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

Socket (UDP)

Emanuele Petriglia

Slide aggiornate al 21 Marzo 2025

Indice dei contenuti

1. Introduzione ai laboratori

Informazioni utili

- Date, orari e luoghi (visibile anche su Gestione Orari):
  - Turno 1: 21/3, 28/3, 4/4, 11/4, 9/5, 16/5, 23/5, 30/5, 6/6.
  - Turno 2: 20/3, 27/3, 3/4, 10/4, 8/5, 15/5, 22/5, 29/5, 5/6.
  - Per entrambi: 8:30-11:30 LAB14A1
- In caso di problemi scrivere sul forum dedicato su e-Learning.
- **Importante**: è richiesto l'uso della macchina virtuale di laboratorio (maggiori info nell'avviso pubblicato).
- Slide ed esercizi (consegne e soluzioni) sul corso e-Learning.

Attenzione ai link!

Nelle slide il testo colorato di blu rappresenta un link che si può cliccare ed aprire.

Obiettivi

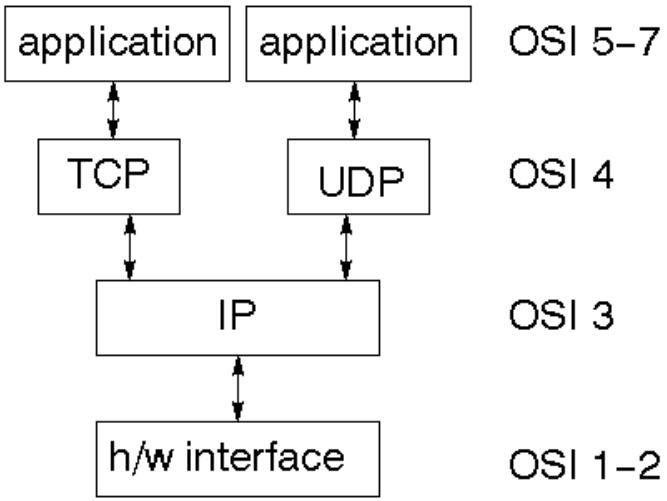
- **Applicazione pratica dei concetti teorici**
  - IP/TCP, architettura dei server, protocolli...
- **Sviluppo di competenze tecniche**
  - Java, JavaScript, Visual Studio Code...
- **Promozione del pensiero critico e della risoluzione dei problemi**
  - Come strutturare un protocollo o un'interfaccia? Come risolvere un problema di compilazione?
- **Rafforzamento dell'apprendimento collaborativo**
  - Sia durante i laboratori sia per il progetto di gruppo opzionale.

Piano

1. Socket (UDP) ← **laboratorio di oggi**
2. Socket (TCP)
3. Concorrenza 1 (Thread)
4. Concorrenza 2 (Sincronizzazione)
5. Servlet
6. REST
7. JavaScript (DOM)
8. JavaScript (AJAX)

2. IP, UDP e TCP in Java

Stack IP/TCP



Fonte

- **Indirizzi IP**: identifica un nodo in una rete.
  - Nei laboratori useremo sempre `localhost` o `127.0.0.1`.
- **Numero di porta**: identificano un servizio all'interno di un nodo.
  - Sia il client che il server devono conoscere la porta.
  - Nei laboratori useremo numeri 1024-15000.
- **Socket**: combinazione di un indirizzo IP e di un numero di porta.
  - Esempio: `127.0.0.1:8080`.

UDP e TCP

- **IP**: trasmissione di pacchetti
  - Servizio non orientato alla connessione (ogni pacchetto è indipendente).
  - Controllo dell'errore solo sull'header, non sul contenuto.
- **UDP**: trasmissione di datagrammi
  - Uguale a IP, in aggiunta avviene un controllo dell'errore anche sul contenuto.
- **TCP**: trasmissione di segmenti
  - Orientato alla connessione.
  - Instaura un canale di comunicazione bidirezionale.
  - Verrà approfondito al prossimo laboratorio.

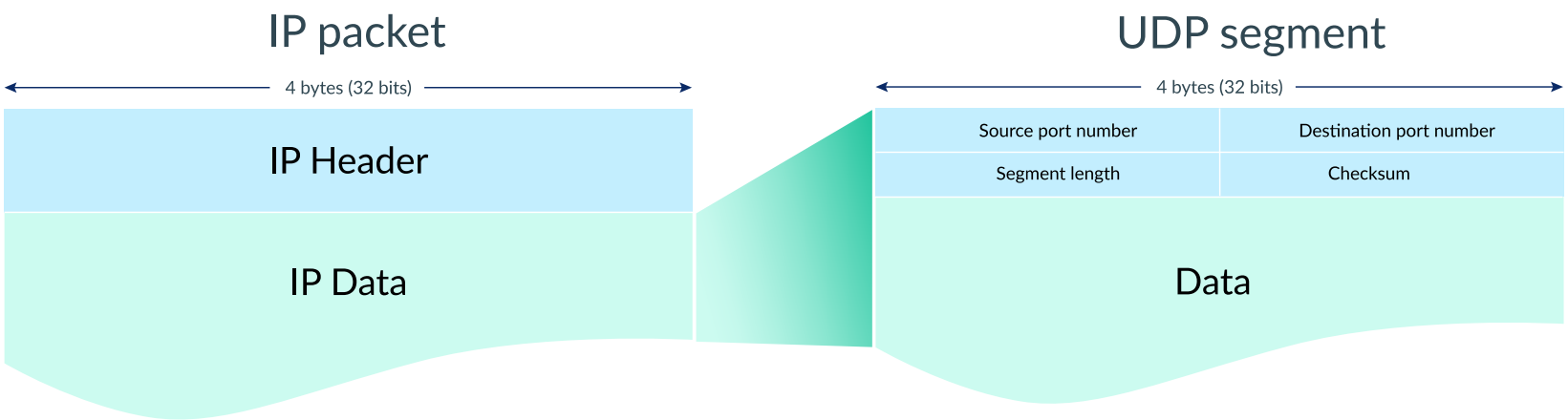
Perché UDP?

