LEZIONE N.7
LPR-A INFORMATICA
IL PROTOCOLLO TCP: STREAM SOCKETS

10/11/2008

Laura Ricci

DATAGRAM SOCKET API: RIASSUNTO DELLE PUNTATE PRECEDENTI

- lo stesso Datagram Socket può essere utilizzato per spedire messaggi verso destinatari diversi
- processi diversi possono inviare datagrams sullo stesso socket di un processo destinatario
- send non bloccante

  se il destinatario non è in esecuzione quando il mittente esegue la send, il messaggio può venir scartato
- receive bloccante

  uso di timeouts associati al socket per non bloccarsi indefinitamente sulla receive
- i messaggi ricevuti possono essere troncati se la dimensione del buffer del destinatario è inferiore a quella del messaggio spedito

DATAGRAM SOCKET API: RIASSUNTO DELLE PUNTATE PRECEDENTI

- protocollo UDP: non implementa controllo del flusso:

  se la frequenza con cui il mittente invia i messaggi è sensibilmente maggiore di quella con cui il destinatario li riceve (li preleva dal buffer di ricezione) è possibile che alcuni messaggi sovrascivano messaggi inviati in precedenza

- Esempio: CountDown Server (vedi esercizio lezione 5).

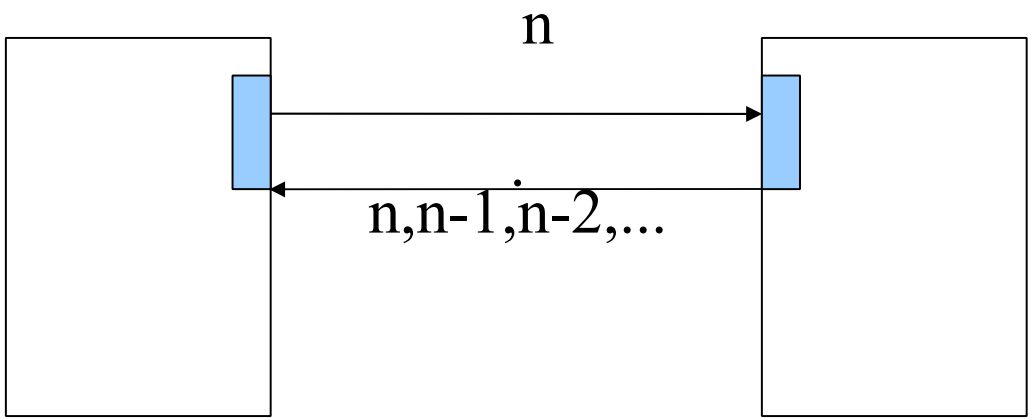
  Il client invia al server un valore di n molto grande

  (provare valori>1000). Allora:

  - il server deve inviare al client un numero molto alto di pacchetti
  - il tempo che intercorre tra l'invio di un pacchetto e quello del pacchetto successivo è basso
  - dopo un certo numero di invii il buffer del client si riempie⇒ perdita di pacchetti

COUNT DOWN SERVER UDP

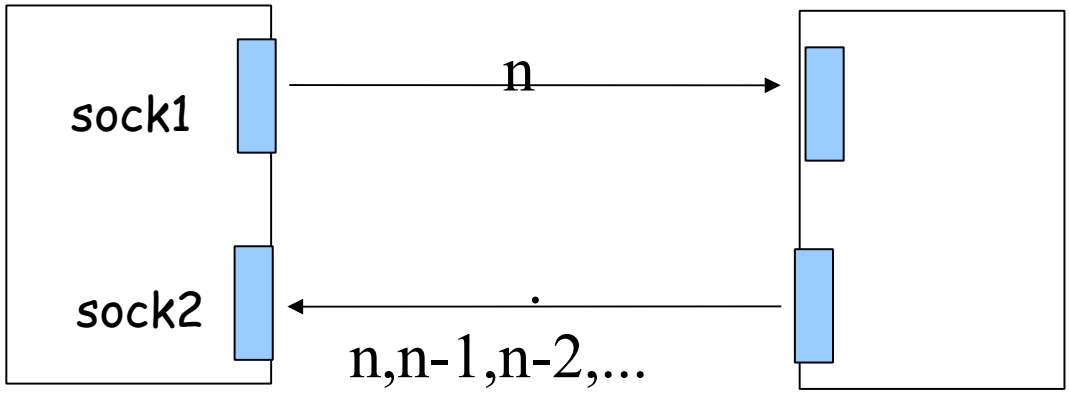
CountDownClient CountDownServer



- Si può utilizzare lo stesso socket per inviare il valore n e per ricevere i risultati
- Quando il CountDownClient invia il valore n il CountDownServer deve aver allocato il socket, altrimenti il Messaggio viene perduto

COUNT DOWN SERVER UDP

CountDownClient CountDownServer



- Posso utilizzare sockets diversi per la spedizione/ricezione
- In questo caso può accadere che sock2 non sia ancora stato creato quando CountDownServer inizia ad iniziare la sequenza di numeri
- E' possibile che il CountDownClient si blocchi sulla receive poiché i dati inviati dal CountDownServer sono stati inviati prima della creazione del socket e quindi sono andati persi

