Università di Verona, Dipartimento di Informatica
Programmazione e Sicurezza delle Reti
Valentina Ceoletta, Marco De Bona, Davide Quaglia

Java Socket

1 Introduzione

I Socket sono il meccanismo software che permette la gestione di un flusso di dati tra due host posizionati in qualsiasi punto di Internet (Figura 1). I Socket possono venir visti come un tubo bidirezionale alle cui estremità è associato un indirizzo IP e un numero di porta di livello trasporto. Quindi un socket è univocamente identificato dalla 5-upla:

(IP_A, IP_B, Porta_A, Porta_B, TCP/UDP)

Java mette a disposizione una serie di semplici classi che consentono la scrittura di applicazioni di rete basate sull'utilizzo dei **Socket**.

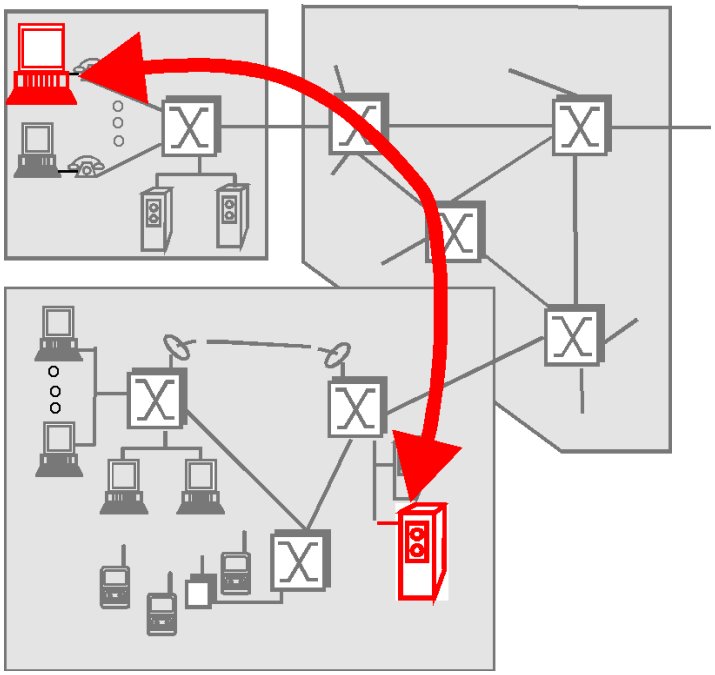
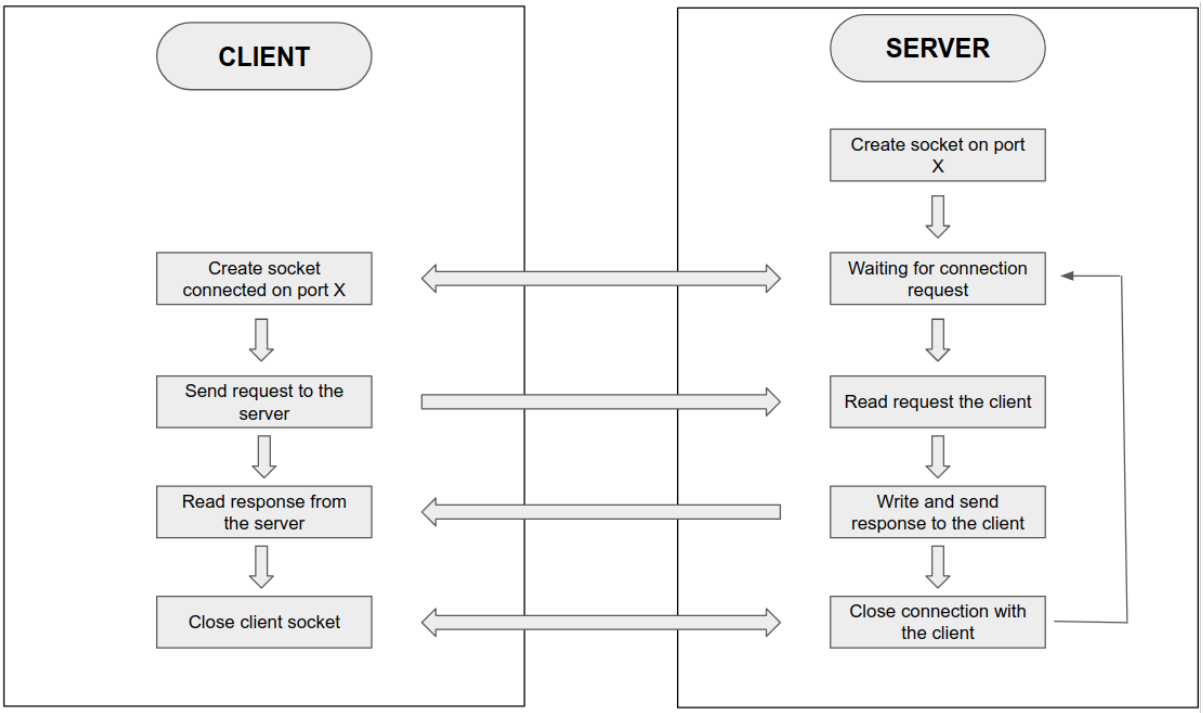


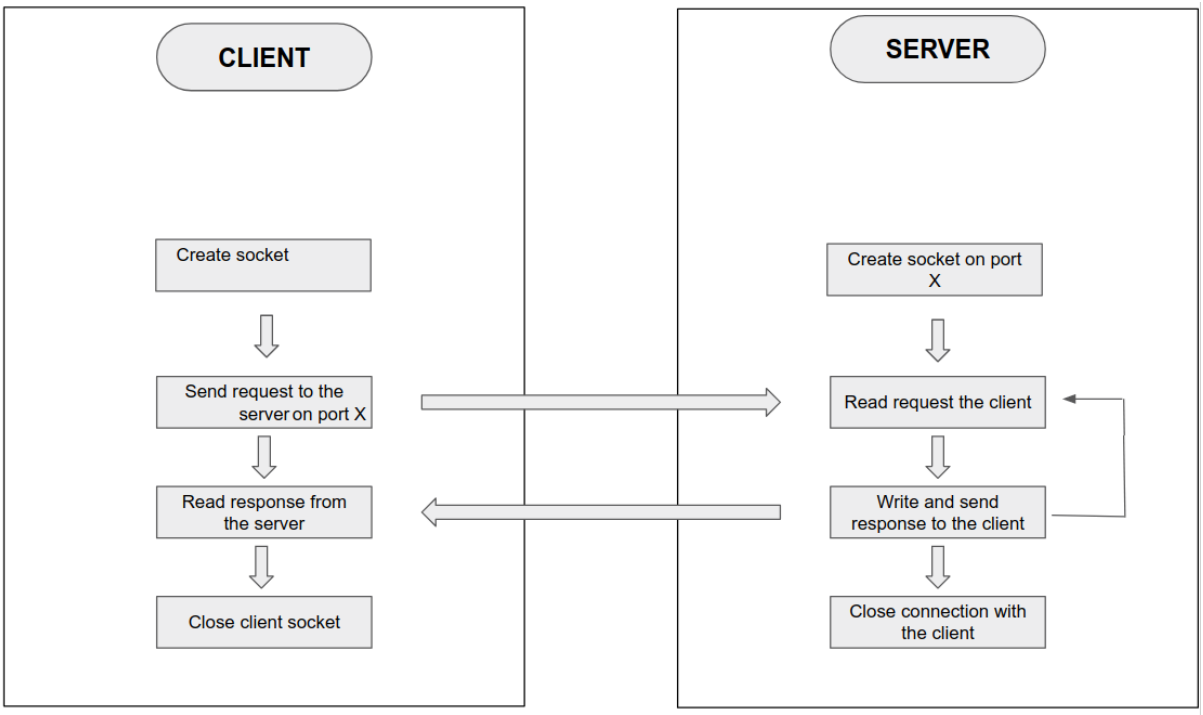
Figura 1. Connessione tramite socket.

