

Reti e Laboratorio III

Modulo Laboratorio III

AA. 2023-2024

docente: Laura Ricci

laura.ricci@unipi.it

Lezione 7

Stream Sockets for servers

02/11/2023

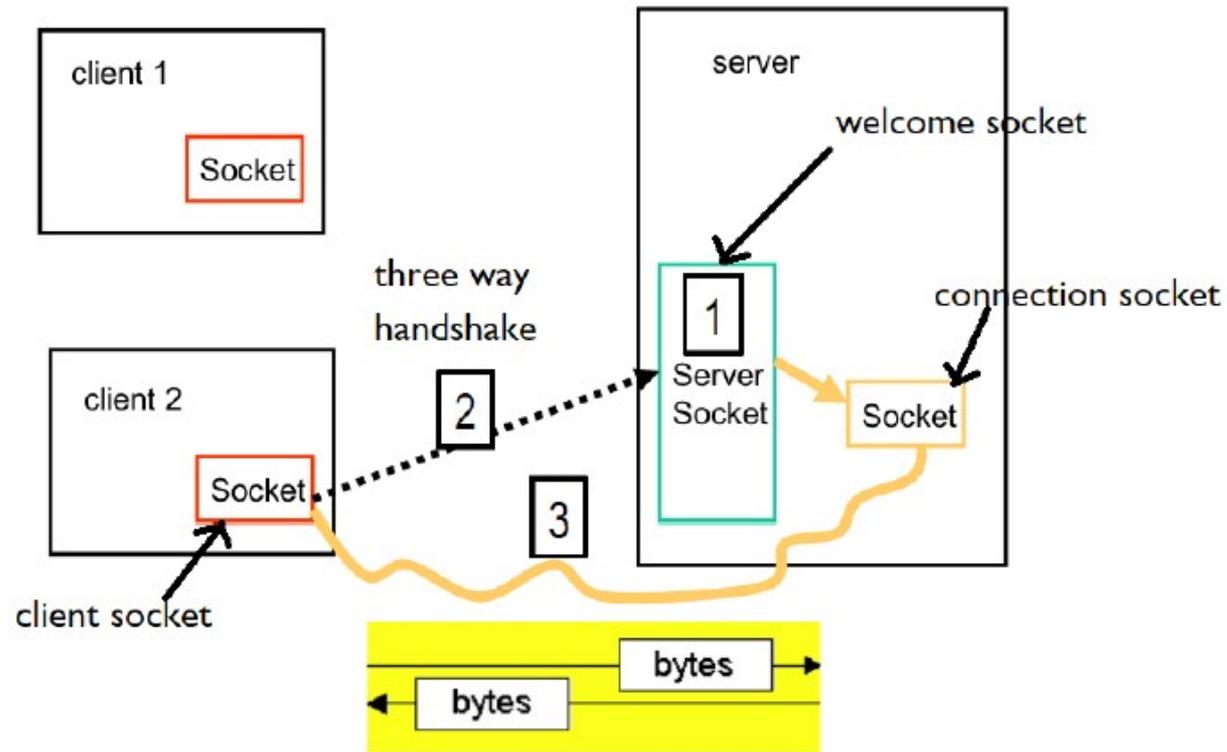
SOCKET LATO SERVER

- esistono due tipi di socket TCP, lato server:
 - **welcome (passive, listening) sockets**: utilizzati dal server per accettare le richieste di connessione
 - **connection (active) sockets**: connettono il server ad un particolare client e supportano lo streaming di byte con essi
- il client crea un active socket per richiedere la connessione
- il server accetta una richiesta di connessione sul welcome socket
 - crea **un proprio connection 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

