Università degli Studi di Pisa
Dipartimento di Informatica

Lezione n.4
LPR INFORMATICA APPLICATA
CONNECTION ORIENTED SOCKETS

10/03/2008

Laura Ricci

RIASSUNTO DELLA PRESENTAZIONE

- Discussione di alcuni esercizi assegnati
- Il concetto di Stream in JAVA
- Meccanismi di comunicazione interprocess (IPC)
- Connection Oriented Sockets
- JAVA:Le classi ServerSocket e Socket

CALCOLO DI π

```
public class PiInterrupt implements Runnable
{   private double lastPiEstimate;
    public PiInterrupt(   ){  };
    private void calcPI(double accuracy) throws InterruptedException{
    lastPiEstimate=0.0;
    long iteration = 0;
    int sign = -1;
    while (Math.abs(lastPiEstimate - Math.PI) > accuracy)   {
     if (Thread.currentThread( ).isInterrupted()) throw new
                                                   InterruptedException();
     iteration ++;
     sign = - sign;
    lastPiEstimate += sign * 4.0 / (( 2*iteration) -1);
    }}}
```

CALCOLO DI π

```
public void run ( ){
    try{
        System.out.println(Math.PI);
        calcPI(0.00000001);
        System.out.println("Latest pi"+lastPiEstimate);}
   catch (InterruptedException x )
   {System.out.println("INTERRUPTED="+lastPiEstimate);   }}
```

CALCOLO DI π

```
public class PiInterruptMain {
 public static void main(String [  ] args)
   {PiInterrupt pi = new PiInterrupt(  );
   Thread t = new Thread(pi);
   t.start();
   try{
        Thread.sleep(100);
   } catch ( InterruptedException x ){    }
   t.interrupt();}}
```

CALCOLO DI π

Se modifico le classi PiInterrupt e ***PiInterruptMain*** come segue

```
public class PiInterrupt extends Thread
public class PiInterruptMain {
 public static void main(String [ ] args)
    {PiInterrupt pi = new PiInterrupt();
    pi.start();
try{
    Thread.sleep(10000);
    pi.interrupt();
} catch ( InterruptedException x ){}}}
```

posso utilizzare if (isInterrupted( )) .....

invece di if (Thread.currentThread ( ).isInterrupted( )) .....

CALCOLO DI π

Compito per casa:considerare la seguente versione (sbagliata!!) del programma,in cui la definizione di calcPI è la seguente:

```
public class PiInterrupt extends Thread {
..............
private void calcPI(double accuracy) throws InterruptedException{
lastPiEstimate=0.0;
long iteration = 0;
int sign = -1;
while (true)
    {  if (isInterrupted() )throw new InterruptedException();
       iteration ++;
       sign = - sign;
     lastPiEstimate += sign * 4.0 / (( 2*iteration) -1); }}
```

CALCOLO DI π

Ed il main è modificato come segue

```
public class PiInterruptMain {
    public static void main(String [ ] args)
      {PiInterrupt pi = new PiInterrupt( );
       Thread t = new Thread(pi);
          t.start();
          try{
           Thread.sleep(100);
          } catch ( InterruptedException x ){}
     t.interrupt();}}
```

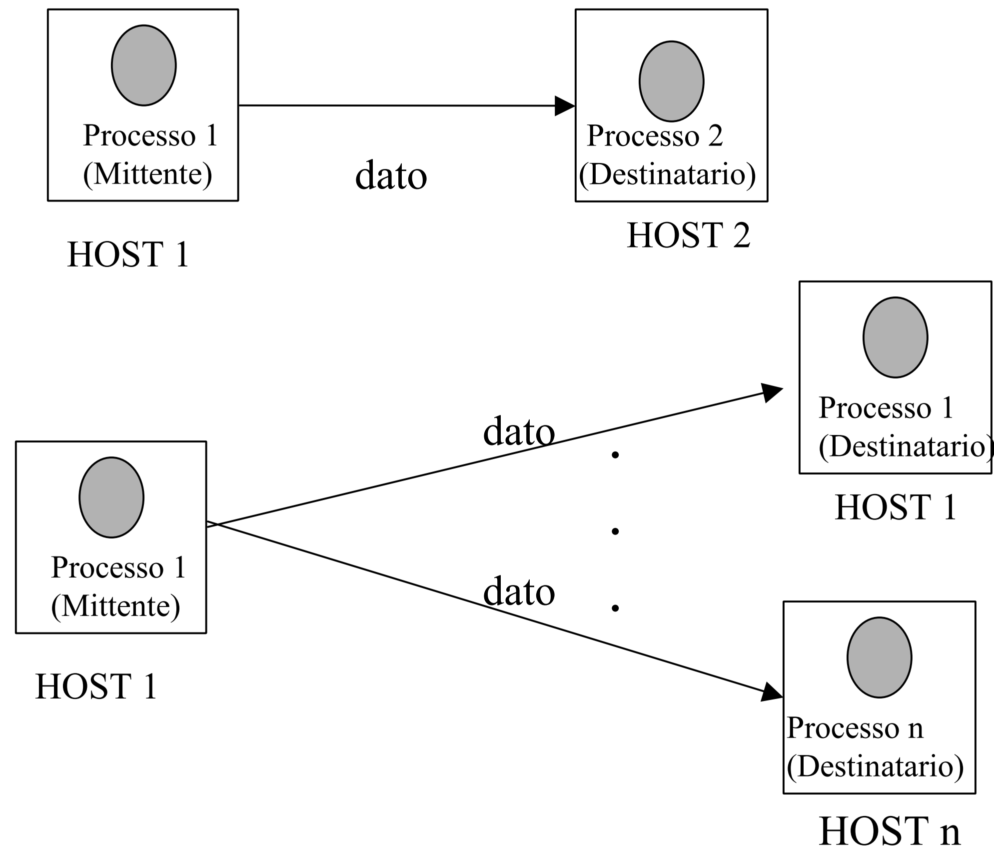
il programma non termina (l'interrupt non viene intercettato. Perchè?