1.1 Implementazione del paradigma Client/Server

Le applicazioni che utilizzano i socket sono quasi sempre strutturate secondo il paradigma client/server. Quando dobbiamo creare una applicazione client/server bisogna definire quali sono i compiti lato client e lato server (Figura 2).
Il server è sempre il primo a partire ed attende richieste dal client, le soddisfa e torna ad attendere. Il client per mandare richieste al server deve conoscere il nome o l'indirizzo IP dell'interfaccia di rete su cui il server è in esecuzione e il numero della porta TCP o UDP sulla quale il server è in ascolto.



(a)



(b)

Figura 2. Paradigma client/server con (a) protocollo orientato alla connessione come il TCP e (b) protocollo non orientato alla connessione come l'UDP.

1.2 Il package java.net

I protocolli coinvolti nell'implementazione dei Socket sono quelli di livello Trasporto:

- **UDP** (*User Datagram Protocol*)

- **TCP** (*Transfer Control Protocol*)

In Java le classi per usare i socket sono contenute nel package java.net

Il package java.net fornisce interfacce e classi per l'implementazione di applicazioni di rete (Figura 3).

Questo package definisce fondamentalmente:

- le classi Socket e ServerSocket per le connessioni TCP
- la classe DatagramSocket per le connessioni UDP
- la classe URL per le connessioni HTTP

più una serie di classi correlate:

- InetAddress per rappresentare gli indirizzi Internet
- URLConnection per rappresentare le connessioni a un URL

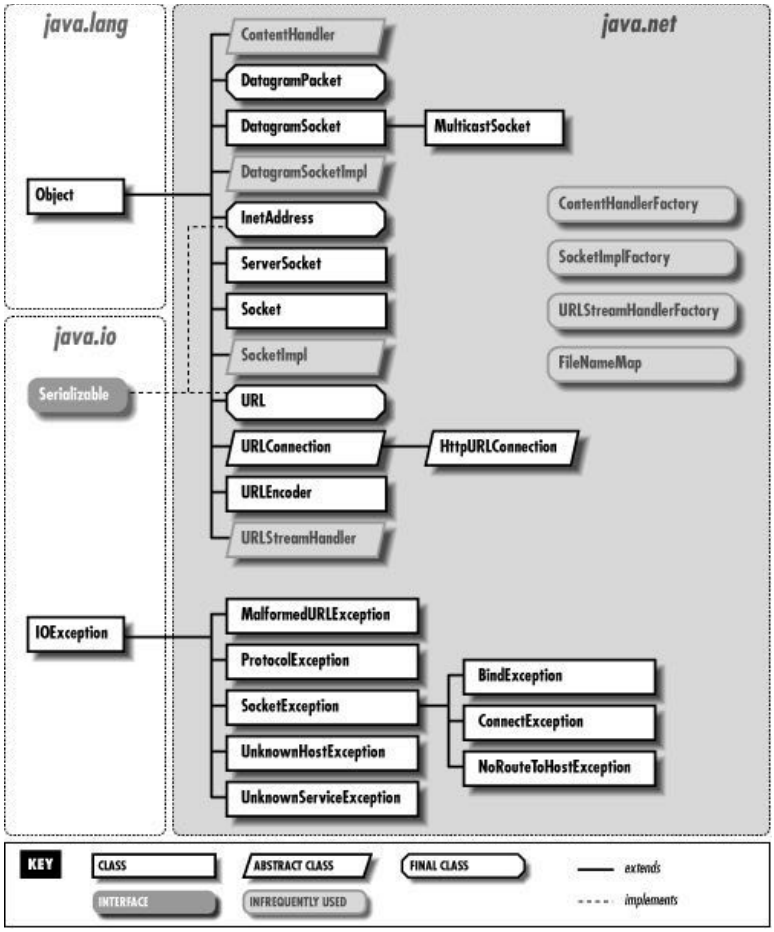


Figura 3. Gerarchia di classi java.net.

1.2.1 Classe DatagramSocket

La classe DatagramSocket implementa comunicazioni dati orientate ai pacchetti senza connessione mediante User Datagram Protocol (UDP). Ogni pacchetto gira attraverso la rete, guidato dal suo indirizzo IP di destinazione. Diversi pacchetti possono prendere percorsi differenti all'interno della rete e potrebbero arrivare in un ordine differente da quello con cui sono stati spediti. Inoltre, il raggiungimento della destinazione da parte del pacchetto non è nemmeno garantito. È compito di un'applicazione che utilizza UDP gestire questi problemi se necessario. Nonostante queste

particolarità sembrino degli svantaggi del DatagramSocket, ci sono dei numerosi vantaggi. In primis, la comunicazione usando DatagramSocket è più veloce della comunicazione a flusso di Socket data la minor quantità di pacchetti che è necessario inviare.

1.2.2 Classi Socket e ServerSocket

Socket

La classe Socket implementa comunicazioni di dati mediante il protocollo TCP che sono affidabili, orientate alle connessioni e basate sul flusso. La classe Socket racchiude la logica del client.

ServerSocket

La classe ServerSocket rappresenta un socket in ascolto per le richieste di connessioni dai client su una porta specifica. Quando una connessione viene richiesta, viene creato un oggetto di tipo Socket per gestire la comunicazione.