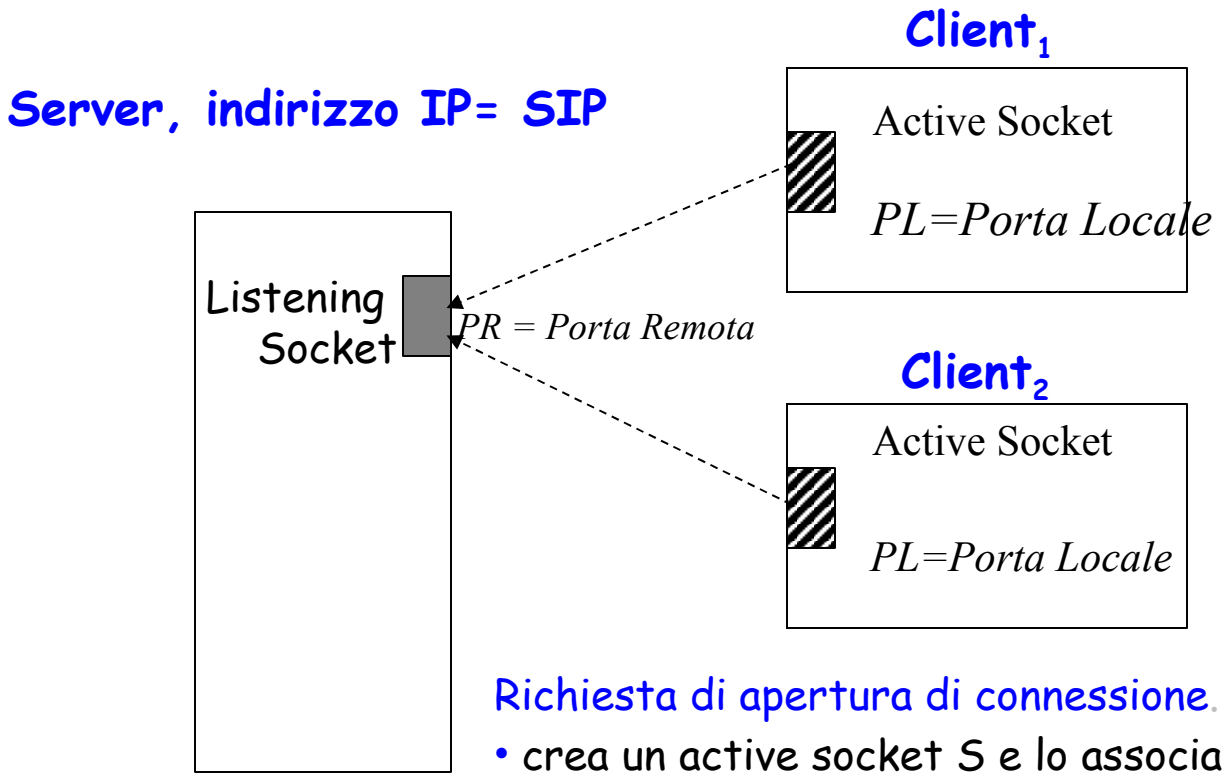
IL PROTOCOLLO TCP: STREAM SOCKETS

- Il protocollo TCP supporta
  - un modello computazionale di tipo client/server, in cui il server riceve dai clients richieste di connessione, le schedula e crea connessioni diverse per ogni richiesta ricevuta
  - ogni connessione supporta comunicazioni bidirezionali, affidabili
- 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
- In JAVA, ogni connessione viene modellata come uno stream continuo di bytes
  - i dati non vengono incapsulati in messaggi (pacchetti)
  - stream sockets: al socket sono associati stream di input/output
  - basato sul modello di I/O basato su streams definito in UNIX e JAVA

IL PROTOCOLLO TCP: STREAM SOCKETS

- Esistono due tipi di socketTCP:
  - Listening (o passive) sockets: utilizzati dal server per accettare le richieste di connessione
  - Active Sockets: supportano lo streaming di byte tra client e server
- Il server utilizza un listening socket per accettare le richieste di connessione dei clients
- Il client crea un active socket per richiedere la connessione
- quando il server accetta 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 sockets presenti nel client e nel server