'COMPITO PER CASA'

- Supponiamo
  - di attivare un insieme di threads in un thread pool
  - Si vogliono quindi interrompere alcuni threads del pool
  - I metodi per la creazione del thread pool non restituiscono puntatori ai threads attivati
  - La interrupt va applicata ad oggetti di tipo threads
  - Formulare una possibile soluzione

MECCANISMI DI COMUNICAZIONE TRA PROCESSI

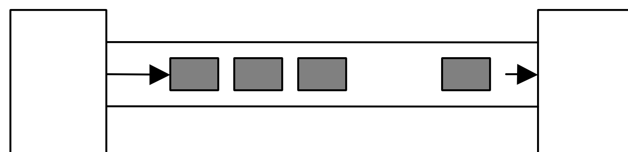
Meccanismi di comunicazione tra processi (IPC)



TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

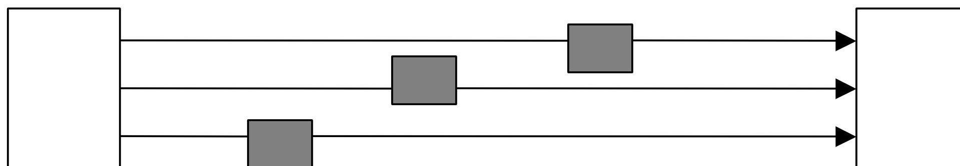
Comunicazione Connection Oriented **(come una chiamata telefonica)**

- creazione di una connessione (canale di comunicazione dedicato) tra mittente e destinatario
- invio dei dati sulla connessione
- chiusura della connessione



Comunicazione Connectionless (come l'invio di una lettera)

- non si stabilisce un canale di comunicazione dedicato
- mittente e destinatario comunicano mediante lo scambio di pacchetti



TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

Connection oriented vs. Connectionless

- Indirizzamento:
  - **Connection Oriented:** l'indirizzo del destinatario è specificato al momento della connessione
  - **Connectionless:** l'indirizzo del destinatario viene specificato in ogni pacchetto (per ogni send)
- Ordinamento dei dati scambiati:
  - **Connection Oriented:** ordinamento dei messaggi garantito
  - **Connectionless:** nessuna garanzia sull'ordinamento dei messaggi
- Utilizzo:
  - **Connection Oriented:** grossi streams di dati
  - **Connectionless:** invio di un numero limitato di dati

TIPI DI COMUNICAZIONE: CONNECTION ORIENTED VS. CONNECTIONLESS

Protocollo UDP = connectionless, trasmette pacchetti dati = Datagrams

- ogni datagram deve contenere l'indirizzo del destinatario
- datagrams spediti dallo stesso processo possono seguire percorsi diversi ed arrivare al destinatario in ordine diverso rispetto all'ordine di spedizione

Protocollo TCP = trasmissione connection-oriented o stream oriented

- viene stabilita una connessione tra mittente e destinatario
- su questa connesione si spedisce una sequenza di dati = stream di dati
- per modellare questo tipo di comunicazione in JAVA si possono sfruttare i diversi tipi di stream definiti dal linguaggio.

IPC: MECCANISMI BASE

Una API per la comunicazione tra processi deve garantire almeno le seguenti funzionalità

- **Send** per trasmettere un dato al processo destinatario
- **Receive** per ricevere un dato dal processo mittente
- **Connect** (solo per comunicazione connection oriented) per stabilire una connessione logica tra mittente e destinatario
- **Disconnect** per eliminare una connessione logica

Possono esistere diversi tipi di send/receive (sincrona/asincrona, simmetrica/asimmetrica)

IPC: MECCANISMI BASE

Un esempio: HTTP (1.0)

- il processo che esegue il Web browser esegue una **connect** per stabilire una connessione con il processo che esegue il Web Server
- il Web browser esegue una **send**, per trasmettere una richiesta al Web Server (operazione GET)
- il Web server esegue una **receive** per ricevere la richiesta dal Web Browser, quindi a sua volta esegue una **send** per inviare la risposta
- i due processi eseguono una **disconnect** per terminare la connessione

HTTP 1.1: Più richieste su una connessione (più send e receive).