1.2.3 Classe InetAddress

La classe InetAddress racchiude un indirizzo IP (Internet Protocol). Gli oggetti InetAddress sono usati dalle varie classi che specificano gli indirizzi di destinazione dei pacchetti in uscita nella rete, come DatagramSocket, MulticastSocket, e Socket. InetAddress non fornisce alcun costruttore pubblico. Bisogna invece usare i metodi statici getAllByName(), getByName(), e getLocalHost() per creare oggetti di tipo InetAddress.

1.2.4 Classe URL

La classe URL rappresenta un Uniform Resource Locator (ad es. http://www.di.univr.it/). Questa classe fornisce metodi per recuperare le varie parti di un URL e inoltre per accedere alla risorsa specificata. Un URL assoluto consiste in un protocollo, un hostname, un numero di porta, un filename e una referenza opzionale, o àncora.

1.2.5 Classe URLConnection

La classe URLConnection è una classe astratta che rappresenta una connessione ad un URL. Una sottoclasse di URLConnection supporta una connessione protocollo-specifica. Una URLConnection può sia leggere che scrivere su un URL.
Un oggetto URLConnection viene creato quando il metodo openConnection() viene chiamato per un oggetto URL. A questo punto la connessione attuale non è ancora stata fatta, quindi i parametri di configurazione e le proprietà generali di richiesta possono essere modificate per una connessione specifica.

1.2.6 I/O stream tramite socket TCP

Siccome il socket TCP permette di instaurare una connessione affidabile e byte-oriented, in molti linguaggi di programmazione per scambiare dati mediante esso viene utilizzato il modello a stream come per i file. In particolare in Java si utilizzano le classi **InputStream** e **OutputStream**. Ciò che avviene realmente è che lo l'output stream associato ad un socket A, invia tramite la rete dei dati che verranno ricevuti dall'input stream del socket B che è connesso con A.

2 Apertura di un URL

La classe URL rappresenta un Uniform Resource Locator, ovvero un puntatore verso una risorsa nel World Wide Web. In questo esempio scrivete una classe Java la quale, dato un indirizzo URL passato tramite linea di comando, visualizza ciò che viene inviato dal server (nell'ipotesi che si tratti di testo).

Ricorda

La sintassi generale di un URL è:
`protocol://username:password@domain:port/path?query_string#fragment_id`
dove la porta è sottintesa se si usa quella di default per il protocollo specificato.
Esempio di URL web (qual è la porta di default?)
`http://www.univr.it`
Esempio di URL riferita ad un file sulla propria macchina (non serve essere connessi in rete … perché?)

In questa lezione vengono utilizzate (tra le altre) le classi URL, URLConnection e MalformedURLException.

La **Javadoc** è vostra amica, trovate i dettagli delle classi ai seguenti link:
URL (http://docs.oracle.com/javase/7/docs/api/java/net/URL.html)
URLConnection (http://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html)
MalformedURLException
(http://docs.oracle.com/javase/7/docs/api/java/net/MalformedURLException.html)

Attenzione: copia&incolla da un pdf, a differenza della Javadoc, è, spesso, tuo nemico. Quindi tutti gli esempi si trovano anche in una cartella di file sorgente.

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

class EsempioURL {
    public static void main(String args[]) {
        String indirizzo="";

        if (args.length > 0) {
            indirizzo = args[0];
        }
        else {
            System.out.println("Uso: java EsempioURL <URL>");
            System.exit(-1);
        }

        URL url = null;
        try {
            url = new URL(indirizzo);
```

```
            System.out.println("URL aperto: " + url);

            URLConnection connection = null;
            DataInputStream istream = null;

            System.out.print("Connessione in corso...");
            connection = url.openConnection();
            connection.connect();
            System.out.println("ok.");
            BufferedReader reader =
                new BufferedReader(
                new InputStreamReader(connection.getInputStream()));
            System.out.println("Lettura dei dati...");
            String str;
            while( (str = reader.readLine()) != null ) {
                System.out.println(str);
            }
        } catch (Exception e) {
                System.out.println(e.getMessage());
        }
    }
}
```

Socket UDP

User Datagram Protocol (UDP) e Transmission Control Protocol (TCP) sono protocolli di livello trasporto dell'architettura di rete Internet. La differenza tra UDP e TCP è che UDP è un protocollo di tipo connectionless (ovvero senza connessione), non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi.
Il TCP, invece, è un protocollo di tipo connection-oriented; esso si occupa di "controllo di trasmissione" e rende affidabile la comunicazione (che è inaffidabile a livello IP), fornendo un servizio di ritrasmissione automatica di pacchetti persi o corrotti e un servizio di ricezione basato su acknowledge (ACK) per segnalare quando una sequenza di dati è stata ricevuta.
Sebbene il protocollo TCP permetta di recuperare automaticamente la trasmissione dopo un blocco, questa funzionalità impiega uno scambio di pacchetti molto maggiore rispetto UDP.
Il protocollo UDP, infatti, è da preferire al protocollo TCP ogniqualvolta non è essenziale avere un sistema di ritrasmissione oppure quando l'acknowledge del pacchetto inviato è implicito nella risposta ottenuta a livello di applicazione. Inoltre la complessità di scrittura di programmi con protocollo UDP è ridotta nel numero di oggetti da istanziare per la comunicazione.

ATTENZIONE: In questa esercitazione occorre utilizzare **Wireshark** in modalità super-user per fare delle catture dei pacchetti scambiati tra i propri programmi Java e il comando **IPTABLES** per bloccare alcuni pacchetti simulando un guasto di rete. **Siccome entrambi richiedono la modalità super-user, essi vanno usati o sul proprio PC di proprietà oppure in una macchina virtuale messa appositamente e temporaneamente a disposizione sul PC del laboratorio.**

Per accendere la macchina virtuale occorre cercarla nel menu applicazioni Unity (quadrato in alto a sinistra) digitando "psr". Dopo l'avvio della macchina virtuale, lanciando il file manager e selezionando il link sulla sinistra "Documenti condivisi" è possibile vedere la propria cartella home del PC del laboratorio. Quindi il codice Java può essere scritto, modificato e compilato dal PC host

(quello vero) usando il solito editor di preferenza. I file .class possono essere spostati sulla macchina virtuale uando il link “Documenti condivisi”. La password dell'utente super-user quando si usa “sudo” con Wireshark e IPTABLES è “PAR”.

3 UDP echo server UPPERCASE

In questo esempio andremo ad implementare un echo server che una volta ricevuta una sequenza di lettere in minuscole, le converte in maiuscolo, e le rimanda al client. Occorre lanciare client e server da due diverse shell. Come nome dell’host verrà usato *localhost* che è un po’ come dire “me stesso”. In tal caso l’interfaccia di rete utilizzata è la cosiddetta *loopback*.

Documentazione:
DatagramSocket (http://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html)
DatagramPacket (http://docs.oracle.com/javase/7/docs/api/java/net/DatagramPacket.html)
InetAddress (http://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html)