SOCKET LATO SERVER

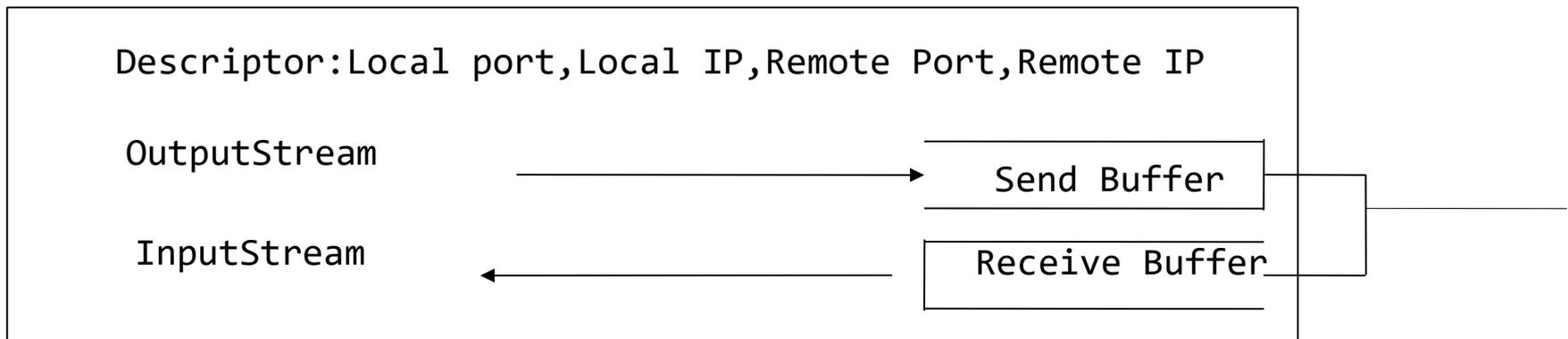
- il server pubblica un proprio servizio
 - gli associa un welcome socket, sulla porta remota PS, all'indirizzo IPS
 - usa un oggetto di tipo ServerSocket
- il client crea un Socket e lo connette all'endpoint IPS + PS
- la creazione del socket effettuata dal client produce in modo atomico la richiesta di connessione al server
 - three way handshake completamente gestito dal supporto
 - se la richiesta viene accettata
 - il server crea un **socket dedicato** per l'interazione con quel client
 - tutti i messaggi spediti dal client vengono diretti **automaticamente** sul nuovo socket creato.

SOCKET LATO SERVER



STREAM BASED COMMUNICATION

- dopo che la richiesta di connessione viene accettata, client e server associano streams di bytes di input/output ai socket dedicati a quella connessione, poichè gli stream sono **unidirezionali**
 - a seconda del servizio può essere necessario un solo stream di output dal server verso il client, oppure una coppia di stream da/verso il client
- la comunicazione avviene mediante **lettura/scrittura di dati sullo stream**
- eventuale utilizzo di filtri associati agli stream



Struttura del Socket TCP

JAVA STREAM SOCKET API: LATO SERVER

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 `port`.
- `length`: lunghezza della coda in cui vengono memorizzate le richieste di connessione.

se la coda è piena, ulteriori richieste di connessione sono 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

JAVA STREAM SOCKET API: LATO SERVER

- accettare una nuova connessione dal `connection socket`

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

metodo della classe `ServerSocket`.

- quando il processo server invoca il metodo `accept()`, pone il server in attesa di nuove connessioni.
- bloccante: se non ci sono richieste, il server si blocca (possibile utilizzo di time-outs)
- quando c'è almeno una richiesta, il processo si sblocca e costruisce un nuovo socket tramite cui avviene la comunicazione effettiva tra cliente server

PORT SCANNER LATO SERVER

- ricerca 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);}
    } }
```

CICLO DI VITA TIPICO DI UN SERVER

```
// 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();
```

DAYTIME SERVER

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
public final static int PORT = 1313;
    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(PORT)) {
            while (true) {
                try (Socket connection = server.accept()) {
                    Writer out = new OutputStreamWriter(connection.getOutputStream());
                    Date now = new Date();
                    out.write(now.toString() + "\r\n");
                    out.flush();
                    // connection.close();
                } catch (IOException ex) {}
            }
        } catch (IOException ex) {
            System.err.println(ex); } } }
```

porte 0-1023 privilegiate

si ferma qui ed aspetta, quando un client
si connette restituisce un nuovo Socket

servizio della
richiesta

inutile perchè si è usato il try with resources

try-with-resource: autoclose

DAYTIME SERVER: CONNETTERSI CON TELNET

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DayTimeServer {
    public final static int PORT = 13;
    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(PORT)) {
            while (true) {
                try (Socket connection = server.accept()) {
                    Writer out = new OutputStreamWriter(connection.getOutputStream());
                    Date now = new Date();
                    out.write(now.toString() + "\r\n");
                    out.flush();
                    connection.close();
                } catch (IOException ex) {}
            } } catch (IOException ex) {System.err.println(ex);}}
```

