavi invalidate, quelle su cui è stato invocato il metodo *cancel()* ma non ancora de-registrate.

All'invocazione di *selector.select()* quello che avviene inizialmente è definito come **"delayed cancellation"** ovvero cancella ogni chiave appartenente al *Cancelled Key Set* dagli altri due insiemi (*cancellando così la registrazione del canale*). Inoltre interagisce col sistema operativo per verificare lo stato di prontezza di ogni canale registrato, per ogni operazione specificata nel suo interest set.
Per ogni canale con almena una operazione pronta controlla che il canale esista già nel *Selected Key Set* e aggiorna il *ready set* della chiave corrispondente al canale pronto calcolando l'or bit a bit tra il valore precedente e la nuova maschera, *"accumulando"* i bit a 1 con delle operazioni pronte. Se invece il canale non è presente nel Selected Key Set resetta il ready set e lo imposta con la chiave dell'operazione pronta aggiungendo infine il canale al *Selected Key Set*.

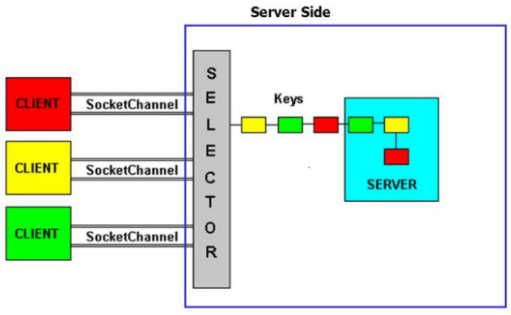
Dal funzionamento del processo di selezione si evince un **comporamento cumulativo** della selezione: una chiave aggiunta al *Selected Key Set* può essere

rimossa solo con una operazione di rimozione esplicita. Inoltre il *ready set* di una chiave inserita nel selected key set non viene mai resettato ma aggiornato incrementalmente (*si può lasciare per scelta architetturale di assegnare al programmatore la responsabilità di aggiornare esplicitamente le chiavi*). Per resettare il ready set rimuove la chiave dall'insieme delle chiavi selezionate.

Vista la teoria, lanciamoci su uno snippet che da l'idea del pattern generale per la selezione:

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator <SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {
SelectionKey key = (SelectionKey) keyIterator.next();
keyIterator.remove();

    if(key.isAcceptable()) {
    // a connection was accepted by a ServerSocketChannel.
    } elseif (key.isConnectable()) {

    // a connection was established with a remote server.
    } elseif (key.isReadable()) {

    // a channel is ready for reading
    } elseif (key.isWritable()) {

    // a channel is ready for writing
    }
}
```

L'iterazione sull'insieme delle chiavi che consente di individuare i canali pronti: dalla chiave si può ottenere un riferimento al canale sui cui si è verificato l'evento. Per la rimozione *keyIterator.remove()* deve essere invocata poiché il *Selector* non rimuove le chiavi. Un server accetta connessioni dai client, ed, una volta che una



connessione è stata accettata, invia un codice numerico al client corrispondente:

sarà dunque necessario registrare sul *Selector* il canale associato a quel client con *Selectkey="WRITE"*.

5.7 UDP Datagram Socket e Channels

In alcuni casi TCP offre più di quanto sia necessario al programmatore poiché ci sono casi in cui non interessa garantire che tutti i messaggi vengano recapitati (*come lo streaming video*) oppure nel caso in cui si vuole evitare l'overhead dovuto alla ritrasmissione dei messaggi. Un terzo caso d'esempio è nel caso in cui non è necessario leggere i dati nell'ordine con cui sono stati spediti.
UDP supporta un tipo di comunicazione **connectionless** e fornisce un insieme molto limitato di servizio rispetto a TCP quali:

1. aggiunge un ulteriore livello di indirizzamento rispetto a quello offerto dal livello IP, quello delle porte

2. offre un servizio di scarto dei pacchetti corrotti (*senza ritrasmissione*)

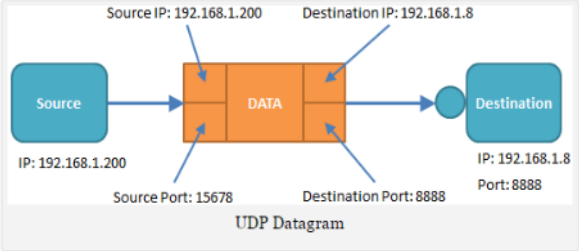
Uno slogan classico per indicare quando utilizzare UDP è *"timely, rather than oderly and reliable delivery"*.
Le API Socket di Java forniscono interfacce diverse per UDP e TCP. Come abbiamo visto, per TCP si fa uso di ***StreamSocket*** in cui occorre connettere il socket al server per aprire una connessione. Per TCP la trasmissione è vista come uno **stream continuo di byte** provenienti dallo stesso mittente. Per UDP si fa uso di un ***DatagramSocket*** in cui un socket **non deve essere connesso ad un altro socket prima di essere utilizzato** ma è semplicemente un canale tramite cui mando pacchetti sulla rete e dunque funge da interfaccia verso la rete. In UDP ogni messaggio è chiamato **Datagram** ed è indipendente dagli altri e porta l'informazione per il suo instradamento. La trasmissione è **orientata ai messaggi** secondo la filosofia *"preserves message boundaries"*: la socket è vista come una mailbox, in essa possono essere inseriti messaggi in arrivo da diverse sorgenti (*mittenti* o i messaggi inviati a diverse destinazioni. Le operazioni per ricevere e inviare datagrammi sono rispettivamente *send* e *receive*. Ogni ricezione si riferisce ad un singolo messaggio inviato tramite unica *send* e dati inviati dalla stessa send non possono essere ricevuti in *receive* diverse.

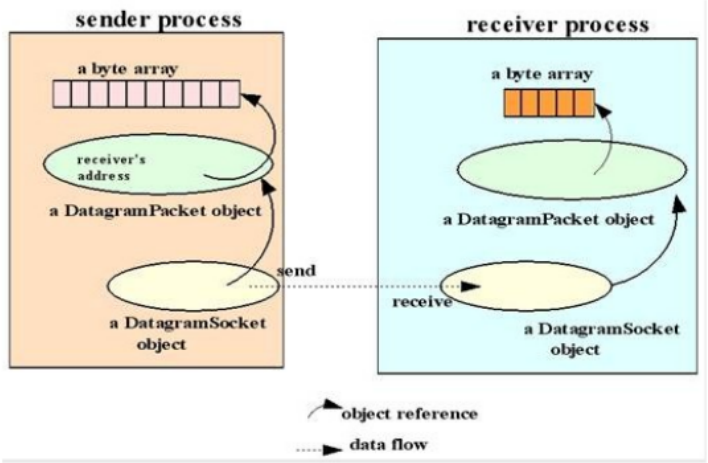
Un Datagram è un messaggio indipendente, self-contained in cui arrivo ed il tempo di ricezione non sono garantiti e che in Java è modellato come un oggetto ***DatagramPacket***. u Il mittente deve inizializzare il campo *DATA* e il *destination IP* e destination port mentre il *source IP* è inserito automaticamente, la *source port* è scelta automaticamente in modo casuale.

5.7.1 DatagramSocket API

Le classi per la gestione di UDP sono principalmente due:

1. **DatagramSocket** per creare i sockets.





2. **DatgramPacket** per costruire i datagram.

Un processo mittente che desidera ricevere/inviare dati su UDP deve istanziare un oggetto di tipo ***DatagramSocket*** collegato ad una porta locale: il processo mittente collega il suo socket ad una porta *PM* che può essere una porta effimera e non è necessario pubblicarla. Dualmente, il destinatario *"pubblica"* la porta a cui è collegato il socket di ricezione affinché il mittente possa spedire pacchetti su quella porta. Di fatto il processo destinatario collega il suo socket ad una porta *PD*.Analogamente per *Socket e ServerSocket* anche *DatagramSocket* è selezionabile tramite un *Selector* in modalità non bloccante.
Dunque, per **inviare un datagramma** è necessario:

1. creare un ***DatagramSocket*** e collegarlo ad una porta, anche effimera
2. creare un oggetto di tipo ***DatagramPacket*** in cui inserire un riferimento ad un byte array contenente i dati da inviare nel payload del datagramma e indirizzo IP e porta del destinatario nell'oggetto appena creato.
3. il *DatagramPacket* tramite una *send* invocata sull'oggetto *DatagramSocket*

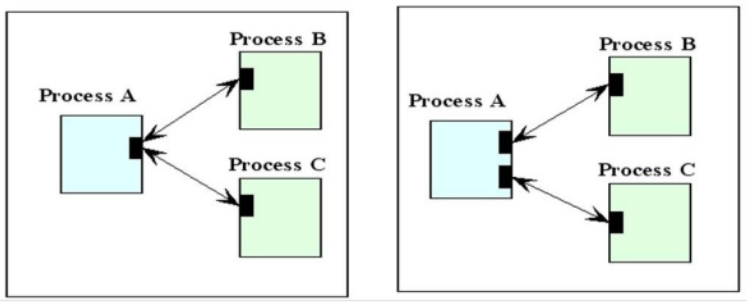
Dualmente, per **ricevere un datagramma** è necessario:

1. creare un ***DatagramPacket*** e collegarlo ad una porta pubblica (*che corrisponde a quella specificata dal mittente nel pacchetto*)

2. creare un *DatagramPacket* per memorizzare il pacchetto ricevuto. Tale oggetto contiene un riferimento ad un byte array che conterrà il messaggio ricevuto.

3. invocare una *receive* sul *DatagramSocket* passando l'oggetto *DatagramPacket* creato al punto precedente.

Un processo può utilizzare lo stesso socket per spedire pacchetti verso destinatari diversi infatti processi (*applicazioni*) diverse possono spedire pacchetti sullo stesso socket allocato dal destinatario: in questo caso l'ordine di arrivo dei messaggi non è deterministico, in accordo con il protocollo UDP. E' comunque possibile utilizzare socket diversi per comunicazioni diverse.




```
//Utilizzato lato client
public class DatagramSocket extends Object
       public DatagramSocket ( ) throws SocketException
```

La classe ***DatagramSocket*** crea un socket e lo collega ad una porta **anonima** o **effimera**, il sistema operativo di fatto sceglie una porta non utilizzata e la assegna al socket. Per reperire la port allocata si usa il metodo *getLocalPort()*. Un comportamento tipico è quello di un client che si connette ad un server mediante una socket collegato ad una porta anonima. Il server dunque preleva l'indirizzo del mittente (*composto da IP + porta*) dal pacchetto ricevuto e può così inviare una risposta. Quando il socket viene chiuso la porta può dunque essere utilizzata per altre connessioni.

```
//Utilizzato lato server
public class DatagramSocket extends Object
       public DatagramSocket(int p) throws SocketException
```

Il server crea una socket sulla porta specificata *int p*. L'invocazione del costruttore può sollevare un'eccezione quando la porta è già utilizzata oppure se si tneta di connettere il socket ad una porta su cui non si hanno i diritti. Uno

scenario classico è quando il server crea un socket collegato ad una porta che rende nota ai clients e solitamente la porta viene allocata permanentemente a quel servizio e dunque non è scelta tra le porte effimere.
Riportiamo un esempio di **port scanning** per individuare le porte libere su un host:

```
mport java.net.*;
public class scannerporte {
    public static void main(String args[ ]){

        for (int i=1024; i<.....; i++){
            try {
                DatagramSocket s =new DatagramSocket(i);
                System.out.println ("Porta libera"+i);
            }catch (BindException e) {
                System.out.println ("porta gi in uso")
            }catch (Exception e) {
                    System.out.println (e);
            }
        }
    }
}
```

La classe ***DatagramPacket*** ha moltissimi costruttori (*vedi API*) diversi per invio/ricezione di pacchetti.