- Nel mondo reale:
  - Streaming audio e video
  - DNS (Domain Name System)
  - Broadcasting e Multicasting (es. IPTV, DHCP...)
  - Internet of Things
- In laboratorio:
  - Semplice da usare
  - Facilita la comprensione di TCP

Fonte

Caratteristiche principali di UDP

1. Ogni datagramma ha dimensione massima di 65 507 byte.
   - In caso di messaggi più grandi bisogna spezzare manualmente il contenuto e gestire l'ordine (per questo c'è TCP).
   - Più un datagramma è grande più è probabile perderlo.
2. L'ordine dei datagrammi non è garantito.
3. Ogni datagramma è indipendente
   - Conseguenza: non c'è una reale distinzione tra client e server.



Fonte

UDP in Java

Due classi principali:

- `DatagramSocket`: socket per invio e ricezione di datagrammi.
- `DatagramPacket`: un singolo datagramma inviato o ricevuto.

Classe di supporto:

- `InetAddress`: rappresenta un indirizzo IP. Nei laboratori useremo sempre l'indirizzo `127.0.0.1`:

```
var serverAddress = InetAddress.getByName("127.0.0.1");
```

DatagramSocket

`DatagramSocket`: socket per invio e ricezione di datagrammi. Metodi principali:

- `send(DatagramPacket p)`: invia un datagramma.
- `receive(DatagramPacket p)`: riceve un datagramma.
- `close()`: chiude il socket.

Attenzione

Il metodo di ricezione blocca il processo finché non si riceve un datagramma!

Costruttori:

- Invio: `DatagramSocket()`: usa una porta temporanea del sistema.
- Ricezione: `DatagramSocket(int port)`: usa la porta specificata.

DatagramPacket

`DatagramPacket`: un singolo datagramma inviato o ricevuto.
Costruttori principali:

- Per inviare: `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
- Per ricevere: `DatagramPacket(byte[] buf, int length)`

Attenzione!

Che sia per inviare o ricevere un datagramma, è obbligatorio preparare un buffer di dimensioni fissate, anche se poi il datagramma è più corto.

Esempio client

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Client {
    public static void main(String[] args) {
        try (var socket = new DatagramSocket()) {
            var message = "Hello World!";
            var payload = message.getBytes();

            var serverAddress = InetAddress.getByName("127.0.0.1");
            var port = 8080;

            var datagram = new DatagramPacket(payload, payload.length,
            ↪  serverAddress, port);
            socket.send(datagram);

            System.out.println("Datagramma inviato!");
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Esempio server

```
1   public class Server {
2       public static void main(String[] args) {
3           var port = 8080;
4           try (var socket = new DatagramSocket(port)) {
5               var buf = new byte[1024];
6
7               System.out.println("Server in ascolto...");
8               while (true) {
9
10                  var datagram = new DatagramPacket(buf, buf.length);
11                  socket.receive(datagram);
12
13                  var message = new String(datagram.getData(), 0,
                    ↪  datagram.getLength());
14                  System.out.printf("Datagramma ricevuto (porta %d) con
                    ↪  messaggio: \"%s\"\n",
15                                  datagram.getPort(),
16                                  message);
17              }
18          } catch (Exception e) {
19              e.printStackTrace();
20          }
21      }
22  }
```

3. Esercizio A

Consegna

Scrivere un client e server che scambiano, tramite protocollo UDP, dei messaggi composti da stringhe nel seguente modo:

- **Client**: in un ciclo infinito legge una riga da standard input, la invia al server e riceve una risposta con la stringa in maiuscolo.
- **Server**: per ogni datagramma ricevuto estrae la stringa, la modifica in maiuscolo e la invia al client.

Porta del server

È importante scegliere una porta per il server! Attenzione a quando si invia la risposta: la porta del client la si può recuperare dal datagramma!

Formato del datagramma

In questo esercizio il datagramma contiene dei byte che rappresentano una stringa (UTF-8). Attenzione che viene incluso anche il ritorno a capo!

