Laboratorio di "Sistemi Distribuiti"
A.A. 2024-2025

Socket (TCP)

Emanuele Petriglia

Slide aggiornate al 27 Marzo 2025

Indice dei contenuti

1. Introduzione a TCP
2. TCP in Java
3. Esercizio A
4. Esercizio B
5. Esercizio C

1. Introduzione a TCP

Riassunto di UDP

Caratteristiche di UDP:

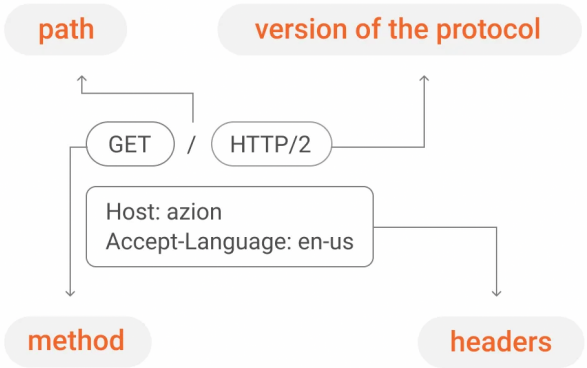
- **Non orientato alla connessione**: ogni datagramma è indipendente.
- **Inaffidabile**: i datagrammi si possono perdere o cambiare ordine di arrivo.
- **Veloce e leggero**: ottimo per applicazioni sensibili alla latenza (streaming) o limitate (DNS, NTP).

UDP in Java:

- Classi `DatagramSocket`, per ricevere/inviare datagrammi, e `DatagramPacket` per costruire un datagramma.
- Contenuto codificato in **array di byte** per invio e ricezione.

Tipologie di protocollo

- **Protocollo testuale**: testo leggibile dall'uomo. Guidato da spazi e ritorni a capo.



Fonte

- **Protocollo binario**: contenuto codificato in campi ben delineati, con regole di costruzione e analisi dei campi.

Fonte

Attenzione!

In entrambe i casi il contenuto è sempre codificato come **array di byte**!

Verso TCP

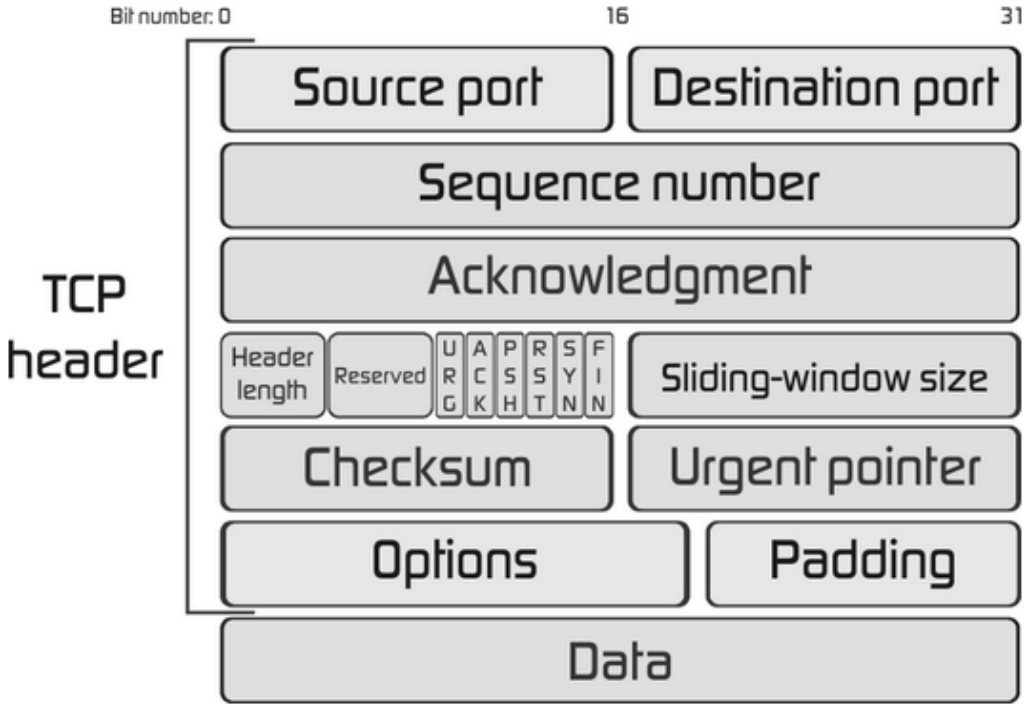
Funzionalità	**UDP**	**TCP**
Tipo connessione	Non orientato	Orientato
Affidabilità	Inaffidabile	Affidabile
Controllo contenuto	Garantito (opzionale)	Garantito
Ordine dei dati	Non garantito	Garantito
Segmentazione dati	Basato su datagrammi	Basato su stream
Complessità	Bassa	Alta
Casi d'uso	Streaming, DNS, NTP	HTTP, SMTP, SSH

Da ricordare!

TCP è: **orientato alla connessione**, **affidabile** e **basato su stream/flusso**.