```
// CLIENT UDP Echo UpperCase

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

class UDPClient {
    public static void main(String args[]) throws Exception {
        try {
            DatagramSocket clientSocket = new DatagramSocket();
            DatagramPacket receivePacket;
            DatagramPacket sendPacket;

            byte[] sendData = new byte[1024];
            byte[] receiveData = new byte[1024];

            BufferedReader inFromUser =
                new BufferedReader(new InputStreamReader(System.in));
            InetAddress IPAddress = InetAddress.getByName("localhost");// IP dst

            System.out.println("Inserisci il messaggio per il server\n");
            String sentence = inFromUser.readLine();
            sendData = sentence.getBytes();
            sendPacket = new DatagramPacket(sendData,sendData.length, IPAddress,
9876); // IP e PORTA dst
            clientSocket.send(sendPacket);

            receivePacket = new DatagramPacket(receiveData,receiveData.length);
            clientSocket.receive(receivePacket);

            String modifiedSentence = new String(receivePacket.getData());
            System.out.println("FROM SERVER:" + modifiedSentence);

            clientSocket.close();
        } catch (Exception e) {
                    System.err.println("Client: errore "+e);
        }
    }
```

```
}

// SERVER UDP Echo UpperCase

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

class UDPServer {
    public static void main(String args[]) throws Exception {
        try {
            DatagramSocket serverSocket = new DatagramSocket(9876);
            DatagramPacket receivePacket;
            DatagramPacket sendPacket;

            System.out.print("Attesa client...");

            while (true) {
                try {
                    byte[] receiveData = new byte[1024];
                    byte[] sendData = new byte[1024];

                    receivePacket =
                        new DatagramPacket(receiveData, receiveData.length);
                    serverSocket.receive(receivePacket);
                    String sentence = new String(receivePacket.getData());
                    System.out.println("RECEIVED: " + sentence);
                    InetAddress IPAddress = receivePacket.getAddress();
                    int port = receivePacket.getPort();
                    String capitalizedSentence = sentence.toUpperCase();
                    sendData = capitalizedSentence.getBytes();

                    sendPacket = new DatagramPacket(sendData,sendData.length,
IPAddress, port);
                    serverSocket.send(sendPacket);
                } catch (UnknownHostException ue) {
                    System.out.println("Errore: " + ue.getMessage());
                }
            }
        } catch (java.net.BindException b) {
            System.err.println("Impossibile avviare il server.");
        }
    }
}
```

Esercizio 1

1. Quali informazioni di rete sono specificate nel codice del client per raggiungere il server?
2. Quali informazioni di rete sono specificate nel codice del server per funzionare correttamente?
3. Cosa succede se lancio da una finestra diversa una nuova istanza del server quando è ancora attiva la prima?
4. Perché il server contiene un ciclo while infinito?
5. Avviate Wireshark in modalità super-user (mediante il comando da shell `sudo wireshark`) e fate partire una cattura sull'interfaccia di loopback.
6. Avviate il server e poi avviate il client.

7. Filtrare i pacchetti UDP applicando il filtro di visualizzazione sulla porta utilizzata.
8. Quanti sono i pacchetti scambiati?

Iptables è un firewall per Linux che consente di specificare regole per il traffico di rete. Per bloccare la ricezione sulla porta 9876 del server eseguire il comando:
`sudo iptables -A INPUT -p udp --dport 9876 -j DROP`
Per riabilitare la ricezione di pacchetti eseguire il comando:
`sudo iptables -D INPUT -p udp --dport 9876 -j DROP`

Esercizio 2

Provate ad usare il servizio di sistema IPTABLES per bloccare i pacchetti in ricezione sulla porta 9876 del server UDP e vedere cosa succede a livello di rete sempre con l'ausilio di wireshark. Provate a bloccare la trasmissione appena dopo che è stato avviato il client e prima di inserire la stringa da tastiera, mandare una richiesta al server e poi sbloccare la trasmissione, infine rimandare la stessa stringa.

4 UDP Calcolatrice client / server

In questo esempio vediamo come sia possibile eseguire operazioni su un server remoto con l'obiettivo di non gravare il client. Il client invia al server una serie di numeri; l'ultimo numero deve essere lo zero. Il server effettua la somma dei valori ricevuti e la trasmette al client.

```
// CLIENT UDP Calcolatrice Client/Server

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

class Client {
    public static void main(String args[]) {
        try {
            DatagramSocket clientSocket = new DatagramSocket();
            DatagramPacket sendPacket;
            DatagramPacket receivePacket;

            byte[] sendData = new byte[1024];
            byte[] receiveData = new byte[1024];

            int val;

            BufferedReader inFromUser =
                new BufferedReader(
                new InputStreamReader(System.in));
            InetAddress IPAddress = InetAddress.getByName("localhost");
            do {
                 System.out.println("Client: inserisci il valore (0 per
terminare):");
                 String val1 = inFromUser.readLine();
                 sendData = val1.getBytes();

                 sendPacket = new DatagramPacket(sendData,sendData.length,
IPAddress, 9876); // IP e PORTA
```

```
                clientSocket.send(sendPacket);
                val = Integer.parseInt(val1.trim());
            } while (val!=0);

            receivePacket =
                new DatagramPacket(receiveData,receiveData.length);
            clientSocket.receive(receivePacket);

            String somma = new String(receivePacket.getData());
            System.out.println("Client: somma ricevuta dal server " +
somma);

            clientSocket.close();
        } catch (Exception e) {
                System.err.println("Client: errore "+e);
        }
    }
}
```

// SERVER UDP Calcolatrice Client/Server

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

class Server {
    public static void main(String args[]) {
        try {
            DatagramSocket serverSocket = new DatagramSocket(9876);
            DatagramPacket receivePacket;
            DatagramPacket sendPacket;

            byte[] sendSomma = new byte[1024];

            int val;
            int somma = 0;

            System.out.println("Attesa client...");

            do {
                    byte[] receiveData = new byte[1024];
                    // riceviamo il numero e facciamo la somma
                    receivePacket =
                    new DatagramPacket(receiveData, receiveData.length);
                    System.out.println("Server: attesa valore");
                    serverSocket.receive(receivePacket);
                    String str1 = new String(receivePacket.getData());
                    val = Integer.parseInt(str1.trim());
                    System.out.println("Server: ricevuto " + val );
                    somma += val;
            } while(val!=0);
```

```
            // rispondiamo al client con la somma
            System.out.println("Server: somma " + somma );
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();
            sendSomma = (Integer.toString(somma)).getBytes();
            sendPacket = new DatagramPacket(sendSomma,sendSomma.length,
IPAddress, port);
            serverSocket.send(sendPacket);
        } catch (Exception e) {
                System.err.println("Server: errore "+e);
        }
    }
}
```

Esercizio 3

1. Provare a bloccare la ricezione e a sbloccarla subito dopo usando IPTABLES in modo da perdere un addendo. Cosa si può notare?
2. Implementare una nuova versione della Calcolatrice UDP con la gestione del rinvio di pacchetti (chiamiamo questa nuova versione Calcolatrice UDP+ACK). Ogni volta che il client invia un pacchetto, dovrà aspettare di ricevere un messaggio di acknowledge. Se l'acknowledge non viene ricevuto entro un certo tempo (**timeout**), bisogna ritrasmettere lo stesso dato.
3. Rifare il punto (1) con la nuova versione del programma. Cosa si può notare?