Suggerimenti

- Partire dallo scheletro fornito su e-Learning, contiene i due file Java di base.
- Usare il metodo `strip()` sulle stringhe per rimuovere eventuali spazi all'inizio e alla fine (compreso il ritorno a capo).
- Una volta realizzato il client e il server, fare delle prove:
  1. Cosa succede se eseguo più client in contemporanea?
  2. Cosa succede se il server non è in esecuzione?
  3. Cosa succede se la dimensione del buffer del datagramma è più piccola della stringa da ricevere?
  4. Cosa succede se il server dopo aver ricevuto un datagramma aspetta qualche secondo prima di rispondere? Usare il seguente codice:

```
1  try {
2      Thread.sleep(5000); // 5 secondi.
3  } catch (InterruptedException e) {
4      System.err.println("Sleep interrotto: " + e.getMessage());
5  }
```

4. Esercizio B

Consegna (pt. 1)

Realizzare un server che, tramite protocollo UDP, esponga un servizio di contatori. La comunicazione con il client avviene tramite un **protocollo binario**. Realizzare pertanto anche il client.

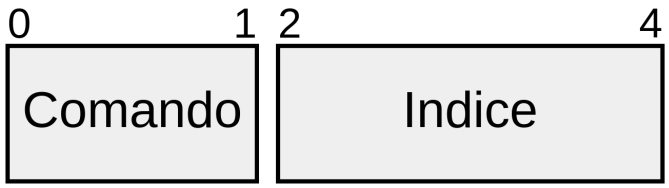
Il server mantiene una mappa di contatori indicizzati da interi. I contatori iniziano da zero. Il server supporta i seguenti comandi:

- **Lettura di un contatore**: dato un indice, restituisce il valore attuale del contatore.
- **Incremento di un contatore**: dato un indice, incrementa di uno il valore del contatore.

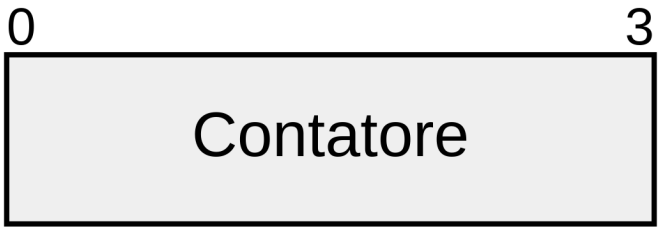
Consegna (pt. 2)

Codifica dei datagrammi:

- Client $\rightarrow$ Server: 5 byte (1 per il comando e 4 per l'indice del contatore).



- Server $\rightarrow$ Client: 4 byte (valore del contatore).



Attenzione!

- Il server invia un datagramma al client **solo** se quest'ultimo richiede il comando di lettura.
- In Java, un intero ha dimensione di 32 bit (4 byte).

Suggerimenti

Partire dallo scheletro! Contiene i seguenti metodi per codificare degli interi:

- `byteArrayToInt(byte[] bytes, int offset)`
- `intToByteArray(int value)`

Esempio di utilizzo:

```
1  int x = 5;
2  var buffer = intToByteArray(x);
3  int y = byteArrayToInt(buffer, 0);
4  // y == x
```

Il comando è rappresentato da un byte, i valori sono a libera scelta (es. 0x01 per la lettura e 0x02 per l'incremento).

Promemoria: usate una `HashMap`:

```
1  Map<Integer, Integer> map = new HashMap<>();
2  var counter = map.getOrDefault(523, 0); // Contatore con indice 523.
3  map.put(622, 4); // Contatore con indice 622.
```

5. Esercizio C

Consegna (pt. 1)

Realizzare il gioco **"Guess My Number"** con un'architettura client/server su protocollo UDP.

Regole del gioco:

- Scelta del numero: il server sceglie un numero segreto casuale $x \in [0, 100]$.
- Tentativi: il client ipotizza un numero y e lo invia al server.
- Feedback: a ogni tentativo il server risponde con tre messaggi (codificabili come si vuole):
  - "Troppo alto" $\rightarrow y > x$.
  - "Troppo basso" $\rightarrow y < x$.
  - "Indovinato" $\rightarrow y = x$.

  Il client termina quando ha indovinato il numero x.

Consegna (pt. 2)

Il server per ogni client mantiene un numero da indovinare diverso (*suggerimento*: usare `HashMap` tra la porta del client e il numero da indovinare).

Il protocollo di comunicazione deve essere progettato e implementato. Due possibilità:

- **Protocollo binario**: formato da datagrammi di dimensione fissa con le informazioni codificate.
- **Protocollo testuale**: formato da datagrammi che contengono testo codificato.

Suggerimento: provare a implementare entrambe le tipologie di protocollo.

Suggerimenti

- Partire dallo scheletro presente su e-Learning!
- Iniziare da un server che gestisce un solo client (o più client con lo stesso numero segreto).
- Iniziare con un protocollo testuale, per poi provare un protocollo binario.
  - Per il protocollo testuale:
    - Stringa → Intero: `Integer.parseInt(String s)`
    - Intero → Stringa: `toString()`
  - Per il protocollo binario, stabilire in anticipo il formato dei messaggi. Un campo (anche di un solo byte) può identificare il tipo di messaggio (risposta del server o tentativo del client). Usare la stessa tecnica dell'esercizio B.