LPR A 2005/2006
Lezione 5

- Riepilogo Caratteristiche Datagram Socket
- *Stream Mode* Socket API
- Sockets Lato *Client*
- Sockets Lato *Server*
- Esempi

DATAGRAM SOCKET API: RIASSUNTO DELLE PUNTATE PRECEDENTI

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

  se il destinatario non è in esecuzione quando il mittente esegue la send, il messaggio può venir scartato dal mittente
- 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 siano sovrapposti a messaggi inviati in precedenza

- Esempio: *CountDown Server* (vedi esercizio lezione precedente).

  Il client invia al server un valore di n molto grande

  (provare valori>400). Allora:

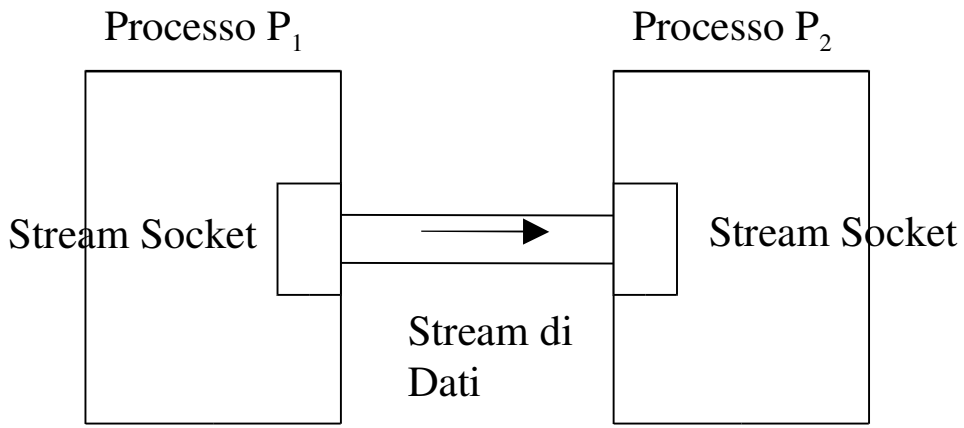
  - il server deve inviare al client un numero molto alto di pacchetti
  - il tempo che intercorre tra l'invio di un pacchetto e quello del pacchetto successivo è basso
  - dopo un certo numero di invii il buffer del client si riempie⇒ perdita di pacchetti

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 NETWORKPROGRAMMING !!

STREAM MODE SOCKET API: LATO SERVER

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:

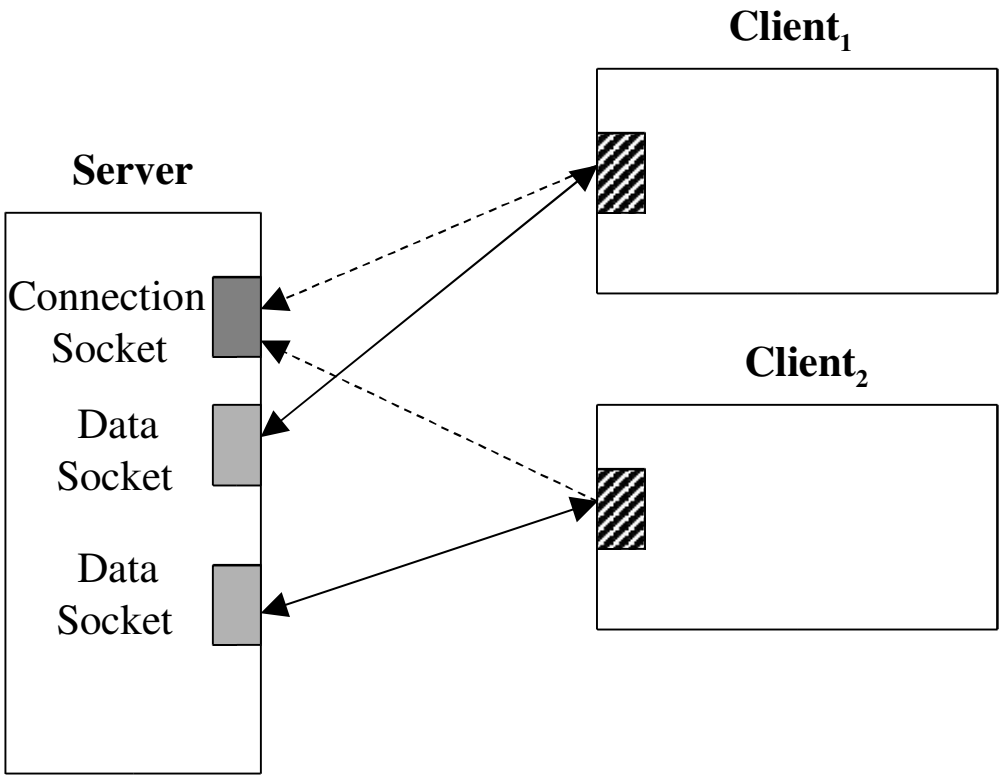
connection socket

utilizzato per accettare richieste di connessione

data socket

utilizzato per scambiare i dati con il client

STREAM MODE SOCKET API



STREAM MODE SOCKET API
LATO CLIENT

- al momento della creazione del socket S, il client deve specificare l' *indirizzo del server* e porta su cui *è attivo il servizio* che intende richiedere tramite quel socket.