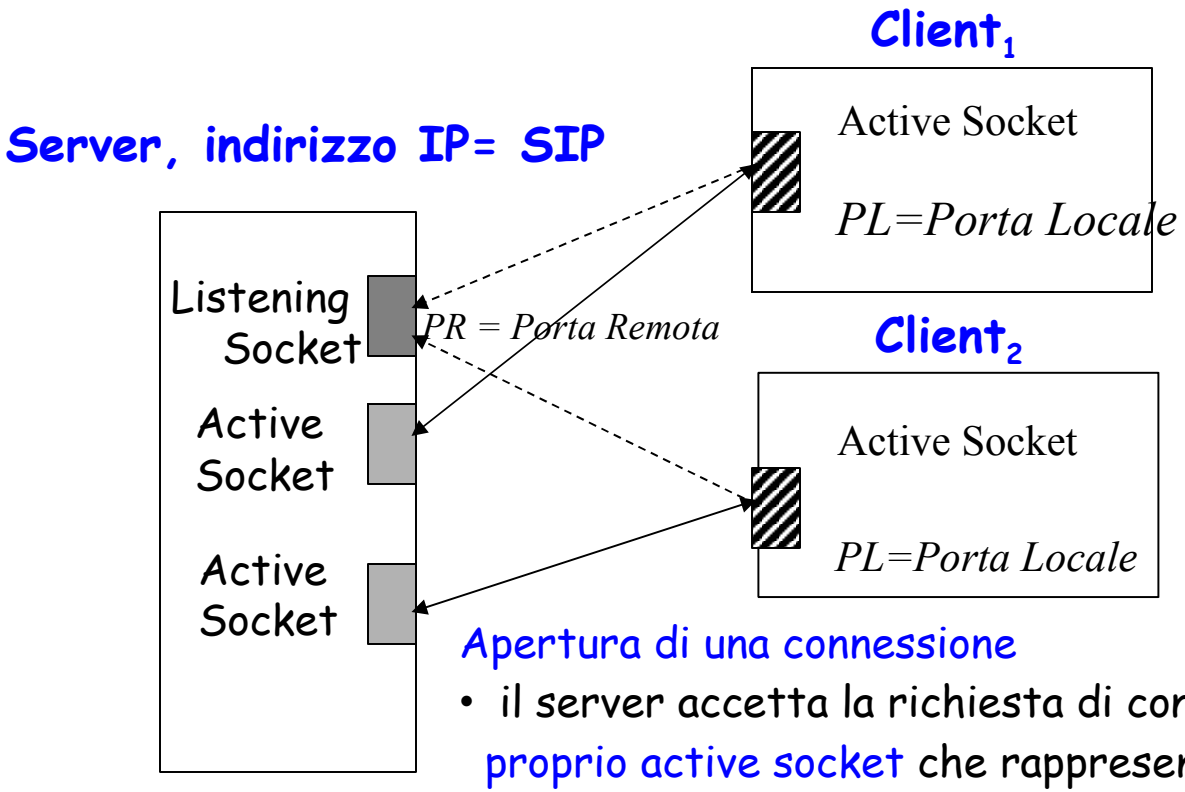
IL PROTOCOLLO TCP: STREAM SOCKETS



Richiesta di apertura di connessione. Il client

- crea un active socket S e lo associa alla sua porta locale PL
- collega S al listening socket presente sul server pubblicato all'indirizzo (SIP,PR)

IL PROTOCOLLO TCP: STREAM SOCKETS



Apertura di una connessione

- il server accetta la richiesta di connessione e crea un proprio active socket che rappresenta il suo punto terminale della connessione
- tutti i segmenti TCP scambiati tra client e server vengono trasmessi mediante la coppia di active sockets creati

IL PROTOCOLLO TCP: STREAM SOCKETS

- Il server pubblica un proprio servizio associandolo al listening socket, creato sulla porta remota PR

- il client C che intende usufruire del servizio deve conoscere l'indirizzo IP del server, SIP ed il riferimento alla porta remota PR a cui è associato il servizio
- la richiesta di creazione del socket
  - produce in modo atomico la richiesta di connessione al server
  - il protocollo di richiesta della connessione viene completamente gestito dal supporto

- quando la richiesta di connessione viene accettata dal server, il supporto in esecuzione sul server crea in modo automatico un nuovo active socket AS.
  - AS è utilizzato per l'interazione con il client. Tutti i messaggi spediti dal client vengono diretti automaticamente sul nuovo socket creato.

STREAM SOCKET JAVA API: LATO CLIENT

Classe **java.net.Socket** : costruttori

public socket(InetAddress host, **int** port) **throws** IOException

crea un active socket 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)

PORT SCANNER: INDIVIDUAZIONE SERVIZI TCP ATTIVI SU UN HOST

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

PORT SCANNER: INDIVIDUAZIONE SERVIZI TCP ATTIVI SU UN HOST

- Nella classe PortScanner
  - il client richiede la connessione tentando di creare un socket su ognuna delle prime 1024 porte di un host
  - nel caso in cui non vi sia alcun servizio attivo, il socket non viene creato e viene invece sollevata un'eccezione
  - Osservazione: il programma effettua 1024 interrogazioni al DNS, una per ogni socket che tenta di creare
- Per ottimizzare il comportamento del programma: 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, dalla InetAddress.getByName
  - viene utilizzato l'InetAddress invece del nome dell'host per costruire i sockets

STREAM SOCKET API

Altri costruttori della Classe java.net.socket

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

tenta di creare una connessione

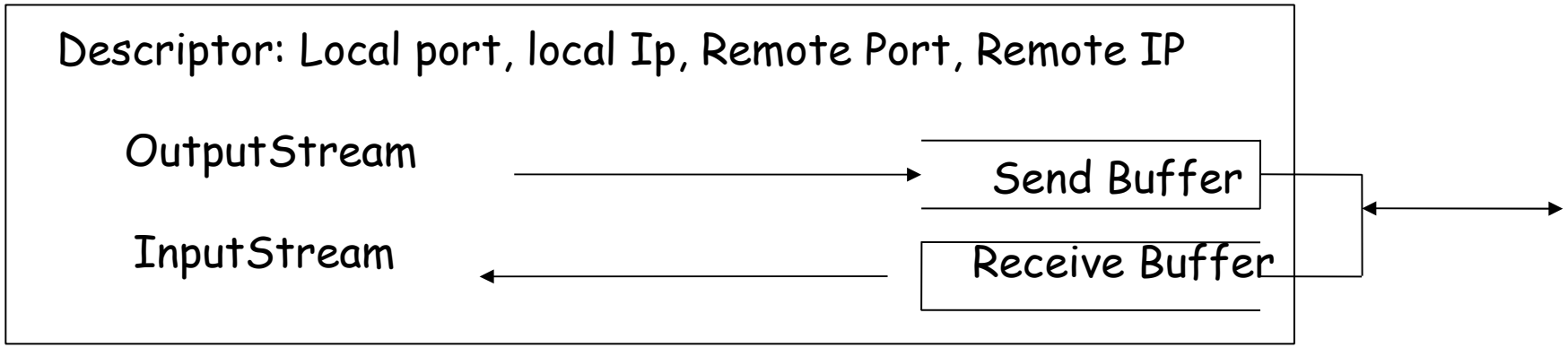
- verso l'host H,
- sulla porta P.
- dalla interfaccia locale IA
- dalla porta locale LP