Segmenti TCP

Struttura di un singolo segmento (header da 20-60 byte!):



Fonte

Segmenti vs datagrammi

- TCP: Segmenti $\rightarrow$ Impliciti (gestiti da TCP).
- UDP: Datagrammi $\rightarrow$ Espliciti (gestiti dallo sviluppatore).

2. TCP in Java

Classi principali

- Per il server: `ServerSocket` per accettare nuove connessioni.
- Per il client: `Socket` per la comunicazione tra client e server.
- Per la trasmissione dati (**flusso di byte bidirezionale!**):
  - `InputStream` rappresenta uno stream di byte in ingresso,
  - `OutputStream` rappresenta uno stream di byte in uscita.

ServerSocket e Socket

- `ServerSocket(int port)`
  - Coda in ingresso di massimo 50 connessioni.
  - `accept()`: restituisce un nuovo socket per una nuova connessione in ingresso (**BLOCCANTE**!).
- `Socket(String host, int port)`
  - Alla creazione si collega in automatico.
  - `getInputStream()`: restituisce un `InputStream` per ricevere byte.
  - `getOutputStream()`: restituisce un `OutputStream` per inviare byte.

Attenzione!

Le istanze di `ServerSocket` e `Socket` devono essere chiuse quando non più necessarie. Suggerimento: usare `try/catch` invece di chiamare a mano `close()`!

Esempio Client (ExampleClient.java)

```java
import java.net.Socket;
import java.util.Scanner;

public class ExampleClient {
    public static void main(String[] args) {
        final var serverAddress = "127.0.0.1";
        final var serverPort = 8080;
        final var messages = 3;

        try (var socket = new Socket(serverAddress, serverPort);
             var scanner = new Scanner(socket.getInputStream())) {
            for (int index = 0; index < messages; index++) {
                var message = scanner.nextLine();
                System.out.printf("Stringa ricevuta dal server: \"%s\"\n",
                ↪  message);
            }
            var output = socket.getOutputStream();
            output.write("OK!\n".getBytes());
        } catch (Exception ex) {
            // ...
        }
    }
}
```

Esempio Server (ExampleServer.java)

```java
public class ExampleServer {
    public static void main(String[] args) {
        try (var serverSocket = new ServerSocket(8080)) {
            while (true) {
                var socket = serverSocket.accept();
                processClient(socket);
            }
        } catch (Exception ex) { /* ... */ }
    }

    private static void processClient(Socket socket) {
        try (socket;
             var scanner = new Scanner(socket.getInputStream())) {
            var output = socket.getOutputStream();
            output.write("Hello\n".getBytes());
            output.write("World\n".getBytes());
            output.write("!\n".getBytes());

            var reply = scanner.nextLine();
            System.out.printf("Il client ha risposto con \"%s\"\n", reply);
        } catch (Exception ex) { /* ... */ }
    }
}
```

InputStream e OutputStream

- `InputStream` con i metodi `read()` e `readN()` per ricevere byte o array di byte.
- `OutputStream` con uno dei tre metodi `write()` per inviare byte o array di byte.

Attenzione!

Entrambi vengono automaticamente chiusi quando il socket associato viene chiuso.

Esempio invio byte (BinaryClient.java)

```java
import java.net.Socket;

public class BinaryClient {
    public static void main(String[] args) {
        final var serverAddress = "127.0.0.1";
        final var port = 8080;

        try (var socket = new Socket(serverAddress, port)) {
            System.out.println("Connesso al server");

            var output = socket.getOutputStream();

            System.out.printf("Invio del numero %d al server...\n",
            ↪  socket.getLocalPort());
            var buf = intToByteArray(socket.getLocalPort());
            output.write(buf); // 4 byte!
            System.out.println("Inviato!");
        } catch (Exception ex) {
            // ...
        }
    }
    // ...
}
```

Esempio ricezione byte (BinaryServer.java)

```java
public class BinaryServer {
    public static void main(String[] args) {
        final var port = 8080;
        try (var serverSocket = new ServerSocket(port)) {

            while (true) {
                try (var socket = serverSocket.accept()) {
                    System.out.printf("Nuovo client (%s) connesso!\n",
                    ↪  socket.getPort());

                    var input = socket.getInputStream();
                    var buf = input.readNBytes(4);
                    var value = byteArrayToInt(buf, 0);
                    System.out.printf("Valore ricevuto: %d\n", value);
                } catch (Exception ex) {
                    // ...
                }
            }
        } catch (Exception ex) {
            // ...
        }
    }
    // ...
}
```

3. Esercizio A

Consegna

Scrivere un'applicazione client/server TCP in Java per un servizio di ora corrente.

Client $\rightarrow$ Server:

- `GET_TIME <timezone>` per ottenere l'ora attuale in un determinato fuso orario (può essere vuoto, in tal caso il fuso orario predefinito).
- `QUIT` per terminare la connessione.

Server $\rightarrow$ Client:

- L'ora attuale come stringa nel fuso orario fornito,
- Un messaggio di errore in caso di fuso orario invalido.

Ogni messaggio del client o del server è separato da un ritorno a capo.