NOTA: per implementare il timeout in ricezione su un `DatagramSocket` utilizzate `setSoTimeout` dentro un blocco `try … catch(SocketTimeoutException)`. Per la documentazione fare riferimento a:
https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html#setSoTimeout(int)

4. Provare a lanciare 2 client diversi (aprire una nuova shell) e "mixare" l'invio di numeri dai due client. La somma ricevuta da un client dipende anche dai numeri inviati dall'altro? E' una caratteristica positiva?

Da UDP a TCP

In questa sezione andremo ad analizzare in pratica le differenze tra un programma basato su protocollo UDP e TCP. Alla fine scopriremo in quali casi è preferibile utilizzare l'uno piuttosto che l'altro.

5 TCP Echo server UPPERCASE

In questo esempio andremo ad implementare una variante dell'esercizio UDP Echo server UPPERCASE vista nelle lezioni precedenti utilizzando il protocollo TCP (chiamiamola TCP Echo server UPPERCASE).

Documentazione:
All about sockets (https://docs.oracle.com/javase/tutorial/networking/sockets/index.html)

Socket (http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html)
ServerSocket(http://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html)
BufferedOutputStream
(https://docs.oracle.com/javase/7/docs/api/java/io/BufferedOutputStream.html)
BufferedReader (https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html)
PrintWriter (https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html)
InputStreamReader (http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html)

```
//CLIENT EchoServer Uppercase

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

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

        // socket
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintWriter outWriter = null;

        try {
            // IP e porta del server
            clientSocket = new Socket("localhost", 11111);
            System.out.println("Socket creata: " + clientSocket);

            // apertura streams
            reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
            outWriter = new PrintWriter(clientSocket.getOutputStream());
            BufferedReader inFromUser = new BufferedReader(
                new InputStreamReader(System.in));

            while(true)
            {
                String line = null;

                System.out.println("Inserisci il testo da inviare");
                line = inFromUser.readLine();

                // Invio messaggio
                outWriter.println(line);
                outWriter.flush();

                // ricezione risposta dal server
                line = reader.readLine();
                System.out.println("Ricevuto: " + line);
                if (line.equals("STOP"))
                    break;
```

```
            }


            // chiusura streams
            clientSocket.close();
            outWriter.close();
            reader.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

// SERVER EchoServer Uppercase

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

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintWriter outWriter = null;

        try {
            System.out.print("Creazione ServerSocket...");
            serverSocket = new ServerSocket(11111);

            System.out.print("Attesa connessione...");
            clientSocket = serverSocket.accept();
            System.out.println("Conness. da " + clientSocket);

            // apertura streams
            reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
            outWriter = new PrintWriter(clientSocket.getOutputStream());

            while(true)
            {
                // ricezione della stringa
                String text = new String(reader.readLine());
                System.out.println(text);

                // invio della nuova stringa in maiuscolo
                outWriter.println(text.toUpperCase());
                outWriter.flush();
```

```
                if (text.equals("stop"))
                    break;
            }

            // chiusura streams
            reader.close();
            outWriter.close();
            clientSocket.close();
            serverSocket.close();
        } catch (Exception e) {
            System.out.println("Errore: " + e);
        }
    }
}
```

Esercizio 4

1. Avviate Wireshark (mediante il comando da shell `sudo wireshark`) e fate partire una cattura sull'interfaccia di loopback. Avviate il server e poi avviate il client per vedere cosa succede a livello di rete.
2. Osservate i pacchetti scambiati dai vostri programmi impostando un opportuno filtro di visualizzazione. Quanti sono i pacchetti scambiati? Qual'è la differenza di pacchetti scambiati rispetto alla versione UDP precedentemente implementata?
3. Provate ad usare il servizio di sistema IPTABLES per bloccare i pacchetti in ricezione sulla porta 11111 del server TCP (attenzione che dovete cambiare 2 parametri) e vedere cosa succede a livello di rete sempre con l'ausilio di wireshark. Provate a bloccare la trasmissione dopo che è stata creata la connessione e prima di inserire la stringa da tastiera, mandare una richiesta al server e poi sbloccare la trasmissione dopo un brevissimo tempo (eventualmente provare con diversi tempi di blocco). Cosa cambia rispetto alla corrispondente versione UDP precedentemente implementata?

Esercizio 5

1. Modificare il sorgente del server in modo da prendere da linea di comando la porta su cui attendere.
2. Modificare il sorgente del client in modo da prendere da linea di comando: IP del server a cui connettersi, porta alla quale connettersi, stringa da inviare per la conversione.
3. Cosa succede se lancio da una finestra diversa una nuova istanza del server quando è ancora attiva la prima? E se cambio la porta del nuovo server? Tale comportamento vale solo per TCP o anche per UDP?
4. Cosa succede se metto in ascolto il server sulla porta 111? E se lo faccio come super-user (nella virtual machine o sul proprio PC di proprietà)?
5. Scoprire l'indirizzo IP della propria interfaccia di rete usando il comando (da usare per collegarsi al proprio server) `$ ifconfig`
6. Connettere il proprio client al server del vicino.
7. Notare l'ordine di chiusura di stream e socket alla fine dei due programmi. Che differenze ci sono e perché?

Il protocollo TCP è utilizzato per fornire affidabilità della comunicazione implementando un ritrasmissione automatica basata su acknowledge per segnalare quando una sequenza di dati è stata ricevuta. Come mostra l'esempio dell'echo server UPPERCASE, la gestione della ritrasmissione dei pacchetti comporta un costo significativo in termini di pacchetti scambiati sulla rete.
La versione TCP ha trasmesso un totale di >>2 pacchetti mentre la versione UDP ha trasmetto solo 2 pacchetti. Il protocollo UDP è da preferire al protocollo TCP ogniqualvolta non è essenziale avere un sistema di ritrasmissione.
Usare il TCP per l'esempio dell'echo server UPPERCASE è uno spreco di risorse. Il fatto stesso che il client riceva (o non riceva) una risposta con il testo in UPPERCASE è un acknowledge implicito.