STREAM BASED COMMUNICATION

Dopo che la richiesta di connessione viene accettata, client e server

- associano all'active socket streams di byte di input/output
- poichè gli stream sono unidirezionali stream diversi, associati allo stesso socket,sono utilizzati rs.per l'input e per l'output
- la comunicazione avviene mediante lettura/scrittura di dati sullo stream
- Utilizzo di filtri associati agli stream

Struttura del Socket TCP



NETSTAT: ANALIZZARE LO STATO DI UN SOCKET

Uno 'snapshot' dei socket a cui sono state associate connessioni attive può essere ottenuto mediante l'utility netstat (network statistics), disponibile sui principali sistemi operativi

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:36045           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:53363           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 128.133.190.219:34077   4.71.104.187:80         TIME_WAIT
tcp        0      0 128.133.190.219:43346   79.62.132.8:22          ESTABLISHED
tcp        0      0 128.133.190.219:875     128.133.190.43:2049     ESTABLISHED
tcp6       0      0 :::22                   :::*                    LISTEN
```

NETSTAT: ANALIZZARE LO STATO DI UN SOCKET

- Proto: protocollo associato al socket (TCP, UDP,...)
- RecV-Q, Send-Q: numero di bytes presenti nel receive buffer e nel send buffer
- Local Address: indirizzo IP + porta locale a cui è associato il socket
- Foreign Address: indirizzo IP + porta a cui è associato il socket
- State: stato della connessione
  - LISTEN : il server sta attendendo richieste di connessione
  - TIMEWAIT: il client ha iniziato la procedura di chiusura della connessione, che non è ancora stata completata
  - ESTABLISHED: Il client ha ricevuto il SYN dal server (3-way handshake completato) e la connessione è stata stabilita
  - Altri stati corrispondono ai diversi stati del 3-way handshake o del protocollo definito da TCP per la chiusura del socket

STREAM BASED COMMUNICATION

Per associare uno stream di input/output ad un socket esistono i metodi

public InputStream getInputStream( ) throws IOException

public OutputStream getOutputStream( ) throws IOException

che applicati ad un oggetto di tipo Socket

- restituiscono uno stream associato al socket
- ogni valore scritto su uno stream di output associato al socket viene copiato nel Send Buffer
- ogni valore letto dallo stream viene prelevato dal Receive Buffer

Il client può leggere dallo stream

- un byte/ una sequenza di bytes
- dati di tipo qualsiasi (anche oggetti) mediante l'uso di filtri (DataInputStream, ObjectInputStream,…)

ECHO CLIENT TCP

```
import java.net.*; import java.io.*; import java.util.*;
public class EchoClient {
public static void main (String args[]) throws Exception
   {   Scanner console = new Scanner( System.in);
       InetAddress ia=InetAddress.getByName("localhost");
      int port=8;    Socket echosocket=null;
      try{   echosocket = new Socket (ia, port);}
      catch (Exception e){System.out.println(e);return;}
      InputStream is = echosocket.getInputStream( );
      DataInputStream NetworkIn = new DataInputStream(is);
      OutputStream os=echosocket.getOutputStream();
      DataOutputStream NetworkOut = new DataOutputStream(os);
```

ECHO CLIENT TCP

```
boolean done=false;
while (! done)
  {String linea = console.nextLine( );
   System.out.println (linea);
   if  (linea.equals("exit")) {NetworkOut.writeUTF(linea);
                               NetworkOut.flush( ); done = true;
                               echosocket.close ( ); }
       else     {NetworkOut.writeUTF (linea);
                 NetworkOut.flush( );
                 String echo=NetworkIn.readUTF( );
                 System.out.println (echo);
                }}}}
```

STRUTTURA DI UN 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 Active Socket su cui avviene la comunicazione con C
- associa all' Active Socket 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 listening 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.

public ServerSocket(int port, int length, Inetaddress bindAddress).....

- permette di collegare il socket ad uno specifico indirizzo IP locale.

- utile per macchine dotate di più schede di rete, ad esempio un host con due indirizzi IP, uno visibile da Internet, l'altro visibile solo a livello di rete locale. Se voglio servire solo le richieste in arrivo 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

ECHO SERVER TCP

Echo Server

- si mette in attesa di richieste di connessione
- dopo aver accettato una connessione, si mette in attesa di una stringa dal client e gliela rispedisce
- quando riceve la stringa 'exit' chiude la connessione con quel client e torna ad accettare nuove connessioni

ECHO SERVER TCP