IPC: MECCANISMI BASE

Comunicazione sincrona (o bloccante): il processo che esegue la send o la receive si sospende fino al momento in cui la comunicazione è completata.

send sincrona = completata quando i dati spediti sono stati ricevuti dal destinatario (è stato ricevuto un ack da parte del destinatario)

receive sincrona = completata quando i dati richiesti sono stati ricevuti

send asincrona (non bloccante) = il destinatario invia i dati e prosegue la sua esecuzione senza attendere un ack dal destinatario

receive asincrona = il destinatario non si blocca se i dati non sono arrivati. Possibile diverse implementazioni

IPC: MECCANISMI BASE

Receive Non Bloccante.

- se il dato richiesto è arrivato, viene reso disponibile al processo che ha eseguito la receive

- se il dato richiesto non è arrivato:

  - il destinatario esegue nuovamente la receive, dopo un certo intervallo di tempo (polling*)*

  - il supporto a tempo di esecuzione notifica al destinatario l'arrivo del dato (richiesta l'attivazione di un event listener)

IPC: MECCANISMI BASE

Comunicazione non bloccante: per non bloccarsi indefinitamente

- **Timeout** – meccanismo che consente di bloccarsi per un intervallo di tempo prestabilito, poi di proseguire comunque l'esecuzione

- **Threads** – l'operazione sincrona può essere effettuate in un thread. Se il thread si blocca su una send/receive sincrona, l'applicazione può eseguire altri thread.

- Nel caso di receive sincrona, gli altri threads ovviamente non devono richiedere per l'esecuzione il valore restituito dalla receive

INVIARE OGGETTI

Invio di strutture dati ed, in generale, di oggetti richiede :

- il mittente deve effettuare la **serializzazione** delle strutture dati (eliminazione dei puntatori)
- il destinatario deve ricostruire la struttura dati nella sua memoria

Da Wikipedia: ....La serializzazione è il processo richiesto per memorizzare un oggetto su un supporto di memorizzazione (un file,...) oppure per trasmettere un oggetto mediante un collegamento di rete, trasformandolo in una sequenza di bytes. La sequenza di bytes può essere utilizzata per ricreare un oggetto con uno stato identico a quello spedito (in modo da creare un clone dell'oggetto originario).

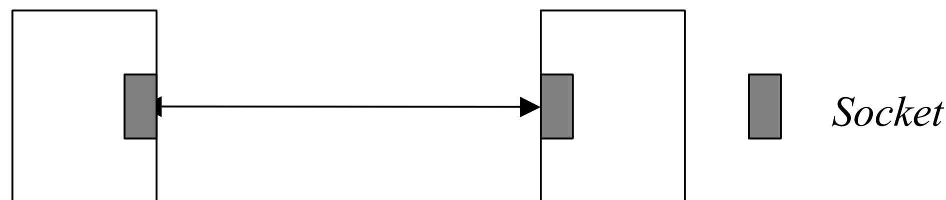
Serializzazione: il processo di serializzare un oggetto viene anche indicato come marshalling (operazione opposta = deserializzazione)

JAVA IPC: I SOCKETS

Socket = presa di corrente

Termine utilizzato in tempi remoti in telefonia. La connessione tra due utenti veniva stabilita tramite un operatore che inseriva fisicamente i due estremi di un cavo in due ricettacoli (*sockets*), ognuno dei quali era assegnato ai due utenti.

Socket è una astrazione che indica una "presa " a cui un processo si può collegare per spedire dati sulla rete. Al momento della creazione un socket viene collegato ad una porta.



JAVA IPC: I SOCKETS

Studiare capitolo 13 Elliotte Rusty Harold

Socket Application Program Interface = Definisce un insieme di meccanismi che supportano la comunicazione di processi in ambiente distribuito.

- JAVA socket API: definisce interfacce diverse per UDP e TCP

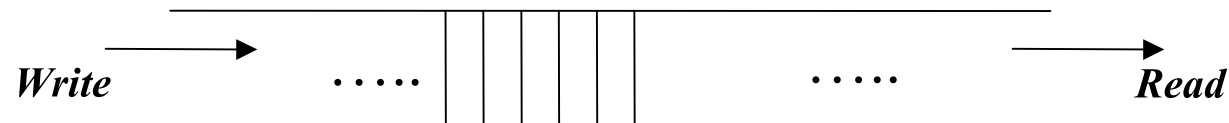
  Protocollo UDP = Datagram Sockets

  Protocollo TCP = Stream Sockets

JAVA: IL CONCETTO DI STREAM

- Streams: introdotti per modellare l'interazione del programma con i dispositivi di I/O (console, files, connessioni di rete,…)

- JAVA Stream I/O: basato sul concetto di *stream: si può immaginare uno stream come una condotta* tra una sorgente ed una destinazione (dal programma ad un dispositivo o viceversa), da un estremo entrano dati, dall'altro escono



- L'applicazione può inserire dati ad un capo dello stream

- I dati fluiscono verso la destinazione e possono essere estratti dall'altro capo dello stream Esempio: l'applicazione scrive su un FileOutputStream. Il dispositivo legge i dati e li memorizza sul file

JAVA: IL CONCETTO DI STREAM

Caratteristiche principali degli streams:

- mantengono l'ordinamento FIFO
- read only o write only
- accesso sequenziale
- bloccanti: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca finchè l'operazione non è completata (ma le ultime versioni di JAVA introducono l' I/O non bloccante
- non è richiesta una corrispondenza stretta tra letture/scritture

  esempio: una unica scrittura inietta 100 bytes sullo stream, che vengono letti con due write successive, la prima legge 20 bytes, la seconda 80 bytes)

JAVA: USO DEGLI STREAM PER LA PROGRAMMAZIONE DI RETE

Come utilizzeremo gli streams in questo corso:

- Trasmissione connection oriented:

  Una connessione viene modellata con uno stream.

  invio di dati = scrittura sullo stream

  ricezione di dati = lettura dallo stream

- Trasmissione connectionless:

  ByteArrayOutputStream, generano streams di bytes che possono essere convertiti in vettori di bytes da spedire con i pacchetti UDP

  ByteArrayInputStream, converte un vettore di bytes in uno stream di byte. Consente di manipolare più agevolmente i bytes

JAVA: STREAMS DI BASE

Streams di bytes:

public abstract class OutputStream

Metodi di base:

public abstract void write (*int* b) *throws* IOException;

public void write(*byte* [ ] data) *throws* IOException;

public void write(***byte*** [ ] data, ***int*** offeset, ***int*** length) *throws* IOException;

write (***int*** b) scrive su un OuputStream il byte corrispondente all'intero passato

Gli ultimi due metodi consentono la scrittura di gruppi di bytes.

Analogamente la classe *InputStream*

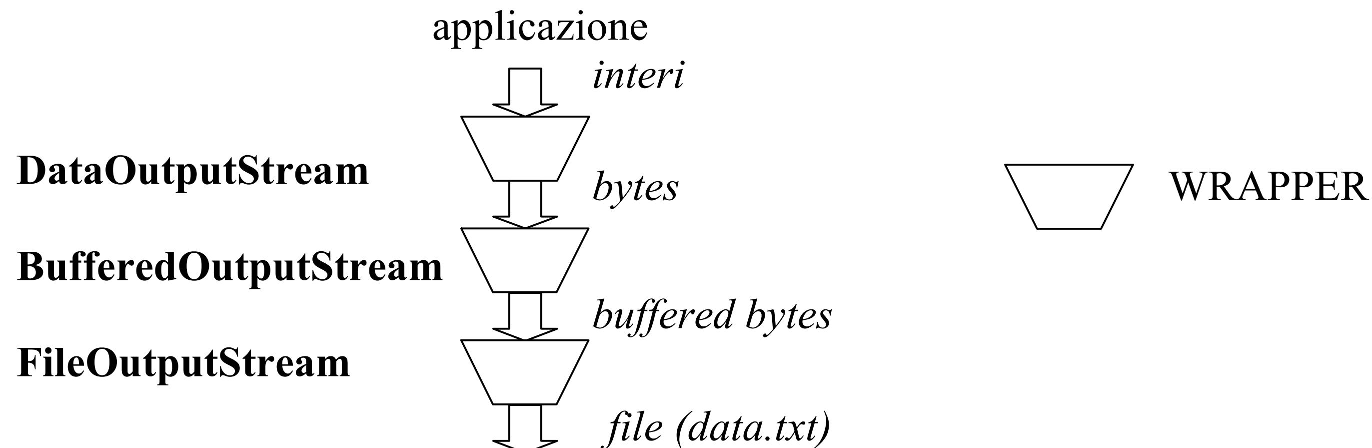
JAVA: STREAMS DI BASE E WRAPPERS

- La classe *OutputStream* ed il metodo *write* sono dichiarati astratti.
- Le sottoclassi descrivono stream legati a particolari dispositivi di I/O (file, console,…) .
- L'implementazione del metodo *write* può richiedere codice nativo (es: scrittura su un file…).
- Stream di base: classi utilizzate
  - *Stream utilizzate per la trasmissione di bytes
  - *Reader o *Writer: utilizzate per la trasmissione di caratteri
- Per non lavorare direttamente a livello di bytes o di carattere
  - Si definisce una serie di wrappers che consentono di avvolgere uno stream intorno all'altro (come un tubo composto da più guaine...)
  - l'oggetto più interno è uno stream di base che 'avvolge' la sorgente dei dati (ad esempio il file, la connessione di rete,....).
  - i wrappers sono utilizzati per il trasferimento di oggetti complessi sullo stream, per la compressione di dati, per la definizione di strategie di buffering (per rendere piu' veloce la trasmissione)

JAVA: STREAMS DI BASE E WRAPPERS

InputStream, OuputStream consentono di manipolare dati a livello molto basso, per cui lavorare direttamente su questi streams risulta parecchio complesso.
Per estendere le funzionalità degli streams di base: classi wrapper

```
DataOutputStream= new DataOutputStream(
                new  BufferedOutputStream(
                    new FileOutputStream("data.txt")))
```



JAVA STREAMS: FILTRI

DataOutputStream consente di trasformare dati di un tipo primitivo JAVA in una sequenza di bytes da iniettare su uno stream.
Alcuni metodi utili:

```
public final void writeBoolean(boolean b) throws IOException;
public final void writeInt (int i) throws IOException;
public final void writeDouble (double d) throws IOException;
```

......