6 Calcolatrice client/server TCP

In questa sezione andremo ad implementare una variante dell'esercizio **Calcolatrice client/server UDP** utilizzando il protocollo TCP (chiamiamola Calcolatrice client/server TCP).

```
// CLIENT Calcolatrice Client/Server

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

class CalcClient {
    public static void main(String args[]) {
    try {
        // socket
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintWriter outWriter = null;

        BufferedReader inFromUser = new BufferedReader(new
InputStreamReader(System.in));
        clientSocket = new Socket("localhost", 11111); // IP e porta del
server
        System.out.println("Socket creata: " + clientSocket);

        // creazione stream per lettura e invio della stringa verso il
server
        reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        outWriter = new PrintWriter(clientSocket.getOutputStream());

        int val;

        do {
           System.out.println("Client: inserisci il valore (0 per
terminare):");
           String val1 = inFromUser.readLine();
           outWriter.println(val1);
           outWriter.flush();
```

```
           val = Integer.parseInt(val1.trim());
        } while (val!=0);

        System.out.println("Attesa risposta...");
        String line = null;

        line = new String(reader.readLine());
        System.out.println("Msg dal server: " + line);

        // chiusura streams
        clientSocket.close();
        outWriter.close();
        reader.close();
    } catch (IOException e) {
            System.err.println(e.getMessage());
    }
    }
}
```

// SERVER Calcolatrice Client/Server

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

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null;
        PrintWriter outWriter = null;

        try {
            System.err.println("Creazione ServerSocket");
            serverSocket = new ServerSocket(11111);

            System.out.println("Attesa connessione...");
            clientSocket = serverSocket.accept();
            System.out.println("Connessione da " + serverSocket);

            // apertura streams per lettura e invio della stringa
            // verso il client
            reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
            outWriter = new PrintWriter(clientSocket.getOutputStream());

            String line;
            int y, x = 0;
            do {
                line = new String(reader.readLine());
                y = Integer.parseInt(line);
                System.out.println("Value: " + y);
                x += y;
            } while (y != 0);
```

```
            outWriter.println("Somma = " + x);
            outWriter.flush();

            // chiusura streams
            outWriter.close();
            reader.close();
            clientSocket.close();
            serverSocket.close();
        } catch (Exception e) {
            System.out.println("Errore: " + e);
        }
    }
}
```

Esercizio 6

1. Provare a bloccare la ricezione e a sbloccarla subito dopo usando IPTABLES in modo da perdere un addendo. Cosa si può notare?
2. Confrontare i sorgenti di Calcolatrice UDP + ACK, Calcolatrice UDP e Calcolatrice TCP:
    a. Qual è il codice Java più complesso?
    b. Osservando con Wireshark, il traffico generato dal codice UDP + ACK è paragonabile a quello generato con la versione TCP?
    c. Qual è la versione ottimale di codice Java per questa applicazione tra UDP, UDP + ACK e TCP?
3. Provare a lanciare 2 client diversi (aprire una nuova shell) e “mixare” l’invio di numeri dai due client. E’ possibile con TCP?
4. Modificare client e server in modo che prendano da riga di comando IP/porta e porta, rispettivamente.
5. Creare una versione del server che calcoli i primi N numeri primi e una versione del client che interroghi il server chiedendo i primi N numero primi.

7 Servizio di trasferimento file

Il client richiede al server un file di testo, il cui nome viene specificato tramite la riga di comando, e il server risponde inviando al client il contenuto del file riga per riga. Gestire le eccezioni (file not found, etc.)

```
//CLIENT trasferimento file

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

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

        // socket
        Socket clientSocket = null;

        // streams
```

```
        BufferedReader reader = null;
        PrintWriter outWriter = null;

        if (args.length == 0) {
            System.out.println("Missing file name");
            System.exit(-1);
        }

        try {
            // IP e porta del server
            clientSocket = new Socket("localhost", 11111);
            System.out.println("Socket creata: " + clientSocket);

            // apertura streams per lettura e
            // invio della stringa verso il server
            reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
            outWriter = new PrintWriter(clientSocket.getOutputStream());
        } catch(IOException e){
            e.printStackTrace();
        }

        // invio messaggio
        System.out.println("Invio richiesta per: " + args[0]);
        outWriter.println(args[0]);
        outWriter.flush();

        // stampa risposta del server
        System.out.println("Attesa risposta...");
        String line = null;

        try {
            while ((line = reader.readLine()) != null) {
                System.out.println("Messaggio: " + line);
            }

            // chiusura streams
            clientSocket.close();
            outWriter.close();
            reader.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

//SERVER trasferimento file

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

class FileServer {
```

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

        // sockets
        ServerSocket serverSocket = null;
        Socket clientSocket = null;

        // streams
        BufferedReader reader = null ;
        PrintStream outStream = null ;
        BufferedReader fileReader = null;

        try {
            System.err.println("Creazione ServerSocket");
            serverSocket = new ServerSocket(11111);

            System.out.println("Attesa connessione...");
            clientSocket = serverSocket.accept();
            System.out.println("Connessione da " + clientSocket);

            // apertura streams per lettura e
            // invio della stringa verso il client
            reader = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
            BufferedOutputStream outBuffer =
            new BufferedOutputStream(clientSocket.getOutputStream());
            outStream = new PrintStream(outBuffer, true);

            // ricezione nome file dal client
            String fileName = new String(reader.readLine());
            System.out.println("File richiesto dal client: " + fileName);

            // invio del file al client
            fileReader  = new BufferedReader(
               new InputStreamReader(new FileInputStream(fileName)));

            String tmp = null;
            while ((tmp = fileReader.readLine()) != null)
                outStream.println(tmp);

            // chiusura streams
            serverSocket.close();
            clientSocket.close();
            reader.close();
            outStream.close();
            fileReader.close();
        } catch(IOException e){
              System.out.println("Errore: " + e);
        }
    }
}
```

Esercizio 7

1. Modificare client e server in modo che prendano da riga di comando IP/porta e porta, rispettivamente.
2. Modificare il client in modo che il contenuto del file ricevuto non venga visualizzato a video ma venga salvato su un file di testo. Il nome file è lo stesso di quello richiesto se non viene passato come parametro via riga di comando.
3. Perché non è conveniente implementare questa applicazione con il protocollo UDP?
4. Cosa succede se provate a richiedere al server il file /etc/shadow ? Perché?