```
import java.net.*; import java.io.*;
public class EchoServer {
    public static void main (String args[ ]) throws Exception{
      int port=.....;   ServerSocket ss= new ServerSocket(port);
  while (true)
    {Socket sdati = ss.accept( );
     InputStream is = sdati.getInputStream( );
     DataInputStream networkIn = new DataInputStream(is);
     OutputStream out=sdati.getOutputStream( );
     DataOutputStream networkOut = new DataOutputStream(out);
      boolean done=false;
      while (!done){
        String echo= networkIn.readUTF( );
        if (echo.equals("exit"))   {System.out.println("finito");  done=true;}
        else {networkOut.writeUTF(echo);} } }  } }
```

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)

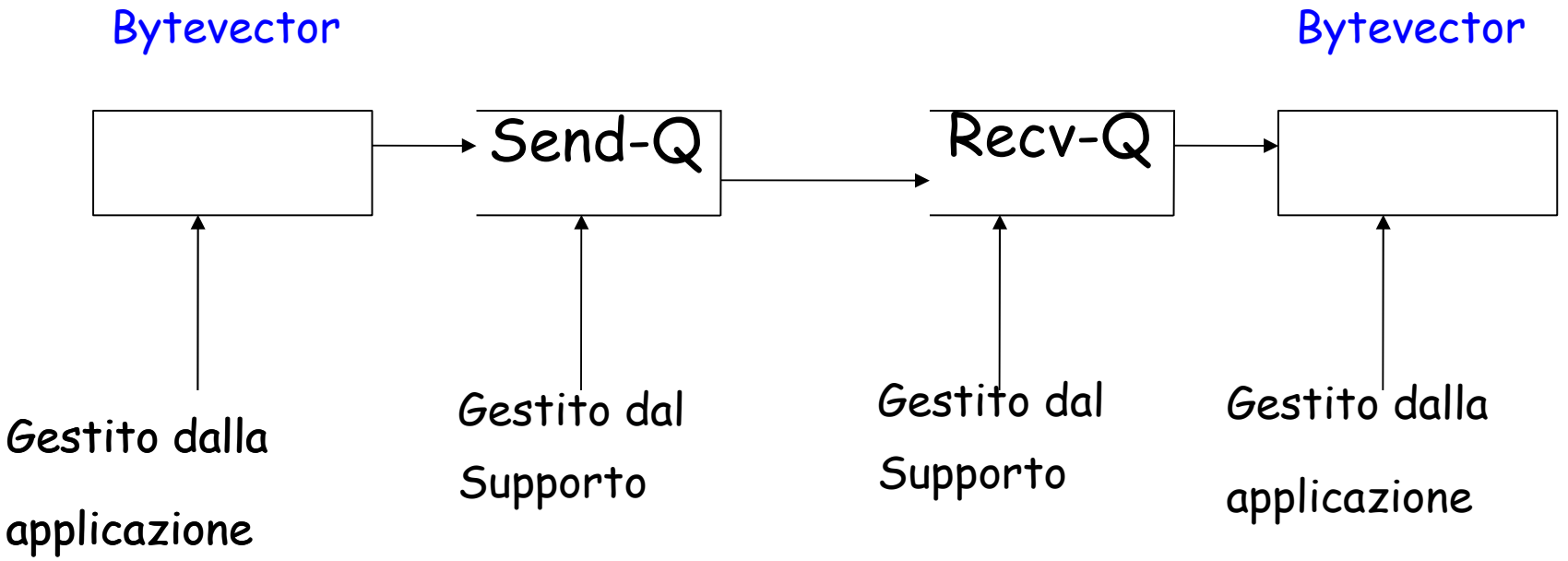
```
if (st.nextToken(" ").equals("GET") && st.nextToken(" ").equals("message"))
     { String message = HtmlWelcomeMessage();
    out.writeUTF("HTTP/1.0 200 OK"+"Content-Length: "
      +message.length()+ "Content-Type: text/html"+""+message);
     } else { out.writeUTF("400 Bad Request");}
     out.flush();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

TCP BUFFERING

Per analizzare il comportamento dei buffers associati ad un socket TCP, consideriamo prima il caso in cui l'applicazione legga/scriva diretamente sequenza di bytes (contenute in vettori di bytes gestiti dalla applicazione) sugli/dagli stream.



TCP BUFFERING

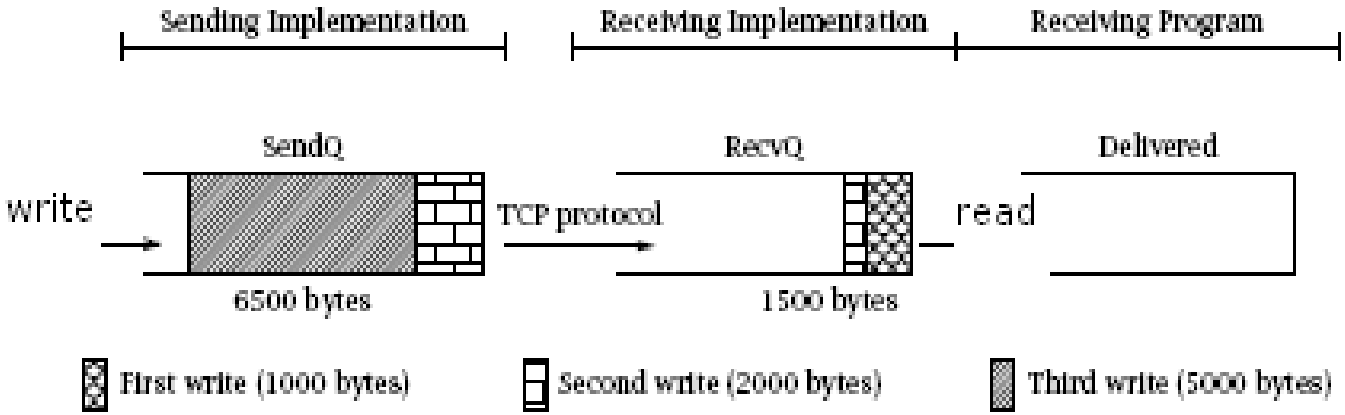
- Ipotesi: si utilizzano write/read, per scrivere/leggere vettori di bytes sugli/dagli streams.
  - La write( ) trasferisce i bytes nel Send-Q buffer, se esiste abbastanza spazio nel buffer. Se non riesce a scrivere tutti i bytes nel send buffer, si blocca.
  - La read( ) legge i dati disponibili al momento della invocazione sull'InputStream. Se Recv-Q buffer non contiene dati si blocca.
- Non esiste, in generale, alcuna corrispondenza tra
  - le scritture effettuate sull'OutputStream ad un capo della comunicazione
  - le letture dall'InputStream effettuate all'altro capo
- I dati scritti sull'OutputStream mediante una singola scrittura possono, in generale, essere letti mediante un insieme di operazioni di lettura