Il filtro produce una sequenza di bytes che rappresentano il valore del dato.
Rappresentazioni utilizzate:

- interi 32 bit big-endian, complemento a due
- float 32 bit IEEE754 floating points

Formati utilizzati dalla maggior parte dei protocolli di rete
Nessun problema se i dati vengono scambiati tra programmi JAVA.

JAVA STREAMS: BUFFERIZZAZIONE

BufferedOutputStream

- memorizza una sequenza di bytes in un buffer B (un *byteArray)*

- copia i dati da B allo stream quando risulta verificata una delle seguenti condizioni:
  - B è pieno
  - viene effettuata una operazione di flush sullo stream:

    esempio: bos.flush ( ), se bos è un buffered outputstream

- la dimensione di B può essere stabilita al momento della costruzione del filtro, oppure stabilita per default (512 bytes), usando un costruttore opportuno

  - public BufferedOutputStream(OutputStream out)
  - public BufferedOutputStream(OutputStream out, *int* bufferSize)

JAVA STREAMS: BUFFERIZZAZIONE

- E' importante valutare la relazione tra dimensione del buffer e quantità di dati da inviare sullo stream
- Esempio: un client utilizza una connessione TCP,a cui è associato uno stream
  - invia una richiesta (300 bytes) ad un server HTTP. Utilizza un BufferOutputStream BS (dimensione del buffer = 1024 bytes)
  - attende risposta dal server prima di inviare una nuova richiesta

$\Rightarrow$

il client si blocca indefinitamente, perchè
  - i dati bufferizzati non riempiono completamente il buffer
  - il buffer non viene spedito
  - il client non riceve alcuna risposta dal server e non può inviare la nuova richiesta
- Soluzione: effettuare il flush dello stream dopo aver inviato la prima richiesta

JAVA STREAMS: INPUTSTREAMS

public abstract class InputStream

metodi:

- **public abstract** *int* read() **throws** IOException
- **public int** *r*ead(byte[]input) **throws** IOException
- **public long** skip(long n) **throws** IOException

- Skip consente di bypassare una certa sequenza di dati senza leggerli

$\Rightarrow$

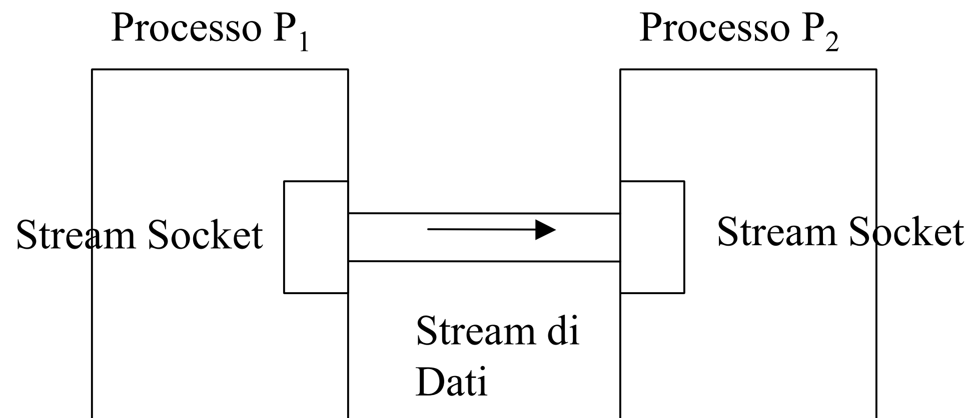
- BufferInputStream: definisce un buffer B che viene riempito *on demand*. Quando l'utente richiede la lettura di una sequenza di k bytes dallo stream, controlla se i k bytes sono disponibili in B. Se sullo stream sono disponibili, in quel momento, n bytes
  - se n>k i bytes rimanenti vengono memorizzati nel buffer per successive letture
  - se n<k la lettura si blocca

STREAM MODE SOCKET API

- **stream mode sockets** : supportano la comunicazione connection oriented

- estensione del modello di base di I/O basato su streams definito in UNIX e JAVA

- concetto base: associare uno stream di input/ouput ad un socket



Studiare capitoli 9, 10 JAVA NETWORK PROGRAMMING !!

STREAM MODE SOCKET API: LATO SERVER

La comunicazione Connection-Oriented prevede due fasi:

- il client richiede una connessione al server
- quando il server accetta la connessione, client e server iniziano a scambiarsi i dati

Stream Mode Socket API: fornisce operazioni per l'implementazione del modello client/server

Il server utilizza diversi tipi di socket:

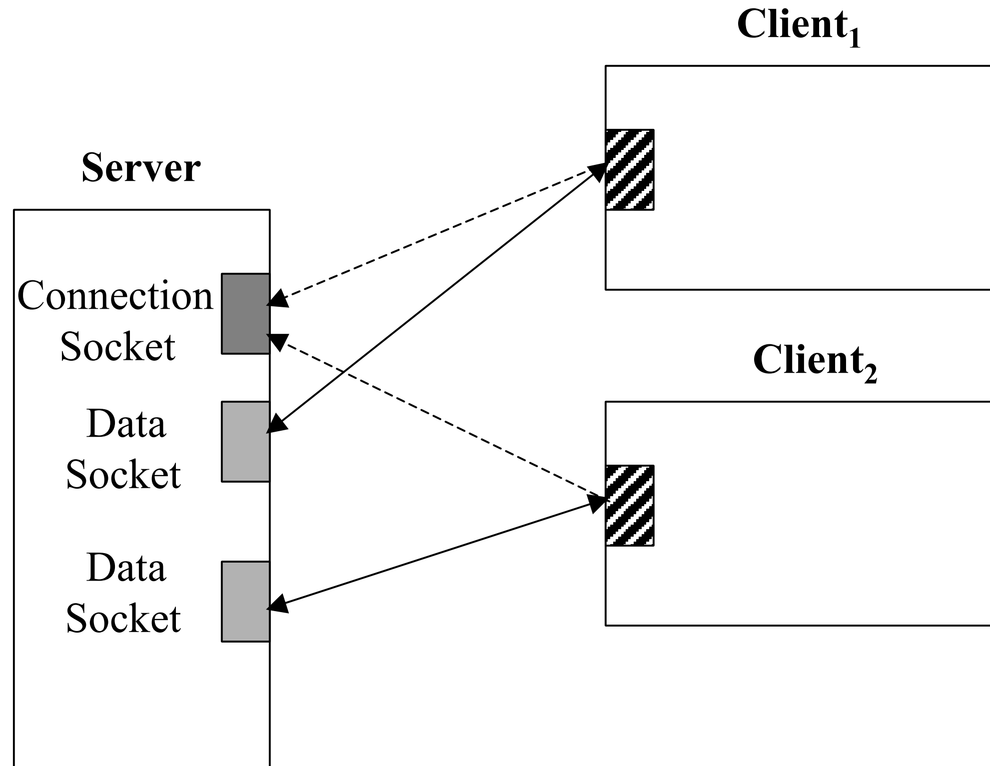
connection socket

per accettare richieste di connessione

data socket

per scambiare i dati con un client con cui si è stabilita una connessione

STREAM MODE SOCKET API



STREAM MODE SOCKET API
LATO CLIENT

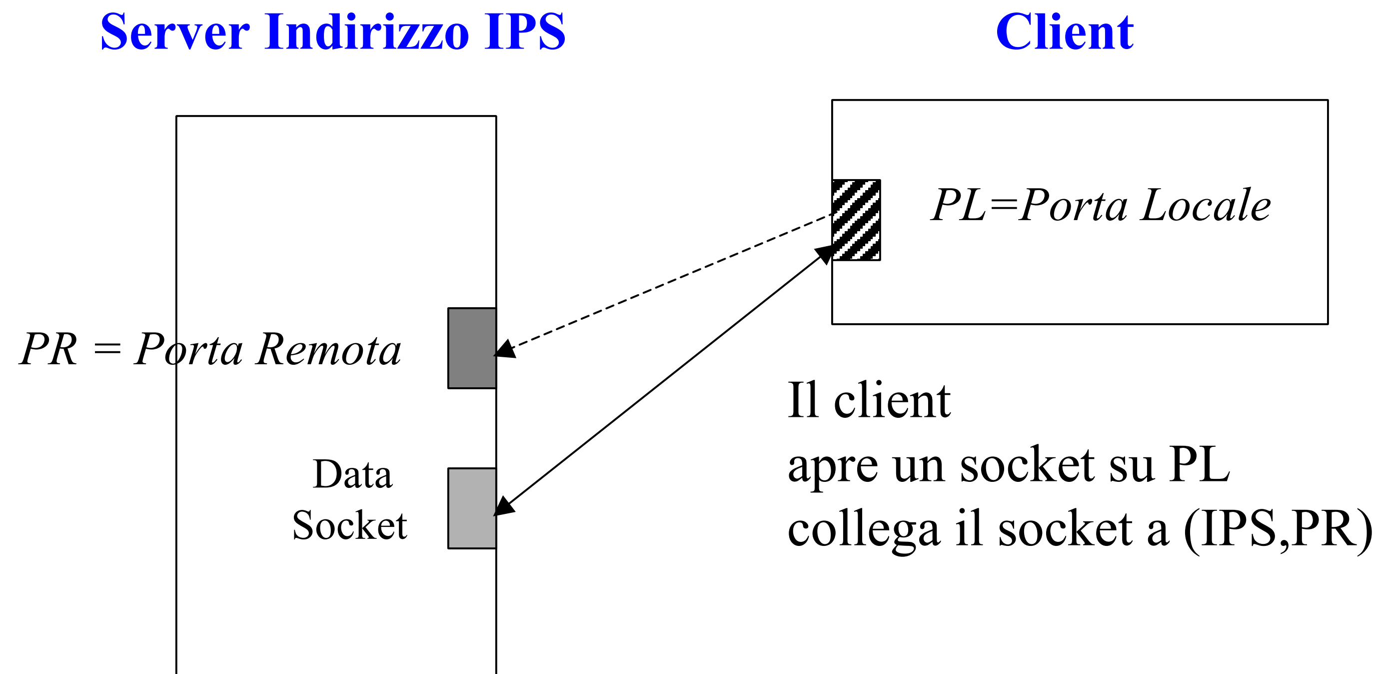
- Il client C intende usufruire di **un servizio** offerto da un Server S, di cui conosce l'indirizzo IP, IPS, su **una porta** remota nota PR.