```
//Costruttori utilizzati lato server
DatagramPacket(byte[ ] buffer, int length)
DatagramPacket(byte[ ] buffer, int offset, int length)

//Costruttori utilizzati lato client
DatagramPacket(byte[ ] buffer, int length, InetAddress remoteAddr,int
    remotePort)
DatagramPacket(byte[ ] buffer, int offset, int length,InetAddress
    remoteAddr, int remotePort)
```

Generalmente un oggetto di tipo *DatagramPacket* contiene un riferimento ad un vettore di byte buffer che contiene i dati da spedire oppure quelli ricevuti ed eventuali informazioni aggiuntive di addressing (*se deve essere spedito*). Per **inviare i dati** si utilizza il costruttore:

```
public DatagramPacket(byte[] buffer,int length,InetAddress
    destination,int port)
```

Tra i parametri del costruttore ne commentiamo alcuni:

- **length**: indica il numero di butes che devono essere copiati dal byte buffer nel pacchetto IP, a partire dal byte 0 da offset, se indica. Se *length* è maggiore di *buffer.length* allora solleva un'eccezione. Generalmente il byte

buffer può contenere più di *length* bytes ma questi non verranno spediti sulla rete.

- **destination + port**: individuano il destinatario

Nota non banale è che per essere memorizzato nel buffer il messaggio deve essere trasformato in una **sequenza di bytes**. Come visto precedentemente un modo semplice è tramite l'invocazione del metodo *getBytes* tuttavia per oggetti con una struttura più complessa o per tipi di dato non primitivi si farà uso (*come vedremo più avanti* dei metodi forniti dalla classe ***java.io.ByteArrayOutputStream***.
Analogamente, per **ricevere i dati** si utilizza il costruttore:

```
public DatagramPacket (byte[ ] buffer, int length)
```

Tale costruttore definsice la struttura utilizzata per memorizzare il pacchetto ricevuto. Generalmente il buffer viene passato vuoto alla *receive* che lo riempie al momento della ricezione di un pacchetto col payload del pacchetto stesso (*se settato un offset, la copia avviene nella posizione indivuata da esso*).
Notare bene che il parametro ***length*** indica il numero massimo di bytes che possono essere copiati nel buffer e tale valore deve essere minore di *buffer.length* altrimenti viene sollevata un'eccezione. La copia del payload nel buffer termina o quando l'intero pacchetto è stato copiato oppure se la lunghezza del pacchetto è maggiore di *length* e quando *lenght* bytes sono già stati copiati. Il metodo *getLength* restituisce il numero di bytes effettivamente copiati.
Ad ogni socket sono associati due buffers: uno per la ricezione ed uno per la spedizione, gestiti dal **sistema operativo e non dalla JVM**.
Appronfondiamo questo aspetto con un esempio:

```
import java.net.*;
public class udpproof {
    public static void main (String args[])throws Exception{
        DatagramSocket dgs = new DatagramSocket( );
        int r = dgs.getReceiveBufferSize();
        int s = dgs.getSendBufferSize();
        System.out.println("receive buffer"+r);
        System.out.println("send buffer"+s);
    }
}
```

Il seguente snippet riporta in stampa: *receive buffer 8192 send buffer 8192*.
Poiché i due buffer (*associati ad ogni socket*) sono gestiti autonomamente dal sistema operativo non è possibile modificare la quantità di bytes che i due buffer possono contenere. Generalmente la dimensione massima di un pacchetto UDP è di ***64k bytes*** tuttavia in molte piattaforma è di ***8k bytes***. Se ci poniamo nello scenario in cui mandiamo un dato più grande di 8kB allora i pacchetti vengono troncati e dunque è buona prassi impostare la singola dimensione di un *DatagramPacket* ad un valore minore di 512 bytes. In questo modo si mandano *chunk* di bytes (*frammenti*) che poi verranno ricostruiti lato ricevente.

La classe *DatagramPacket* fornisce una lunga lista di metodi *setter* e *getter* che commentiamo parzialmente:

- ***byte[] getData()***: restituisce un riferimento all'**intero buffer** associato più recentemente al *DatagramPacket* (*tramite invocazione del costruttore o tramite il metodo setData()*). Tale metodo ignora l'offset e la lunghezza e può provocare problemi nel caso in cui il buffer abbia dimensioni maggiori dell'effetivo dato ricevuto.
- **public int getLength()**: restituisce la lunghezza dei dati ricevuti
- ***void setData(byte[] buffer*** o ***void setData(byte[] buffer, int offset, int length***: consente di mettere i dati nel byte buffer gestito esplicitamente dal programmatore (*non nel buffer del SO*). Si possono mettere i dati nel buffer passandoli direttamente nel costruttore del buffer o, appunto, tramite il metodo *setData()*.

Rispetto al metodo *setData()* vediamo un caso d'uso esplicativo quando si deve mandare una grossa quantità di dati:

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;

while (bytesSent < bigarray.length) {

    socket.send(dp);

    bytesSent += dp.getLength( );
    int bytesToSend = bigarray.length - bytesSent;

    int size = (bytesToSend > 512) ? 512 : bytesToSend;
    dp.setData(bigarray, bytesSent, size);
}
```

Ipotizziamo che l'array *bigarray* abbia una dimensione maggiore di 512 bytes: ciò che avviene è che nel while l'offset viene modificato ad ogni invio controllando la dimensione dei dati da mandare (*se i byte da mandare byteToSend sono maggiori di 512 allora invio un "chunk" da 512, altrimenti invio i bytes rimanenti bytesToSend*).

5.7.2 Generazione dei pacchetti

I dati inviati mediante UDP devono essere rappresentat come **vettori di bytes**. Alcuni metodi per la conversione di stringhe in vettore di bytes quali:

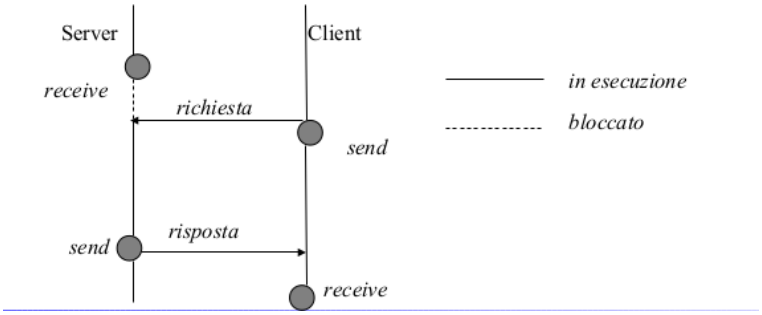
- ***Byte[] getBytes[]***: viene applicato ad un oggetto String e restituisce una sequenza di bytes che codificano i caratteri della stringa usando la codifica di default dell'host e li memorizza nel vettore.

- ***String(byte[] bytes, int offset, int length***: costruisce un nuovo oggetto di tipo String prelevando *length* bytes dal vettore bytes a partire dalla posizione offset.

Sarà necessario utilizzare altri meccanismi per generare pacchetti a partire da dati strutturati come l'utilizzo di **filtri** per generare stream di bytes a partire da dati strutturati/ad alto livello.
L'**invio di pacchetti** e la **relativa ricezione** si articola dunque in due istruzioni operate rispettivamente da mittente e ricevente:

```
sock.send(dp);
//sock indica il socket attraverso il quale voglio spedire il
    pacchetto dp

sock.receive(buffer);
//sock indica il socket attraverso il quale ricevo il pacchetto e
    buffer indica la struttura in cui memorizzo il pacchetto ricevuto
```

Il metodo ***send*** è **non bloccante** sotto un'accezione diversa da quella vista finora ovvero nel senso che il processo che esegue la send prosegua la sua esecuzione senza attendere che il destinarario abbia di fatto ricevuto il pacchetto. Dualmente, il metodo ***receive*** è **bloccante** poiché il processo che esegue la receive si blocca fino al momento in cui viene ricevuto un pacchetto. Al fine di evitare attese indefinite è possibile associare al socket un **timeout** allo scadere del quale viene sollevata una ***InterruptedIOException***. Nella



receive con timeout, il parametro *SO_ TIMEOUT* è associato alla socket e indica l'**intervallo di tempo** (*in millisecondi*) di attesa di ogni receive eseguita su quel socket. Nel caso in cui l'intervallo di tempo scada prima che venga ricevuto un pacchetto dalla socket viene sollevata una eccezione di tipo *InterruptedException*. Il metodo usato per la gestione del timeout è:

```
public synchronized void setSoTimeout(int timeout) throws
    SocketException
// Esempio: se ds  un datagram socket, l'invocazione del metodo:
//            ds.setSoTimeout(30000);
```

```
    // associa un timeout di 30 secondi al socket ds.
```

Diamo un occhiata al seguente snippet:

```
class TestSocket {
    public static void main(String[] args) throws IOException {

        DatagramSocket ds = new DatagramSocket(50000,
        InetAddress.getByName("localhost"));
        final DatagramPacket dp = new DatagramPacket(new byte[1024],
             1024);
        //thread used to try to access the packet's data asynchronously

        new Thread() {
            public void run() {
            while (true){
                try { sleep(1000); }
                    catch (InterruptedException e){
                                e.printStackTrace(); }

                System.out.println("Will try to call getData on dp");
                dp.getData(); //should block
                System.out.println("getData ran");
            }
            }
        }.start();
        ds.receive(dp);
    }
}
```

L'idea alla base del programma è che il programma principale si blocca su una receive mentre un thread controlla, in modo asincrono, il contenuto del buffer di ricezione tuttavia l'istruzione *System.out.println("getData ran");* non viene mai eseguita poiché la receive acquisisce una lock sul *DatagramPacket* e non la lascia fino a che la receive non è completata tuttavia poiché non c'è alcun nodo che invia messaggi sul socket, la lock non viene rilasciata e la *getData()* rimane bloccata in attesa di acquisire la lock.

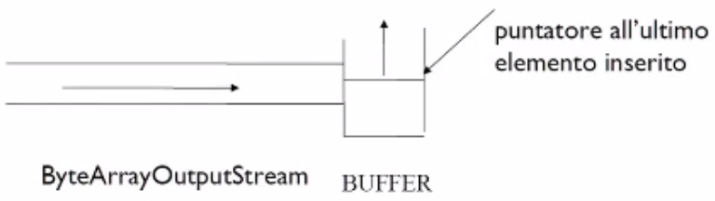
5.7.3 UDP Strucutured Data

Per rappresentare (*e poter inviare in UDP*) dati di tipo strutturato faremo uso principalmente di due classi ***ByteArrayOutputStream*** e ***ByteArrayInputStream***. Guardiamo la prima:

```
public ByteArrayOutputStream();
public ByteArrayOutputStream(int size);
```

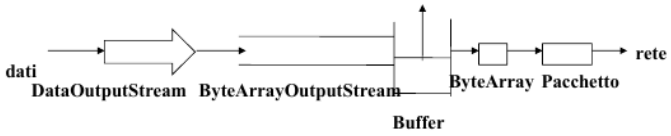
Gli oggetti istanze della classe ***ByteArrayOutputStream*** rappresentano uno stream di bytes: ogni dato scritto sullo stream viene riportato in un **buffer**

di memoria a dimensione variabile con dimensione di default 32 bytes. Quando il buffer si riempie la sua dimensione viene raddopiata automaticamente, seguendo lo schema riportato:



Ad un *ByteArrayOutputStream* può essere collegato un filtro: i dati presenti nel buffer B associato ad un *ByteArrayOutputStream baos* possono essere copiati in un array di bytes:

```
ByteArrayOutputStream baos= new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos)

byte [ ] barr = baos.toByteArray( )
//prende tutti i dati dallo stream, li mette in un vettore di byte che
    verr messo nel pacchetto spedito in rete
```



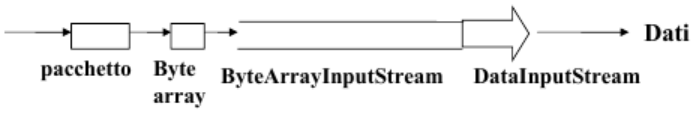
Tra i metodi disponibili della classe ***ByteArrayOutputStream*** per la gestione dello stream vi sono utili:

- ***public int size()***: restituisce count, cioè il numero di bytes memorizzati nello stream (*non la lunghezza del vettore buf*).
- ***public synchronized void reset()***: svuota il buffer, assegnando 0 a count. Tutti i dati precedentemente scritti vengono eliminati.
- ***public synchronized byte toByteArray()***: restituisce un vettore in cui sono stati copiati tutti i bytes presenti nello stream. L'invocazione non modifica count e non svuota il buffer.

```
public ByteArrayInputStream ( byte [ ] buf);
public ByteArrayInputStream ( byte [ ] buf, int offset,int length);
```

Dualmente, i costruttori della classe ***ByteArrayInputStream*** consentono di creare uno stream di byte a partire dai dati contenuti nel vettore di byte *buf*.