TCP BUFFERING: UN ESEMPIO

```
byte [ ] buffer0 = new byte[1000];
byte [ ] buffer1 = new byte[2000];
byte [ ] buffer2 = new byte[5000];
      ....
Socket s = new Socket(destAddr, destPort);
OutputStream out= s.getOutputStream();
            .....
out.write(buffer0);   ....
out.write(buffer1);   ....
out.write(buffer2);   ....
s.close();
```

TCP BUFFERING: STATO DEI BUFFERS

Stato dei buffer dopo l'esecuzione l'esecuzione di tutte le write( ), ma prima di una qualsiasi operazione di read, uno scenario possibile



Questo scenario può essere analizzato mediante l'esecuzione di netstat

Mittente

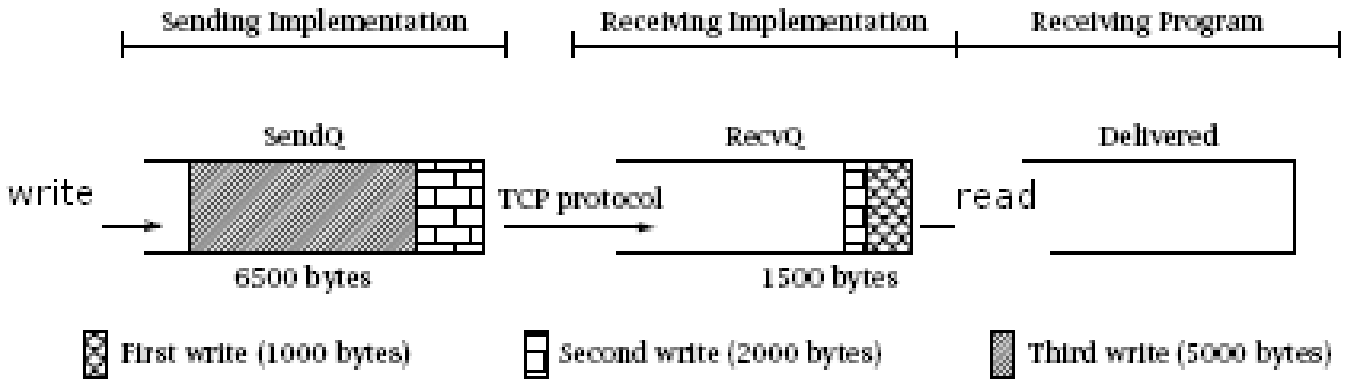
```
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address      State
tcp        0   6500 10.21.44.33:43346 192.0.2.8:22         ESTABLISHED
```

Destinatario

```
Active Internet connections
Proto Recv-Q Send-Q Local Address      Foreign Address      State
tcp     1500      0 192.0.2.8:22       10.21.44.33:43346    ESTABLISHED
```

TCP BUFFERING: STATO DEI BUFFERS

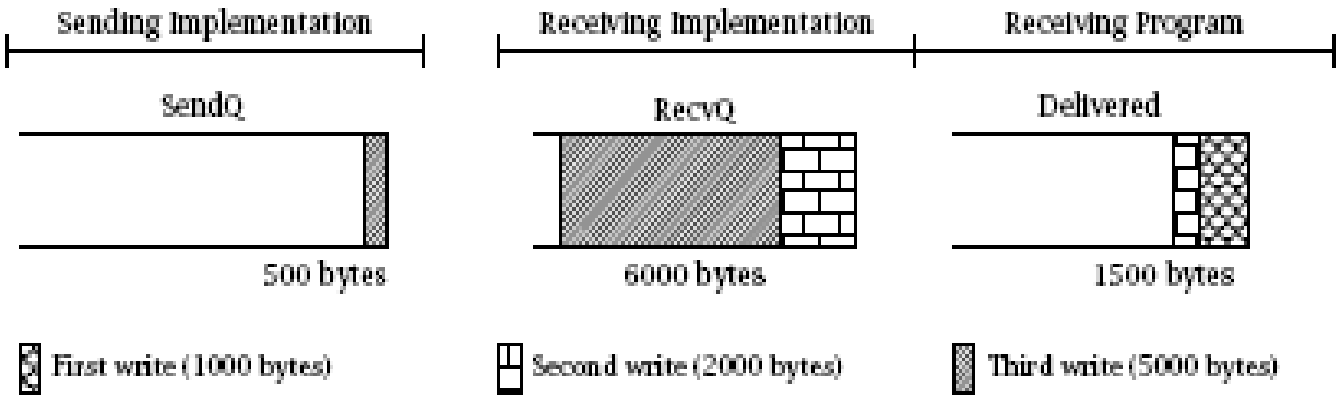
- Se il ricevente esegue una read con un byte array di dimensione 2000, nella situazione mostrata dalla figura mostrata, la read
  - riempe parzialmente completamente il byte array
  - L'applicazione riceve 1000 byte prodotti dalla prima write( ) e 500 dalla seconda
- Se necessario, l'applicazione deve utilizzare opportuni meccanismi per delimitare i dati prodotti da write( ) diverse