  - C chiede l'apertura di una connessione
  - la richiesta di apertura viene effettuata mediante
    - la creazione di un connection socket SS
    - il collegamento del socket all'indirizzo IPS, sulla porta PR

- SS quindi collega
  - una **porta locale PL di C** che può essere anonima

    con
  - la **porta remota PR** sull'host di indirizzo IPS

STREAM MODE SOCKET API
LATO CLIENT



Il client
apre un socket su PL
collega il socket a (IPS,PR)

STREAM MODE SOCKET API
LATO CLIENT

- la richiesta di creazione del socket produce in modo atomico la richiesta di connessione al server o lancia una eccezione se la connessione non viene accettata

- il protocollo di richiesta della connessione viene gestito dal supporto

- il socket viene utilizzato solo per quella connessione, connessioni diverse utilizzano sockets diversi

- quando la richiesta di connessione viene accettata dal server, il supporto in esecuzione sul server crea in modo automatico un nuovo socket data socket DS.

- DS è utilizzato per l'interazione con il client. Tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

STREAM MODE SOCKET API
LATO CLIENT

Dopo che la richiesta di connessione viene accettata, il client

- associa uno stream di bytes di input e/o di output al socket
- allo stream di bytes possono essere associati opportuni filtri
- la comunicazione con il server avviene mediante la lettura/scrittura di dati sullo stream creato
- alla fine dell'interazione, il client chiude il socket

STREAM MODE SOCKET API: LATO CLIENT

Gestione sockets lato client:i costruttori definiti in JAVA tendono a nascondere diversi dettagli implementativi.

Classe *java.net.Socket :* costruttori

public socket(InetAddress host, int port) throws IOException

Crea un socket TCP e tenta di stabilire, tramite esso, una connessione con l'host individuato da InetAddress, sulla porta port. Se la connessione viene rifiutata, lancia una eccezione di IO

public socket (String host, int port) throws UnKnownHostException, IOException

Come il precedente, l'host è individuato dal suo nome simbolico (interroga automaticamente il DNS)

INDIVIDUAZIONE DI SERVIZI TCP

Esercizio: ricerca dei servizi TCP attivi sulle prime 1024 porte di un host H.

PortScanner: *si richiede la connessione* ad ognuna delle 1024 portedi H, mediante la creazione di un socket su quella porta. Si verifica la presenza/ assenza del servizio mediante la rilevazione di un'eccezione

```
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"; };
```

STREAM MODE SOCKET API: LATO CLIENT

```
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);}
     ;}   }     }
```

STREAM MODE SOCKET API: LATO CLIENT

- PortScanner: effettua 1024 interrogazioni al DNS, una per ogni socket che tenta di creare
- Ottimizzazione: utilizzare il costruttore

```
public Socket(InetAddress host, int port) throws IOException
```

- il DNS viene interrogato una sola volta, prima di entrare nel ciclo di scanning (InetAddress.getByName)
- Si utilizza l'InetAddress invece del nome dell'host per costruire i sockets

STREAM MODE SOCKET API: LATO CLIENT

Altri costruttori della Classe java.net.socket

public Socket (String H, int P, InetAddress IA, int LP)

tenta di creare una connessione

- *verso* /host H,
- sulla porta P.
- dalla interfaccia locale IA
- dalla porta locale LP

STREAM MODE SOCKET API
LATO CLIENT

Associazione di streams di input/output ad un socket connection- oriented

public **InputStream getInputStream( )** throws **IOException**

Associa un InputStream (stream di bytes) ad un oggetto di tipo Socket.

Il client può leggere successivamente dallo stream

- un byte per volta
- dati di tipo qualsiasi (anche oggetti) mediante l'uso di filtri (DataInputStream, ObjectInputStream,...)

public **OutputStream getOutputStream( )** throws **IOException**

Associa un OutputStream ad un socket.

STREAM MODE SOCKET API: INTEZIONE CON SERVERS PREDEFINITI

Esercizio: considerare un servizio attivo su una porta pubblicata da un Server (es *23 Telnet, 25 SMTP, 80 HTTP).* Definire un client JAVA che utilizzi tale servizio. Si possono considerare i seguenti semplici servizi (vedere JAVA Network Programming)

Daytime(porta 13): il client richiede una connessione sulla porta 13, il server invia la data e chiude la connessione

Echo (port 7): il client apre una connesione sulla porta 7 del server ed invia un messaggio. Il server restituisce il messaggio al client

Finger (porta 79): il client apre una connessione ed invia una query, il Server risponde alla query

Whois (porta 43): il client invia una stringa terminata da return/linefeed.La stringa può contenere, ad esempio, un nome. Il server invia alcune informazioni correlate a quel nome