Nello specifico, il secondo costruttore copia *lenght* bytes iniziando dalla posizione *offset*. Analogamente a prima, è possibile concatentare un *ByteArrayInputStream* ad un *DataInputStream*.



Vediamo un esempio che si basa sull'ipotesi semplificativa in cui non consideriamo la perdita/ordinamento di pacchetti:

```
importimportjava.io.*;
java.net.*;
public class multidatastreamsender{
    public static void main(String args[ ]) throws Exception{

    // fase di inizializzazione
    InetAddress ia=InetAddress.getByName("localhost");
    int port=13350;

    DatagramSocket ds= new DatagramSocket();
    ByteArrayOutputStream bout= new ByteArrayOutputStream();
    DataOutputStream dout = new DataOutputStream (bout);
    byte [ ] data = new byte [20];
    DatagramPacket dp= new DatagramPacket(data,data.length, ia ,port);
    for (int i=0; i< 10; i++){

        dout.writeInt(i);
        data = bout.toByteArray();
        dp.setData(data,0,data.length);
        dp.setLength(data.length);
        ds.send(dp);
        bout.reset( );
        dout.writeUTF("***");
        data = bout.toByteArray( );
        dp.setData (data,0,data.length);
        dp.setLength (data.length);
        ds.send (dp);
        bout.reset();
        }
    }
 }
```

L'idea alla base è quella di scrivere in *dout* (*che è un oggetto ByteArrayOutputStream a cui concateno un DataOutputStream*) una serie di interi (*da 1 a 10*) che vengono convertiti in byte, messi nel buffer e infine inseriti nel buffer *data* tramite l'invocazione di *toByteArray*. Inserisco l'intero in byte all'interno del

pacchetto UDP *dp* tramite l'invocazione di *setData* e infine lo spedisco.
Effettuo una *reset* sull'oggetto *ByteArrayOutputStream bout* poiché voglio inserire la stringa *"***"* tramite una *writeUTF()*. Analogamente a prima, converto la stringa in byte tramite l'invocazione del *toByteArray*, inserisco i byte nel pacchetto e lo invio.
Vediamo come il ***receiver*** interpreta quanto appena mostrato:

```
importimportjava.io.*;
java.net.*;
public class multidatastreamreceiver{

    public static void main(String args[ ]) throws Exception{

    // fase di inizializzazione
    FileOutputStream fw = new FileOutputStream("text.txt");
    DataOutputStream dr = new DataOutputStream(fw);
    int port = 13350;

    DatagramSocket ds = new DatagramSocket (port);
    byte [ ] buffer = new byte [200];
    DatagramPacket dp= new DatagramPacket(buffer, buffer.length);

    for (int i=0; i<10; i++){

        ds.receive(dp);
        ByteArrayInputStream bin= new ByteArrayInputStream
        (dp.getData(),0,dp.getLength());
        DataInputStream ddis= new DataInputStream(bin);
        int x = ddis.readInt();
        dr.writeInt(x);
        System.out.println(x);

        ds.receive(dp);
        bin= new ByteArrayInputStream(dp.getData(),0,dp.getLength());
        ddis= new DataInputStream(bin);
        String y=ddis.readUTF( );
        System.out.println(y);
        }
    }
}
```

Il funzionamento del programma appena mostrato potrebbe non essere sempre corretto. Poniamoci nell'ipotesi che il mittente alterni l'invio di pacchetti contenente interi all'invio di pacchetti contenenti stringhe: analogamente il receiver dovrà alternare letture di interi a letture di stringhe tuttavia se un pacchetto viene perso le letture da parte del destinatario possono non corrispondere alle scritture effettuate dal mittente.
Una possibile soluzione potrebbe essere quella di implementare una sorta di

protocollo affidabile in UDP implementando l'invio di **pacchetti ACK** per confermare la ricezione che tramite l'uso di identificatori univoci consentono un corretto riscontro di ogni pacchetto.

5.7.4 Serializzazione

Analizziamo il seguente snippet:

```
import java.io.*;
public class Test {
    public static void main (String Args[ ]) throws Exception
    {

    ByteArrayOutputStream bout = new ByteArrayOutputStream( );
    System.out.println (bout.size( ));
    // Stampa 0

    ObjectOutputStream out= new ObjectOutputStream(bout);
    System.out.println (bout.size( ));
    // Anche se non ho scritto niente sulla stream, stampa 4 ,l'header
        stato scritto sullo stream !!

    out.writeObject("ciao mondo");
    byte[] b=bout.toByteArray();

    ByteArrayInputStream bin = new ByteArrayInputStream(b);
    ObjectInputStream oin = new ObjectInputStream(bin);

    String s = (String) oin.readObject();
    System.out.println(s);
    // Stampa  ciao   mondo

    bout.reset();
    //riflettere sul fatto di inserire o meno l'istruzione precedente!

    out.writeObject("ciao ciao");
    b = bout.toByteArray();
    bin = new ByteArrayInputStream(b);
    oin = new ObjectInputStream(bin);
    s = (String) oin.readObject();
    System.out.println(s);

    s = (String) oin.readObject();
    System.out.println(s);
    //Exception in thread "main" java.io.StreamCorruptedException:
    //invalid stream header: 74000963
    //at java.io.ObjectInputStream.readStreamHeader(Unknown Source)
    //at java.io.ObjectInputStream.<init>(Unknown Source)
    //at Test.main(Test.java:20)
    }
```

```
}
```

L'eccezione ***StreamCorruptedException*** viene sollevata poiché dopo l'invocazione del metodo *bout.reset()* rimuovo non solo la stringa *"ciao mondo"* ma anche l'header che aveva automaticamente creato quando ho creato l'ObjectOutputStream. Se elimino l'istruzione *bout.reset()* non resetto il buffer e dunque quando effettuo la seconda deserializzazione trovo nel buffer sia i byte che rappresentano il primo oggetto sia quelli del secondo, ottenendo, per il codice sopra riportato, la seguente stampa:

```
0
4
ciao mondo
ciao mondo
ciaociao
```

Se inserisco l'istruzione *bout.reset()* quando vado a serializzare il secondo oggetto, resetto il buffer e distruggo lo stream header; dunque quando deserializzo la JVM segnala l'eccezione perché non trova lo streamheader e ottengo, per il codice sopraripportato, la seguente stampa:

```
0
4
ciao mondo
//Exception in thread "main" java.io.StreamCorruptedException:
//invalid stream header: 74000963
//at java.io.ObjectInputStream.readStreamHeader(Unknown Source)
//at java.io.ObjectInputStream.<init>(Unknown Source)
//at Test.main(Test.java:20)
```

Una soluzione a questo secondo scenario è quella di ricreare l'oggetto *ObjectOutputStream* poiché consentirebbe di ricreare lo streamheader.

5.7.5 Datagram Channels

Analogamente ai modelli visti precedentemente, anche UDP interagisce con i *channels*. Vediamo alcuni esempi che rispecchiano i pattern già visti per TCP:

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();
// Prepare buffer for reading

channel.receive(buf);
```

In **modalità bloccante** la *receive* si blocca fino a che non è stato ricevuto

un *DatagramPacket* e funziona così come la receive di *DatagramSocket* ma il trasferimento è più efficiente tramite buffer. Analogamente, la *buf.clear()* reinizializza *position* e *limit*.

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
String newData = "New String to write to file..."

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();

buf.put(newData.getBytes());
buf.flip();

int bytesSent = channel.send(buf, new InetSocketAddress("www.google.it",
    80));
```

L'operazione di write si blocca fino a che non c'è spazio disponibile nel send buffer.

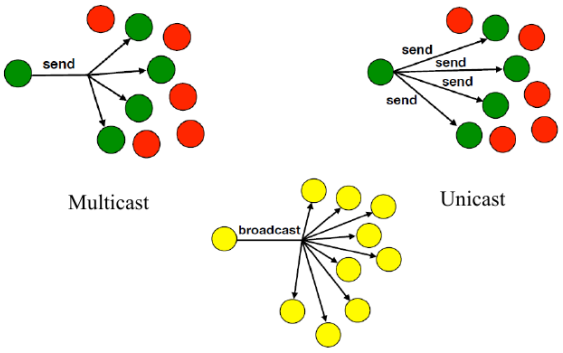
```
ByteBuffer buffer = ByteBuffer.allocate(8192);
DatagramChannel channel= DatagramChannel.open();

channel.configureBlocking(false);
channel.socket().bind(new InetSocketAddress(9999));
SocketAddress address = new InetSocketAddress(localhost,7);

buffer.put(...);
buffer.flip();
while (channel.send(buffer, address) == 0);
   //do something useful ...

buffer.clear();

while ((address = channel.receive(buffer))==null);
   //do something useful ...
```

Nello specifico, il comportamento della *send* varia poiché se restituisce 0 indica che il *send-buffer* è pieno mentre se restituisce *null* allora non è stato ricevuto alcun *Datagram*. Il programma può eseguire altre operazione nell'attesa che il canale sia disponibile per l'operazione.

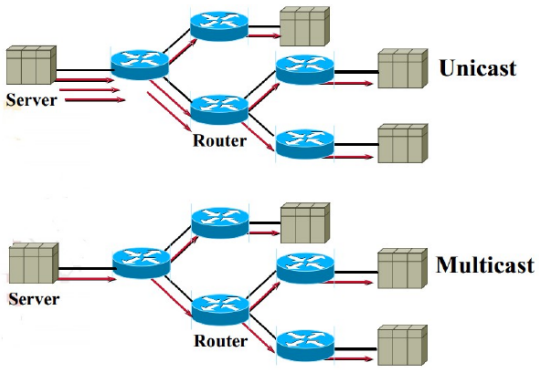
5.8 Multicast

Con il termine **multicast** si indica genericamente la distribuzione simultanea di informazione verso un **gruppo di destinatari**, cioè la possibilità di trasmettere la medesima informazione a più dispositivi finali, senza dover indirizzare questi ultimi ***singolarmente*** e senza avere, quindi, la necessità di duplicare

per ciascuno di essi l'informazione da diffondere. Possiamo vedere il broadcast



come un'estensione del multicast: è il caso in cui l'informazione è diretta a tutti gli host della rete (*o sottorete*) indistintamente. Noi tratteremo il mul-



ticast supportato dal livello IP, differente dal multicast implementato a livello applicativo. Nello specificio, **IP multicast** è un metodo per inviare datagrammi IP a un gruppo di riceventi interessati, in una singola trasmissione. A tale fine, IP multicast è basato sul concetto di **gruppo** ovvero un insieme di processi in esecuzione su hos diversi. Tutti i membri del gruppo di multicast ricevono un messggio spedito su quel gruppo e non occorre essere membri del gruppo per inviare i messaggi su di esso. Tale modalità è gestita a livello IP dal **protocollo IGMP (*Internet Group Management Protocol*)**. Le multicast API devono contenere le primitive necessarie ad eseguire una serie di azioni sul gruppo:

- unirsi ad un gruppo di multicast
- lasciare un gruppo di multicast
- spedire messaggi ad un gruppo (*il messaggio viene recapitato a tutti i processi che fanno parte del gruppo in quel momento*)

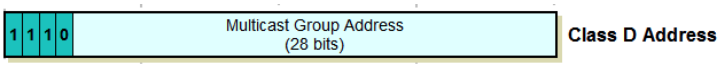
- ricevere messaggi indirizzati ad un gruppo

Il supporto deve fornire uno **schema di indirizzamento** per identificare univocamente un gruppo ed un meccanismo che **registri la corrispondenza tra gruppo ed i suoi partecipanti *IGMP***.