Esempio

Esempio di interazione con il client:

```
1  > Inserisci un fuso orario ("Esci" per uscire): America/New_York
2  Risposta dal server: "2025-03-26T14:10:18.073609-04:00[America/New_York]"
3  > Inserisci un fuso orario ("Esci" per uscire):
4  Risposta dal server: "2025-03-26T19:10:20.761529800+01:00[Europe/Rome]"
5  > Inserisci un fuso orario ("Esci" per uscire): Europe/Rome
6  Risposta dal server: "2025-03-26T19:10:24.536166900+01:00[Europe/Rome]"
7  > Inserisci un fuso orario ("Esci" per uscire): BlahBlah
8  Errore: fuso orario sconosciuto!
9  > Inserisci un fuso orario ("Esci" per uscire): Esci
```

Attenzione!

Quando l'utente invia il fuso orario "`America/New_York`", il client invia al server il comando "`GET_TIME America/New_York`".

Che succede se più client si collegano e inviano comandi? Provare!

Suggerimenti

- Partire dallo scheletro!
- `ZoneId` e `ZonedDateTime` per il fuso orario e l'ora in Java:

```
1  import java.date.*;
2  var rawTimezone = "Europe/Rome";
3  var timezone = ZoneId.of(rawTimezone);
4  var timezone = ZoneId.systemDefault();
5  var time = ZonedDateTime.now(timezone);
6  var str = time.toString();
```

- `Scanner` e in particolare `nextLine()` per leggere una linea.
- Per dividere una stringa in base agli spazi:

```
var arrayStringhe = "Ciao mondo".split("\\s+")
```

4. Esercizio B

Consegna

Estensione dell'esercizio C del laboratorio su UDP: convertire il codice in modo che utilizzi TCP.

Convertire sia la soluzione che usa il **protocollo binario** e la soluzione che usa il **protocollo testuale**.

Se non si ha svolto l'esercizio si può ottenere la soluzione da e-Learning.

Fare infine delle prove di esecuzione con più client.

Attenzione!

Nella conversione ricordatevi di gestire i casi di errore (il client e il server vanno chiusi correttamente!

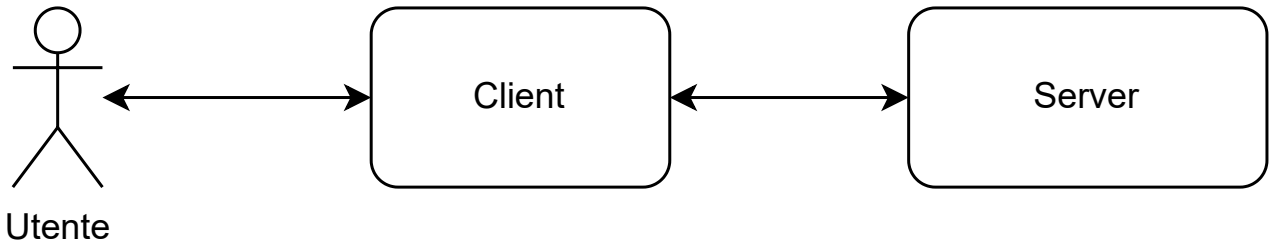
5. Esercizio C

Consegna (pt. 1)

Scrivere un'applicazione client/server per una calcolatrice:

- Il **client** interagisce con l'utente in un ciclo infinito, invia i comandi di calcolo al server e visualizza la risposta con il risultato.
- Il **server** calcola le operazioni richieste dal client e risponde con il risultato.

Il protocollo è di tipo **testuale** ed è specificato nella slide successiva.



Consegna (pt. 2)

Ogni comando è composto da un nome e dagli argomenti. A partire dal nome si determina quanti argomenti ci sono. I **comandi disponibili** sono:

- `CALC ADD`, `CALC SUB`, `CALC MUL` e `CALC DIV` effettuano l'addizione, sottrazione, moltiplicazione e divisione tra due operandi.
- `CALC SIN`, `CALC SQRT` effettuano il seno e la radice quadrata di un operando.
- `RANDOM` restituisce un numero casuale tra 0 e 1.
- `HISTORY` restituisce l'elenco completo dei comandi inviati in precedenza. Non ha argomenti.
- `QUIT` termina l'invio dei comandi.

Il server risponde con due messaggi di riposta:

- `RESULT` seguito dal risultato.
- `ERROR` seguito da un messaggio di errore.

Consegna (pt. 3)

- **Gestione degli errori**: eventuali errori (es. divisione per zero o numeri a virgola mobile malformati) si devono gestire sia lato client sia lato server!
- **Usare il carattere newline** come separatore tra nome del comando e i singoli argomenti.
- Esempio di interazione:

```
> CALC ADD 10 5
RESULT 15
> CALC DIV 10 0
ERROR Divisione per zero
> CALC DIV 10 3
RESULT 3.3333333333333335
> CALC SQRT 25
> RANDOM 5
RESULT 0.42
> QUIT
```

Attenzione!

I numeri sono di tipo virgola mobile (`double` in Java).