8 Java WEB Server

Esercizio 8

Con le conoscenze sui socket acquisite, create un piccolo WEB server in Java che sia possibile interrogare da browser. Il server dovrà essere in ascolto sulla porta 80 e, una volta ricevuta una connessione da parte del client, dovrà leggere la richiesta del client, stamparla a video, spedire un header HTTP + il contenuto di una pagina HTML e chiudere il socket.
Un esempio di header HTTP è:

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: My Java Web Server
```

Attenzione all'ultima riga vuota, che serve da delimitatore tra header HTTP e pagina HTML della risposta del server. Si veda l'esercitazione sulla cattura Wireshark SimpleHTTP.cap per un esempio di comunicazione HTTP in una connessione TCP.

Una volta che il server è in ascolto, usate il browser per navigare su l'indirizzo localhost per visualizzare la risposta del server. Se non è possibile usare la porta 80 (perché?) allora provare sulla porta 8080 e modificare la URL nel browser (si veda il formato delle URL al Capitolo 2 di questa esercitazione).

Thread in Java

Java, al contrario di molti altri linguaggi, fornisce un supporto incorporato per la programmazione multithread. Un programma multithread contiene due o più parti che possono essere eseguite contemporaneamente. Ogni parte di un programma di questo tipo viene definita thread e ogni thread definisce un percorso separato di esecuzione. Questo significa che un singolo programma può eseguire contemporaneamente due o più attività (parallele) a differenza di un programma puramente a esecuzione sequenziale (Figura 5).

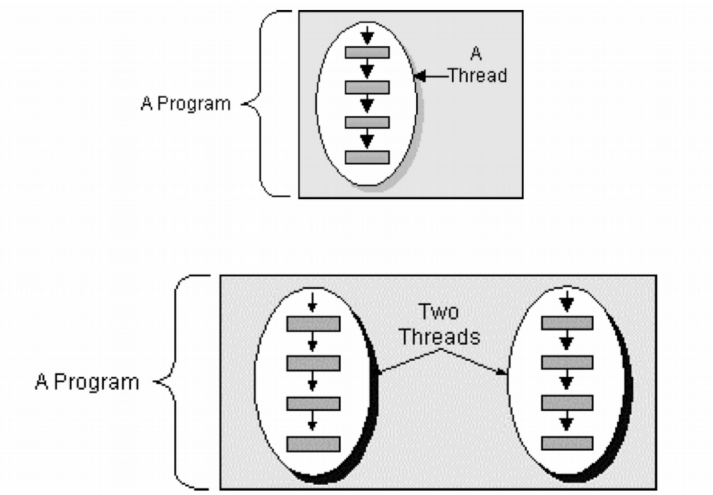


Figura 5. Programma sequenziale e programma multi-thread.

Esistono due modalità per implementare le thread in Java (noi andremo ad utilizzare la prima):

1. come sottoclasse della classe **Thread :**
   - ❏ ridefinizione del metodo **run()** nella classe che estende Thread in cui si andranno a specificare le istruzioni che vogliamo eseguire parallelamente
   - ❏ creazione di un'istanza della sottoclasse tramite **new**
   - ❏ esecuzione della Thread creata andando a chiamare il metodo **start()** sull'istanza creata; la chiamata al metodo andrà implicitamente ad eseguire il metodo run() implementato (ATTENZIONE: NON si deve avviare una Thread chiamando esplicitamente il metodo run())

```
public class MyClass extends Thread {

     public MyClass () {
         // costruttore ...
     }
     public void run () {
         // esecuzione parallela ...
     }
}

public class TestMain {
     public static void main (String[] args) {
         new MyClass().start();
     }
}
```

2. come classe che implementa l'interfaccia **Runnable :**
   - ❏ ridefinizione del metodo run() nella classe che implementa **Runnable**
   - ❏ creazione un'istanza della classe appena estesa tramite **new**
   - ❏ creare un'istanza della classe **Thread** con una nuova **new** e passargli come parametro l'istanza della classe estesa con Runnable
   - ❏ invocare il metodo **start()** sul Thread appena creato

```
public class MyClass implements Runnable {

     public MyClass () {
         // costruttore ...
     }

     public void run () {
         // esecuzione parallela
     }
}

public class TestMain {
     public static void main (String[] args) {
         MyClass c = new MyClass();
         Thread t = new Thread(c);
         t.start();
     }
}
```

Quale dei due meccanismi è preferibile?

Il secondo meccanismo, quello che implementa l'interfaccia Runnable, è più generale perché la classe creata può ereditare da una classe diversa da Thread; quindi è un approccio più flessibile. Il primo meccanismo, quello che andremo ad utilizzare nel prossimo esercizio, è molto facile da utilizzare nelle piccole applicazioni ma è limitato dal fatto che la classe creata debba essere sottoclasse di Thread.

9 TCP Chat client/server

In questo esempio andremo ad implementare una chat testuale. In questo esercizio si andrà ad utilizzare la classe Thread appena illustrata.

Per testare il server utilizzate il programma di sistema TELNET in questo modo:
`telnet localhost 1025`
Quando Telnet sarà connesso al server, ogni volta che scriverete qualcosa e darete invio, lo vedrete apparire sul server e replicato sul client. Per chiudere la sessione in Telnet occorre digitare il comando *close*, oppure *quit* oppure CTRL-C.

```
// SERVER

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

class ChatServer {
    public static void main(String args[]) throws Exception {
        ServerSocket conn = new ServerSocket(1025);
        new Server(conn.accept()).run();
    }
}
```

```
class Server {

    Socket socket;

    public Server(Socket s) {
        this.socket = s;
    }

    public void run() {
        String from;
        BufferedReader reader = null;
        PrintStream outStream = null;

        try {
            reader = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            outStream = new PrintStream(socket.getOutputStream());