5.8.1 Addressing e scoping

Si basa sull'idea di riservare un insieme di indirizzi IP per multicast. Rispettivamente:

1. **IPv4**: l'indirizzo di un gruppo è un indirizzo di classe D (***da 224.0.0.0 a 239.255.255.255***) in cui i primi 4 bit del primo ottetto sono fissati a *1110* e i restanti bit identificano il particolare gruppo.
2. **IPv6**: tutti gli indirizzi di multicast iniziano con FF (*riservati dalla IANA*).



E' necessario scegliere un indirizzo di multicast in modo tale che deve essere noto collettivamente a tutti i partecipanti al gruppo. Nel caso di indirizzi statici sono scelti da un'autorità a livello internet per un certo servizio e conosciuta a livello globale. Nel caso di indirizzi dinamici i gruppi di host si mettono d'accordo sull'utilizzare un certo indirizzo tuttavia sono possibili collisioni nel caso in cui due gruppi scelgano lo stesso indirizzo.
Si introduce il multicast scoping che consente di limitare la diffusione di un pacchetto generalmente tramite due metologie:

- **TTL scoping**: il TTL limita la diffusione del pacchetto infatti ad ogni pacchetto IP viene associato un valore rappresentato su un byte (*riferito come Time-to-Live del pacchetto*). Indica il numero massimo ri routers attraversati dal pacchetto che viene scartato dopo aver attraversato *TTL* routers. Generalmente si associano valori di TTL per aree geografiche.
- **Administrative scoping**: a seconda dell'indirizzo in classe D scelto, la diffusione del pacchetto viene limitata ad una parte della rete i cui confini sono definiti da un amministratore di rete.

Gli indirizzi possono essere:

- **statici**: assegnati da una autorità di controllo, utilizzati da particolari protocolli/ applicazioni.L 'indirizzo rimane assegnato a quel gruppo, anche se, in un certo istante non ci sono partecipanti
- **dinamici**: si utizzano protocolli particolari che consentono di evitare che lo stesso indirizzo di multicast sia assegnato a due gruppi diversi. Esistono solo fino al momento in cui esiste almeno un partecipante e richiedono un protocollo per l'assegnazione dinamica degli indirizzi

Gli **indirizzi statici** possono essere assegnati da ***IANA*** o dall'*amministratore di rete*. Quelli concernenti allo IANA sono validi per tutti gli host della rete e possono essere cablati nel codice delle applicazioni. (*ad esempio l'indirizzo multicast 224.0.1.1 è assegnato al* ***network time protocol*** *utilizzato per sincronizzare i clocks di più hosts*. Per gli indirizzi assegnati dall'amministratore di una certa organizzazione valgono per tutti gli host della rete amministrata. Per ottenere un indirizzo di multicast in modo dinamico è necessario utilizzare un protocollo opportuno, rispettivamente:

- *Multicast Address Dynamic Client Allocation Protocol* ***(MADCAP)***: nell'ambito di una sottorete gestita mediante un unico dominio amministrativo
- *Multicast Address Set Claim* ***(MASC, SSM, SDR)***: nell'ambito più generale della rete

Il multicast utilizza il paradigma **connectionless (UDP)** poiché se usassimo un tipo di comunicazione *connection-oriented* dovremmo gestire *n*(n-1)* connessioni per un gruppo di n applicazioni. La comunicazione connectionless adatta per il tipo di applicazioni verso cui è orientato il multicast in cui è accetabile la perdita occasionale di un frame piuttosto che un ritardo costante tra la spedizione di due frames successivi. Esistono in Java librerie non standard che forniscono un **multicast affidabile** garantendo che il messaggio venga recapitato una solta volta a tutti i processi del gruppo e garantire proprietà di ordinamento dei pacchetti spediti. Nel corso tuttavia trattando il **multicast IP** non abbiamo nessun tipo di garanzia in merito.

5.8.2 Java Multicast API

Si fa uso principalmente della classe **MulticastSocket** che estende *DatagramSocket* che rapprsenta una socket su cui ricevere i messaggi da un gruppo di multicast. Effettua overriding dei metodi esistenti in *DatagramSocket* e fornisce nuovi metodi per l'implementazioni di funzionalità tipiche del multicast.

```
import java.net.*;
import java.io.*;
public class multicast{
    public static void main (String [ ] args){

         try {
            MulticastSocket ms = new MulticastSocket(4000);
            InetAddress ia=InetAddress.getByName("226.226.226.226");
            ms.joinGroup (ia);
        }
        catch (IOException ex) {System.out.println("errore"); }
    }
}
```

L'invocazione di *joinGroup* è necessaria nel caso si vogliano ricevere messaggi dal gruppo di multicast. Tale metodo lega il **multicast socket** ad un gruppo

multicast e dunque tutti i messaggi ricevuti tramite quel socket provengono da quel gruppo (*si rende necessaria la gestione dell'IOException che viene sollevata se l'indirizzo di multicast è errato*).

```
import java.io.*;
import java.net.*;
public class provemulticast {
    public static void main (String args[]) throws Exception{

        byte[] buf = new byte[10];
        InetAddress ia = InetAddress.getByName(228.5.6.7 );

        DatagramPacket dp = new DatagramPacket (buf,buf.length);
        MulticastSocket ms = new MulticastSocket (4000);

        ms.joinGroup(ia);
        ms.receive(dp);
     }
 }
```

Lo snippet precedente illustra come nel caso di due istante attive di *provemulticast* sullo stesso host non viene sollevata una ***BindException*** rispetto al classico funzionamento dei socket visto finora. Ciò è reso possibile dalla proprietà **reuse socket** del *MulticastSocket* che consente, se settata a *true* di associare più socket alla stessa porta, svincolando dunque la corrisponde biunivoca porta-servizio. Generalmente *reuse socket* è settato di default a true e tale valore può essere manipolato tramite il metodo specifico:

```
try{
        sock.setReuseAddress(true);

    }catch (SocketException se) { //manage exception}
```

Per dare una prima idea di come utilizzare multicast si riporta il codice di uno sniffer multicast in cui dopo aver ricevuto in input un nome simbolico di un gruppo di multicast si unisce al gruppo e *sniffa* i messaggi spediti sul gruppo, stampandone il contenuto.

```
import java.net.*; import java.io.*;
public class multicastsniffer {
    public static void main (String[] args){

        InetAddress group = null;
        int port = 0;

        try{
            group = InetAddress.getByName(args[0]);
            port = Integer.parseInt(args[1]);
```

```
        } catch(Exception e){
            System.out.println("Uso:java multicastsniffer
                multicast_address port");
            System.exit(1);
        }


        MulticastSocket ms=null;
        try{
            ms = new MulticastSocket(port);
            ms.joinGroup(group);
            byte [ ] buffer = new byte[8192];
            DatagramPacket dp=new DatagramPacket(buffer,buffer.length);

            while (true){
                try{
                    ms.receive(dp);
                    String s = new String(dp.getData());
                    System.out.println(s);
                }
                catch (IOException ex){System.out.println (ex);}
                finally{ if (ms!= null) {
                        try {
                            ms.leaveGroup(group);
                            ms.close();
                        } catch (IOException ex){}
                    }
                }
            }
        }
    }
}
```

Per **spedire messaggi** ad un gruppo di multicast è necessario:

- creare un *DatagramSocket* su una porta anonima
- non è necessario collegare il socket ad un gruppo multicast
- creare un pacchetto inserendo nell'intestazione l'indirizzo IP del gruppo di multicast a cui si vuole inviare il pacchetto
- spedire il pacchetto tramite il socket creato (*tramite il metodo public void send (DatagramPacket p) throws IOException*

Traduciamo quanto detto in codice Java:

```
import java.io.*;
import java.net.*;
public class multicast {
    public static void main (String args[]){
```

```
        try{
            InetAddress ia=InetAddress.getByName("228.5.6.7");
            byte [] data;
            data="hello".getBytes();
            int port= 6789;
            DatagramPacket dp = new
                DatagramPacket(data,data.length,ia,port);
            DatagramSocket ms = new DatagramSocket(6789);
            ms.send(dp);
            Thread.sleep(80000);
        } catch(IOException ex){
            System.out.println(ex);
            }
    }
}
```

E' possibile utilizzare il **TTL scoping** nel seguente modo:

- il mittente specifica un valore TTL per i pacchetti spediti
- il TTL viene memorizzato nel campo dell'header del pacchetto IP
- il TTL viene decrementato per ogni router attraversato
- se $TTL = 0$ il pacchetto viene scartato

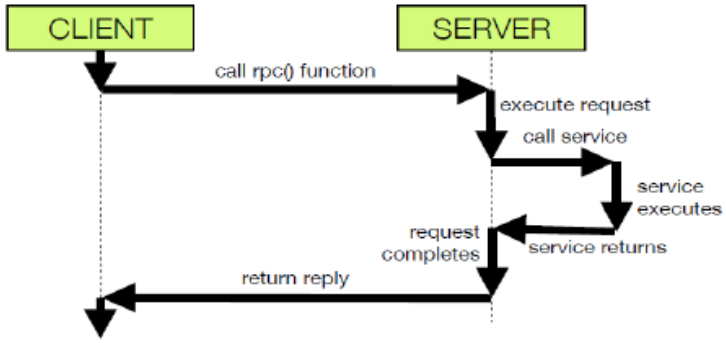
Generalmente il valore del TTL è impostato ad 1 se i pacchetti multicast non possono lasciare la rete locale. E' possibile modificare il valore del TTL associato al *multicast socket* tramite:

```
MulticastSocket s = new MulticastSocket();
s.setTimeToLive(1));
```

6 RMI - Remote Method Invocation

Un'applicazione Java distribuita generalmente è composta da computazioni eseguite su **JVM differenti** in esecuzioni su host differenti comunicanti tra loro (*il multithreading non è distribuito essendo sulla stessa macchina*). Il meccanismo dei socket è flessibile e potente per la programmazione di applicazioni distribuito ma è di basso livello e ciò richiede la progettazione di veri e propri protocolli di comunicazione e la verifica non banale delle loro funzionalità nonostante, come abbiamo visto, la serializzazione consente di ridurre la complessità dei protocolli grazie all'invio di dati strutturati.
Un'alternativa è rappresentata dall'utilizzo di una tecnologia più ad alto livello, originariamente nota come **Remote Procedure Call (*RPC*)** evolutasi poi in **Remote Method Invocation (*RMI*)** che consiste in un'interfaccia alla comunicazione di rete rappresentata dall'invocazione di una **procedura/metodo remoto** invece che un'interfaccia più a basso livello come i socket.

La **Remote Procedure Call** è un paradigma di interazione a **domanda/rispsota** in cui il client invoca una procedura del server remoto e quest'ultimo esegue la procedura con i parametri passati dal client e restituisce a quest'ultimo il risultato ell'esecuzione. La connessione remota è trasparente rispetto a client e server ed in genere provede meccanismi di affidabilità a livello sottostante.



Riportiamo a titolo chiarificatore un esempio banale d'interazione in cui, il client:

- richiede ad un server la stampa di un messaggio.
- attende l'esito dell'operazione *(il server restituisce un codice che indica l'esito della operazione)*
- la richiesta del client al server è implementata come una invocazione di una procedura definita sul server

I meccanismi utilizzati dal client sono gli stessi utilizzati per una normale invocazione di procedura tuttavia l'invocazione di procedura avviene sull'host su cui è in esecuzione il client e la procedura viene eseguita sull'host su cui è in esecuzione il server. I parametri della procedura vengono inviati automaticamente sulla rete dal supporto all'RPC, in modo invisibile al programmatore.
L'interazione appena descrutta consente una semplificazione del lavoro svolto dal programmatore che non si deve più preoccupare di sviluppare protocolli per il trasferimento, verifica, codifica/decodifica di dati poiché tali operazioni sono interamente gestite dal supporto. Come approfondiremo dopo, tale semplificazione è in parte dovuta anche all'uso di **stub** o **proxy** che sono dei *rappresentati del server* sul client e che consentono di astrarre sulle operazioni effettuate dal supporto dell'RPC.
Poiché non tutto è bianco o nero, anche tale tecnologia ha i propri limiti poiché i parametri ed i relativi risultati da restituire al client devono essere dati di tipo primitivo. Inoltre tale approccio consente unicamente l'uso del paradigma della programmazione procedurale e la localizzazione del server non è trasparente poiché richiede al client di conoscere l'IP e la porta su cui il server è in esecuzione. Da tali critiche all'RCP si è sviluppato un **paradigma ad oggetti**

distribuiti noto come **Remote Method Invocation (*RMI*)**.