```
$ telnet localhost 1313
trying 127.0.0.1...
connected to localhost
San Oct 17 23:16:12 CEST 2021
```



TRY WITH RESOURCES

- introdotto in JAVA 7, aggiornato in JAVA 9
- chiusura sistematica ed automatica delle risorse usate da un programma
- un blocco try con uno o più argomenti tra parentesi.
 - argomenti = risorse che devono essere chiuse quando il try block termina
 - le variabili che rappresentano le risorse non devono essere riutilizzate
- suppressed exceptions:
 - quando si verificano delle eccezioni sia nel blocco try-with-resources sia durante la chiusura, la JVM sopprime l'eccezione generata nella chiusura automatica.
- generalizzazione: implementazione della AutoCloseable interface

TRY WITH RESOURCES

- una certa risorsa è chiusa “automaticamente”, dopo che è stata utilizzata
 - risorsa: file, stream, reader o socket
 - tecnicamente ogni oggetto che implementi l'interfaccia AutoClosable

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
  
// w.close() is called automatically
```

- in questo esempio, `w.close()` viene chiamata indipendentemente dal fatto che la `write` sollevi o meno una eccezione
- concettualmente simile ad aggiungere `w.close()` in un blocco `finally`
- possibile usare più risorse in un blocco `try with resources`, vengono chiuse in senso inverso rispetto all'ordine con cui sono state dichiarate

TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- una eccezione può essere sollevata nei seguenti statement
 - `new FileWriter("file.txt")`
 - `w.write("Hello World")`
 - implicitamente da `w.close()`
- eccezione sollevata nel costruttore: nessun oggetto da chiudere, si propaga la eccezione senza eseguire la `write()`

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World");  
}  
// no call to w.close()
```

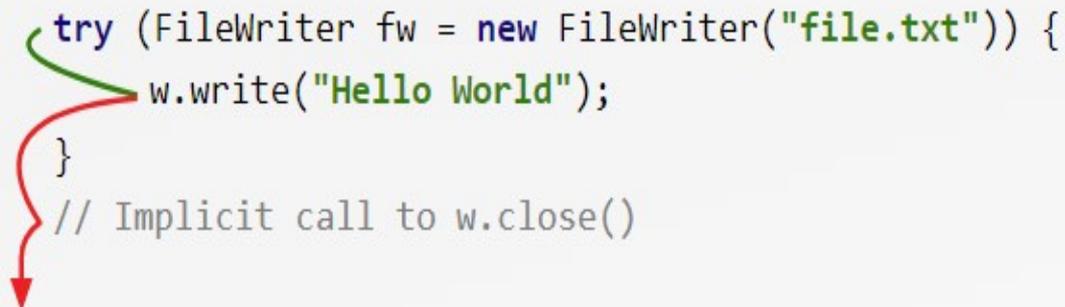


TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- eccezione sollevata nella write() : viene invocato w.close(), poi si propaga l'eccezione



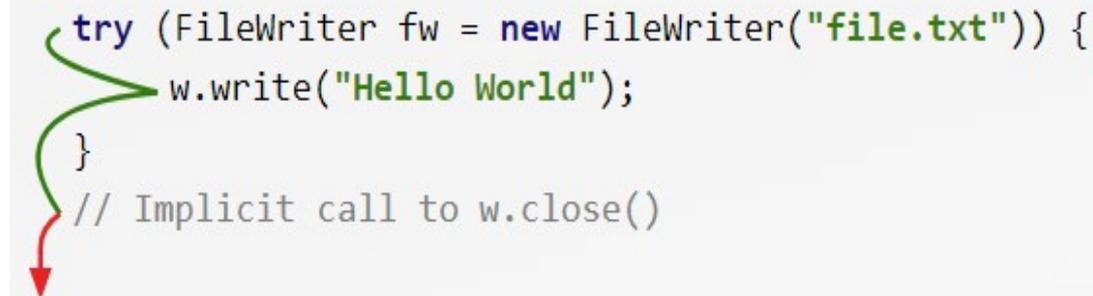
```
try (FileWriter fw = new FileWriter("file.txt")) {  
    w.write("Hello World");  
}  
    // Implicit call to w.close()
```

TRY WITH RESOURCES: ECCEZIONI

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- eccezione sollevata nella chiamata implicita alla `close()` : viene propagata la eccezione



```
try (FileWriter fw = new FileWriter("file.txt")) {  
    w.write("Hello World");  
}  
// Implicit call to w.close()
```

The diagram illustrates the flow of an exception. A green bracket on the left side of the try block indicates the scope of the resource. A red arrow points from the end of the try block (the closing curly brace) down to the comment indicating the implicit call to `w.close()`, showing that any exception thrown within the try block is propagated to the `close()` method.

TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

- nel seguente esempio