TCP BUFFERING: STATO DEI BUFFERS

Se il ricevente esegue una read con un byte array di dimensione 4000, nella situazione mostrata in figura, la read

- riempe completamente il byte array
- restituisce 1500 caratteri prodotti dalla seconda write( ) e 2500 dalla terza
- alcuni bytes rimangono nel receive buffer e verranno recuperati con una successiva read( )



TCP BUFFERING: DEADLOCK

- Meccanismo di Controllo del Flusso: quando il RecvQ è pieno, TCP impedisce il trasferimento di ulteriori bytes dal corrispondente SendQ
- Questo meccanismo, unito al fatto che i buffer sono di dimensione finita, può provocare situazioni di deadlock
- La situazione di deadlock può essere generata nel caso di due programmi che si inviano simultaneamente grosse quantità di dati
- Esempio: client e server si scambiano files di grosse dimensioni
  - il receive buffer del server viene riempito così come il send buffer del client
  - l'esecuzione del client viene bloccata a causa di un'ulteriore write( ).
  - il server non svuota il proprio receive buffer perchè bloccato, a sua volta, nell'invio di una grossa quantità di dati alclient

SOCKETS: CHIUSURA

- Un socket (oggetto della classe Socket) viene chiuso automaticamente a causa di:
  - garbage collection
  - terminazione del programma
- In certi casi (esempio un web browser)
  - il numero di sockets aperti può essere molto alto
  - il numero massimo di sockets supportati può essere raggiunto prima della garbage collection
  - può essere necessario chiudere esplicitamente alcuni sockets che non vengono più utilizzati
  - chiusura esplicita di un socket s : s.close( )

SOCKETS: CHIUSURA

```
for (int i = 1; i< 1024; i++)
   {try
        { s = new Socket(host, i);
             System.out.println("Esiste un servizio sulla porta"+i); }
   catch (UnknownHostException ex)
         {System.out.println("Host  Sconosciuto"); }
   catch (IOException ex) {System.out.println("Non esiste un servizio
                                              sulla porta"+i);}
   finally{
     try{
         if (s!=null)     {s.close( ); s=null;  System.out.println("chiuso");}
        } catch(IOException ex){   };      }    }
```

SOCKETS: CHIUSURA

- ShutdownInput, ShutdownOutput
  - consentono di chiudere indipendentemente gli stream di ingresso/ uscita associati al socket

- Esempio:
  - un client non deve inviare ulteriori dati al socket, ma deve attendere una risposta dal socket stesso
  - Il client può chiudere lo stream di output associato al socket e mantenere aperto lo stream di input per ricevere la risposta

- La lettura di un dato da un socket il cui corrispondente OutputStream è stato chiuso, restituisce il valore -1, che può essere quindi utilizzato come simbolo di fine sequenza

ESERCIZIO; COMPRESSIONE DI FILE

Progettare un'applicazione client/server in cui il server fornisca un servizio di compressione di dati.

Il client legge chunks di bytes da un file e li spedisce al server che provvede alla loro compressione. Il server restituisce i bytes in formato compresso al client che provvede a creare un file con lo stesso nome del file originario e con estensione gz, che contiene i dati ricevuti dal server.

La comunicazione tra client e server utilizza il protocollo TCP.

Per la compressione si può utilizzare la classe JAVA GZIPOutputStream.

Individuare le condizioni necessarie affinchè il programma scritto generi una situazione di deadlock e verificare che tale situazione si verifica realmente quando tali condizioni sono verificate.

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

CLASSE SOCKET: OPZIONI

- la classe socket offre la possibilità di impostare diverse proprietà del socket
- Opzioni
  - SO_TIMEOUT
  - SO_RCVBUF
  - SO_SNDBUF
  - SO_KEEPALIVE
  - TCP_NODELAY
  - SO_LINGER

  .....

CLASSE SOCKET: SO_TIMEOUT

SO_TIMEOUT – consente di associare un time out al socket

```
if (s.getSoTimeout( )== 0) s.setSoTimeout(1800000);
```

- Il timeout viene specificato in millisecondi
- Quando eseguo una lettura bloccante dal socket, l'operazione si può bloccare in modo indefinito
- SO_TIMEOUT: definisce un intervallo di tempo massimo per l'attesa dei dati
- Nel caso in cui il time out scada prima della ricezione dei dati, viene sollevata una eccezione

CLASSE SOCKET: SO_RCVBUF, SO_SNDBUF

- SO_RCVBUF Controlla la dimensione del buffer utilizzato per ricevere i dati.
  - E' possibile impostare la dimensione del buffer di ricezione

    Socket.setReceiveBufferSize(4096)
  - La modifica non viene garantita su tutti i sistemi operativi
  - Per reperire la dimensione del buffer associato

    int size = sock. getReceiveBufferSize( )
  - Alternativa: utilizzare i BufferedInputStream/BufferedReader.