6.1 Schema architetturale

Il **Remote Method Invocation** consente di avere oggetti remoti attivabili, ovvero **servizi** che si attivano *on demand* cioè a seguito di una invocazione, e che si disattivano quando non utilizzati. La critica alla trasperenza mossa all'RPC viene risolta poiché l'utilizzo di oggetti remoti risulta largamente più trasparente in quanto, una volta localizzato l'oggetto, il programmatore utilizza i metodi dell'oggetto come se questi fosse locale. Inoltre codifica, decodifica, verifica e trasmissione dei dati sono effettuati dal **supporto RMI** in maniera trasparente all'utente.
Attualmente l'implementazione di Java prevede 3 tipologie di oggetti:

1. **di tipo unicast**: ad un certo istante esiste una sola istanza dell'oggetto remoto nella rete
2. **volatili** il tempo di vita di un oggetto è al più pari a quello del processo che l'ha creato
3. **non-rilocabili**: l'oggetto remoto non può essere rilocato in un altro processo

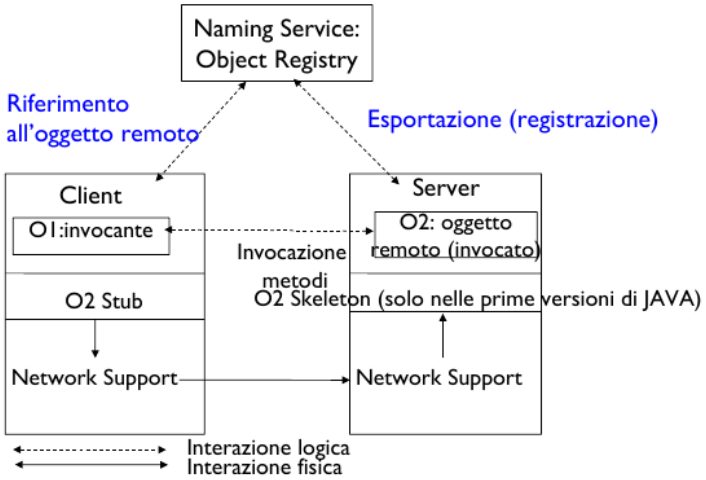
Un **oggetto remoto** è un oggetto i cui metodi possono essere acceduti da un diverso spazio di indirizzamento (*una JVM diversa che è potenzialmente in esecuzione su un altro host*) e consente di sfruttare le caratteristiche della programmazione ad oggetti (*ereditarietà, polimorfismo, incapsulamento*).
Java mette a disposizione un insieme di classi **Java RMI** che consentono l'uso del supporto di un **Java Security Manager** addetto a controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite oltre a supportare il meccanismo di **Distributed Garbage Collection (*DGB*)** per disallocare quegli oggetti remoti per cui non esistano più referenze attive.
Iniziamo ad illustrare l'architettura di RMI ad alto livello tramite le sue entità principali:

- **client**: cerca i riferimenti di oggetti remoti nel *registry* tramite un'operazione di **lookup** a partire dal nome pubblico dell'oggetto.
- **server**: si occupa di **esportare** gli oggetti remoti e registrare i riferimenti agli oggetti remoti nel *registry* tramite la ***bind***.
- **registry**: permette di mettere in comunicazione client e server (*fornendo i riferimenti al client e consentendo di esporre gli oggetti esportati dal server*) rappresentando di fatto un servizio di naming (*simile alle funzioni svolte dal DNS*).

Quando il client vuole accedere all'oggetto remoto cerca un **riferimento** dell'oggetto remoto nel **registry** e invoca il servizio mediante chiamate di metodi

che sono le stesse delle invocazioni locali (*oggetto.metodo();*). L'invocazione dei metodi di un oggetto remoto possono essere osservate sotto due punti di vista:

- **livello logico** in cui l'invocazione è analoga all'invocazione di un metodo di un oggetto locale
- **livello di supporto** in cui il supporto provvede a trasformare i parametri della chiamata remota in dati da spedire sulla rete e gestisce l'invio dei dati in rete.

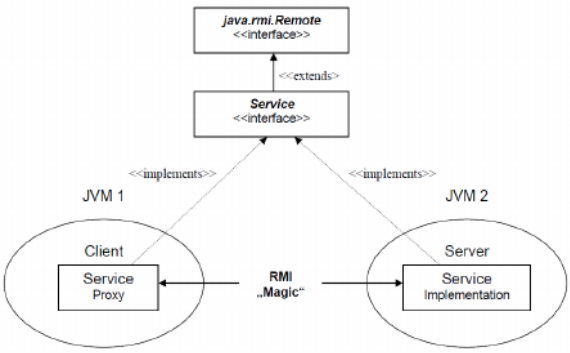


6.2 RMI Step-by-step: EUStatsService

Affrontiamo la teoria in parallelo ad un esempio di applicazione che fa uso di RMI. Il seguente esempio definisce un server che mediante un oggetto remoto restituisce le principali informazioni relative ad un paese dell'Unione Europea di cui il client ha specificato il nome. Il server restituisce *lingua ufficiale, popolazione* e *nome della capitale*.
Innazitutto è necessario creare un'**interfaccia Java** che dichiara i metodi accessibili da remoto: tale interfaccia è implementata da due classi:

1. sul **server**, dalla classe che implementa il servizio
2. sul **client**, dalla classe che implementa il **proxy** (*il rappresentante del servizio sul client*) del servizio remoto

Un'interfaccia è remota se e solo se estende ***java.rmi.Remote***. Di fatto ***Remote*** è una *tag interface* ovvero non definisce alcun metodo poiché ha il solo scopo di identificare gli oggetti che possono essere utilizzati in remoto. I metodi remoti devono dichiarare di sollevare **eccezioni remote** della classe *java.rmi.RemoteException* oppure di una sua superclasse *java.io.IOException,*



java.lang.Exception, etc (*guarda API Java*). Dunque il primo step è quello di definire l'interfaccia remota:

```
//######## STEP 1: DEFINIZIONE DELL'INTERFACCIA REMOTA
import java.rmi.RemoteException;
import java.rmi.Remote;

public interface EUStatsService extends Remote {
    String getMainLanguages(String CountryName) throws RemoteException;
    int getPopulation(String CountryName) throws RemoteException;
    String getCapitalName(String CountryName) throws RemoteException;
}
```

Il **secondo step** consiste nell'implementare l'interfaccia remota lato server definendo il codice specifico dei metodi.
Vi sono alcune differenze rispetto alla implementazione degli oggetti standard poiché le operazioni base della classe *Object* quali *equals, hashCode e toString* devono essere ridefiniti per gli oggetti remoti poiché hanno una **semantica diversa** ovvero si tenta di unificare il comportamento di tali metodi rendendolo omogeneo per qualsiasi tipo d'invocazione (*il metodo equals potrebbe avere implementazioni diverse sulle diverse macchine che eseguono l'invocazione remota dei metodi*).
Il package *Java.rmi.server* supporta la definizione degli oggetti remoti e consente una **ridefinizione standard** delle operazioni precedentemente elencate (*nonostane sia comunque possibile ridefinirli manualmente, ad opera del programmatore*).
La semanica del metodo **equals** per oggetti remoti consiste nel verificare se i riferimenti considerati sono uguali tra loro. Una uguaglianza di tipo ***content equality*** è stata scartata dai progettisti poiché innanzitutto implicherebbe chiamate remote per il confronto dei singoli attributi con un notevole aumento dei costi computazionali e la differenza di semantica tra chiamate remote e chiamate locali del metodo equals creerebbe la necessità di intercettare eccezioni *Remote* e un carico di lavoro ed un aumento della complessità del tutto inutile.

La semantica del metodo **hashCode** è definita standard in cui si calcola il valore hash dell'oggetto e restituisce lo stesso valore per tute le refeenze remote che puntano allo stesso *RemoteObject*. Infine, la semantica del metodo **toString** è anch'essa definita standard e restituisce una rappresentazione sottoforma di stringa per l'oggetto remoto (*ad esempio la stringa può includere host name+porta+identificatore oggetto*).
Le classi del package *java.rmi.server* sono molteplici:

$$UnicastRemoteObject < RemoteServer < RemoteObject < Object$$

dove A < B indica che A è sottoclasse di B. La **definizione dell'oggetto remoto** può essere effetutata in modi diversi:

- utilizzando la classe ***RemoteObject*** oppure ***UnicastRemoteObject*** in cui si effettua l'overriding di alcuni metodi della classe *Object* per adattarli al comportamento degli oggetti remoti
- definendo un oggetto come istanza della classe *Object* che consente di effettuata esplicitamente una ridefinizione di questi metodi (*ad opera del programmatore*).

Dunque l'implementazione dell'interfaccia può far uso delle classi sopra descritte, fornendo di fatto 3 strategie d'implementazione:

1. **Soluzione 1**: definire una classe che implementi i metodi della interfaccia remota ed estenda la classe RemoteObject
2. **Soluzione 2**: definire una classe che implementi i metodi della interfaccia remota ed estenda la classe UnicastRemoteObject
3. **Soluzione 3**: definire una classe che implementi i metodi della interfaccia remota senza estendere alcuna delle classi viste.

6.2.1 Soluzione 1

```
public class MyServerRemoto extends RemoteServer implements IntRemota{

    public MyServerRemoto() throws RemoteException {
     ..... }
......}
```

I vantaggi di questa soluzione risiedonjo nel fatto che si eredita la ridefinizione della semantica degli oggetti remoti definita da ***RemoteServer***. Dualmente, gli svantaggi sono rappresentati dall'**ereditarietà singola** che non consente di estendere altre classi e inoltre l'oggetto deve poi essere esportato esplicitamente (*al fine di renderlo reperibile dal registry e dunque accessibile al client*).

6.2.2 Soluzione 2

```
public class MyServerRemoto extends UnicastRemoteObject implement
    IntRemota {

   public MyServerRemoto() throws RemoteException {
      super(); //  E  qui che il server viene esportato
   .....}
}
```

La classe sopra definita, estendendo *UnicastRemoteObject* estende dunque anche *RemoteServer* che estende a sua volta *RemoteObject*. Al suo interno contiene metodi per costruire ed esportare un oggetto remoto con semantica di tipo **unicast, volatile, non migrabile**. Ricordiamo che invocare il metodo *exportObject()* consente di creare e attivare lapplicazione remota.
Il vantaggio rispetto alla *soluzione 1* è che l'estendere *UnicastRemoteObject* consente in automatico di ridefinire i metodi base *equals, hashCode, toString* e che l'oggetto remoto viene esportato automaticamente alla chiamata del costruttore.

6.2.3 Soluzione 3

```
public class MyServerRemoto extends MyClass implement IntRemota{
   public MyServerRemoto() throws RemoteException {.....}
.....
}
```

In quest'ultima soluzione l'oggetto remoto richiede **esportazione esplicita**. Un ovvio vantaggio deriva dalla possibilità di estendere un'altra classe utile per l'applicazione (*come MyClass, nello snippet*) mentre uno svantaggio è dato dal fatto che la semantica degli oggetti è delegata al programmatore che dunque deve effettuare l'overriding di *equals,hashCode e toString*.

Torniamo all'esempio di prima, applicando i concetti visti all'implementazione del servizio *EUStatsServer*. Si fa uso di una *hash table* per memorizzare i dati riguardanti le nazioni europee definendo dunque un semplice database delle nazioni tramite l'uso della classe *EUData* per definire oggetti che desrivono la singola nazione.

```
public class EUData {

   private String Language;
   private int population;
   private String Capital;

   EUData(String Lang, int pop, String Cap) {

      Language = Lang;
```

```
        population = pop;
        Capital = Cap;
    }

    String getLangs( ) {
        return Language;
    }

    int getPop( ){
        return population;
    }

    String getCapital( ) {
        return Capital;
    }

}
```

Definiamo una classe ***EUStatsServiceImpl*** che utilizzando la classe *EUData* implementa l'oggetto remoto e ne crea un'istanza. Tale classe implementa l'interfaccia *EUStatsService* definita al passo 1 e nel main della classe è presente un *"launch code"* dell'oggetto remoto che consente di creare un'istanza dell'oggetto remoto, esportare l'oggetto remoto e associare all'oggetto remoto un nome simbolico, registrando il collegamento nel **registry**.