STREAM MODE SOCKET API: STRUTTURA DEL SERVER

Comportamento di un **Server Sequenziale**:

- crea un **connection socket CS** sulla porta associata al servizio pubblicato.
- si mette in ascolto su CS (si blocca fino al momento in cui arriva una richiesta di connessione)
- quando accetta una richiesta di connessione da parte di un client C, crea un nuovo **Data Socket** su cui avviene la comunicazione con C
- associa al **DataSocket** uno o più stream (di input e/o di output) su cui avverrà la comunicazione con il client
- quando l'interazione con il client è terminata, chiude il data socket e torna ad ascoltare su CS ulteriori richieste di connessione

STREAM MODE SOCKET API
LATO SERVER

Classe *java.net.ServerSocket:* costruttori

```
public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int length) throws BindException,
                                                        IOException
```

- costruisce un connection socket, associandolo alla porta p. Length indica la lunghezza della coda in cui vengono memorizzate le richieste di connessione. (lunghezza massima della coda stabilita dal sistema operativo).
- se la coda è piena, eventuali ulteriori richieste di connessione **vengono rifiutate.**

STREAM MODE SOCKET API
LATO SERVER

Gestione sockets lato server:

Classe *java.net.ServerSocket* : altri costruttori

```
public ServerSocket(int port, int length, Inetaddress bindAddress)
                throws BindException,IOException
```

- permette di collegare il connection socket ad uno specifico indirizzo IP locale.
- utile per macchine dotate di più schede di rete.

Esempio: un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale. Posso decidere di accettare connessioni solo dalla rete locale ⇒ associo il connection socket all'indirizzo IP locale

STREAM MODE SOCKET API
LATO SERVER

Esempio: ricerca dei servers 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);}
        } }
```

STREAM MODE SOCKET API
LATO SERVER

Accettare una nuova connessione dal **connection socket**

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

metodo della classe *ServerSocket*. Comportamento:

- 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 (possibile utilizzo di time-outs)
- se c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket S tramite cui avviene la comunicazione effettiva tra cliente server

PROTOCOLLO HTTP

Messaggio di richiesta HTTP GET /dir/requestpage.html HTTP/1.1

Host: www.dipinf.edu

Connection: close

User-agent: mozilla/4.0

Accept-languare:it

Messaggio di risposta HTTP/1.0 200 ok

Connection: close

Date:...

Server: Apache/1.3.0

Content Length:6821

Content-Type: text/html

<dati><dati><dati><dati><dati>

UN SERVER HTTP (SEMPLIFICATO)

```
import java.io.*; import java.net.*; import java.util.*;
public class HttpWelcome {
    private static int port = 80;
    // ipotesi: il welcome message contiene il contenuto della pagina richiesta
    private static String HtmlWelcomeMessage () {
    return "<html>\n"+
    " <head>\n"+" <title>UNIPI - Corso di Laurea in Informatica</title>\n"+
    " </head>\n"+
    " <body>\n"+ " <h2 align=\"center\">\n"+
    " <font color=\"#0000FF\">Benvenuti al Corso di"+
    " Laboratorio di Programmazione di Rete</font>\n"+ " </h2>\n"+
    " </body>\n"+
    "</html>";  };
```

UN SERVER HTTP (SEMPLIFICATO)

```
public static void main (String args[  ]) {
try { ServerSocket server = new ServerSocket(port);
        System.out.println("HTTP server running on port: "+port);
         while (true) {
             Socket client = server.accept();
             DataInputStream in = new DataInputStream(client.getInputStream());
             DataOutputStream out =
               (newDataOutputStream(client.getOutputStream()));
             String request = in.readUTF();
             System.out.println("Request: "+request);
             StringTokenizer st = new StringTokenizer(request," ");
```

UN SERVER HTTP (SEMPLIFICATO)

```
if (st.nextToken(" ").equals("GET") && st.nextToken(" ").equals("message"))
    { String message = HtmlWelcomeMessage();
    out.writeUTF("HTTP/1.0 200 OK"+"Content-Length: "
      +message.length()+ "Content-Type: text/html"+""+message);
    } else { out.writeUTF("400 Bad Request");}
    out.flush();
    client.close();
    }
    } catch (Exception e) { System.err.println(e); }}
```

UN CLIENT HTTP (SEMPLIFICATO)

```
import java.io.*;
import java.net.*;
public class httpclient {
public static void main (String args [ ])throws Exception
    {InetAddress ia=InetAddress.getLocalHost();
    Socket s =new Socket(ia,80);
    DataOutputStream out =
    (new DataOutputStream(s.getOutputStream()));
    out.writeUTF("GET"+" "+"message"+" "+"HTTP/1.0"+" ");
    DataInputStream in = new DataInputStream(s.getInputStream());
    String reply = in.readUTF(); System.out.println(reply);} }
```

ESERCIZIO

- Si vuole progettare una mini-calcolatrice client / server
  - Il client invia al server una serie di righe di testo, contenenti numeri interi (uno per riga); l'ultima riga contiene lo zero.
  - Il server effettua la somma dei valori ricevuti e la ritrasmette al client.
- Analizzare attentamente le API JAVA per decidere quale classi filtro utilizzare.