- la porta a cui è collegato S può essere *anonima*

- *la creazione del socket produce in modo atomico la richiesta di connessione al server o lancia una eccezione se la connessione non viene accettata* (protocollo richiesta di connessione gestito dal supporto)

- quando la richiesta di connessione viene accettata dal server, il supporto crea in modo automatico un nuovo socket sul server che 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: STRUTTURA DEL SERVER

Comunicazione client/server mediante *stream mode sockets*, operazioni principali effettuati nel *Server (supponiamo un comportamento sequenziale)*

- crea un *connection socket CS* su una porta specifica. Il numero della porta a cui è associato il socket deve essere pubblicato (per l'individuazione del servizio)
- 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 il client
- associa al *DataSocket* uno o più stream (di input e/o di output) su cui avverrà la comunicazione con il client
- quando la interazione con il client è terminata, chiude il data socket e torna ad ascoltare sul Connection Socket ulteriori richieste di connessione

STREAM MODE SOCKET API

Comunicazione Client/Server mediante *stream mode sockets*, operazioni principali effettuate dal *client*:

- crea un socket S su cui inviare la richiesta di connessione al server.
- quando la richiesta viene accettata, associa un data stream di input e/o di output per la comunicazione sul data socket
- invia/riceve dati dallo stream
- alla fine dell'interazione, 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 connettersi mediante il socket creato all'host individuato da InetAddress e 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)

STREAM MODE SOCKET API:
LATO CLIENT

Esempio: ricerca dei servizi TCP attivi sulle prime 1024 porte di un host specificato.
Soluzione: PortScanner un programma che cerca di connettersi ad ognuna delle 1024 porte tentando di collegare un socket ad ognuna di quelle porte.

```
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 serviziosulla porta"+i);
                      }
                      catch (UnknownHostException ex)
                                  {System.out.println("Host Sconosciuto");
                                    break
                                   };
                      catch (IOException ex) {       };
                      }
        }
}
```

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

  - interrogare il DNS una sola volta all'inizio prima di entrare nel ciclo di scanning (InetAddress.getByName)

  - utilizzare l'InetAddress invece del nome dell'host per costruire i sockets

STREAM MODE SOCKET API:
LATO CLIENT

Classe java.net.socket
Altri costruttori

public Socket (String h, *int* p, *InetAddress* i, *int* lp)

tenta di creare una connessione *verso* l'host h, sulla porta p. Il socket eventualmente creato per la connessione viene legato alla interfaccia locale i, ed alla porta locale lp

STREAM MODE SOCKET API
LATO CLIENT

Metodi per l'associazione di streams di input/output ai sockets

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 SERV ERS PREDEFINITI

Esercizi: considerare un servizio attivo su una porta pubblicata da un host (esempio:*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 queery
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

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);
            String hostname=……; int port=7;
                        Socket echosocket = new Socket (hostname, port);
            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);
                                      }
…… }
```

STREAM MODE SOCKET API
LATO SERVER

Gestione sockets lato server:

Classe *java.net.ServerSocket*: costruttori
public ServerSocket(*int* port) *throws BindException, IOException*
costruisce un connection socket sulla porta specificata

Esempio: ricerca dei servers attivi sull'host locale

```
public class LocalPortScanner {
        public static void main(String args[])
                {for (int port= 1; port<= 65535; port ++)
                        try {ServerSocket server = new ServerSocket(port)}
                        catch (BindException ex) {port + "occupata"} } } }
```

STREAM MODE SOCKET API
LATO SERVER

Gestione sockets lato server:

Classe *java.net.ServerSocket*: altri costruttori

public ServerSocket(*int* p, *int* queueLength) *throws IOException*,*BindException*

- Costruisce un connection socket, associandolo alla porta p. queueLength rappresenta 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* p, *int* queueLength, *InetAddress* bindAddress)
throws IOException,*BindException*

- 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 $\Rightarrow$ associo il connection socket all'indirizzo IP locale

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'e' 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.*;
import java.util.*;

    public class EchoServer {

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

            Scanner console = new Scanner(System.in);
            String hostname=……;
            int port=…..;
            ServerSocket ss= new ServerSocket(port);
```

ECHO SERVER TCP

```
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"))
            sdati.close( ); done = true;}
            else {networkOut.writeUTF(echo);}
}}}}
```

OPSIONI DI SOCKETS

TCP_NODELAY – i messaggi vengono inviati non appena disponibili

SO_LINGER – specifica come trattare i pacchetti non ancora spediti al momento della chiusura di un socket (opzioni possibili: scarto immediato dei messaggi, attesa per un intervallo di tempo dal momento della chiusura, poi scarto dei messaggi).

SO_TIMEOUT – associazione di un time-out al socket

SO_RCVBUF /SO_SNDBUF – Per impostare le dimensioni del buffer di ricezione/spedizione

ESERCIZIO

Sviluppare una applicazione che offra il servizio di trasferimento di files da un client *RemoteCopyClient* ad un server *RemoteCopyServerServer.* Il client richiede, in modo interattivo, il nome del file da trasferire all'utente, e, se il file esiste, richiede una connessione al Server. Quando la connessione viene accettata, invia al server il nome del file seguito dal suo contenuto. Infine il client attende l'esito della operazione dal server , quindi torna a proporre una nuova richiesta di trasferimento all'utente.
RemoteCopyServer riceve una richiesta di connessione e salva il file richiesto in Una directory locale. Alla fine del download del file, *RemoteCopyServer* invia al client l'esito della operazione. L'esito può essere di due tipi:

- *update:* il file esisteva già ed è stato sovrascritto

- *new:* è stato creato un nuovo file