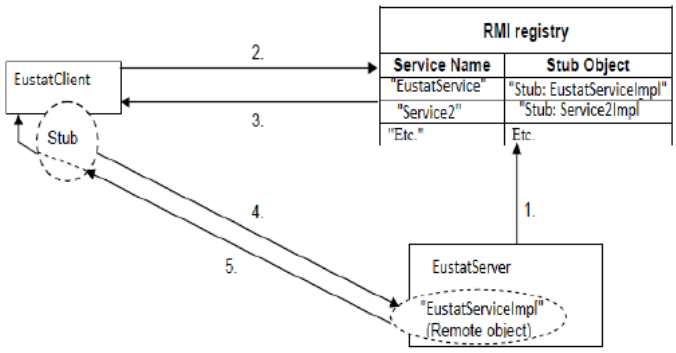
```
import java.rmi.*;          // Classes and support for RMI
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.*;  // Classes and support for RMI servers
import java.util.Hashtable; // Contains Hashtable class

public class EUStatsServiceImpl extends RemoteServer implements
    EUStatsService {

/* Store data in a hashtable */
Hashtable <String, EUData> EUDbase = new Hashtable<String, EUData>();


/* Constructor - set up database */

    EUStatsServiceImpl() throws RemoteException {

      EUDbase.put("France", new EUData("French",57800000,"Paris"));
      EUDbase.put("United Kingdom", new
          EUData("English",57998000,"London"));
      EUDbase.put("Greece", new EUData("Greek",10270000,"Athens"));
    }

/* implementazione dei metodi dell'interfaccia */
```

```
public String getMainLanguages(String CountryName) throws
    RemoteException {

    EUData Data = (EUData) EUDbase.get(CountryName);

    return Data.getLangs();
}

public int getPopulation(String CountryName) throws RemoteException {
    EUData Data = (EUData) EUDbase.get(CountryName);
    return Data.getPop();
 }

public String getCapitalName(String CountryName) throws RemoteException {
    EUData Data = (EUData) EUDbase.get(CountryName);
    return Data.getCapital( );
}


//############## STEP 3: Attivazione servizio
public static void main (String args[]) {
    try {

    /* Creazione di un'istanza dell'oggetto EUStatsService */
    EUStatsServiceImpl statsService = new EUStatsServiceImpl();

    /* Esportazione dell'Oggetto */
    EUStatsServiceImpl stub = (EUStatsServiceImpl)
        UnicastRemoteObject.exportObject(statsService, 0);

    // Creazione di un registry sulla porta args[0]
    LocateRegistry.createRegistry(args[0]);
    Registry r = LocateRegistry.getRegistry(args[0]);

    /* Pubblicazione dello stub nel registry */
    r.rebind("EUSTATS-SERVER", stub);
    System.out.println("Server ready");

    }
    /* If any communication failures occur... */
    catch (RemoteException e) {
        System.out.println("Communication error " + e.toString());
    }
  }
 }
}
```

Il **passo 3** consiste nella generazione dello stub e nell'attivazione del servizio. Prendiamo intanto in analisi lo **stub**: è un oggetto che consente di interfacciarsi

con un altro oggetto (*il target, ovvero l'oggetto remoto*) in modo da sostituirsi ad esso. Si comporta da intermediario, inoltrando le chiamate che riceve al suo target. Per generare un'istanza dello *stub* è necessario generare lo *stato dell'oggetto + metodi* associati all'oggetto. I meccanismi per generare lo stub sono diversi in relazione alla versione di Java tuttavia nel nostro caso faremo uso della **Reflection** senza tuttavia approfondirne il funzionamento.
Dunque il servizio viene **creato** allocando una istanza dell'oggetto remoto mentre viene **attivato** mediante la creazione dell'oggetto remoto e la registrazione dell'oggetto in un ***registry***.
Il main della classe *EuStatsServerImpl* si occupa di:

- creare un'istanza del servizio (*cioè l'oggetto remoto*)
- invoca il metodo statico *UnicastRemoteObject.exportObject(obj,0)* che esporta dinamicamente l'oggetto (*se si indica la porta 0 viene utilizzata una porta scelta dal supporto*)
- restituisce un'istanza dell'oggetto che rappresenta l'oggetto remoto mediante il suo riferimento
- pubblica il riferimento dell'oggetto remoto nel ***registry***
- il main termina ma il **thread** in attesa di invocazione di metodi remoti rimane attivo, impedendo la terminazione del server



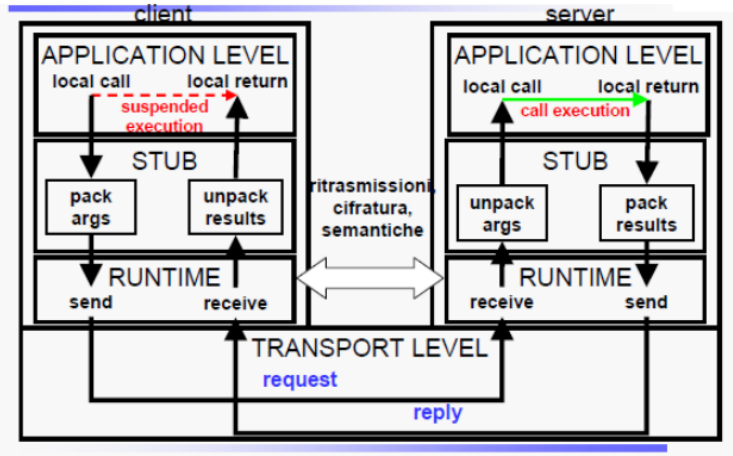
In relazione allo schema sopra, le interazioni sono, in ordine:

1. EutStatServer genera lo stub per l'oggetto remoto EustatServiceImpl e lo registra nel Registry RMI con il nome: "EustatService"
2. EustatClient effettua una look up nel Registry (Naming.lookup(...))
3. il registry RMI restituisce lo stub al client
4. il clien t EustatClient effettua l'invocazione di metodo remoto mediante lo stub

5. viene restituito al client il servizio richiesto

Ricapitolando, un **server RMI** ha la responsabilitá di creare unistanza delloggetto remoto, effettuare la registrazione delloggetto remoto nellRMI registry ed esportare loggetto remoto.

In figura è riportato uno schema generale delle operazioni svolte dal supporto RMI e dai livelli dello stack protocollare IP: in particolare si effettua un **pack args** (*nello specifico, si esegue il* ***marshalling arguments***) che consente di serializzare i dati e rispettivamente, lato server, di effettuare l'**unpack args** (*o* ***unmarshalling***) al fine di eseguire la chiamata locale del metodo e di restituire i risultati replicando la procedura di **pack-unpack** del risultato dell'invocazione del metodo.



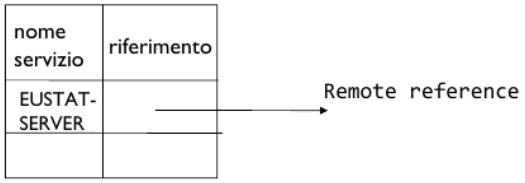
6.2.4 RMI Registry

Analizziamo brevemente il ruolo del **registry** in RMI: mette a disposizione del programmatore un semplice servizio di naming che consente la registrazione ed il reperimento di *stub*. E' dunque un servizio di ascolto sulla porta indicata, simile ad un DNS per oggetti remoti che contiene i legami tra il nome simbolico dell'oggetto remoto ed il riferimento effettivo all'oggetto.
Per creare e gestire direttamente da programma oggetti di tipo *Registry* si fa uso della classe ***LocateRegistry*** di cui ne riportiamo alcuni metodi statici implementati:

- *public static Registry createRegistry(int port)*: consente di lanciare un servizio di registry RMI sull'host locale, su una porta specificata e restituisce un riferimento al registro
- *public static Registry getRegistry(String host, int port)*: reperisce e restituisce un riferimento ad un registro RMI su un certo host ad una certa

porta. Una volta ottenuto il riferimento al registro RMI si possono invocare i metodi definiti dall'interfaccia Registry; solo allora viene creata una connessione col registro (*e viene restituita una eccezione se non esiste il registro RMI corrispondente*).

```
LocateRegistry.createRegistry(args[0]);
Registry r=LocateRegistry.getRegistry(args[0]);
```