- SO_SNDBUF : analogo per il buffer associato alla spedizione

  int size = sock.getSendBufferSize( );

- Attenzione: questi comandi non sono implementati correttamente su alcuni sistemi operativi

CLASSE SOCKET: SO_KEEPALIVE

- So_keepalive: tecnica utilizzata per monitorare le connessioni aperte e controllare se il partner risulta ancora attivo
- introdotto per individuare i sockets "idle" su cui non sono stati inviati dati per un lungo intervallo di tempo
- Per default, su ogni socket vengono spediti solo dati inviati dalla applicazione
- Un socket può rimanere inattivo per ore, o anche per giorni
  - Esempio: crash di un client prima dell'invio di un segnale di fine sequenza. In questo caso, il server può sprecare risorse (tempo di CPU, memoria,...) per un client che ha subito un crash
  - Consente un'ottimizzazione delle risorse

CLASSE SOCKET: SO_KEEPALIVE

Socket.setSoKeepAlive(true) abilita il keep alive.

- il supporto invia periodicamente dei messaggi di keep alive sul socket per testare lo stato del partner.
- se il partner è ancora attivo, risponde mediante un messaggio di ack
- nel caso di mancata risposta viene reiterato l'invio del keep alive per un certo numero di volte
- se non si riceve alcun acknowledgment, il socket viene portato in uno stato di 'reset'
- ogni lettura, scrittura o operazione di chiusura su un socket posto in stato di reset(), solleva un'eccezione
- questa funzionalità può non essere implementata su alcune piattaforme,nel qual caso il metodo solleva un'eccezione

CLASSE SOCKET: TCP_NODELAY

Algoritmo di Nagle:

- Introdotto per evitare che il TCP spedisca una sequenza di piccoli segmenti, quando la frequenza di invio dei dati da parte della applicazione è molto bassa
- riduce il numero di segmenti spediti sulla rete fondendo in un unico segmento più dati
- Applicazione originaria dell'algoritmo
  - Sessioni Telnet, in cui è richiesto di inviare I singoli caratteri introdotti, mediante keyboard, dall'utente
  - Se l'algoritmo di Nagle non viene applicato, ogni carattere viene spedito in un singolo segmento,(1 byte di data e decine di byte di header del messaggio
- Motivazioni per disabilitar l'algoritmo di Nagle: trasmissioni di dati in 'tempo reale', ad esempio movimenti del mouse per un'applicazione interattiva come un gioco multiplayer

CLASSE SOCKET: TCP_NODELAY

Algoritmo di Nagle:

- In generale, per default, l'algoritmo di Nagle risulta abilitato
- tuttavia alcuni sistemi operativi disabilitano l'algoritmo di default
- per disabilitare l'algoritmo di Nagle

  sock.setTcpNoDelay(true)

  disabilita la bufferizzazione (no delat= non attendere, inviare subito un segmento, non appena l'informazione è disponibile)
- JAVA RMI disabilita l'algoritmo di Nagle:lo scopo è quello di inviare prontamente il segmento contenente i parametri di una call remota oppure il valore restituito dall'invocazione di un metodo remoto

CLASSE SOCKET: SO_LINGER

La proprietà SO_LINGER (to linger= indugiare) viene utilizzata per specificare cosa accade quando viene invocato il metodo close( ) su un socket TCP.

A seconda del valore di SO_LINGER può accadere che

- Linger= false (default): il contenuto del buffer di invio associato al socket viene inviato al destinatario, mentre i dati nel buffer di ricezione vengono scartati. Il thread che esegue il metodo close( ) non attende la terminazione di queste attività che avvengono quindi in background..

  Questo è lo scenario di default, che però non garantisce che i dati vengano consegnati correttamente. In caso di crash del destinatario, ad esempio, i dati nel buffer di spedizione non vengono consegnati

CLASSE SOCKET: SO_LINGER

- Linger=true, Linger time=0 Vengono scartati sia gli eventuali dati nel buffer di ricezione che quelli da inviare. Come prima, lo scarto avviene in background
  - Utilizzato quando si vuole terminare la connessione immediatamente, senza spedire i dati
- Linger=true e linger time!=0 Vengono inviati eventuali dati presenti nel buffer al destinatario e si scartano gli eventuali dati nel buffer di ricezione. Il thread che esegue il metodo close( ) si blocca per il linger time oppure fino a che tutti i dati spediti sono stati confermati a livello TCP. Dopo linger time viene sollevata un'eccezione
  - Quando si vuole garantire che il metodo close( ) ritorni solo quando i dati sono stati consegnati, oppure che sollevi un'eccezione nel caso in cui scatti il time-out definito da linger-time

CLASSE SOCKET:SO_LINGER

```
public void setSoLinger (boolean no, int seconds)
                                                throws SocketException
public int getSoLinger ( ) throws SocketException
```

- per default, SO_LINGER=false: il supporto tenta di inviare i datagrams rimanenti, anche dopo che il socket è stato chiuso
- per controllare la gestione dei dati presenti al momento della chiusura

```
if (s.getSoLinger( )== -1) s.setSoLinger(true,240);
```

il metodo close( ) si blocca ed attende 240 secondi (4 minuti) prima di eliminare i datagrams rimanenti. Se il tempo di attesa viene impostato a 0, i datagram vengono eliminati immediatamente.