            System.out.println("Connected");
            while ((from = reader.readLine()) != null
                        && !from.equals("")) {
                System.out.println(from);
                outStream.print(from + "\r\n");
            }
            socket.close();
        } catch (IOException e) {
            System.out.println(e);
        }
        System.out.println("Disconnected");
    }
}
```

Esercizio 9

1. Un problema di questo server è che termina non appena il client si disconnette, invece di aspettare una nuova connessione. Cosa succede se provate a connettere due client Telnet contemporaneamente? NOTA: dovete usare due shell diverse.
2. Modificare il server in modo che, una volta terminata una connessione con un client, resti in ascolto di altre connessioni. Cosa succede adesso se provate a connettere due client Telnet contemporaneamente?
3. Modificare il server in modo che accetti connessioni simultanee da più client. Ciascun client deve rimanere connesso e funzionare indipendentemente dagli altri.
4. Ancora non abbiamo implementato una vera chat, infatti i client si connettono al server ma non sono in grado di comunicare tra di loro. Come possiamo fare in modo che i messaggi ricevuti dal server vengano inviati alla lista di tutti i client che sono connessi?
5. Cosa succede col server del punto precedente quando un client qualsiasi di quelli connessi si disconnette? Come fare perché il server continui a propagare sui client rimanenti i messaggi ricevuti?
6. Implementare un client in java che si colleghi al server e che invii e riceva i messaggi. Vogliamo poter continuare ad inviare messaggi senza dover bloccare il client fino a quando non riceve un messaggio di risposta. L'utilizzo delle thread consente di gestire in parallelo

invio e ricezione di messaggi senza bloccare il client. Il client dovrà implementare una thread che riceve i messaggi del server e una interazione con l'utente che scrive i messaggi.

7. ...e se volessimo implementare anche la gestione dei nickname?

Socket in C

Adesso vedremo brevemente la programmazione via socket in C. Andremo ad implementare un server sempre in esecuzione e che, appena un client si connette, invia data e ora.

Documentazione:
socket() (http://man7.org/linux/man-pages/man2/socket.2.html)
bind() (http://unixhelp.ed.ac.uk/CGI/man-cgi?bind+2)
struct sockaddr_in (http://www.gta.ufrj.br/ensino/eel878/sockets/sockaddr_inman.html)
connect() (http://linux.die.net/man/2/connect)

Server:

- La chiamata alla funzione socket() crea un socket nel kernel e restituisce un intero chiamato descrittore del socket. A questo passiamo come argomento AF_INET per indicare che vogliamo utilizzare il protocollo IPv4. Il secondo parametro SOCK_STREAM dice che vogliamo che il canale sia affidabile e che quindi implementi l'invio di ack. Il terzo parametro è generalmente lasciato a 0 per indicare al kernel di scegliere il protocollo (TCP in questo caso) (riferirsi alla documentazione per ulteriori dettagli)
- La struttura sockaddr_in contiene le informazioni IP porta e dettagli della connessione
- La funzione bind() assegna i dettagli della struttura sockaddr_in al socket precedentemente creato.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

// SERVER
int main(int argc, char *argv[]){
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;

    char sendBuff[1025];
    time_t ticks;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, '0', sizeof(serv_addr));
```

```
    memset(sendBuff, '0', sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

    listen(listenfd, 10);

    while(1){
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        ticks = time(NULL);
        snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));
        write(connfd, sendBuff, strlen(sendBuff));
        close(connfd);
        sleep(1);
     }
}
```

Client:

- Anche qui viene creato un socket tramite la chiamata alla funzione socket() e viene utilizzata la struttura sockaddr_in per specificare i dettagli del server a cui connettersi.
- A questo punto viene effettuata la connessione tramite la chiamata alla funzione connect() che tenta di connettere il socket creato, con il socket remoto del server.
- Una volta che il socket è connesso, il client riceve le informazioni e le visualizza a video, e poi termina.

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

//CLIENT
int main(int argc, char *argv[]){
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;

    if(argc != 2){
```

```
        printf("\n Usage: %s <ip of server> \n",argv[0]);
        return 1;
    }

    memset(recvBuff, '0',sizeof(recvBuff));
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        printf("\n Error : Could not create socket \n");
        return 1;
    }

    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);

    if(inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0){
        printf("\n inet_pton error occured\n");
        return 1;
    }

                if(connect(sockfd,    (struct    sockaddr    *)&serv_addr,
sizeof(serv_addr))<0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }
    while ( (n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0){
        recvBuff[n] = 0;
        if(fputs(recvBuff, stdout) == EOF){
            printf("\n Error : Fputs error\n");
        }
    }

    if(n < 0){
        printf("\n Read error \n");
    }

    return 0;
}
```

Client/Server UPPERCASE-C

```c
//CLIENT CODE

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

int main(){
  int clientSocket, portNum, nBytes;
  char buffer[1024];
  struct sockaddr_in serverAddr;
  socklen_t addr_size;

  clientSocket = socket(PF_INET, SOCK_STREAM, 0);

  portNum = 7891;

  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(portNum);
  serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);

  addr_size = sizeof serverAddr;
  connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);

  while(1){
    printf("Type a sentence to send to server:\n");
    fgets(buffer,1024,stdin);
    printf("You typed: %s",buffer);

    nBytes = strlen(buffer) + 1;

    send(clientSocket,buffer,nBytes,0);

    recv(clientSocket, buffer, 1024, 0);

    printf("Received from server: %s\n\n",buffer);
  }

  return 0;
}
```

Esercizio 10

Dato il codice del Client specificare in quali righe avvengono le seguenti azioni:

1. Creazione del Socket
2. Configurazione delle impostazioni riguardanti l'indirizzo del Server
3. Configurazione della porta alla quale connettersi
4. Configurazione dell'indirizzo IP alla quale connettersi
5. Connessione del Socket al Server
6. Invio messaggio al Server
7. Ricezione messaggio dal Server
8. Stampa del messaggio ricevuto dal server

```c
//SERVER CODE
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>

int main(){
  int welcomeSocket, newSocket, portNum, clientLen, nBytes;
  char buffer[1024];
  struct sockaddr_in serverAddr;
  struct sockaddr_storage serverStorage;
  socklen_t addr_size;
  int i;

  welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);
  portNum = 7891;
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(portNum);
  serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
  memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
  bind(welcomeSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr));

  if(listen(welcomeSocket,5)==0)
    printf("Listening\n");
  else
    printf("Error\n");

  addr_size = sizeof serverStorage;
  /*loop to keep accepting new connections*/
  while(1){
    newSocket = accept(welcomeSocket, (struct sockaddr *) &serverStorage, &addr_size);
    /*fork a child process to handle the new connection*/
    if(!fork()){
      nBytes = 1;
      /*loop while connection is live*/
      while(nBytes!=0){
        nBytes = recv(newSocket,buffer,1024,0);

        for (i=0;i<nBytes-1;i++){
          buffer[i] = toupper(buffer[i]);
        }
        send(newSocket,buffer,nBytes,0);
      }
      close(newSocket);
      exit(0);
    }
    /*if parent, close the socket and go back to listening new requests*/
    else{
      close(newSocket);
    }
  }
  return 0;
}
```

Esercizio 11

Dato il codice del Server specificare in quali righe avvengono le seguenti azioni:

1. Creazione del Socket
2. Configurazione della struttura del Server
3. Configurazione della porta alla è possibile connettersi
4. Configurazione dell'indirizzo IP alla quale connettersi
5. Connessione del Socket al Server
6. Attesa di un Client che si connetta
7. Accettazione della connessione
8. Lettura del messaggio ricevuto dal Client
9. Invio risposta al Client