6.2.5 RMI Client

Torniamo dopo svariate deviazioni al nostro esempio sull'EUStatsService.
Consideriamo il comportamento del client: per accedere all'oggetto remoto il client deve ricercare lo **stub** dell'oggetto remoto dunque accede al *registry* attivato sul server effettuando una ricerca con il nome simbolico dell'oggetto remoto. Il riferimento restituito è un riferimento allo *stub dell'oggetto* che, nel caso in cui si faccia uso della reflection, restituisce anche il codice dello stub. Il riferimento restitutio è di tipo generico *Object* che dunque necessita il casting al tipo definito dall'**interfaccia remota**.
Il client dunque può effettuare l'invocazione dei metodi dell'oggetto remoto come fossero metodi locali a patto di intercettare le *RemoteException*.

```
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class EUStatsClient {
    public static void main (String args[]) {

        EUStatsServiceImpl serverObject;
        Remote RemoteObject;
        /* Check number of arguments */
        /* If not enough, print usage string and exit */
        if (args.length < 2) {
            System.out.println("usage: java EUStatsClient port
                countryname");return;

        }
        /* Set up a security manager as before */
        /* System.setSecurityManager(new
        RMISecurityManager()); */
```

```
        try {
            Registry r = LocateRegistry.getRegistry(args[0]);
            RemoteObject = r.lookup("EUSTATS-SERVER");
            serverObject = (EUStatsServiceImpl) RemoteObject;

            System.out.println("Main language(s) of " + args[1] + "is/are
                " + serverObject.getMainLanguages(args[1]));
            System.out.println("Population of " + args[1] + " is "+
                serverObject.getPopulation(args[1]));
            System.out.println("Capital of " + args[1] + " is " +
                serverObject.getCapitalName(args[1]));

            }catch (Exception e) {

                System.out.println("Error in invoking object method " +
                e.toString() + e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

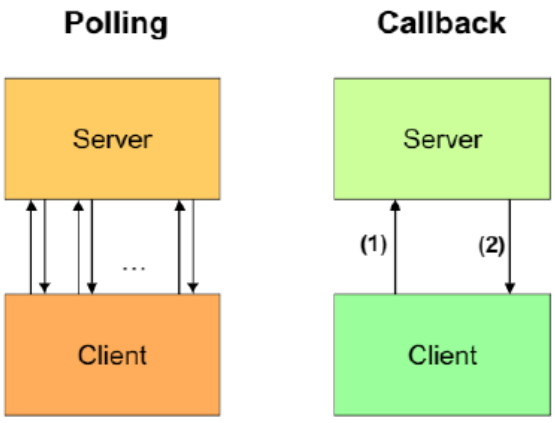
6.3 Callbacks

Il meccanismo delle **callback** consente di realizzare il pattern ***Observer*** in ambiente distribuito che risulta utile quando un client è interessato allo stato di un oggetto remoto o quando vuole ricevere una **notifica asincrona** quando lo stato viene modificato. Tale pattern di programmazione è anche noto come **publish-subscribe**.
Si applica a diversi contesti come, ad esempio, un utente partecipa ad una chat e vuole essere avvertito quando un nuovo utente entra nel gruppo o nello scenario di un gioco multiplayer quando i giocatori modificato lo stato del gioco e rispettivamente il server deve avvisare ogni giocatori di eventuali modifiche che il gioco subisce.

6.3.1 Meccanismi di notifica

Il client verifica l'occorenza dell'evento atteso, interrogando ripetutamente il server (**polling**) tramite l'invocazione ripetita di un metodo remoto tramite RMI. Un evidente svantaggio è dato dall'uso non efficiente delle risorse di sistema.

Un'alternativa al *polling* è data dalle **callback** in cui si notifica in modo asincrono il verificarsi di un evento. Il client registra il suo interesse per un evento ed il server notifica il verificarsi di tale evento. I vantaggi evidenti sono che il client non si blocca e viene avvertito solo quando l'evento avviene.



Il concetto delle callbacks è indipendente dall'RMI che sappiamo essere utile per l'interazione client-server (*per la registrazione da parte del client dell'interesse per un evento*) sia per l'interazione server-client (*notificare al client il verificarsi di un evento*).

Il **server** definisce un'interfaccia remota **ServerInterface** con un metodo remoto utilizzato dal client per registrare il suo interesse per un certo evento (*chiamato anche* ***bootstrap*** *della callback*). Dualmente, il client definisce una interfaccia remota **ClientInterface** con un metodo remoto utilizzato dal server per notificare un evento al client. Il client reperisce il riferimento all'oggetto remoto tramite ***registry*** ed invia al server lo **stub** del suo oggetto remoto. Il server non utilizza il *registry* per indivduare l'oggetto remoto del client poiché lo riceve dal client come parametro del metodo di registrazione della callback.
Il **server** definisce un oggetto remoto ***ROS (RemoteObject Server)*** che implementa *ServerInterface* contenente il metodo per la registrazione della callback che ha come parametro il riferimento allo stub del client. Quando il server riceve una invocazione del metodo remoto, riceve il ***ROC*** **(RemoteObject Client)** ovvero il riferimento all'oggetto remoto del client e lo memorizza in una propria struttura dati. Al momento della notifica, il server utilizza il *ROC* per invocare il metodo remoto sul client, notificando l'evento.

Dualmente il **client** definisce un oggetto remoto **ROC** che implementa ***ClientInterface*** contenente un metodo che consente la notifica dell'evento atteso (*invocato dal server*). Il client ricerca l'oggetto remoto *ROS* del server che contiene il metodo per la registrazione mediante un servizio di **registry**. Al momento della registrazione sul server, passa al server lo *stub* di ***ROC*** (*dunque non registra l'oggetto remoto su un registry ma lo passa al server*). Ció consente di implementare la **notifica asincrona** mediante due **invocazioni remote sincrone**.

6.4 Concorrenza

É possibile che i metodi di un oggetto remoto vengano invocati da piú client (*oppure da thread diversi dello stesso client, in un contesto multithreading*) in modo concorrente. Analizzeremo come il *supporto RMI* gestisce invocazioni multiple.
Dalla documentazione Java: *"A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Calls originating from different clients Virtual Machines will execute in different threads. From the same client machine it is not guaranteed that each method will run in a separate thread".*

Nel caso in cui l'invocazione di metodi remoti provenga da **client diversi (*diverse JVM*)** sono eseguite da thread diversi e dunque consente di non bloccare un client in attesa della terminazione dell'esecuzione di un metodo invocato da un altro client. Dualmente, invocazioni concorrenti provenienti dallo stesso client possono essere eseguite dallo stesso thread o da thread diversi.

La politica di Java RMI di implementare automaticamente il multithreading per **chiamate diverse** presenta il vantaggio di evitare all'utente di scrivere codice per i thread (*lato server*) tuttavia il **server non é thread-safe** e dunque richieste concorrenti di client diversi possono portare ad uno stato **inconsistente** e dunque necessita implementare il server in modo che l'accesso all'oggetto remoto sia correttamente sincronizzato. Per rendere ***atomica*** l'esecuzione di un metodo occorre utilizzare opportuni meccanismi di sincronizzazione (*come i metodi synchronized*).

Per invocazioni multiple provenienti dallo stesso client (*in multithreading, ad esempio*) la documentazione Java é opaca, non esplicita una politica specifica.

6.5 Dynamic Class Loading

RMI offre la capacitá di **reperire dinamicamente** la definizione di classe di un oggetto se non presente nei *local path*. Il meccanismo del **Dynamic Class Loading** fornise la possibilitá di definire applicazioni espandibili che estendano dinamicamente il proprio comportamento. Nello specifico, tramite il *Dynamic Class Loading* vi é la possibilitá di scaricare un file *.class* e quindi di eseguirlo. Tra i meccanismi necessari per implementare tale funzionamento vi sono quelli di individuazione del **repository remoto** delle classi e la gestione della **sicurezza**.

Il class loader di default é utilizzato per caricare la classe il cui metodo *main* é eseguito utilizzando il comando *java*. Effettua una ricerca nel *CLASSPATH locale* e ricerca successivamente tutte le classi utilizzate, ricercandole nel *CLASSPATH locale*. Differentemente, l'**RMIClassLoader** é utilizzato per il caricamento remoto delle classi e puó essere invocato esplicitamente al momento della ricezione di **oggetti remoti serializzati** (*come nel caso in cui si*

riceva un oggetto di cui non si possiede la classe per la deserializzazione).

Alcuni possibili scenari di applicazione sono, ad esempio, i **thin client**: tali client scaricano dinamicamente il loro codice dal server consentendo una semplificazione delle applicazioni oltre ad aggiornamenti automatici in caso di rilascio di versioni/patch. L'idea alla base é quella di sviluppare un *client minimale* che carica dinamicamente il proprio codice da un'area condivisa dove l'ha memorizzato il server e consentendo di evitare la reinstallazione dei client e la presenz adi una unica copia del client presente in un'area condivisa. Il client scarica l'ultima versione del codice dal server e lo esegue tramite un caricamento esplicito di classi dal server. Il metodo ***RMIClassLoader.loadClass*** carica dinamicamente ed in modo esplicito classi remote tuttavia per essere reperite necessitano di un URL che riferisce la directory da cui scaricare il codice.
Nelle prime versione di RMI era richiesta il caricamento dinamico degli **stub** consentendo l'applicazione del **polimorfismo** in cui il caricamento dinamico delle classi permette **lato server** di passare come parametro di una invocazione remota un oggetto di una sottoclasse di quella dichiarata nell'interfaccia. Dualmente, **lato client**, gli oggetti restituiti dal server al client possono essere di una sottoclasse del tipo dichiarato come parametro di ritorno.
Ricordiamo che il principio del **polimorfismo** nei linguaggi ad oggetti denota la possibilitá di usare un oggetto di una sottoclasse B ogni volta che viene indicato l'uso di un oggetto di classe A con una releazione di sottoclasse tra B ed A ($B < A$).
Per rendere lo stub disponibile al client si puó copiare offline il file *.class* dello stub in una directory riferita dal *CLASSPATH* del client o utilizzare il **dynamic class loading**. Il caricamento dinamico dello stub permette di evitare la copia esplicita dello stub dal client al server e nelle ultime versioni di RMI non é piú necessario lo stub poiché viene generato dinamicamente con il meccanismo delle **reflection**.

6.5.1 Passaggio parametri a metodi remoti

In Java standard, distinguiamo classicamente il passaggio di parametri in:

- *void f(int x)*: il parametro x é passato per **valore**
- *void g(Object k)*: il parametro k é passato per **riferimento**

In **Java RMI** invece abbiamo una semantica differente:

- *void h(Object k*: se Object non é un oggeto remoto, il parametro viene passato per valore. L'oggeto dunque viene copiato da un host all'altro usando il meccanismo di serializzazione dell'oggetto (*marshalling*). Invece se l'oggetto é remoto viene passato un **riferimento remoto**.

Il client puó passare al momento dell'invocazione di un **metodo remoto**:

- un oggetto di un tipo di dato primitivo

- un oggetto di un tipo (*classe*) sconosciuto al server remoto in cui la classe dell'oggetto é passata dal client al server esplicitamente mediante altre applicazioni oppure offline
- un oggetto di un tipo (*classe*) sconosciuto al server remoto: tale scenario puó verificarsi poiché il server conosce l'interfaccia (*dunque l'oggetto é compilato, attivato ed esportato*) e dunque tramite il **polimorfismo** é possibile fare uso di una sottoclasse di un tipo noto sia al client che al server. Il server conosce il *tipo* T del parametro, ma il client passa un oggetto di un sottotipo che estende T.

Chiariamo con un esempio:

```
//############ Classe Studente
public class Studente implements java.io.Serializable{

    private static final long serialVersionUID=1L;
    private String nome;
    private int matricola;
    private String corsoDiLaurea;

    public Studente (String n, int m, String c){
        nome = n;
        matricola =m;
        corsoDiLaurea = c;
    }

    public String toString(){
       return ("Nome"+ nome + ", matricola=" + matricola + ", corso di
           Laurea=" + corsoDiLaurea);
    }
}

//Interfaccia del servizio DBStudentiServer
import java.rmi.*;
public interface DBStudentiServer extends Remote{
    public boolean insert (Studente s) throws RemoteException;
}

//############ Implementazione del servizio -- Server

import java.rmi.*;
import java.rmi.server.*;
public class DBStudentiServerImpl extends UnicastRemoteObject implements
     DBStudentiServer{
    private static final long serialVersionUID = 1L;

    public DBStudentiServerImpl() throws java.rmi.RemoteException{ };

    public boolean insert (Studente s) throws RemoteException{
```

```
        // inserisce in un Data Base lo studente
        // se andato a buon fine restituisce true, altrimenti false

        System.out.println("Inserito:"+s); return true;
    }

    public static void main(String args[]){
        // System.setSecurityManager(new SecurityManager());
        DBStudentiServerImpl server;
        try{
            server = new DBStudentiServerImpl();
            Naming. rebind("DBServer",server);
            System.out.println("pronto per Connessioni al DB...");
        }
        catch (Exception e){e.printStackTrace();}
    }

}

//############ Client del servizio senza uso della sottoclasse

import java.io.*;
import java.rmi.*;
public class DBStudentiClient {

    final String pippo = "localhost";
    public static void main(String [] args){

        BufferedReader in = new BufferedReader (new
            InputStreamReader(System.in));
        String nome ="";
        int matricola=0;
        String corsoDiLaurea="";

        try{
            System.out.println("Inserire Nome");
            nome = in.readLine();
            System.out.println("Inserire Matricola");
            matricola = Integer.parseInt(in.readLine());
            System.out.println("Inserire Corso di Laurea");
            corsoDiLaurea = in.readLine();

        }catch (Exception e){System.out.println(e.getMessage());
            System.exit(-1);}

        // System.setSecurityManager(new SecurityManager());
        DBStudentiServer server;
        try {
            server = (DBStudentiServer) Naming.lookup("DBServer");
            Studente s=new Studente(nome, matricola, corsoDiLaurea);
```

```
            server.insert(s);
        } catch (Exception e){System.out.println(e.getMessage());}

    }
}

//############ Sottoclasse di Studente
public class StudenteErasmus extends Studente {

    private static final long serialVersionUID = 1L;
    private String nazione;
    public StudenteErasmus(String n, int m, String c, String naz){

        super (n, m, c);
        nazione= naz;
    }

    public String toString() {
        return (super.toString()+ ",nazione=" + nazione);
    }
}


//############ Client del servizio con uso della sottoclasse
import java.io.*;
import java.rmi.*;
public class DBStudentiClientErasmus {
    public static void main(String [] args){

        BufferedReader in = new BufferedReader (new
            InputStreamReader(System.in));
        String nome =" ";
        int matricola=0;
        String corsoDiLaurea="";
        String nazione=" ";

        try{
            System.out.println("Inserire Nome");
            nome = in.readLine();
            System.out.println("Inserire Matricola");
            matricola = Integer.parseInt(in.readLine());
            System.out.println("Inserire Corso di Laurea");
            corsoDiLaurea = in.readLine();
            System.out.println("Inserire Nazione di Provenieneza");
            nazione = in.readLine();
        }
        catch (Exception e){e.printStackTrace(); System.exit(-1);}