```
try (FileWriter w = new FileWriter("file.txt")) {  
    w.write("Hello World"); }  
    // w.close() is called automatically
```

- cosa accade se la `w.write()` solleva un'eccezione ed anche la chiamata implicita alla `w.close()` la solleva?
- la prima eccezione “vince” sulla seconda e la seconda viene soppressa



TRY WITH RESOURCES: SUPPRESSED EXCEPTIONS

```
import java.io.*;

public class trywithresources
{ public static void main (String args[])throws IOException {
    try(FileInputStream input = new FileInputStream(new File("immagine.jpg"));
        BufferedInputStream bufferedInput = new BufferedInputStream(input))
    {
        int data = bufferedInput.read();
        while(data != -1){
            System.out.print((char) data);
            data = bufferedInput.read();
        }
    }
}
```

- risolve il problema delle “suppressed exceptions”
 - eccezioni possono essere sollevate nel blocco try, oppure nel blocco finally,
 - un'eccezione rilevata nella finally sopprimerebbe l'eccezione rilevata nel blocco try
- con il try with resources viene propagata l'eccezione rilevata nel blocco try

MULTITHREADED SERVER

- nello schema del lucido precedente, la fase “communicate and work” può essere eseguita in modo concorrente da più threads
- un thread per ogni client, gestisce le interazioni con quel particolare client
- il server può gestire le richieste in modo più efficiente
- tuttavia....threads: anche se processi lightweighth ma tuttavia utilizzano risorse !
 - esempio: un thread che utilizza 1MB di RAM. 1000 thread simultanei possono causare problemi !
- Soluzioni alternative:
 - Thread Pooling
 - ServerSocketChannels di NIO

A CAPITALIZER SERVICE: SERVER

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

A CAPITALIZER SERVICE: SERVER

```
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); }
    }
}
```

A CAPITALIZER SERVICE: CLIENT

```
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;
```

A CAPITALIZER SERVICE: CLIENT

```
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();}
}
```

ASSIGNMENT 7: DUNGEON ADVENTURES

- sviluppare un'applicazione client server in cui il server gestisce le partite giocate in un semplice gioco, “Dungeon adventures” basato su una semplice interfaccia testuale
- ad ogni giocatore viene assegnato, ad inizio del gioco, un livello X di salute e una quantità Y di una pozione, X e Y generati casualmente
- ogni giocatore combatte con un mostro diverso. Anche al mostro assegnato a un giocatore viene associato, all'inizio del gioco un livello Z di salute generato casualmente

ASSIGNMENT 7: DUNGEON ADVENTURES

- il gioco si svolge in round, ad ogni round un giocatore può
 - *combattere con il mostro*: il combattimento si conclude decrementando il livello di salute del mostro e del giocatore. Se LG è il livello di salute attuale del giocatore e MG quello del mostro, tale livello viene decrementato di un valore casuale X , con $0 \leq X \leq LG$. Analogamente, per il mostro si genera un valore casuale K , con $0 \leq K \leq MG$.
 - *bere una parte della pozione*, la salute del giocatore viene incrementata di un valore proporzionale alla quantità di pozione bevuta, che è un valore generato casualmente
 - *uscire dal gioco*. In questo caso la partita viene considerata persa per il giocatore
- il combattimento si conclude quando il giocatore o il mostro o entrambi hanno un valore di salute pari a 0.
- se il giocatore ha vinto o pareggiato, può chiedere di giocare nuovamente, se invece ha perso deve uscire dal gioco.

ASSIGNMENT 7: DUNGEON ADVENTURES

- sviluppare una applicazione client server che implementi Dungeon adventures
 - il server riceve richieste di gioco da parte dei cliente e gestisce ogni connessione in un diverso thread
 - ogni thread riceve comandi dal client li esegue. Nel caso del comando “combattere”, simula il comportamento del mostro assegnato al client
 - dopo aver eseguito ogni comando ne comunica al client l'esito
 - comunica al client l'eventuale terminazione del del gioco, insieme con l'esito
- il client si connette con il server
 - chiede iterativamente all'utente il comando da eseguire e lo invia al server. I comandi sono i seguenti 1:combatti, 2: bevi pozione, 3: esci del gioco
 - attende un messaggio che segnala l'esito del comando
 - nel caso di gioco concluso vittoriosamente, chiede all'utente se intende continuare a giocare e lo comunica al server