        //System.setSecurityManager(new SecurityManager());
        DBStudentiServer server;
```

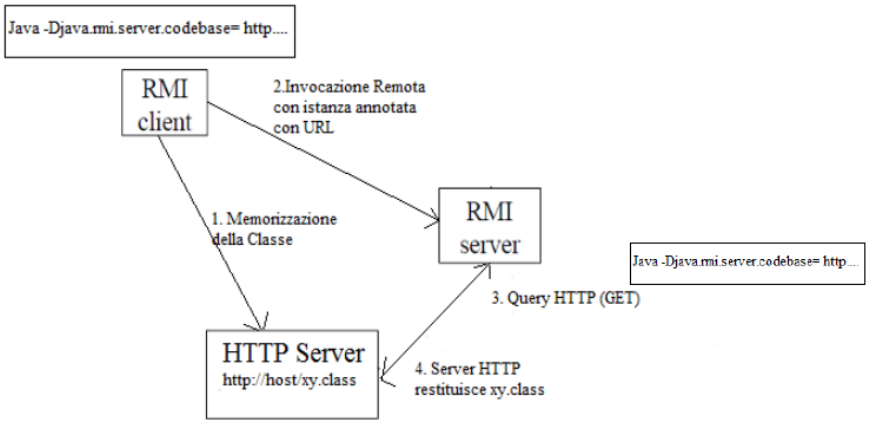
```
        try{
            server = (DBStudentiServer) Naming.lookup("DBServer");
            server.insert(new StudenteErasmus(nome, matricola,
            corsoDiLaurea,nazione));

        } catch (Exception e){System.out.println(e.getMessage());
        }
    }
}
```

6.5.2 Individuazione class repository

Come giá precedentemente accennato meccanismo del **Dynamic Class Loading** fornise la possibilitá di definire applicazioni espandibili che estendano dinamicamente il proprio comportamento. Nello specifico, tramite il *Dynamic Class Loading* vi é la possibilitá di scaricare un file *.class* e quindi di eseguirlo. Tra i meccanismi necessari per implementare tale funzionamento vi sono quelli di individuazione del **repository remoto** delle classi che permetta di scaricare i file(*.class*) dinamicamente su un server web o ftp. Il server che svolge tale funzione é chiamato **codebase**.
Per passare al server un riferimento alla **URL del codebase** Java fa uso di un meccanismo automatico poiché insieme alla **rappresentazione serializzata** dell'oggetto che viene trasmesso al client o al server viene annotata **automaticamente** anche la URL del codebase. Quando l'oggetto viene deserializzato la VJM puó scaricare il *bytecode* dalla classe dell'oggetto da deserializzare in modo trasparente, utilizzando l'annotazione. I due server, quello sui cui é in esecuzione l'oggetto remoto e quello che pubblica i file *.class* possono essere ospitati o meno su macchine diverse.
É possibile utilizzare piú meccanismi di individuazioni del codebase che vanno a settare la proprietá *java.rmi.server.codebase* della JVM (*il valore della proprietá é un riferimento alla URL del codebase*). Le classi RMI che effettuano la serializzazione di un oggetto dunque leggono il valore di tale proprietá e incorporano la URL nella rappresentazione serializzata dell'oggetto.
Si possono modificare le proprietá della JVM modificando i **VM arguments** ovvero gli argomenti in ingresso alla JVM (*analoghi agli argomenti del main*) che permettono il setting e la modifica di varie impostazioni della JVM quali impostare il caching degli indirizzi IP mediante DNS, passare un riferimento al **file di policy** (*vedi sezione Sicurezza*) per la sicurezza in caso di code mobility, controllo dello heap, etc.
Generalmente, per modificare i valori dei parametri della JVM si usano due metologie:

- **setting da linea di comando**: al momento del lancio del programma si usa l'opzione *-D* per distinguere gli argomenti del programma da quelli della JVM (*java –Dnomeproprieta=valore MyClass*).

- **setting tramite system call**: tramite i metodi forniti dalla classe *System* in cui i parametri *prop e val* sono di tipo *String* (*System.setProperty(prop, val)*



6.6 Sicurezza

La **mobilitá del codice** pone ovvi problemi di sicurezza. Da Java 2 in poi la poltiica di sicurezza impone all'utente di definire esplicitamente i permessi di cui deve disporre un'applicazione scaricata dalla rete. Tali permessi definiscono una **sandbox** in cui l'applicazione deve essere eseguita: generalmente una sandobox dovrebbe contenere il minimo numero di permessi che consentono l'esecuzione dell'applicazione. Tali permessi sono elencati in un file di policy di cui é possibile farne un uso da **amministratore di sistema** il quale utilizza tale file per indicare cosa possono fare i programmi lanciati all'interno del sistema, ed un uso da **sviluppatore dell'applicazione** che distribuisce il file di policy che elenca i permessi di cui l'applicazione ha bisogno.

La garanzia che vengano rispettati i permessi, fissati nel **file di policy**, durante l'esecuzione del codice é fornita installando un'istanza della classe ***SecurityManager***. Quando un programma tenta di eseguire un'operazione che richiede un esplicito permesso (*ad esempio una lettura su un file o una connessione via socket*) viene interrogato il SecurityManager.

I programmi che scaricano dinamicamente del codice dalla rete devono impostare un **SecurityManager** poiché RMI abilita il *caricamento dinamico del codice* solo in presenza del *SecurityManager* che se non viene settato allora l'applicazione sará solo in grado di caricar ele classi dal *CLASSPATH locale*. Per attivare il *Security Manager*, all'inizio del programma:

```
System.setSecurityManager(new SecurityManager())
```

O in alternativa definendo la proprietá di sistema da linea di comando: *java -Djava.security.manager applicazione* che consente la creazione e l'installazzione di un'istanza del Security Manager prima dell'esecuzione dell'applicazione.

6.7 REST - Representational State Transfer

É uno **stile architetturale per sistemi software distribuiti**. Il termine é stato introdotto e definito nel 2000 nella tesi di dottorato di *Roy Fielding*, uno degli autori delle specifiche del protocollo *HTTP*. La tesi indica una serie di principi architetturali per la progettazione di servizi web. Inizialmente REST é stata descritta da Fielding nel contesto el protocollo HTTP tuttavia sono possibili varie implementazioni indipendenti da HTTP. Generalmente le interazioni con REST fanno uso di *JSON*.
Un servizio web conforme alle **specifiche REST** deve avere alcune caratteristiche specifiche:

- **architettura client server**
- **stateless**: ogni ciclo di *request-response* deve rappresentare un'interazione completa del client con il server. In questo modo non é necessario mantenere informazione sulla sessione di un utente, minimizzando l'uso di memoria del server e la sua complessitá.
- **uniformemente accessibile**: ogni risorsa deve avere un indirizzo univoco e ogni sistema presenta la stessa interfaccia per identificare le risorse, in genere individuata dal protocollo HTTP.

Gli **elementi fondamentali** di un web service basato su *rest* sono i seguenti:

- **risorse**: una risorsa puó rappresentare un insieme di altre risorse o una singola risorsa (*come un prodotto o un insieme di ordini effettuati da un utente*).
- **rappresentazione delle risorse**: ogni risorsa é identificato da una URL specifica e le risorse sono entitá estratte caratterizzate da uno stato che puó cambiare nel tempo
- **operazioni sulle risorse**

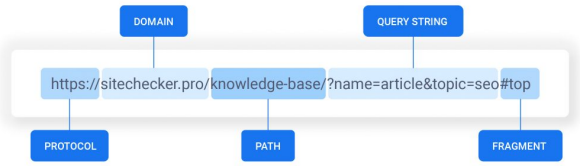
Durante la comunicazione, client e server si scambiano una **rappresentazione dello stato delle risorse** il cui formato della rappresentazione deve essere concordato tra client e server precedentemente. (*HTML per la visualizzazionr browser, XML per l'elaborazione da altre applicazioni, JSON per il js di una pagina web, PDF per la stampa, etc*).
Nel protocollo HTTP lo stato di una risorsa viene identificato attraverso la codifica definita dallo standard ***MIME - Multimedia Internet Mail Extension***

che consente di specificare il formato in due modi distinti quali *Type/subtype* o *Type/subtype; options*. Lo IANA ha assegnato oltre 1000 tipi di MIME quali *text/plain* o *text/plain; charset=UTF-8*, *text/html* e molti altri.
Nell'approccio REST si usa un numero limitato di operazioni per leggere o modificare lo stato delle risorse. Le operazioni corrispondono ai metodi definiti nel protocollo HTTP:

- **POST**: crea una sottorisorsa (*fa riferimento ad una risorsa preesistente, in pratica è un'operazione di "append"*)
- **GET**: restituisce una rappresentazione della risorsa (*é un'operazione di sola lettura, é un metodo idempotente e la risorsa puó essere oggetto di caching*).
- **PUT**: inizializza o aggiorna lo stato di una risorsa
- **DELETE**: elimina una risorsa
- **HEAD**: legge i metadati di una risorsa (GET senza response body)
- **OPTIONS**: elenco di operazioni possibili su una risorsa
- **PATCH**: modificare una parte di una risorsa

6.7.1 Java REST Client

La definizione di un client che accede ad un server REST puó essere effetuata in Java tramite l'uso delle classi *URL, URLConnection, HTTPURLConnection* mentre la definizione di un **server REST** é piú complessa e fuori dagli obiettivi di questo corso. Rispolveriamo la struttura di un URL:



Le classi *URL e URLConnection* incapsulano la maggior parte della complessitá relativa al ritrovamento di una informazione da un sito remoto. Per specificare una URL:

```
URL url = new
    URL("https://docs.oracle.com/javase/tutorial/networking/urls/creatingUrls.htm");
```

Per scaricare i contenuti di una risorsa é possibile usare il metodo *openStream* della classe URL che restituisce un oggetto di tipo *InputStream* che permette la lettura dei contenuti della risorsa. Il metodo *url.openStream()* é la forma abbreviata di *url.openConnection().getInputStream()*.
Vi sono svariati modi di costruire un URL e recuperare le relative risorse associate:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
public class JavaNetURLExample {
public static void main(String[] args) {
    try {

    // Generate absolute URL
    URL url1 = new URL("http://www.gnu.org");
    System.out.println("URL1: " + url1.toString());
    // stampa URL1: http://www.gnu.org

    // Generate URL for pages with a common base URL = www.gnu.org
    URL url2 = new URL(url1, "licenses/gpl.txt");
    System.out.println("URL2: " + url2.toString());
    // stampa URL2: http://www.gnu.org/licenses/gpl.txt

    // Generate URL from different pieces of data
    URL url3 = new URL("http", "www.gnu.org", "/licenses/gpl.txt");
    System.out.println("URL3: " + url3.toString());
    // stampa URL3: http://www.gnu.org/licenses/gpl.txt

    URL url4 = new URL("http", "www.gnu.org", 80, "/licenses/gpl.txt");
    System.out.println("URL4: " + url4.toString() + "\n");
    //stampa URL4: http://www.gnu.org:80/licenses/gpl.txt

    // Open URL stream as an input stream and print contents to command
        line
    try (

        BufferedReader in = new BufferedReader(new
            InputStreamReader(url4.openStream()))){
           String inputLine;
           // Read the "gpl.txt" text file from its URL representation
           System.out.println("/***** File content (URL4) *****/\n");
           while((inputLine = in.readLine()) != null) {
               System.out.println(inputLine);
            }
        } catch (IOException ioe) { ioe.printStackTrace(System.err); }
    } catch (MalformedURLException mue) {
```

```
            mue.printStackTrace(System.err);
        }
    }
}
```

Per effettuare altre operazioni sulla risorsa oltre alla lettura o nel caso in cui si voglia effettuare una modifica occorre usare la classe ***URLConnection*** che fornisce piú funzionalitá rispetto alla classe URL. L'invocazione del metodo *openConnection* per ottenere un oggetto *URLConnection* instaura di fatto una connessione HTTP:

```
URLConnection connection = url.openConnection();
```

É possibile settare proprietá specifiche della connessione quali *SetDoInput, SetDoOutput, SetIfModifiedSince, SetUseCaches, SetAllowUserInteraction, SetRequestProperty*. Dopo aver effettuato la onnessione, si ricevono diverse informazioni che descrivono il contenuto della risorsa. Per recuperare il valore degli header é possibile utilizzare i metodi *getHeaderFieldKey* e *getHeaderField* oltre ad una serie di metodi specifici che consentono di reperire informazioni specifiche dallo header quali *getContentType, getContentLength, getDate, getExpiration (vedi API).*